A l'intérieur d'un solveur SAT

François Schwarzentruber

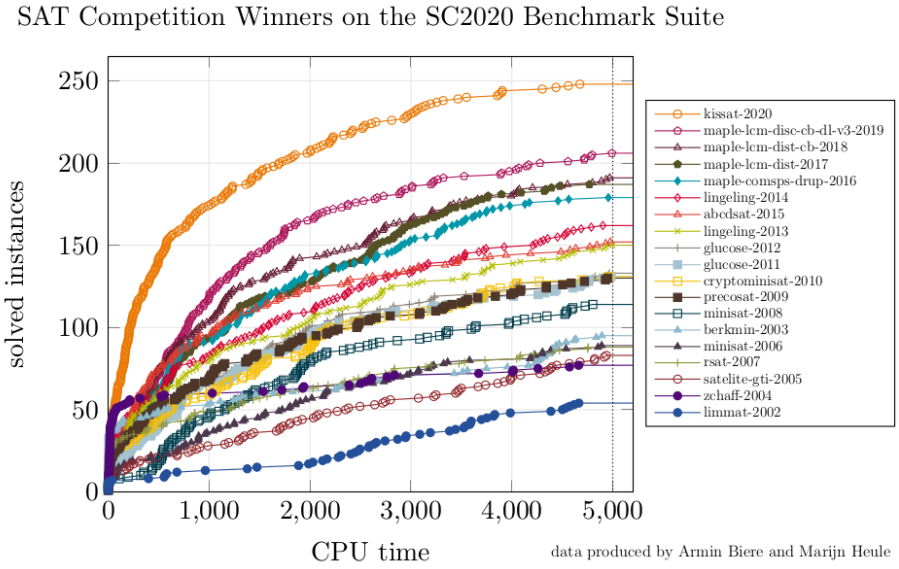
7 mars 2023

SAT
entrée : une formule φ en forme normale conjonctive
sortie : une valuation qui satisfait φ si φ est satisfiable ; unsat si φ est insatisfiable.

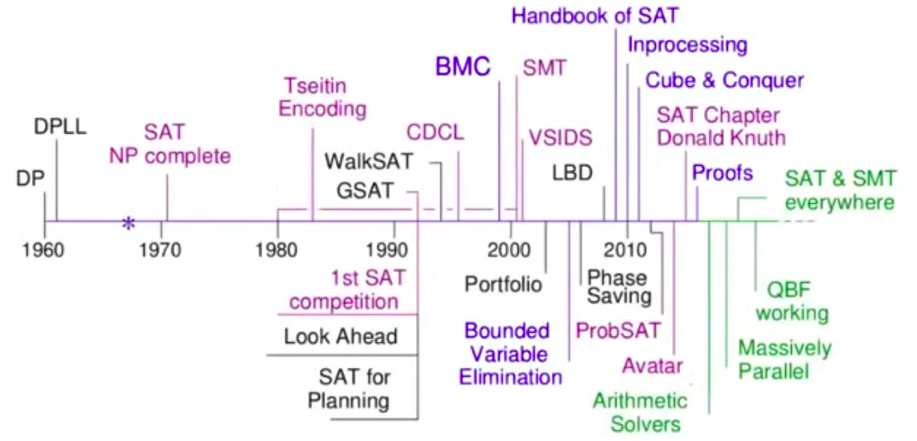
1 Applications

— bounded model checking
— planification automatique
— sous-routine pour des solveurs d'autres logiques (SMT, QBF, logique du premier ordre, ASP, etc.)
— résoudre des problèmes mathématiques (problème des triplets pythagoriciens booléens) [HKM16]

2 Performances des solveurs SAT



3 Historique



par Armin Biere

4 Notations

Soit AP un ensemble dénombrable de variables propositionnelles.

Définition 1 (valuation partielle) Une valuation partielle est une fonction partielle de AP dans $\{0,1\}$.

Exemple 1
$\nu = \emptyset$
$\nu = \{(p,0),(q,1)\}$.

Remarque 2 Soit ℓ un littéral. $\neg\ell = \begin{cases} \neg p & \text{si } \ell = p \\ p & \text{si } \ell = \neg p \end{cases}$

Définition 3 Pour tout $i \in \{0,1\}$, on note $\begin{cases} \nu[p := i] & = \nu \cup \{(p,i)\} \\ \nu[\neg p := i] & = \nu \cup \{(p,1-i)\}. \end{cases}$

Définition 4 Soit ℓ un littéral. On définit la notation $\nu \models \ell$ pour dire que $\begin{cases} (p,1) \in \nu & \text{si } \ell = p \\ (p,0) \in \nu & \text{si } \ell = \neg p \end{cases}$

Définition 5 (littéral assigné) ℓ est ν-assigné si $\nu \models \ell$ or $\nu \models \neg\ell$.

Définition 6 (condition de vérité d'une forme normale conjonctive)
— $\nu \models \bigwedge_{i\in\{1,\dots,n\}} \bigvee_{j\in\{1,\dots,k_i\}} \ell_{i,j}$ si pour tout $i \in \{1,\dots,n\}$, il existe $j \in \{1,\dots,k_i\}$ tel que $\nu \models \ell_{i,j}$.
— $\nu \models \neg \bigwedge_{i\in\{1,\dots,n\}} \bigvee_{j\in\{1,\dots,k_i\}} \ell_{i,j}$ si il existe $i \in \{1,\dots,n\}$ pour tout $j \in \{1,\dots,k_i\}$, on a $\nu \models \neg\ell_{i,j}$.

5 Backtracking

— entrée : une valuation partielle ν, une formule φ sous forme normale conjonctive
— sortie : *true* si ν peut être étendue en une valuation totale qui satisfait φ

fonction *dpll*(ν,φ)
  si $\nu \models \neg\varphi$ **alors renvoyer** *false*
  si $\nu \models \varphi$ **alors renvoyer** *true*
  sinon
    choisir une variable p non ν-assignée
    renvoyer *dpll*$(\nu[p := 0],\varphi)$ ou *dpll*$(\nu[p := 1],\varphi)$

Exemple 7 $\overbrace{(p \vee q)}^{\alpha} \wedge \overbrace{(p \vee \neg q)}^{\beta} \wedge \overbrace{(\neg p \vee q)}^{\gamma} \wedge \overbrace{(\neg p \vee \neg q)}^{\delta} \wedge \overbrace{(r \vee s)}^{\epsilon}$.

6 Davis–Putnam–Logemann–Loveland (DPLL)

6.1 Clause unitaire

Définition 8 (clause unitaire) Une clause $\bigvee_j \ell_j$ est ν-unitaire s'il existe j_0 tel que pour tout $j \neq j_0$, $\nu \models \neg\ell_j$ et ℓ_{j_0} non ν-assigné.

$$\not{p} \vee \not{q} \vee \not{s} \vee \ell.$$

Exemple 2 (élimination des littéraux purs) Suppose the formula φ is

$$(p \vee q \vee \ell) \wedge (r \vee s \vee \neg u \vee \ell) \wedge (s \vee \neg t).$$

If ℓ is ν-unassigned, we replace ν by $\nu[\ell := 1]$.

6.2 Pseudo-code standard

fonction $dpll(\nu, \varphi)$
  ν := propagationsUnitaires(ν, φ)
  si $\nu \models \neg\varphi$ **alors renvoyer** $false$
  si $\nu \models \varphi$ **alors renvoyer** $true$
  sinon
    choisir une variable p non ν-assignée
    renvoyer $dpll(\nu[p := 0], \varphi)$ ou $dpll(\nu[p := 1], \varphi)$

— entrée : une valuation partielle ν, une CNF φ
— sortie : ν' qui étend ν telle que ν peut être étendue en une valuation satisfaisant φ ssi ν' peut être étendue en une valuation satisfaisant φ
fonction propagationsUnitaires(ν, φ)
  tant que il y a une clause ν-unitaire dans φ avec ℓ non ν-assigné
    $\nu := \nu[\ell := 1]$
  renvoyer ν

Exemple 3 Let us consider the following set of clauses :
Let us choose a value for a and for b : a, b true !
Then if we suppose p is true, we infer q by δ and there is a contraction with ϵ. Then if we suppose p is false, we infer q by β and there is a contraction with γ. (*)
But the boring reasoning (*) also occur for $a, \neg b$, $\neg a, b$ and $\neg a, \neg b$!

6.3 Pseudo-code avec backtracking explicite

fonction $dpll(\nu, \varphi)$
  si $\nu = \bullet$ **alors renvoyer** false
  sinon
    ν := propagationsUnitaires(ν, φ)
    si $\nu \models \neg\varphi$ **alors renvoyer** $dpll($ $backtrack(\nu, \varphi), \varphi$ $)$
    si $\nu \models \varphi$ **alors renvoyer** $true$
    sinon
      choisir une variable p non ν-assignée
      renvoyer $dpll(\nu[\ell := 1], \varphi)$

To make this algorithm work, the structure ν is augmented as follows. We mark each chosen literal in by :

p	:	p has been derived by unit propagation ;
$\neg p$	:	$\neg p$ has been chosen ;
p	:	p has been derived by backtracking.

Exemple 9 Here are examples of returned values for $backtrack(\nu, \varphi)$.

ν	$backtrack(\nu, \varphi)$
$\neg p\ q\ r$	p
$p\ q$	$\bullet$
$\neg p\ q\ r\ \neg s\ t$	$\neg p\ q\ r\ s$
$\neg p\ q\ r\ \neg s\ t\ u\ v\ w\ a\ e\ b$	$\neg p\ q\ r\ \neg s\ t\ u\ v\ w\ a\ \neg e$
$\neg p\ q\ r\ s$	$\neg p$

To test whether φ is satisfiable or not, we call $dpll(\epsilon, \varphi)$ where ϵ is the empty valuation. The empty valuation is different from $\bullet$ that denotes the contradiction.

7 Conflict-driven clause learning (CDCL)

```
fonction cdcl(ν, φ)
    si ν = • alors renvoyer false
    sinon
        ν := propagationsUnitaires(ν, φ)
        si ν ⊨ ¬φ alors renvoyer dpll( backtrack − learning(ν, φ) )
        si ν ⊨ φ alors renvoyer true ;
        sinon
            choisir une variable p non ν-assignée
            renvoyer dpll(ν[ℓ := 1], φ)
```

Exemple 10

$$\overbrace{(\neg x1 \vee x2)}^{\alpha} \wedge \overbrace{(\neg x1 \vee x5 \vee x3)}^{\beta} \wedge \overbrace{(x4 \vee \neg x2)}^{\gamma} \wedge \overbrace{(\neg x_3 \vee \neg x_4)}^{\delta} \wedge \overbrace{(x1 \vee x5 \vee \neg x2)}^{\epsilon} \wedge \overbrace{(x3 \vee x2)}^{\theta} \wedge \overbrace{(x2 \vee \neg x3)}^{\lambda} \wedge \overbrace{(\neg x5 \vee x6)}^{\mu} \wedge \overbrace{(x2)}^{\nu} .$$

Exemple 11

$$\overbrace{(x1 \vee x2)}^{\alpha} \wedge \overbrace{(\neg x2 \vee \neg x3)}^{\beta} \wedge \overbrace{(x3 \vee \neg x12)}^{\gamma} \wedge \overbrace{(\neg x2 \vee \neg x4 \vee \neg x5)}^{\delta} \wedge \overbrace{(x3 \vee x5 \vee x6 \vee x7)}^{\epsilon} \wedge \overbrace{(\neg x7 \vee x8 \vee \neg x9)}^{\theta} \wedge \overbrace{(x6 \vee \neg x7 \vee x9 \vee x10)}^{\lambda} \wedge \overbrace{(\neg x7 \vee \neg x10 \vee x8 \vee x11)}^{\mu} \wedge \overbrace{(x13 \vee \neg x14 < \vee \neg x15)}^{\nu} \wedge \overbrace{(x8 \vee \neg x7 \vee x11)}^{\rho} \wedge \overbrace{(\neg x11 \vee \neg x13)}^{\sigma} \wedge \overbrace{(\neg x11 \vee x14)}^{\tau} \wedge \overbrace{(x12 \vee x15)}^{\chi}$$

The problem with backtracking is that the order in which we choose the literals may be far from optimal ! In DPLL, as seen we may have :

before	after
$\neg p$ q r $\neg s$ t u v w a e b	$\neg p$ q r $\neg s$ t u v w a $\neg e$

Our dream is to have :

before	after	
$\neg p$ q r $\neg s$ t u v w a e b	$\neg p$ q r $\neg s$ t	for sure a
	with a minimum number of choices	

This is called *backjumping*. First, we recall the algorithm we want to design with backtrack-learning function. Then we will see that we represent the partial valuation ν with a graph G. This graph is called the *implication graph*. Then we will see how the backtrack-learningfunction works.

7.1 Implication graph

A partial valuation ν is now represented by an implication graph that also contains information about unit propagation.

7.1.1 Definition

Définition 12 (implication graph) An implication graph is a graph $G = (V, E)$ such that :

— A node in V is :

— either a *chosen literal node* depicted in yellow :

where ℓ is a litteral and $n \in \mathbb{N}$ is called the level ;

— a *deduced literal node* depicted in blue :

where ℓ is a litteral and $n \in \mathbb{N}$ is called the level ;

— or a *contradiction node* depicted in orange :

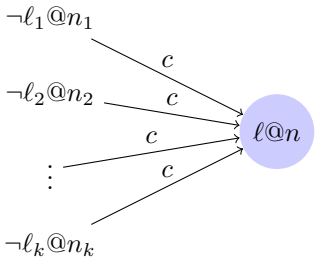
— Edges in E are oriented and labelled by clauses.

7.1.2 Operations

During the process, the following operations may occur :

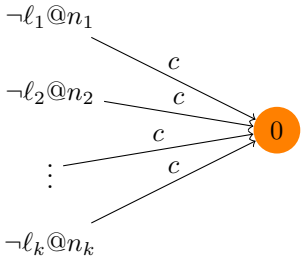
— Choosing a litteral consists in adding a new chosen literal node with level $n+1$ where n is the current level (that is the maximum level appearing in the current graph) ;

— Deduce a litteral by unit propagation. For all unit-clauses of the form $c = \ell \vee \ell_1 \vee \cdots \vee \ell_k$, if litterals $\neg\ell_1, \ldots \neg\ell_k$ are in the graph, then we can add ℓ to the graph :

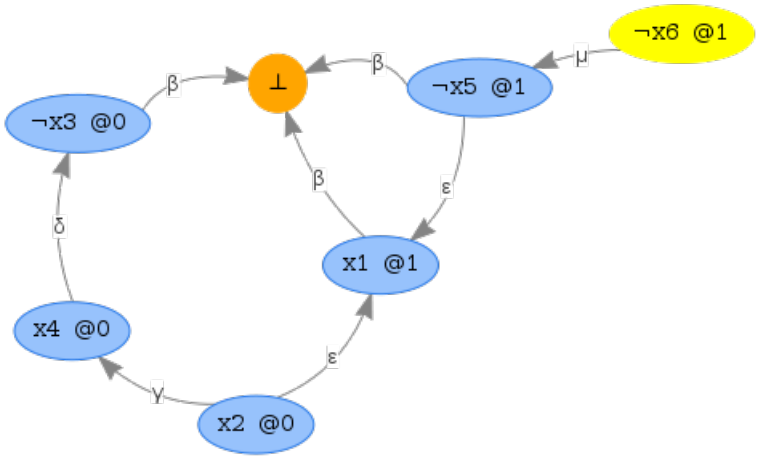


where $n = max_i n_i$. We note $reason(\ell) = c$.

— Reach a contradiction. If a conflicting clause $c = \ell_1 \vee \cdots \vee \ell_k$ is found, that is all litterals $\neg\ell_1, \ldots \neg\ell_k$ are in the graph, then we add a contradiction node.



Exemple 13 The following graph is an implication graph :



where Greek letters denotes the following clauses :

Yellow nodes are literals that have been selected. For instance, $\neg x_6$ has been chosen at decision level 1. $x1$ has been derived from clause ϵ because $\neg x5$ and $x2$ were true.

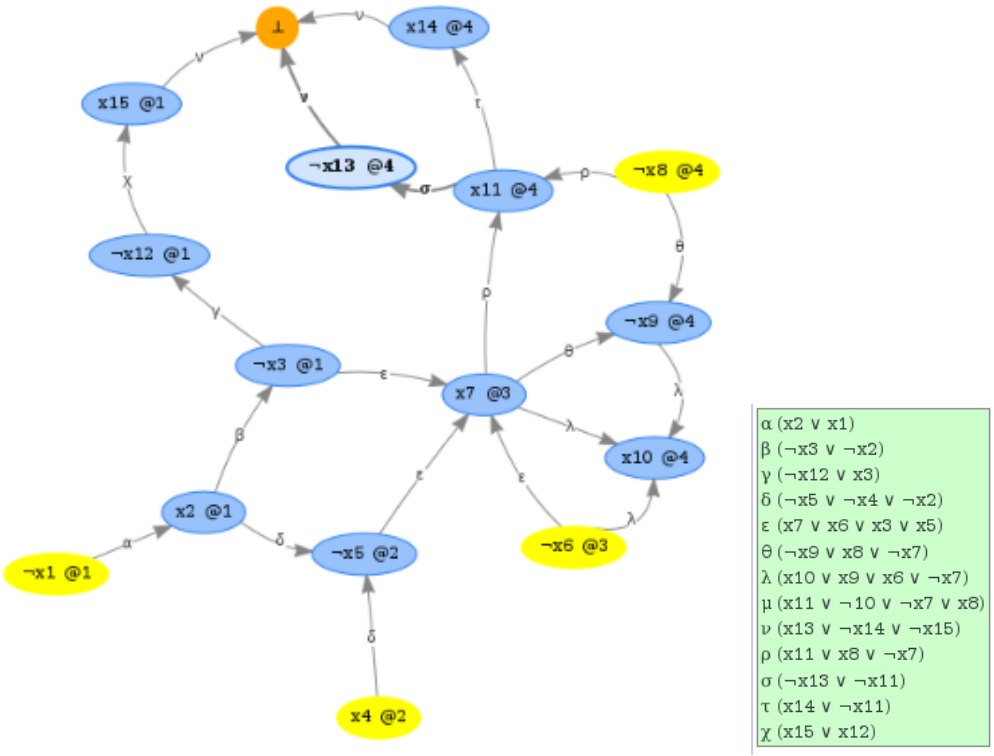
Exemple 14 (this example comes from a presention at IAF2007 by Pascal Nicolas and Laurent Simon)

Let us construct the implication graph corresponding to the following conjunctive normal form :

with respect to the following order for the chosen literals :

```
((not x1)     x4     (not x6)     (not x8))
```

We obtain :



7.2 Backtracking as learning a clause

We see that

$$\neg x1@1 \qquad x4@2 \qquad \neg x6@3 \qquad \neg x8@4$$

leads to a contradiction.
Backtracking corresponds to :
— delete all nodes of level 4
— Derive $x8@3$ by backtracking.
We can see backtracking as :
— Learning the clause $(x1 \vee \neg x4 \vee x6 \vee x8)$;
— delete all nodes of level 4 ;
— Derive $x8@3$ by unit propagation from $(x1 \vee \neg x4 \vee x6 \vee x8)$.

7.3 Our objective

Définition 15 A clause is in the good form if it is of the form :

$$c_L := \underbrace{\ell}_{\text{of level } n} \vee \underbrace{\ell_1 \vee \ell_2 \vee \ldots}_{\text{of level } n' < n \text{ (and if possible very low)}}$$

where n is the current level.

The objective is to learn a better clause in the good form.
Then backjumping is :

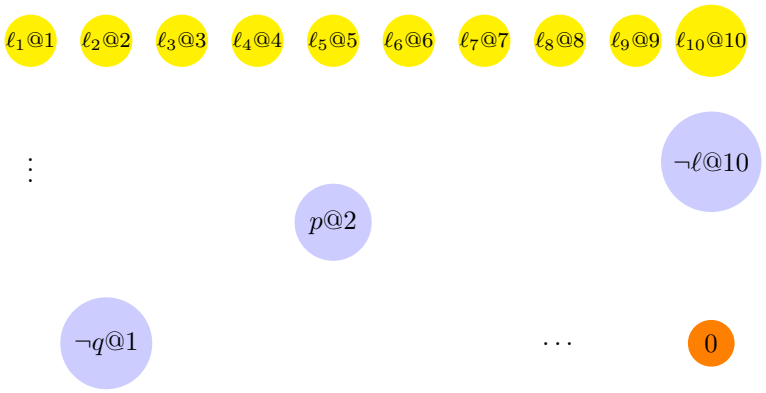
1. Learn the clause c_L ;
2. Delete all nodes of level $> n'$;
3. Derive $\ell@n'$ by unit propagation from c_L.

Step 2. is efficient since we have n' very small. Step 3. is possible because c_L only contains one literal at level n.

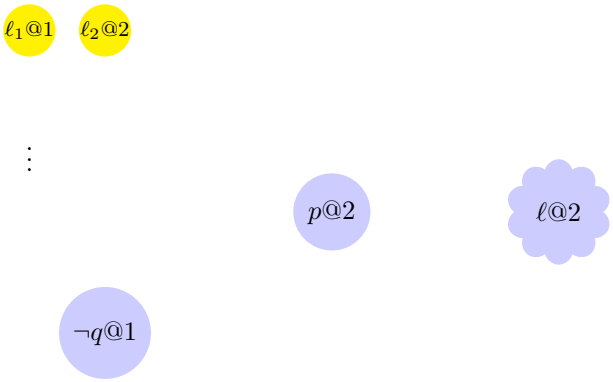
Exemple 16 For instance, suppose that the learned clause is

$$\underbrace{\ell}_{\text{at level 10 (current level)}} \vee \underbrace{\neg p}_{\text{level 2}} \vee \underbrace{q}_{\text{level 1}}$$

and that $\ell_1, \ldots, \ell_{10}$ are the chosen literals.

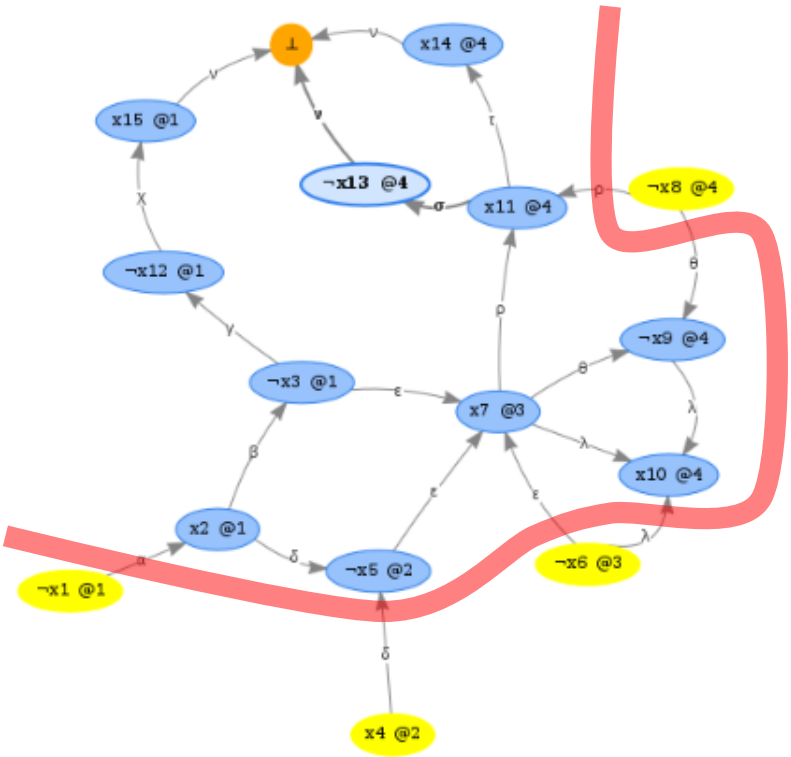


After the backjumping, we forget everything strictly after level 2. The chosen literals are ℓ_1, ℓ_2. As p and $\neg q$ are still in the implication graph, we deduce ℓ via the learned clause (at level 2).



7.4 Learning clause = cut

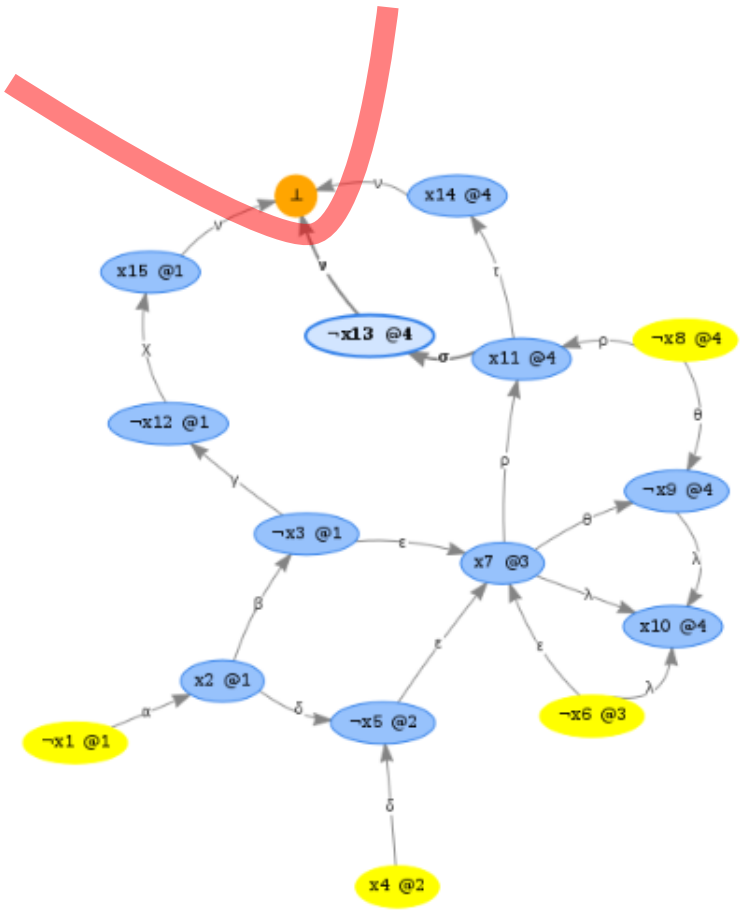
Exemple 17 The cut corresponds to the leanring clause for backtracking :



On the border (reason side), you read :

$\neg x1@1$ $x4@2$ $\neg x6@3$ $\neg x8@4$

Exemple 18 Another cut is when 0 is alone in the conflict side but it may not correspond to a clause in the good form to be learned.



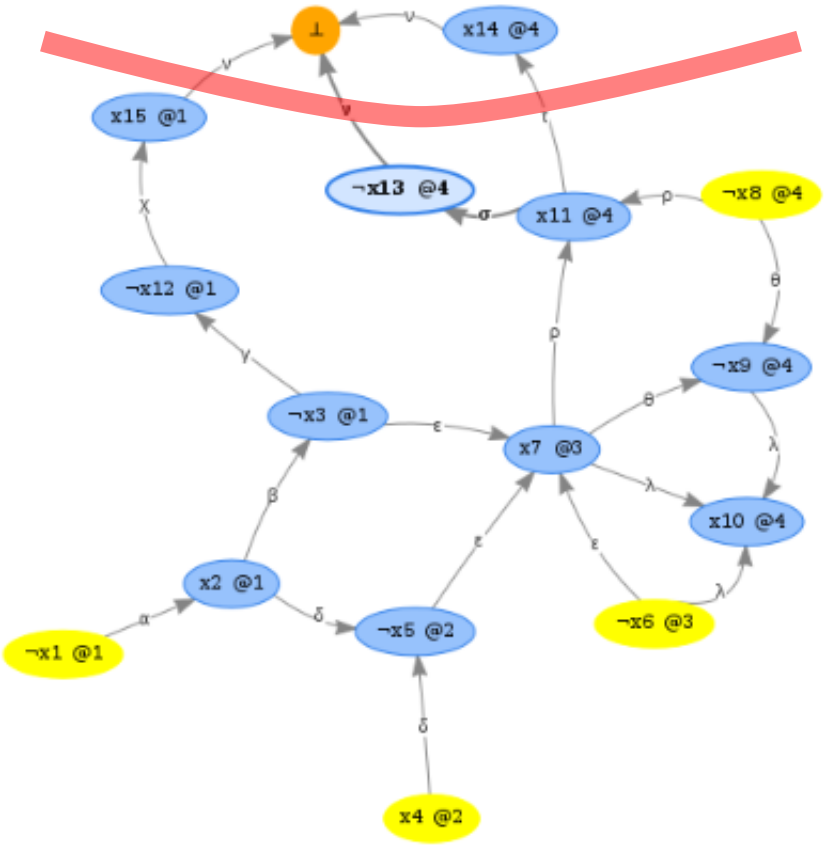
Here you read

$$x15 \qquad \neg x13 \qquad x14$$

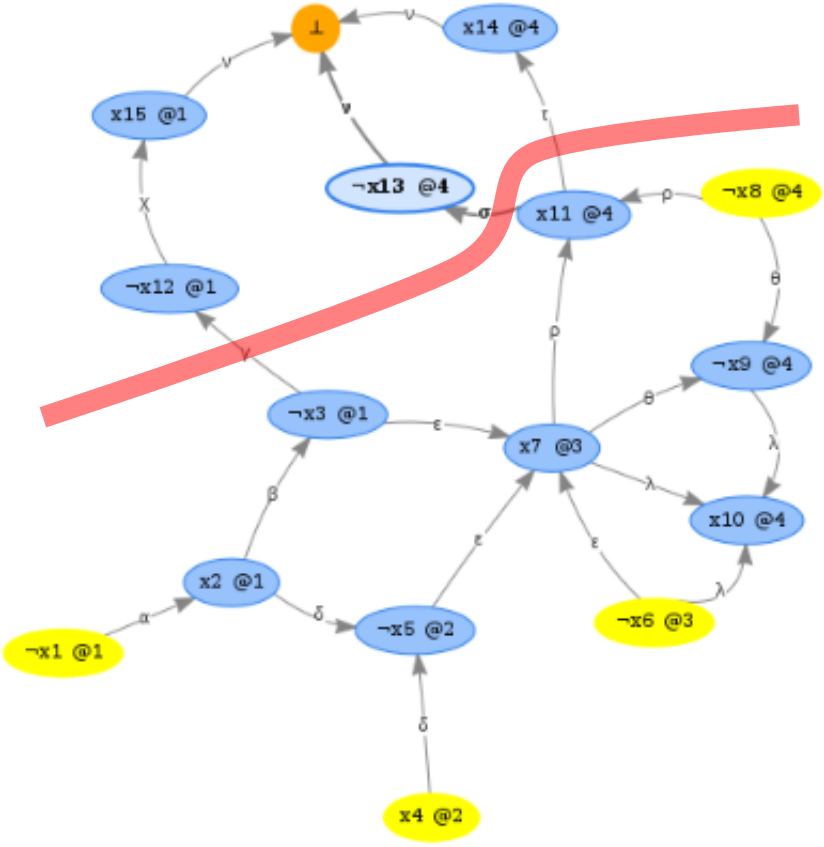
on the border because it is exactly the clause $x13 \vee \neg x14 \vee \neg x15$ that is contradicted. But it is not in the good form because both $\neg x13$ and $x14$ are of level 4.

Interestingly, if you consider other cuts, you read other contradicted clauses.

Exemple 19 The following cut is not very interesting because among $x15$, $\neg x13$ and $x11$ both $\neg x13$ and $x11$ are at level 4 :



Exemple 20 The following cut is more interesting because it gives a clause in the good form :



Indeed,the clause to be learning is $(\neg x3@4 \wedge x11@1)$.
Then :
— We learn $(\neg x11 \wedge x3)$;
— We delete all nodes of level > 1 ;
— We derive $\neg x11@1$ by unit propagation with $(\neg x11 \wedge x3@1)$.

8 Backjumping and learning : algorithm

$backtrack - learning(\nu, \varphi)$ returns a new ν and a new φ. We describe here is a possible implementation of $backtrack - learning(\nu, \varphi)$.

8.1 Learning as resolution steps

Définition 21 (resolution) We note $\bigodot$ the resolution operator. For all clauses c_1, c_2 for which there is a unique variable r such that one clause has a literal r and the other has literal $\neg r$, $c_1 \bigodot c_2$ contains all the literals of c_1 and c_2 with the exception of r and $\neg r$.

Exemple 22 For instance : $(p \vee q \vee \neg r) \bigodot (r \vee s \vee u) = (p \vee q \vee s \vee u)$.

Proposition 23 Let ν be a (complete) valuation. Let c_1 and c_2 be two clauses for which there is a unique variable r such that one clause has a literal r and the other has literal $\neg r$.

$$(\nu \models c_1 \text{ and } \nu \models c_2) \text{ implies } \nu \models (c_1 \bigodot c_2).$$

Définition 24 (computation of the learned clause) Let n be the current level. We define the following sequence of clauses :

$c_0 =$ the conflict clause

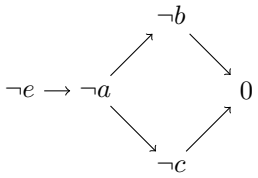
$$c_{i+1} = \begin{cases} c_i \bigodot reason(\ell) & \text{if there are at least two literals in } c_i \text{ at the level } n \\ & \text{and } \ell \text{ is a deduced literal at level } n \text{ such that } \neg\ell \text{ is in } c_i \text{ and } reason(\ell) \text{ is defined} \\ c_i & \text{otherwise} \end{cases}$$

Note that the 'otherwise' means that there is exactly one literal at level n.

Exemple 25 See example 20.

i	c_i	
0	$\neg x15 \vee x13 \vee \neg x14$	
		both $\neg x13$ and $x14$ are at the level 4 (current level)
		resolution with $reason(x14) = (x14 \vee \neg x11)$
1	$\neg x15 \vee x13 \vee \neg x11$	
		both $\neg x13$ and $x11$ are at the level 4 (current level)
		resolution with $reason(\neg x13) = (\neg x13 \vee \neg x11)$
2	$\neg x15 \vee \neg x11$	
		both $x11$ is at level 4 (current level) but $x15$ is older

Exemple 26 Considérons ce graphe d'implication :



La clause conflictuelle est $b \vee c$.
On peut faire : $(b \vee c) \bigodot (a \vee \neg b) = a \vee c$
$(a \vee c) \bigodot (\neg a \vee e) = e \vee c$
$(e \vee c) \bigodot (\neg c \vee a) = e \vee a$
$(e \vee a) \bigodot (\neg a \vee e) = e$

ou alors :
$(b \vee c) \bigodot (a \vee \neg b) = a \vee c$
$(a \vee c) \bigodot (\neg c \vee a) = a$

Proposition 27 Il existe i_0 tel que c_{i_0} contient exactement un littéral de niveau n.

Démonstration. On note $G = (V, E)$ le graphe d'implication. Nous avons l'invariant suivant : le graphe d'implication est acyclique. En effet, le graphe est initialement vide donc acyclique, puis le graphe reste acyclique si on ajoute un noeud de décision, et faire une propagation unitaire. Nous considérons alors un tri topologique, i.e. on peut ordonner les sommets comme suit : $v_1, \ldots, v_{|V|}$ avec $v_k \to v_{k'}$ implique $k < k'$.

Par contradiction, supposons qu'il n'y a pas de tel i_0. Mais alors on a pour tout $i \in \mathbb{N}$, $c_{i+1} = c_i \bigodot reason(\ell_i)$ pour un certain noeud ℓ_i.

On représente une clause c par un vecteur $(\beta_{|V|}, \ldots, \beta_1) \in \{0,1\}^{|V|}$ où $\beta_k = \begin{cases} 1 \text{ si } \neg v_k \text{ apparaît dans } c \\ 0 \text{ sinon} \end{cases}$. On dira que $c < c'$ si le vecteur de c est plus petit que le vecteur de c' pour l'ordre lexicographique sur les mots de longueur $|V|$ sur l'alphabet $\{0,1\}$.

Exemple 28 $011000 < 011100$.

Fait 29 On a $c_{i+1} < c_i$.

Démonstration. $c_{i+1} := c_i \bigodot reason(\ell_i)$. En notant $c_i := \neg \ell_i \vee c_i'$ et $reason(\ell_i) := \ell_i \vee r$ où c_i' et r sont des clauses, on a

$$c_{i+1} = c_i' \vee r.$$

ℓ_i est un certain v_k. Les négations des littéraux dans r sont des $v_{k'}$ avec $k' < k$. Ainsi, $c_{i+1} < c_i$:

$$\begin{array}{lccc} & & v_k & \\ c_i & = (xxxxxxx & 1 & yyyyyyy) \\ c_{i+1} & = (xxxxxxx & 0 & zzzzzzz) \end{array}$$

■

Contradiction car nous avons la suite $(c_i)_{i \geq 0}$ qui est strictement décroissante dans l'ensemble $\{0,1\}^N$. ■

8.2 UIP

Définition 30 Un UIP est un sommet u tel que tout chemin du nombre de décision au niveau n à 0 contient u.

8.3 Pseudo-code

fonction *backtrack* − *learning*(ν, φ)
  si there are no decision nodes in ν **alors**
    renvoyer $\bullet$
  c_L = compute the limit of $(c_i)_{i\in\mathbb{N}}$
  n := the maximum level in c_L
  n' := the second maximum level in c_L
  ν' := ν where we forgot all nodes of level $> n'$
  renvoyer $\nu', \varphi \wedge c_L$

See demonstration : `https://francoisschwarzentruber.github.io/dpll/`

9 Proofs

This subsection is inspired from Chapter 4 of [ZM03].

Proposition 31 $\models (\varphi \to c_L)$.

Démonstration. We prove that $\models \varphi \to c_n$ for all $n \in \mathbb{N}$. Let us do it by recurrence on n. Let P_n be the following property :

$$\models \varphi \to c_n.$$

— the conflicting clause c_0 is already a clause of φ so P_0 is true ;
— Suppose P_n. Let us prove that P_{n+1}. We have $\models \varphi \to c_n$. If $c_{n+1} = c_n$, P_{n+1} is true. Otherwise, $c_{n+1} = c_n \odot reason(\ell)$ where $\neg\ell$ is in c_n and ℓ is a deduced literal at the current decision level.
Let ν be a (total) valuation such that $\nu \models \varphi$. We have $\nu \models c_n$ by P_n. We have $\nu \models reason(\ell)$ as $reason(\ell)$ is a clause of φ. Therefore, by Proposition 23, $\nu \models c_{n+1}$. Hence, we proved that $\models \varphi \to c_{n+1}$. The property P_{n+1} is true.
We proved the proposition by recurrence.
■

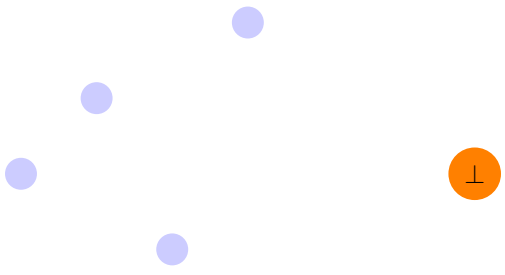
Corollaire 32 $\models (\varphi \leftrightarrow (\varphi \wedge c_L))$.

Proposition 33 If the procedure returns *true*, then the initial formula φ is satisfiable.

Démonstration. The algorithm returns true when a (partial) valuation ν such that $\nu \models \varphi$ is found. ν can be extended into a total valuation. Therefore, the formula φ is satisfiable. ■

Proposition 34 If the procedure returns $false$, then the initial formula φ is unsatisfiable.

Démonstration. If the procedure returns $false$ then $\nu = \bullet$ is reached. $\nu = \bullet$ is reached because $backtrack - learning(\nu, \varphi)$ returns a $\nu = \bullet$. This means that a conflict was detected and no literals are chosen. At this stage, the implication graph is full of deduced literals at level 0 and a 0 node.



By contradiction, suppose that the initial formula φ^{ini} is satisfiable. Therefore, by proposition 31, the current formula φ, that is the conjunction of φ^{ini} and all learned clauses is also satisfiable. Let ν be a (total) valuation such that $\nu \models \varphi$.

Lemme 35 All added literals ℓ in the implication graph are such that $\nu \models \ell$.

Démonstration. They are added via unit propagation. For the first added literal ℓ, ℓ itself should be a clause in φ. Thus, $\nu \models \ell$. Then at each step, such that $\nu \models \ell'$ for all ℓ' already in the implication graph. Let ℓ be the next added literal. We should have a clause $\ell_1 \vee \cdots \vee \ell_k \vee \ell$ of φ such that $\neg\ell_1, \ldots, \neg\ell_k$ are in the implication graph. As $\nu \models \varphi$ and $\nu \models \neg\ell_i$ we have $\nu \models \ell$. ■
Therefore, the conflict clause c is such that $\nu \models \neg c$. So $\nu \not\models \varphi$. There is a contradiction. ■

Proposition 36 The call $dpll(\epsilon, \varphi)$ terminates.

DÉMONSTRATION. By contradiction. Suppose that $dpll(\epsilon, \varphi)$ does not terminate. We exhibit a strictly increasing sequence $(x^{(t)})_t$ of elements in a finite set with a strict total order. This leads to a contradiction.

Definition of $(x^{(t)})_t$. Given a partial valuation ν (where assigned litterals have levels), let $k(\nu, i)$ be the number of assigned (chosen or deduced) literals at level i in ν. Let us consider the t^{th} call of *dpll*. Let

$$x^{(t)} = (k(\nu, 0), \ldots, k(\nu, n)) \quad \in \quad \{0, \ldots, n\}^{n+1}$$

where ν is the valuation at t^{th} call of *dpll* after saturation by unit propagation.

Evolution of $x^{(t)}$. There are two reasons for a $t+1^{th}$ call of *dpll* :

— Either we chose a new literal. For instance :

before	after
$\neg p$ q r $\neg s$ t u v w a	$\neg p$ q r $\neg s$ t u v w a e

In the example :

$x^{(t)} = (0, 3, 2, 1, 1, 2, 0, 0, \ldots);$
$x^{(t+1)} = (0, 3, 2, 1, 1, 2, \mathbf{1}, 0, \ldots).$

— Or we performed backjumping. For instance :

before	after
$\neg p$ q r $\neg s$ t u v w a e b	$\neg p$ q r $\neg s$ t for sure a $\ldots$

In the example :

$x^{(t)} = (0, 3, 2, 1, 1, 2, 2, 0, \ldots);$
$x^{(t+1)} = (0, 3, > \mathbf{2}, 0, 0, \ldots).$

Lexicographical order. That is why we introduce the lexicographic order $>$ on $\{0, \ldots, n\}^{n+1}$. Given $x, y \in \{0, \ldots, n\}^{n+1}$, we have $x > y$ iff there exists an integer $i \in \{0, \ldots, n\}$ such that $x_i > y_i$ and for all $j < i$, we have $x_j = y_j$.

Conclusion. The sequence $(x^{(t)})_t$ is strictly increasing and takes only a finite number of values. Contradiction.

■

10 Implementation details

10.1 Two literal watching

The problem is to know efficiently which clauses are unit. Note that clauses of formula φ are *not* modified during the algorithm (clauses are just added). But clauses themselves are not modified (otherwise the backtrack is too difficult to implement). In particular, unit clauses are clauses such that all literals except one have been assigned to false but they are not clauses with only one literal.

Exemple 37 If we consider the clause $p \vee q \vee \neg s$ and we set p to false, the clause is *not* replaced by $q \vee \neg s$.

In each clause c, two unassigned literals ℓ_1, ℓ_2 are selected. We say that the two unassigned literals (occurences of literals) are *watched*. The implementation works as follows. For each atomic proposition p, we maintain a list of clauses in which some occurrence of p is watched and a list of clauses in which some occurrence of $\neg p$ is watched.

— When some clause is added, we select randomly two unassigned literals in it ;
— When some atomic proposition p is assigned, we browse all the clauses c in which p or $\neg p$ is watched :
  — if all literals except one in c are assigned (to false), then the last litteral is assigned to true ;
  — if all literals in c are assigned to false, then we backtrack ;
  — if one litteral in c is assigned to true, we continue ;
  — otherwise, let ℓ be a unassigned litteral in the clause c, make ℓ to be watched and the litteral about p unwatched.

	Naive implementation with counters	VS	2 literal watching
When a clause is added	We update counter for that clause		We select randomly two unassigned literals in it
When a literal is set	We update counters for all clauses		We update the watched literals
When backtrack	We recompute the counters		We do nothing !

Proposition 38 (invariant) For all clauses c, if c contains more than 2 unassigned literals, then the two watched literals are unassigned.

In practice, if c is a clause, watched literals are $c[0]$ and $c[1]$ (and we swap literals to garantee this property). If p is an atomic proposition, $p.positiveWatches$ is an array of clauses where p appears positively and $p.negativeWatches$ is an array of clauses where p appears negatively.

Exemple 39

Operations	Valuations	My clause				
		p	q	r	s	
We set r	r	p	q	r	s	
We set $\neg p$	$r, \neg p$	q	r	p	s	
We set $\neg q$	$r, \neg p, \neg q$	r	s	p	q	
Backtrack 2 steps	r	r	s	p	q	(we do nothing)

10.2 Efficient algorithm for computing FUIP

See Javascript code of `sat.js`.

11 Améliorations

11.1 Redémarrage

— redémarrer après un certain nombre de backtracking, avec un seuil qui augmente (sinon on ne serait pas complet !)

11.2 Heuristiques

— Choix de la variable à décider, qui apparaît le plus fréquemment dans les clauses (Chaff)
— priorité aux variables qui apparaissent dans les clauses les plus récentes
— utilisation du machine learning

11.3 Suppression de clauses apprises

On peut apprendre un nombre exponentiel de clauses...
— On apprend que des clauses assez courtes (moins de n littéraux où n est fixé)
— On oublie une clause dès qu'au moins m littéraux sont non assignés

12 Recherche locale

fonction GSAT(φ)
  ν := valuation au hasard
  pour $i := 1$ à N **faire**
    si $\nu \models \varphi$ **alors renvoyer** ν
    v := variable parmi lesquelles, l'échange de la valeur $\nu(v)$ donne le plus de clauses satisfaites
    $\nu(v) := 1 - \nu(v)$
  renvoyer échec

Définition 40 Le break-count d'une variable est le nombre de clauses sont actuellement satisfaites mais qui deviennent insatisfaites si la variable est flipée.

```
fonction walkSAT(φ)
    ν := valuation au hasard
    pour i := 1 à N faire
        si ν ⊨ φ alors renvoyer ν
        C := une clause non satisfaite choisie au hasard
        si il existe une variable x dans C avec break-count = 0 alors v := x
        sinon
            avec probabilité p : v := une variable de C au hasard
            avec probabilité 1 − p : v := une variable de C avec break-count le plus petit
        ν(v) := 1 − ν(v)
    renvoyer échec
```

13 Notes bibliographiques

Les éléments techniques de ce cours viennent de [KS08] et [Har09]. La terminaison de l'algorithme CDCL est très bien expliquée dans la thèse [ZM03].

Références

[Har09] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[HKM16] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2016.

[KS08] Daniel Kroening and Ofer Strichman. *Decision procedures*, volume 5. Springer, 2008.

[ZM03] Lintao Zhang and Sharad Malik. *Searching for truth : techniques for satisfiability of boolean formulas*. PhD thesis, Princeton University Princeton, 2003.