SAT
Comment le résoudre efficacement ?

cours de Laurent Simon

simon@lri.fr

Scribe : Benjamin Hellouin

22 Janvier 2009

ENS-Lyon Master Systèmes Complexes

Le contenu de ce cours inclut :
- Les motivations à l'origine du travail sur SAT (pourquoi lui ?) ;
- Les problèmes sur lesquels ça marche ;
- L'histoire de cette quête à la performance ;
- Les algorithmes qui ont menés aux meilleurs algorithmes actuels.

1 Introduction au premier cours

1.1 Introduction

Depuis "The complexity of theorem proving procedures" (Cook, 1971), SAT est la brique de base de la théorie de la complexité : à la fois trés étudié pour ses limitations théoriques et pour sa résolution pratique.

Il y a 10 à 20 ans, la réduction polynomiale de tout problème à SAT était vu comme une question *seulement* d'intérêt théorique : un problème semble difficile à résoudre en pratique ? « Réduisez-le à SAT, et montrez qu'on ne peut pas le résoudre ! ». Aujourd'hui, la réponse serait plutôt : « Réduisez-le à SAT, et résolvez-le grâce à lui ! ».

Ceci à tel point que Moshé Vardi considère que beaucoup des problèmes logiques les plus compliqués dans les prochaines années tireront un gros avantage de la résolution effective de SAT.

Ceci s'explique par d'incroyables progrès pratiques et théoriques.

- Du point de vue pratique :
  Dans les années 90, les formules traitables étaient de quelques centaines de clauses sur une station de recherche... là où aujourd'hui, sur un PC *familial*, on peut résoudre des problèmes de taille industrielle en :
  - Electronic Design Automation (EDA) ;
    - ▷ Vérification de Microprocesseurs ;
    - ▷ Génération automatique de tests.

– (Bounded) Model Checking (des solveurs incorporés dans des outils de model checking *industriels*) ;
– Planification ;
– Preuve de théorèmes de logiques plus expressives (au dessus de NP).

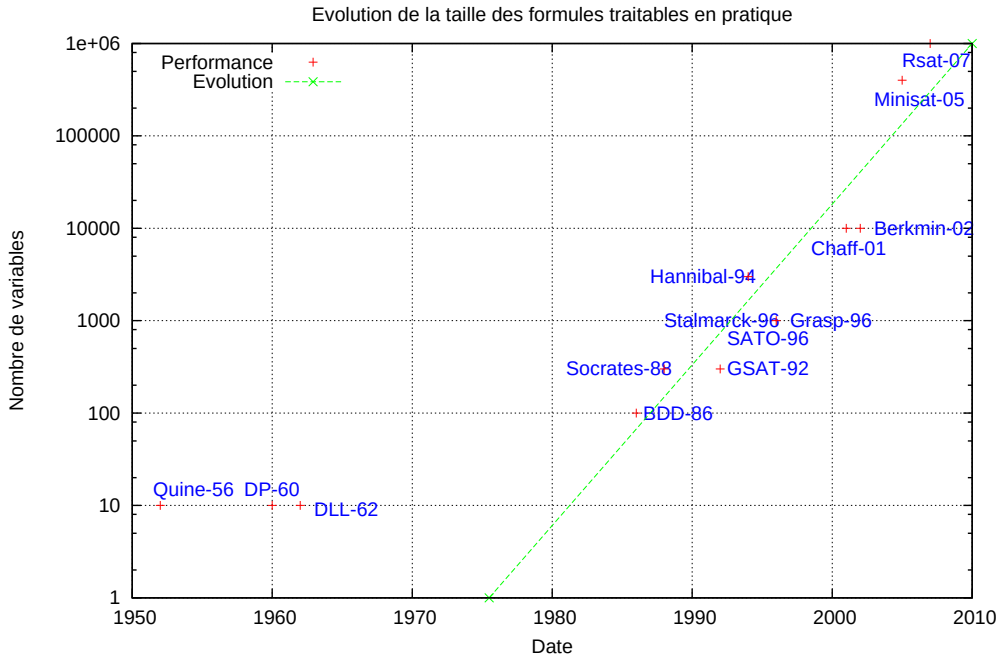
- Du point de vue théorique, sur des instances aléatoires :
  Aujourd'hui, on arrive à résoudre des problèmes UNSAT jusqu'à 600 variables au seuil, et plusieurs dizaines de milliers de variables du côté SAT.

Cependant, pour l'instant, les bons résultats sur les exemples industriels sont très liés aux améliorations expérimentales, et non théoriques...

Trois explications (parmi d'autres) :

1. Algorithmes sophistiqués **et** simples ;
2. Implantations *ingénieuses* ;
3. Seuil de capacité franchi par les machines ?

Pour avoir une idée des progrés effectués :



Depuis DLL62 , le nombre de variables croît exponentiellement, et l'espace de recherche en double exponentielle : la croissance est plus rapide que la loi de Moore !

Cependant, le comportement des solveurs n'est aujourd'hui pas du tout le même suivant le type de problèmes :
– Problèmes aléatoires ;
– Problèmes « industriels » ;

– Problèmes académiques ou typiquement difficiles.

La séparation des problèmes permet de spécialiser les démonstrateurs. Par exemple, sur les exemples industriels, KCNFS (spécialisé dans les problmèes aléatoires) quitte tout de suite sans même essayer. On va présenter un petit historique de cette spécialisation.

Dans les annèes 90, on obtient d'assez bons résultats sur quelques exemples jouets et on effectue des comparaison avec d'autres techniques (construction de BDD). Les méthodes incomplètes donnent de très bons résultats, et la communauté cherche alors des problèmes sur lesquels tester leurs programmes :
– Des problèmes académiques (pigeons, coloration de graphes, ...) ;
– Des problèmes de planification (monde des blocks, ...) ;
– Des problèmes aléatoires.

Cependant :
– Les problèmes résolus sont petits et les programmes de l'époque moins performants que des algorithmes spécialisés du domaine ;
– L'expérimentation gagne en importance, mais on ignore où sont les « bons » problèmes ;
– La génération aléatoire est née, mais les tests son effectués « au jugé ».

Dans les années 2000, l'expérimentation atteint sa maturité. « Hard and Easy distributions of SAT problems » vient démontrer l'existence d'un pic de difficulté sur les instances aléatoires uniformes.

Dés lors, des algorithmes spécialisés apparaissent (comme kcnfs). Les algorithmes stochastiques sont paramétrés pour les instances aléatoires au seuil.

Par exemple,
– Sur les instances aléatoires, on cherche des backbones (littéraux vrais pour tout assignement valide) ;
– Sur les instances industrielles, on utilise des backdoors (variables qui déterminent la valeur d'autres variables).

Ensuite, sur les instances aléatoires, on a assisté à une séparation complète des méthodes (survey propagation, ...). Du point de vue industriel, on a d'abord de bons résultats en recherche locale. Les algorithmes stochastiques donnent de bons résultats sur des problèmes de planification (walksat, Gsat).

Certains exemples pathologiques pour les algorithmes spécialisés sont résolus en passant par SAT. D'autre part, l'introduction du Bounded Model Checking fournit de " vrais " problèmes de grande taille à la communauté. La communauté de Model Checking commençait à être confrontée à des limites.

En conséquence, les structures de données s'adaptent à ces tailles, les heuristiques et les techniques suivent.

1.2 Le cycle des compétitions

Avant 2001, il y avait déjà eu quelques compétitions :

– Paderborn en 1991/1992 ;

– the Second DIMACS Challenge, en 1992/1993 ;
– Beijing competition, en 1996.

Depuis 2002, une nouvelle compétition est organisée et permet de :
– *mesurer* l'impact de `zchaff` ;
– motiver toute la communauté, fournir des benchmarks ;
– offrir une vitrine *officielle* et *uniforme* ;
– ouvrir la problématique *d'expérimentation* d'algorithmes.

En 2007, les meilleurs programmes dépassaient le million de variables, ce qui nous amène à ce pradoxe : les meilleurs algorithmes actuels pour SAT sont exponentiels en théorie, et presque linéaires en pratique !

1.3 Rappels sur la logique

Le langage des formules propositionnelles se définit à partir d'un ensemble de variables propositionnelles $\mathcal{V}$, de deux constantes $vrai$ et $faux$ et d'un ensemble de *connecteurs logiques* : $\neg$ (négation), $\vee$ (disjonction), $\wedge$ (conjonction), $\rightarrow$ (implication), $\leftrightarrow$ (équivalence), $\oplus$ (exclusion).

Définition 1 (Le langage des propositions $\mathcal{LP}$) *Etant donné un ensemble de variables propositionnelles $\mathcal{V} = \{x_1, \ldots, x_n\}$, le langage $\mathcal{LP}$ des **formules propositionnelles** (ou **propositions**) est le plus petit langage construit inductivement en appliquant un nombre fini de fois les règles :*
– $vrai \in \mathcal{LP}$, $faux \in \mathcal{LP}$
– $\forall x \in \mathcal{V}$, $x \in \mathcal{LP}$
– $\forall f \in \mathcal{LP}$, $\neg f \in \mathcal{LP}$
– $\forall f, g \in \mathcal{LP}$, $f \vee g$, $f \wedge g$, $f \rightarrow g$, $f \leftrightarrow g$, $f \oplus g$ *et* (f) *sont dans* $\mathcal{LPXS}$.

La sémantique classique de la logique propositionnelle repose sur la notion *d'interprétation*. Intuitivement, on se donne un ensemble $\mathbb{B} = \{\mathbf{T}, \mathbf{F}\}$, constitué de deux *valeurs de vérité*, et l'on associe chaque variable propositionnelle à une de ces valeurs.

Définition 2 (Interprétation) *Une **interprétation** I d'un ensemble de variables $V \subseteq \mathcal{V}$ est une application ayant pour domaine V et pour co-domaine $\mathbb{B} = \{\mathbf{T}, \mathbf{F}\}$. On parle d'interprétation totale lorsque $V = \mathcal{V}$; Autrement, l'interprétation est dite partielle.*

Définition 3 (Interprétation d'une formule) *Étant donnée une interprétation I d'un ensemble de variables V et une formule propositionnelle f telle que $Var(f) \subseteq V$, la valeur de l'interprétation de f dans I (notée $[\![f_I]\!]$) est définie récursivement par :*
– $vrai_I = \mathbf{T}$ *et* $faux_I = \mathbf{F}$
– *si* $x \in \mathcal{V}$, $x_I = I(x)$
– $\forall f, g \in \mathcal{LP}$, $[\![\neg f_I]\!] = \mathbf{T}$ *si* $[\![f_I]\!] = \mathbf{F}$ *et* $\mathbf{F}$ *sinon*
$[\![f \wedge g_I]\!] = \mathbf{F}$ *si* $[\![f_I = \mathbf{F}$ *ou* $[\![g_I]\!] = \mathbf{F}$, *et* $\mathbf{T}$ *sinon*
$[\![f \vee g_I]\!] = \mathbf{T}$ *si* $f_I = \mathbf{T}]\!]$ *ou* $[\![g_I]\!] = \mathbf{T}$, *et* $\mathbf{F}$ *sinon*
$[\![f \rightarrow g_I]\!] = \mathbf{F}$ *si* $[\![f_I = \mathbf{F}]\!]$ *et* $[\![g_I]\!] = \mathbf{T}$, *et* $\mathbf{T}$ *sinon*
$[\![f \leftrightarrow g_I]\!] = \mathbf{T}$ *si* $[\![f_I]\!] = [\![g_I]\!]$, *et* $\mathbf{F}$ *sinon*
$[\![f \oplus g_I]\!] = \mathbf{T}$ *si* $[\![\imath nt f_I]\!] \neq [\![g_I]\!]$, *et* $\mathbf{F}$ *sinon*
$[\![(f)]\!] = [\![f]\!]$

Définition 4 (Restriction d'une formule) *Soient f une formule et I une interprétation sur $V \subseteq \mathcal{V}$. La* ***restriction de*** *f* ***à*** *I (notée $f|_I$) est la formule $f\sigma_I$ où σ_I est la substitution caractérisée par $\sigma = \{x \mapsto vrai/x \in V \text{ et } I(x) = \mathbf{T}\} \cup \{x \mapsto faux/x \in V \text{ et } etI(x) = \mathbf{F}\}$*

On peut naturellement appliquer les simplifications usuelles pour éliminer toutes les occurrences de $vrai$ et $faux$ dans $f\sigma_I$. Notons que si $Var(f) \subseteq V$ on a nécessairement soit $f\sigma \equiv vrai$, soit $f\sigma \equiv vrai$. Sinon, les variables apparaissant dans $f\sigma$ sont nécessairement non interprétées dans I.

Exemple 1 $((x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (\neg y \vee t))|_{\{x,\neg t\}} = ((y \vee \neg z) \wedge \neg y)$

Rappels de logique propositionnelle

- *$V = \{x_1, \ldots, x_n\}$ est l'ensemble des variables propositionnelles, l_i est un littéral, c-à-d une variable x_i ou sa négation $\neg x_i$.*
- *Une clause est une disjonction de littéraux $c_i = l_1 \vee l_2 ... \vee l_{n_i}$. Une clause unitaire ne contient qu'un seul littéral.*
- *Une formule Σ est sous forme normale conjonctive ($\mathcal{CNF}$) lorsque Σ s'écrit comme une conjonction de clauses $\Sigma = c_1 \wedge c_2 ... \wedge c_m$ (on peut aussi parler d'ensembles de clauses).*
- *Problème SAT : existe-t-il une interprétation permettant d'évaluer une formule sous $\mathcal{CNF}$ à vrai ?*
- *Note : Toute formule peut se réécrire sous $\mathcal{CNF}$. Comment ?*

2 Intuitions et problèmes NP-complets

2.1 L'intuition derrière NP

L'intuition qui se cache derrière les problèmes NP est la suivante :

- Il est difficile de trouver une aiguille dans une botte de foin (mais on peut être chanceux).
- Il est facile de vérifier que l'aiguille est là où on nous l'a dit (l'oracle !).
- Il est très difficile de prouver qu'il n'y a pas d'aiguille (même en étant chanceux).
  $\rightarrow$ La réponse "oui" est courte (un lieu), la réponse "non" est exponentiellement longue (tous les lieux).

2.2 Rappels plus formels sur NP

Définition 5 (Classe de complexité P) *La classe P est l'ensemble des problèmes de décision pouvant être résolus en temps polynômial.*

En terme de fonction calculable, ces problèmes sont caractérisés par la réponse à la question « Que vaut $f(x)$? », avec f calculable en temps polynômial par rapport à la taille de la donnée x.

Définition 6 (Classe de complexité NP) *La classe NP est l'ensemble des problèmes de décision pouvant être résolus en temps polynômial de manière non déterministe.*

La classe de complexité NP correspond à l'ensemble des problèmes de décision s'exprimant logiquement comme

$$\exists x.f(x)$$

« existe-t-il un x tel que l'on ait $f(x)$? », avec f une fonction calculable en temps polynômial.

Définition 7 (Classe de complexité $CoNP$) *La classe $CoNP$ est l'ensemble des problèmes dont le problème complémentaire appartient à NP.*

Définition 8 (Réduction α_p) *Un problème $\mathcal{P}_1$ est polynômialement réductible à un autre problème $\mathcal{P}_2$ si et seulement si il existe une fonction f calculable en temps polynômial telle que pour tout problème de décision x de $\mathcal{P}_1$, $\mathcal{P}_2$ répond identiquement sur l'entrée $f(x)$. On le note $\mathcal{P}_1\ \alpha_p\ \mathcal{P}_2$.*

Définition 9 (Classe NP-complet) *Soit $\mathcal{P}$ une classe de problèmes. On dit que $\mathcal{P}$ est NP-Complet si et seulement si (1) $\mathcal{P} \in NP$; (2) $\forall \mathcal{P}' \in NP,\ \mathcal{P}'\alpha_p\mathcal{P}$.*

Définition 10 (Classe CoNP-complet) *Soit $\mathcal{P}$ une classe de problèmes. On dit que $\mathcal{P}$ est $CoNP$-Complet si et seulement si (1) $\mathcal{P} \in CoNP$; (2) $\forall \mathcal{P}' \in CoNP,\ \mathcal{P}'\alpha_p\mathcal{P}$.*

La notion de problème NP est très fortement reliée à la garantie de l'existence d'un certificat court, et facilement vérifiable. En pratique, on peut observer d'énormes différences de performances.

A noter : les algorithmes « modernes » pour SAT ne montrent aucune différence de performance pour les problèmes SAT ou les problèmes UNSAT.

2.3 Attaquer $P \neq NP$

Les problèmes posés se présentent sous un encodage différent :
- Sudoku (pas encore un problème industriel)
- Recherche d'haplotype dans une population
- (Bounded) Model Checking
- Logiques modales / Logiques temporelles
- Raisonnement sur les ontologies
- Organisation de tournois de golf
- ...

Pourquoi tant de problèmes sont-ils ramenés à SAT ?
- Soit ça marche, et leur problème est résolu
- Soit ça ne marche pas et leur problème devient un benchmark de valeur

Voici un exemple d'utilisation de SAT pour un problème logique. La formulation STRIPS se présente sous cette forme :

```
MOVE(x,y,z)
PRE : CLEAR(x), ON(x,y), CLEAR(z)
ADD : CLEAR(y), ON(x,z)
DEL : CLEAR(z), ON(x,y)
```

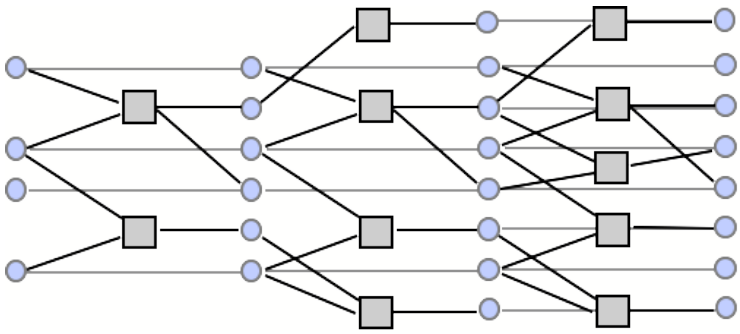
Ici, il s'agit d'exprimer les pré-requis et les effets du déplacement d'un objet x, initialement situé sur y, sur d'un objet z.

Etant donné un but à atteindre, les étapes nécessaires sont déterminées en effectuant les opérations suivantes :

1. Empiler le but à atteindre
2. Effacer le haut de la pile si c'est un fait valide
3. Si c'est une conjonction, empiler tous ses faits
4. Sinon chercher à unifier une action dont l'un des effet est justement le fait recherché, l'ajouter à la pile avec toutes ses préconditions
5. Si c'est une action, l'executer
6. Boucler tant que la pile n'est pas vide

La révolution a été de se ramener à un objet dit GraphPlan [95], dont voici les règles :

1. Génèrer un GraphPlan de longueur k
2. Transformer les contraintes du graphe en clauses. Chaque instance d'une action ou d'un fait à chaque moment est une proposition.
3. Utiliser un solveur SAT
4. Si on ne trouve pas de solution (UNSAT ou TimeOut), incrémenter k et recommence
5. Si on trouve une solution, transformer la solution en solution de planification
6. Simplifier la solution.



Les bons résultats de cette approche ont ouverts la porte aux encodages SAT de problèmes industriels.

1992 la recherche locale donne de bons résultats

2001 bénéficie de zchaff, berkmin et siege

2004 premier prix de la compétition (domaines prop.)

2006 idem, ex aequo avec un autre SAT-planificateur

Par exemple, cette approche s'est révélée fructueuse en Model Checking :

Définition 11 (Principes du Model Checking) *Étant donné un automate décrivant les états possibles d'un système, on cherche à :*

- *Vérifier qu'un état n'est jamais atteint*
  C'est généralement la recherche de bug. Un état spécial « erreur » est utilisé dans la modélisation.
- *Vérifier que le système ne bloque jamais*
  Tout état doit être accessible depuis tout état, à n'importe quel moment du futur (impossible de construire une boucle infinie).

Ces problèmes ont des liens étroits avec les logiques temporelles. Avant SAT, ils étaient résolus à l'aide de BDD

Définition 12 (Principes du (Bounded) Model Checking) *Comme pour la planification, on se borne à un k fixé, qu'on incrémente à loisir.*

- *L'automate est représenté par la fonction caractéristique T de ses transitions entre états.*
  Exemple (2-bit additionneur) : $(a' \leftrightarrow \neg a) \wedge (b' \leftrightarrow a \oplus b)$
- *La propriété à vérifier est une formule logique sur l'état du système*
  Exemple : $a \wedge b$ *(l'état (11) peut-il être atteint ?)*
- *L'état initial sera une affectation le codant au temps 0*

Exemple 2 (Déroulement de boucles) *Vérifions si l'état (11) est atteignable en deux itérations dans notre 2-bit additionneur, depuis (00)*

$I(s_0) = \neg a_0 \wedge \neg b_0$
$T(s_0, s_1) = (a_1 \leftrightarrow \neg a_0) \wedge (b_1 \leftrightarrow a_0 \oplus b_0)$
$T(s_1, s_2) = (a_2 \leftrightarrow \neg a_1) \wedge (b_2 \leftrightarrow a_1 \oplus b_1)$
$p(s_2) = a_2 \wedge b_2$
$p(s_0) = a_0 \wedge b_0$
$p(s_2) = a_1 \wedge b_1$

Au final,

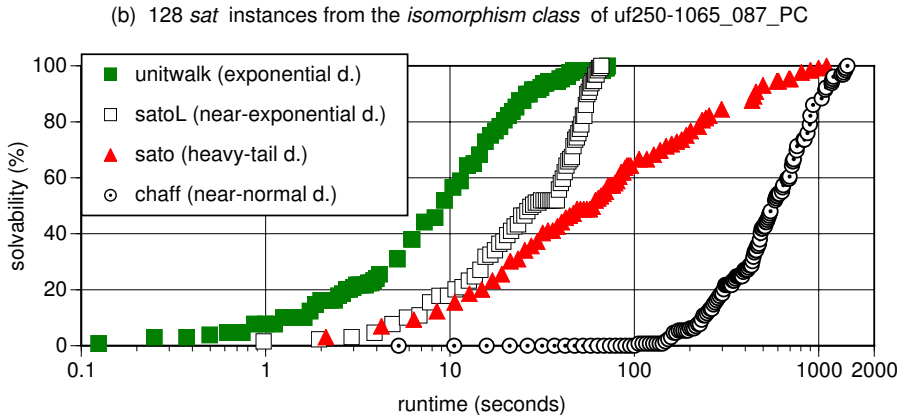
$$(\neg a_0 \wedge \neg b_0) \wedge ((a_1 \leftrightarrow \neg a_0) \wedge (b_1 \leftrightarrow a_0 \oplus b_0)) \wedge ((a_2 \leftrightarrow \neg a_1) \wedge (b_2 \leftrightarrow a_1 \oplus b_1)) \wedge (a_0 \wedge b_0) \wedge (a_1 \wedge b_1) \wedge (a_2 \wedge b_2) \quad (1)$$

est-elle satisfiable ?

On a tout de même quelques limites :

- Perte de la structure
- Difficulté d'encoder des relations entre nombres (ensembles non discrets ou grands)
- Difficulté d'encoder certaines contraintes implicites
- Difficulté d'encoder certains problèmes de maths
- Comportement des démonstrateurs... non encore compris

Sur ce dernier point, voici par exemple le comportement de quatre SAT-solvers sur un ensemble de problèmes : sans surprise, certains problèmes nécessitent beaucoup plus de temps de calcul, mais les comportements sont très variés.



3 Algorithmes complets pour SAT (et UNSAT

3.1 DP 1960

Définition 13 (Règle de Résolution (Coupure) [Gentzen, 1934) *, [Robinson, 1965]]*
Soient $c_1 = (x \vee a_1 \vee a_2 \vee \ldots a_n)$ *et* $c_2 = (\neg x \vee b_1 \vee b_2 \vee \ldots b_m)$
$c = (a_1 \vee a_2 \vee \ldots a_n \vee b_1 \vee \ldots b_m)$ *est appelée* résolvante *des clauses* c_1 *et* c_2 *(sur la variable* x*).*

Aristote dirait : « Il s'agit de la transitivité du syllogisme conjonctif ! »

Si on sait que
- A implique B
- B implique C
- Alors on peut déduire, en plus, que A implique C

Définition 14 (Preuve par résolution) *Une preuve par résolution dans* Σ *d'une clause* c' *fondamentale est une suite de clauses* $P = c_1, \ldots, c_k$ *telle que* $c_k = c'$ *et l'on ait, pour tout* $i \leq k$*,*
- $c_i \in \Sigma$*,*
- *ou il existe* c_m *et* c_n *(*$m < i$ *et* $n < i$*) dans* P *tels que* c_i *soit la*[1] *résolvante de* c_m *et* c_n*.*

On note $\Sigma \vdash_{\mathcal{R}} c'$ *le fait qu'il existe une telle preuve. Sauf mention contraire, nous notons simplement* $\Sigma \vdash c'$ *pour* $\Sigma \vdash_{\mathcal{R}} c'$*.*

Définition 15 (Complétude pour la réfutation) *Un système de preuve* $\mathcal{S}$ *est complet pour la réfutation ssi, pour toute théorie* Σ *inconsistante, la clause vide peut être prouvée à partir de* Σ *dans* $\mathcal{S}$*.*

Proposition 1 *La résolution est complète pour la réfutation.*

[1] l'unicité de la résolvante est dûe au fait que, si plus d'une résolution est possible entre les deux clauses, alors toute résolvante est tautologique.

Définition 16 (Preuve par résolution dirigée) *Soit un ordre total strict $<$ sur les variables d'une base de clauses Σ, Une preuve par résolution dirigée de la clause c', d'aprés la base Σ et selon l'ordre $<$ est une suite de clauses $P = c_1, \ldots, c_k$ telle que $c_k = c'$ et l'on ait, pour tout $i \leq k$,*

- *$c_i \in \Sigma$,*
- *ou il existe c_m et c_n (avec $m < i$ et $n < i$) dans P tels que :*
  - *c_i est la résolvante de c_m et c_n sur la variable x et*
  - *aucune des clauses c_j apparaissant après c_i dans la preuve n'est obtenue grâce à une résolution sur une variable y telle que $y < x$.*

On peut noter $\Sigma \vdash_{\mathcal{RD}} c'$ le fait qu'il existe une telle preuve.

Exemple 3

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

La preuve par résolution dirigée de l'inconsistance de Σ, suivant l'ordre $x_1 \leq x_2 \leq x_3$ est l'ensemble des clauses :

$$\begin{aligned} P \;=\; \{&(x_1 \vee \neg x_2 \vee \neg x_3)_I,\ (\neg x_1 \vee \neg x_2 \vee \neg x_3)_I, \\ &(x_2 \vee x_3)_I,\ (x_2 \vee \neg x_3)_I,\ (\neg x_2 \vee x_3)_I, \\ &(\neg x_1 \vee \neg x_2)_{x_3},\ (x_2)_{x_3},\ (x_1 \vee \neg x_2)_{x_3},\ (\neg x_1)_{x_2},\ (x_1)_{x_2},\ (faux)_{x_1}\} \end{aligned}$$

Proposition 2 *La résolution dirigée est complète pour la réfutation et la production d'impliqués premiers.*

On peut aussi relaxer la contrainte d'ordre global sur toutes les variables :

Définition 17 (Preuve par résolution régulière) *Soit un ordre $<$ sur les variables d'une base de clauses Σ, Une preuve par résolution dirigée de la clause c', d'après la base Σ et selon l'ordre $<$ est un ensemble de clauses $P = c_1, \ldots, c_k$ tel que $c_k = c'$ et l'on ait, pour tout $i \leq k$,*

- *Soit $c_i \in \Sigma$,*
- *Soit il existe $c_{m<i}$ et $c_{n<i}$ dans P tels que et que :*
  - *c_i est la résolvante de c_m et c_n sur la variable x et*
  - *ni c_m ni c_n n'ont été obtenues par résolution sur la même variable x*

On peut noter $\Sigma \vdash_{\mathcal{RR}} c'$ le fait qu'il existe une telle preuve.

Toute preuve par résolution dirigée est donc une preuve par résolution régulière. Ce système permet de maintenir le résultat de complétude suivant :

Proposition 3 *La résolution régulière est complète pour la réfutation et la production d'impliqués premiers.*

Enfin, nous présentons un système de preuve très utilisé en pratique, sur certaines bases de clauses ayant de *bonnes* propriétés :

Définition 18 (Preuve par résolution unitaire) *Une preuve par résolution unitaire dans Σ d'une clause c est une preuve par résolution dans laquelle toutes les résolutions sont unitaires (au moins une des deux clauses est unitaire). On note $\Sigma \vdash_u c$ le fait qu'il existe une telle preuve.*

La résolution est complète pour la réfutation. Une première idée est donc d'essayer de générer toutes les clauses possibles.

Dés qu'un point fixe est atteint, deux cas se présentent :

1. La clause vide a été produite : la formule est UNSAT
2. Il n'y a pas de clause vide : la formule est SAT

Cependant, cette solution se révèle extrêmement coûteuse en pratique : on peut avoir à produire 3^n clauses ! Il faut donc trouver le moyen de simplifier en éliminant les clauses sous-sommées dés que possible. On va se servir des impliqués premiers.

Définition 19 (Impliqué) *Une clause c fondamentale est un **impliqué** d'une formule f si $f \models c$.*

Définition 20 (Impliqué premier) *Une clause c fondamentale est un **impliqué premier** d'une formule f si $f \models c$ et si, pour tout impliqué c' de f, $c' \models c$ implique $c' = c$ (à l'équivalence logique prés).*

Définition 21 (Impliqués premiers) *On note $\mathcal{PI}(f)$ l'ensemble des impliqués premiers d'une formule f. Pour toute formule f, $\mathcal{PI}(f)$ est défini de manière unique à l'équivalence logique près.*

Les impliqués premiers caractérisent toutes les clauses de longueur minimale qui sont des conséquences logiques de la formule considérée. Toute clause fondamentale, conséquence logique de f, est donc soit un impliqué premier soit une clause sous-sommée par un impliqué premier.

Si l'on considère l'ensemble $\mathcal{PA}(f)$ comme une $\mathcal{DNF}$ et $\mathcal{PI}(f)$ comme une $\mathcal{CNF}$, on a la propriété suivante :

Propriété 1 $f \equiv \mathcal{PA}(f) \equiv \mathcal{PI}(f)$.

Il est aussi intéressant de noter qu'en raison de la dualité des connecteurs $\vee$ et $\wedge$ on a :

Propriété 2 $\neg\mathcal{PA}(f) \equiv \mathcal{PI}(\neg f)$.

Plutôt que de faire les résolutions dans n'importe quel ordre en attendant d'atteindre le point fixe, on se donne un ordre global sur les variables, et on effectue les résolutions dans cet ordre (résolution dirigée).

Cet algorithme a été :
- Proposé en 1960 par Davis et Putnam (DP-60) ;
- Revisité en 1994 par Dechter et Rish (appelé DR pour Directional Resolution), mais implantant le principe de la Bucket Elimination ;
- Passé à l'échelle en 2001 par utilisation de structures de données compactes ;
- A la base de beaucoup d'algorithmes de raisonnement.

Une idée très simple, liée à la décomposition de Shannon :

Définition 22 (Décomposition de Shannon) *Si la variable x apparaît dans f, alors*

$$f \equiv (x \wedge f_x) \vee (\neg x \wedge f_{\neg x})$$

On note que x a « disparu » dans les f simplifiés de la partie droite.

3.2 DLL 1962

Voici un algorithme naïf pour résoudre SAT à l'aide des décompositions de Shannon :

Algorithme 1 : BT($\Sigma, \mathcal{I}$)

Data : Σ Une formule sous CNF
Data : $\mathcal{I}$ Une interprétation partielle
Result : SAT si un modèle est trouvé, UNSAT sinon
begin
  if $\Sigma|\mathcal{I}$ *est vide* **then** retourner SAT ;
  if $\Sigma|\mathcal{I}$ *contient* $\perp$ **then** retourner UNSAT ;
  Soit l un littéral de $\Sigma|\mathcal{I}$;
  if *BT(*$\Sigma, \mathcal{I}.l$*) retourne* SAT **then** retourner SAT ;
  retourner BT($\Sigma, \mathcal{I}.\neg l$)
end

$\Sigma|\mathcal{I}$ est la formule réduite par $\mathcal{I}$, dans laquelle toutes les clauses satisfaites par $\mathcal{I}$ sont retirées, et tous les littéraux faux par $\mathcal{I}$ sont retirés de leurs clauses.

On peut l'améliorer grandement en repérant les clause unitaires et les littéraux purs, qui n'ont qu'un seul assignement valable :

Algorithme 2 : DPLL($\Sigma, \mathcal{I}$) (La procédure DPL-62)

Données : Σ une formule CNF
Données : $\mathcal{I}$ une interprétation partielle
Résultat : SAT si un modèle est trouvé, UNSAT sinon
début
  si $\Sigma|\mathcal{I}$ *est vide* **alors** retourner SAT ;
  si $\Sigma|\mathcal{I}$ *contient* $\perp$ **alors** retourner UNSAT ;
  si $\Sigma|\mathcal{I}$ *contient une clause unitaire* l **alors** retourner DPLL($\Sigma, \mathcal{I}.l$);
  si $\Sigma|\mathcal{I}$ *contient un littéral pur* l **alors** retourner DPLL($\Sigma, \mathcal{I}.l$);
  Soit l un littéral de $\Sigma|\mathcal{I}$;
  si *BT(*$\Sigma, \mathcal{I}.l$*) retourne* SAT **alors** retourner SAT ;
  retourner BT($\Sigma, \mathcal{I}.\neg l$)
fin

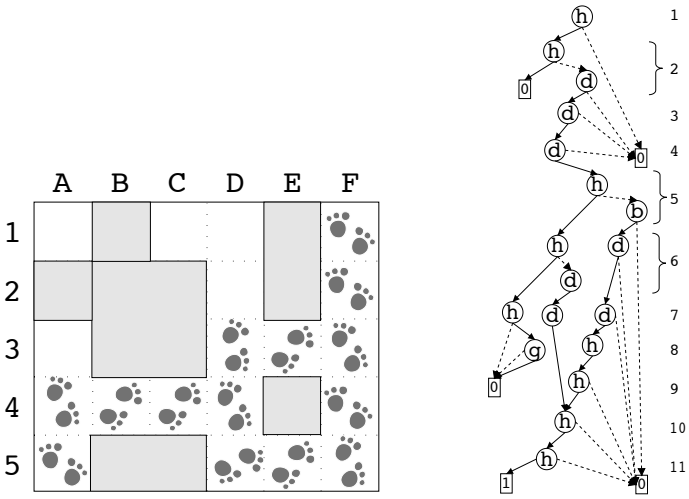
3.3 Construction de ROBDD

Dans les années 90, les techniques SAT n'offrent de bonnes performances que sur des problèmes jouets, de petite taille.

Les industriels se lancent dans le Model Checking et SAT ne suit pas. Les procédures n'arrivent pas à capturer la structure de ces problèmes.

Les industriels cherchent à montrer
- l'équivalence de deux circuits
- l'atteignabilité d'un état

L'état de l'art de l'époque est basé sur les ROBDD.



Chaque variable du BDD est du type h_i, b_i, d_i, g_i signifiant qu'au pas i, on choisit les directions respectives haute, bas, droite ou gauche.

FIG. 1 – Codage sous forme de BDD des chemins sans cycle allant de $A5$ à $F1$.

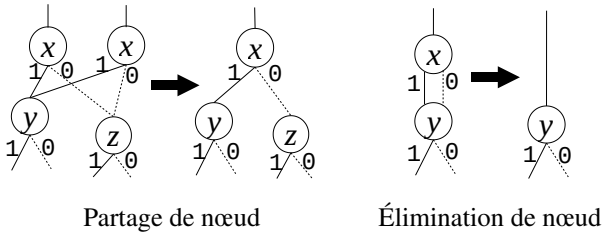


FIG. 2 – Règles de réductions dans les BDDs

Définition 23 (Diagramme de Décision Binaire) *Un graphe dirigé G est un BDD sur un ensemble de variables V s'il possède les caractéristiques suivantes :*

- *G est un graphe dirigé sans cycles ;*
- *Il possède un unique nœud racine et seulement deux types de feuilles (1 ou 0) ;*
- *Ses nœuds internes, notés $\Delta(x_i,\ A_1,\ A_2)$:*
  - *sont étiquetés par une variable x_i de V ;*
  - *ont exactement deux fils, notés respectivement A_1 et A_2 ;*

Définition 24 (OBDD) *Étant donné un ensemble de variables V et une relation d'ordre total sur V, noté $<$, un BDD Δ est un OBDD (pour « Ordered ») ssi pour tout nœud interne de Δ, tout fils Δ' de Δ correspondant à un nœud interne est tel que $label(\Delta) < label(\Delta')$.*

Définition 25 (ROBDD) *Un OBDD est un ROBDD (pour « Reduced ») s'il satisfait aux règles de réduction 1, 2 et 3.*

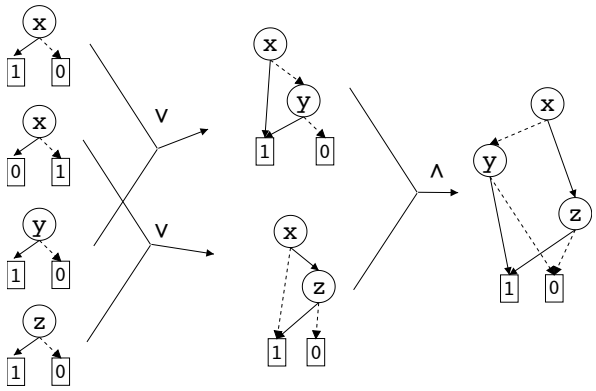


FIG. 3 – Construction récursive d'un ROBDD

Opération	Description	Complexité
SAT ?	*La fonction représentée est-elle satisfiable ?*	$O(1)$
UnModèle	*Exhiber un modèle de la fonction*	$O(n)$
NbModèles	*Compter le nombre de modèles*	$O(\|A\|)$
$A \odot B$	*Appliquer l'opérateur logique* $\odot$	$O(\|A\|.\|B\|)$

TAB. 1 – Complexité en temps des opérations de base sur les ROBDD, n représentant le nombre de ses variables

Définition 26 (Opérateur ite**)** *L'opérateur logique ternaire* ite *est défini par* $ite(f, f_1, f_0) \equiv (f \rightarrow f_1) \wedge (\neg f \rightarrow f_0)$

Notons que ce connecteur ternaire code « *Si* f *Alors* f_1 *Sinon* f_0 ».

Proposition 4 (Canonicité des BDD, [Bryant1986a) *] Soit* f *une fonction booléenne construite sur les variables propositionnelles* $x_1, \ldots, x_n$*. Soit un ordre* $<$ *donné sur ces variables, alors il existe un unique ROBDD codant* f *et respectant l'ordre* $<$

La construction d'un BDD peut s'effectuer récursivement en partant des connecteurs logiques les plus imbriqués et en recomposant récursivement la formule.

Exemple 4 *Pour construire de la formule simple* $(x \vee y) \wedge (\neg x \vee z)$*, il faut d'abord disposer de* 4 *BDD (pour* x*,* $\neg x$*,* y *et* z *respectivement), puis construire les BDD encodant les formules* $x \vee y$ *et* $\neg x \vee z$*, puis calculer la conjonction des deux BDD.*

4 Heuristiques pour DPLL

A chaque étape, le choix du littéral dans la boucle principale de DPLL a une grande importance sur l'efficacité du programme, comme le montre la figure 4. Les heuristiques utilisées portent principalement sur la fréquence des littéraux, leur forme et la longueur de leurs clauses.

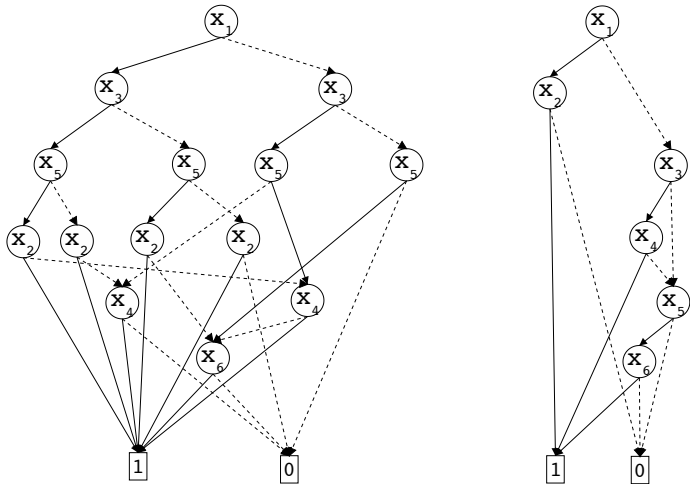


FIG. 4 – Importance de l'ordre dans les BDD. La formule est $(x_1 \wedge x_2) \vee (x_3 \wedge x4) \vee (x_5 \wedge x_6)$. À gauche, l'ordre est $x_1 \leq x_3 \leq x_5 \leq x_2 \leq x_4 \leq x_6$. À droite, l'ordre est $x_1 \leq x_2 \leq x_3 \leq x_4 \leq x_5 \leq x_6$

Définition 27 (Heuristique de Böhm [Burö, Klein-Büning, 1992])

$$H_i(x) = \alpha max(h_i(x), h_i(\neg x)) \ + \ \beta min(h_i(x), h_i(\neg x))$$

$h_i(x)$ *est le nombre de clauses non résolues contenant* x*, et de longueur* i

Il est suggéré de prendre $\alpha = 1$ et $\beta = 2$ L'intuition derrière cette heuristique est que l'on préfère satisfaire en priorité les clauses courtes, ou obtenir encore plus de clauses courtes.

Définition 28 (Heuristique de MOM (Maximum Occurences on clauses of Minimum size) [Dubois et al., 96])

$$[f^*(x) + f^*(\neg x)] * 2^k + f^*(x) * f^*(\neg x)$$

Ici, les clauses courtes pèsent *exponentiellement* plus lourd, mais la recherche reste équilibrée.

Définition 29 (Heuristique de Jeroslow-Wang [Jeroslom and Wang, 90])

$$J(l) = \Sigma 2^{-|\omega|}$$

Toujours un poids exponentiellement plus important aux clauses courtes. On peut mesurer le littéral avec le plus grand $J(l)$ ou la variable dont la somme $J(x) + J(\neg x)$ est la plus grande (2-Sided JW).

Définition 30 (Heuristique DLIS (Dynamic Largest Individual Sum)[Marques, 99])
Tout simplement :

$$C_p + C_n$$

où C_p *est le nombre de clauses non résolues.*

Il s'agit d'une simplification de Böhm qui donne des résultats pas si mauvais que cela... et même parfois bons.

Dans satz, l'heuristique utilisée regarde en avance la variable qui va générer le plus de propagation unitaire des deux côtés de l'arbre.