SAT

- Qu’est-ce que le problème SAT ?
- Pour résoudre quels problèmes ?
- De SAT à CSP, et réciproquement
- Comment comparer l’efficacité de solveurs SAT
- DPLL : un algorithme de base de type backtrack (amélioré)

Slides en anglais empruntés aux tutoriels SAT de Anbulagan & J. Rintanen, et D. Le Berre

Petit historique

- Problème fondamental en théorie de la complexité
  - Premier problème prouvé NP-Complet [Cook 1971]

- Résolution pratique
  - au départ, chercheurs en raisonnement automatique
  - puis : vérification de circuits électroniques
    planification (IA)
    gestion de circulation automobile
    vérification de logiciel ou matériel
    Ex : Processeur Intel Core i7,
    Eclipse (pour gérer les dépendances entre composants)
    problèmes combinatoires : sudoku, n-reines, *etc.*

Rappels de logique propositionnelle

- symbole propositionnel, *variable propositionnelle*, atome x
- *littéral* : variable ou négation d'une variable x ¬x
- *clause* : disjonction de littéraux $(x1 \lor \neg x2 \lor x4)$
- *formule* $\mathcal{F}$ sous forme normale conjonctive (CNF) :
  conjonction de clauses
  $\mathcal{F} = (x1 \lor \neg x2 \lor x4) \land (x4 \lor \neg x5) \land (\neg x3)$

- Une **clause** est *satisfaite* si **au moins un** de ses littéraux est vrai
- Une **CNF** est *satisfaite* si **toutes** ses clauses sont satisfaites

- clause *unitaire* : un seul littéral
- clause *vide* : sans littéraux

SAT / UNSAT

- Soit A un ensemble de variables propositionnelles
  Une formule $\mathcal{F}$ construite sur A est **satisfiable** s'il existe un **modèle** de $\mathcal{F}$ c'est-à-dire une valuation v : A → {vrai, faux} qui rend $\mathcal{F}$ vraie
- Problème **SAT** (« satisfiability »)
  Entrée : une formule $\mathcal{F}$ sous forme normale conjonctive
  Question : $\mathcal{F}$ est-elle satisfiable ?
- Problème **UNSAT** (« unsatisfiability »)
  Entrée : une formule $\mathcal{F}$ sous forme normale conjonctive
  Question : $\mathcal{F}$ est-elle insatisfiable ?

En pratique : si $\mathcal{F}$ est satisfiable, donner une **solution** (un modèle de $\mathcal{F}$), sinon donner un sous-ensemble minimal de clauses qui **ne peut être satisfait**

Exemple : « Student-Courses »

A student would like to decide on which subjects he should take for the next session. He has the following requirements:

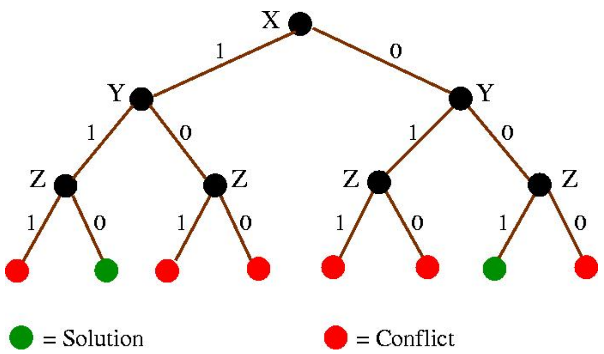
- He would like to take Math or drop Biology
- He would like to take Biology or Algorithms
- He does not want to take Math and Algorithms together

Which subjects the student can take?

X : Math
Y : Biology
Z : Algorithms

$$\mathcal{F} = (X \vee \neg Y) \wedge (Y \vee Z) \wedge (\neg X \vee \neg Z)$$

Binary tree of all possible assignments $\mathcal{F} = (X \vee \neg Y) \wedge (Y \vee Z) \wedge (\neg X \vee \neg Z)$

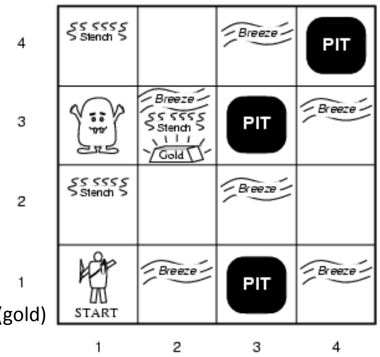


There are 2 possible solutions:

- He could take Math and Biology together. (X=1, Y=1, Z=0)
- He could only take Algorithms. (X=0, Y=0, Z=1)

Exemple en déduction automatique

Monde du Wumpus



Buts de l'agent : (1 : priorité) rester en vie
(2) trouver l'or

Actions possibles : se déplacer
tirer une flèche (1 seule)
ramasser l'or

Perceptions : détecter un souffle (breeze)
une puanteur (stench)
quelque chose qui brille (gold)
un hurlement

Base de connaissances de l'agent :

- les lois générales du monde du Wumpus
  ex : s'il y a un trou, on perçoit une brise dans les cases adjacentes
- les faits qu'il a découvert sur le monde qu'il explore
  ex : on perçoit une brise en [1,2], il y a un trou en [1,3]

Lois générales du monde *ex :* instanciées sur les perceptions en position [1,1]

Les trous causent des brises perceptibles dans les cases adjacentes
(et ce sont les seules causes de brises)

brise-1-1 ↔ trou-1-2 ∨ trou-2-1

(→) ¬ brise-1-1 ∨ trou-1-2 ∨ trou-2-1
(←) (¬ trou-1-2 ∨ brise-1-1) ∧ (¬ trou-2-1 ∨ brise-1-1)

Le wumpus cause une puanteur perceptible dans les cases adjacentes
(et c'est la seule cause de puanteur)

pue-1-1 ↔ wumpus-1-2 ∨ wumpus-2-1

L'or cause un éclat brillant perceptible dans la salle
(et c'est la seule cause de brillance)

éclat-1-1 ↔ or-1-1

Lorsque le wumpus meurt, son hurlement est perceptible partout

wumpus-meurt ↔ cri-1-1

Il y a un et un seul wumpus dans ce monde

(wumpus-1-1 v ....) ∧ (¬ wumpus-1-1 ∨¬ wumpus-1-2) ∧ ...

Problème : quand l'agent arrive en position [i,j], quels nouveaux faits peut-il déduire à partir de sa base de connaissances ?

Faits obtenus par ses perceptions (+ éventuellement : faits déjà déduits)
Lois générales du monde

Base de connaissances (BC)

Peut-on en déduire tel fait ?

ex : or-i-j ?
¬ trou-(i+1)-j ?
wumpus-i-(j+1) ?

De BC peut-on inférer F ?
Est-il certain que dans toutes les situations qui respectent la BC, F est vrai ?

BC ⊨ F ? Autrement dit : **(BC ∧ ¬ F) est-elle insatisfiable ?**

Remarque :

si (BC ∧ ¬ F) est satisfiable, cela signifie que **BC ⊭ F**
il n'est pas certain que F
au moins une situation possible où F est faux

ce qui est différent de **BC ⊨ ¬ F**
il est certain que F est faux

L'agent est en [1-1] et ne perçoit rien de spécial. Que peut-il déduire sur [1-2] et [2-1] ?

Nouveau fait : ¬ brise-1-1

¬ brise-1-1 ∨ trou-1-2 ∨ trou-2-1
¬ trou-1-2 ∨ brise-1-1
¬ trou-2-1 ∨ brise-1-1

brise en 1-1 ssi
trou en 1-2 ou trou en 2-1

BC ⊨ **¬ trou-2-1** ?

BC ∧ **trou-2-1** (in)satisfiable ?

¬ brise-1-1
trou-2-1
¬ brise-1-1 ∨ trou-1-2 ∨ trou-2-1
¬ trou-1-2 ∨ brise-1-1
¬ trou-2-1 ∨ brise-1-1

BC ∧ **trou-2-1** insatisfiable, autrement dit BC ⊨ **¬ trou-2-1**

¬ trou-2-1 est donc certain

Partant de 1-1, l'agent est allé en 1-2 puis en 2-1 (en repassant par 1-1).

¬ trou-1-1, ¬ wumpus-1-1 (toujours vrai)

- rien perçu en 1-1 :
  ¬ brise-1-1, ¬ pue-1-1, ¬ brille-1-1, ¬ cri-1-1

BC ⊨ ¬ trou-1-2, ¬ wumpus-1-2 (donc case 1-2 ok)
¬ trou-2-1, ¬ wumpus-2-1 (donc case 2-1 ok)
¬ or-1-1, ¬ wumpus-meurt

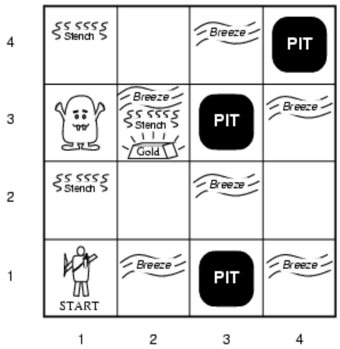
- brise en 1-2 (et c'est tout)
  brise-1-2, ¬ pue-1-2, ¬ brille-1-2, ¬ cri-1-2

BC ⊨ ¬ wumpus-2-2 (entre autres)

- puanteur en 2-1 (et c'est tout)
  pue-2-1, ¬ brise-2-1, ¬ brille-2-1, ¬ cri-2-1

BC ⊨ ¬ trou-2-2 (donc case 2-2 ok)
wumpus-3-1 (en utilisant ¬ wumpus-1-1 et ¬ wumpus-2-2)

etc.



Remarque : à part les faits disant que [1,1] est sûre, les faits à mémoriser sont les **perceptions.** Les autres faits **peuvent se déduire** des perceptions et lois générales

DE SAT à CSP

Problème de décision SAT
Données : une CNF propositionnelle $\mathcal{F}$ « *instance* de SAT »
Question : $\mathcal{F}$ est-elle satisfiable ?

Problème de décision CSP
Données : un réseau (X,D,C) « *instance* de CSP »
Question : ce réseau admet-il une solution ?

Peut-on résoudre SAT en utilisant un algorithme qui résout CSP ?

(et on aimerait bien aussi trouver les solutions)

Réduction de problèmes

- Soient deux problèmes de décision P1 et P2.
  P1 se réduit à P2 s'il existe une transformation **t** qui,
  à toute instance **I1** de **P1** associe une instance **t(I1)** de **P2**,
  tel que la réponse à **t(I1)** est oui **ssi** la réponse à **I1** est oui

- La transformation **t** est appelée **réduction**.
  Elle est dite **polynomiale**
  si elle se calcule en temps polynomial en la taille de **I1**

- Elle **préserve les solutions** si **t** définit une **bijection** entre les solutions à **I1** et les solutions à **t(I1)**

De SAT à CSP

Exemple :
« student problem »

Plusieurs réductions connues. En voici une simple :

- Symboles propositionnels de $\mathcal{F}$ ⇒ variables de X
- Pour tout x ∈ X, D(x) = {1,0}
- Chaque clause de $\mathcal{F}$ fournit une contrainte de C :

  clause $\{l_1, \ldots, l_k\}$ — on suppose qu'il n'y a pas deux littéraux opposés (sinon on peut ignorer cette clause)

  ⇒ contrainte
  - de portée $(x_1, \ldots, x_k)$ où x_i est le symbole du littéral l_i
  - excluant la seule valuation des symboles prop. qui rend la clause fausse

- La transformation Sat2Csp est une **réduction** de SAT à CSP :
  $\mathcal{F}$ est satisfiable *ssi* Sat2Csp($\mathcal{F}$) admet une solution
- Sat2Csp est **polynomiale**
  (si contraintes en extension : il faut que l'arité des clauses soit bornée)
- Sat2Csp **préserve les solutions**

De CSP (contraintes en extension) à SAT

Exemple : coloration de l'Australie

- Idée : « *la variable x peut prendre la valeur v* » se traduit par le symbole xv
  → Ensemble des symboles de la formule F = { xv | x ∈ X et v ∈ D(x) }

- Traduction des **variables** et des **domaines** :
  → pour tout x ∈ X avec $D(x) = \{v_1, ..., v_n\}$ on a les clauses :
  $(xv_1 \vee ... \vee xv_n)$ *x prend* au moins une *valeur de D(x)*
  $(\neg xv_i \vee \neg xv_j)$ pour $i < j$ et $1 \leq i,j \leq n$ *x prend* au plus une *valeur de D(x)*

- pour chaque **contrainte** sur $(x_1 ... x_k)$, on a un ensemble de clauses de taille k construites à partir des tuples interdits par la contrainte :
  si le tuple $(v_1, ..., v_k)$ est interdit par la contrainte
  on pose $\neg(x_1v_1 \wedge ... \wedge x_kv_k)$
  d'où la clause $(\neg x_1v_1 \vee ... \vee \neg x_kv_k)$

Csp2Sat est une **réduction polynomiale** qui préserve les solutions

Résolution pratique de SAT / UNSAT ?

- Pendant longtemps, en face d'un problème difficile, on disait :
  « Réduisez SAT à votre problème, et montrez qu'on ne peut pas le résoudre efficacement ! »
- Evolution extrêmement rapide de la taille des formules traitables
  - 1996 : 1000 variables
  - 2006 : un million de variables
    [aujourd'hui : plusieurs millions de variables]

  ... sur des problèmes industriels
  (tailles moins grandes sur des instances aléatoires)
- Facteurs de progrès
  - Progrès matériels (processeurs)
  - Algorithmes simples mais heuristiques sophistiquées et implémentations ingénieuses

Génération aléatoire

format DIMACS

```
c   1
p  cnf   64  254
-9  -31  -50    0
 3  -32  -46    0
-26  -53   64   0
 9   11  -27    0
-10   55  -59   0
-21  -36  -51   0
 39  -48  -53   0
 27  -33   37   0
 43  -58  -64   0
-12   21  -59   0
 10  -27  -43   0
-15  -31  -62   0
  1  -20   28   0
 -9  -14  -64   0
 -5   37   53   0
-32  -37   49   0
.....
.....
```

Instance (CNF) avec **64 variables** **254 clauses**

Chaque clause est de taille 3

Pour générer cette instance, on a fixé :
- la taille des clauses (α = 3)
- le nombre de variables (**n = 64**)
- le nombre de clauses (**m = 254**)

Algo :

On a **n** variables : $x_1 \dots x_n$
Pour i de 1 à m
    Choisir aléatoirement α variables
    Pour j de 1 à α
        choisir aléatoirement {+, -}
    // on obtient α littéraux

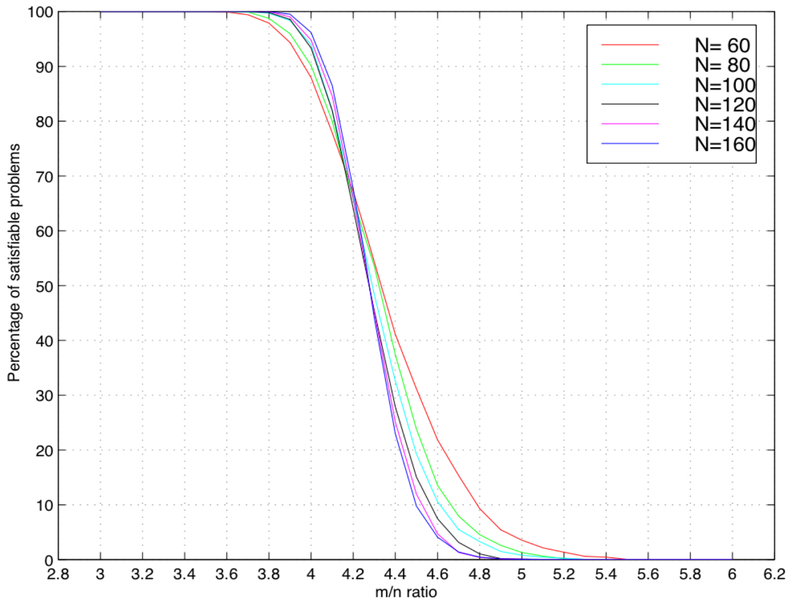
Comment comparer l'efficacité de solveurs SAT ?

- sur des instances issus de problèmes industriels
- sur des instances aléatoires
  - **paramètres** :
    - nombre de variables (n)
    - nombre de clauses (m)
    - taille des clauses : ici on fixe la même taille pour toutes les clauses (α)
  - **question** :
    y a-t-il une combinaison de ces paramètres qui rend les instances **intrinsèquement** difficiles ?
    C'est-à-dire difficiles pour tous les algorithmes ?

Réponse (expérimentale) : **oui**

Paramètre clé : **densité = m/n**

→ une certaine valeur de **m/n** pour une taille de clause α



α= 3

Pour chaque valeur de n : on génère aléatoirement des instances en faisant grandir la densité (ratio m/n) et on mesure le % d'instances satisfiables à chaque valeur de m/n

Phénomène de transition de phase

- On observe un phénomène de **transition de phase** :
  lorsqu'on augmente m/n, le % d'instances satisfiables passe **brutalement** de **100%** à **0%**
- Pour chaque taille de clause α, on observe une valeur de m/n critique
  α= 2 → m/n = 1
  α= 3 → m/n ≈ 4.3 (environ)
- Cette valeur divise l'espace des instances en 3 régions
  - **sous-contraintes** : presque toutes les instances sont satisfiables
    et il y a beaucoup de solutions (→ faciles à résoudre)
  - **sur-contraintes** : presque toutes les instances sont insatisfiables
    et les échecs sont rapides (→ faciles à résoudre)
  - **critiques** : environ 50% des formules sont satisfiables
    les formules satisfiables ont peu de solutions
    les formules insatisfiables sont « presque satisfiables »
    (→ difficiles à résoudre)

Transition de phase et difficulté de résolution des instances SAT

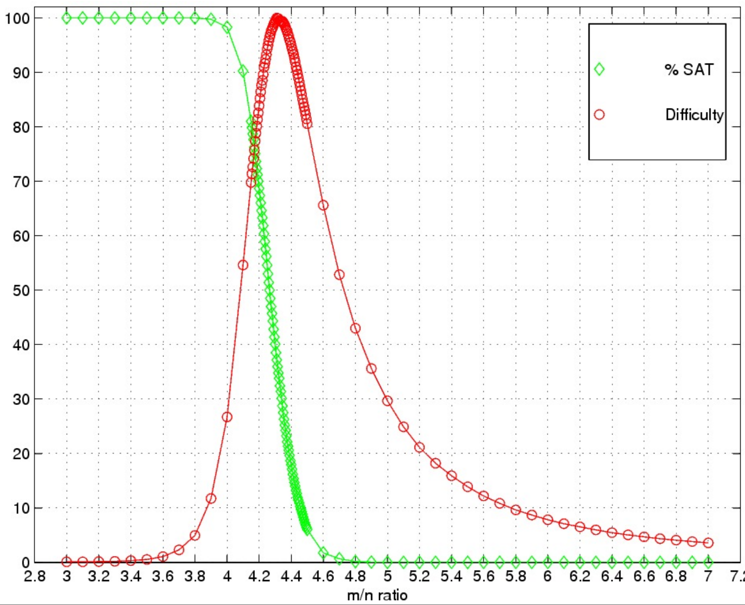
- Phénomène de **transition de phase** :

  observé expérimentalement
  et partiellement prouvé d'un point de vue théorique

- Dans la région critique :
  - les formules satisfiables ont peu de solutions
    → donc solutions difficiles à trouver
  - les formules insatisfiables sont « presque satisfiables »
    → donc les échecs surviennent après l'affectation d'un grand nombre de variables

- On observe expérimentalement que les algorithmes de résolution de SAT trouvent tous la région critique difficile

Phase Transition and Difficulty Level (at n = 200)



α= 3

SAT

- Qu'est-ce que le problème SAT ?
- Pour résoudre quels problèmes ?
- De SAT à CSP, et réciproquement
- Comment comparer l'efficacité de solveurs SAT
- DPLL : un algorithme de base de type backtrack (amélioré)

ALgorithme DPLL

Davis–Putnam–Logemann–Loveland

- Algorithme de type **backtrack** :
  - assigner les variables de F à {vrai,faux} tout en satisfaisant les clauses
  - principe de solution partielle qu'on étend
  - en cas de conflit (une clause ne peut pas être satisfaite), backtrack
- Soit une clause **C** :
  - si C a **un** littéral assigné à **vrai**, elle est satisfaite
    → on n'a plus besoin de la considérer
  - si C a **tous** ses littéraux devenus **faux**, elle ne peut être satisfaite
  - si C a **tous** ses littéraux **assignés sauf un**, elle devient **unitaire** et il existe une seule façon de la satisfaire
- On **retarde** le plus possible le moment de faire un choix : si une clause C est (devenue) **unitaire**, on effectue l'assignation qui la satisfait et on propage les effets de cette affectation (**propagation unitaire**)
- Quand on a fait un choix, on utilise aussi la propagation unitaire pour propager les effets de cette assignation

Propagation unitaire (UP)

UP($\mathcal{F}$) :
Tant que $\mathcal{F}$ contient une clause unitaire l

1) supprimer toutes les clauses qui contiennent l
// ces clauses sont satisfaites

2) supprimer $\neg l$ de toutes les clauses de $\mathcal{F}$ (*)
// ceci peut vider une clause, ce qui provoquera un échec
// ceci peut aussi rendre une clause unitaire

Fin Tant que
Retourner $\mathcal{F}$

(*) Règle de « résolution unitaire » :
de l et $\neg l \vee \Phi$, on déduit Φ

Algorithme DPLL ($\mathcal{F}$)

Assigner une variable = ajouter la clause unitaire qui correspond à cette assignation

Si $\mathcal{F} = \emptyset$, retourner vrai *// toutes les clauses satisfaites*
Si $\mathcal{F}$ a une clause vide, retourner faux *// cette clause ne peut être satisfaite*
Si $\mathcal{F}$ a une clause unitaire, retourner DPLL (**UP**($\mathcal{F}$))
Choisir une variable non assignée x
Retourner DPLL ($\mathcal{F} \cup \{x\}$) *// clause unitaire x ajoutée*
ou DPLL($\mathcal{F} \cup \{\neg x\}$) *// clause unitaire ¬x ajoutée*
[« ou » : choisir l'ordre]

UP = Unit Propagation (propagation unitaire)

Tant que $\mathcal{F}$ contient une clause unitaire
- Supprimer les clauses satisfaites par l'affectation correspondante
- Enlever des clauses les littéraux non satisfaits par cette affectation

Exemple : « Student-Courses »

A student would like to decide on which subjects he should take for the next session. He has the following requirements:

- He would like to take Math or drop Biology
- He would like to take Biology or Algorithms
- He does not want to take Math and Algorithms together

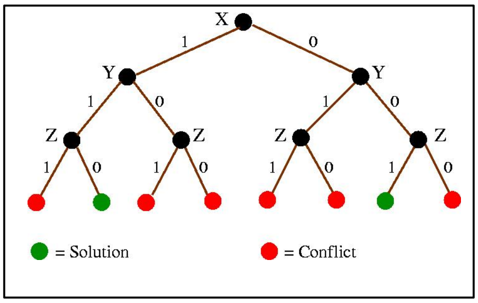
Which subjects the student can take?

X : Math
Y : Biology
Z : Algorithms

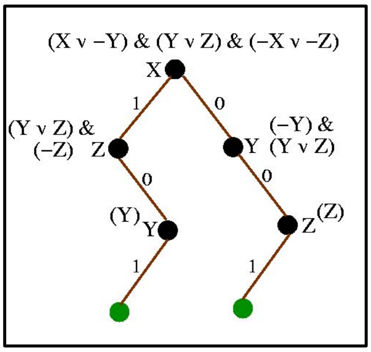
$$\mathcal{F} = (X \vee \neg Y) \wedge (Y \vee Z) \wedge (\neg X \vee \neg Z)$$

On the « student-courses » example :

$$\mathcal{F} = (X \vee \neg Y) \wedge (Y \vee Z) \wedge (\neg X \vee \neg Z)$$



Complete search tree
(with variable order X Y Z)



With DPLL procedure :
The only branching variable is *X*
There is no backtrack

Algorithme DPLL ($\mathcal{F}$, sol)

(appel initial avec sol = ∅)

Retourne une solution étendant *sol* s'il en existe une, sinon échec

Si $\mathcal{F}$ = ∅, retourner sol   *// toutes les clauses satisfaites*
Si $\mathcal{F}$ a une clause vide, retourner échec   *// une clause ne peut être satisfaite*
Si $\mathcal{F}$ a une clause unitaire
    UP($\mathcal{F}$, sol)   *// simplifie $\mathcal{F}$ et ajoute dans sol les affectations faites*
    retourner DPLL ($\mathcal{F}$, sol)
Choisir une variable non assignée x
Retourner DPLL ($\mathcal{F}$ ∪{x}, sol)   *// clause unitaire x ajoutée*
        ou DPLL($\mathcal{F}$ ∪ {¬x}, sol) *// clause unitaire ¬x ajoutée*

UP($\mathcal{F}$,sol) :

Tant que $\mathcal{F}$ contient une clause unitaire $l = x$ ou $l = \neg x$

- Ajouter $x \mapsto$ *vrai* à sol si $l = x$ et $x \mapsto$ *faux* sinon
- Supprimer de $\mathcal{F}$ les clauses satisfaites par l'assignation de x
- Enlever des clauses de $\mathcal{F}$ les littéraux non satisfaits par l'assignation de x

The imagination driving Australia's ICT future.

NICTA

Heuristics for the DPLL Procedure

Objective: to reduce search tree size by choosing a best branching variable at each node of the search tree.

Central issue:
how to select the next best branching variable?

Example: MOMS

MOMS (Maximum Occurrences in Minimum Sized clauses) heuristics: pick the literal that occurs most often in the (non-unit) minimum size clauses.

Méthodes stochastiques

Exemple

```
function GSAT(CNF c, int maxtries, int maxflips) {
    // DIVERSIFICATION STEP
    for (int i =0; i< maxtries ; i++) {
        m = randomAssignment();
        // INTENSIFICATION STEP
        for (int j=0; j<maxflips; j++) {
            if (m satisfies c)
                return SAT;
            flip(m);
        }
    }
    return UNKNOWN;
}
```

Changer la valeur d'une variable qui augmente le nombre de clauses satisfaites

- Incomplètes : peuvent ne pas trouver de solution même s'il en existe une
- Pas adaptées à UNSAT (cf. algo : retourne SAT ou UNKNOWN)

Conclusion

- A l'origine, SAT était un problème étudié par les théoriciens de la complexité et du raisonnement automatique
- Utilisé maintenant pour une très grande variété de problèmes car les solveurs SAT actuels sont surprenants d'efficacité !
- Les serveurs SAT actuels les plus performants sont basés sur :

DPLL + **analyse des conflits** (vidage d'une clause) :

backtrack non chronologique (choix de la variable de retour)

ajout d'une clause qui empêche le conflit de se reproduire

= CDCL (conflict-driven clause learning)