# Méthodes et modélisation pour l'optimisation

M1 informatique, 2019–2020

06 — DPLL

# Algorithme DPLL

- Entrée : une formule en CNF
- Sortie : **SAT** si la formule est satisfaisable, **UNSAT** sinon
- Algorithme naïf : essayer toutes les $2^n$ affectations

$$f : \{x1, \ldots, xn\} \rightarrow \{0,1\}$$

- Problème NP-complet — en général, on ne peut pas faire (beaucoup) mieux.
- Amélioration : étape de raisonnement ajoutée pour éviter de considérer toutes les possibilités.

# Propagation unitaire

$(x_P)$, $(\neg x_N)$, $(\neg x_J \vee x_N)$, $(x_P \vee x_J)$

Si on veut satisfaire toutes les clauses, il faut satisfaire la clause (unitaire) $(x_P)$

~~$(x_P)$~~, $(\neg x_N)$, $(\neg x_J \vee x_N)$, ~~$(x_P \vee x_J)$~~

$(\neg x_N)$, $(\neg x_J \vee x_N)$

Si on veut satisfaire toutes les clauses, il faut satisfaire la clause (unitaire) $(\neg x_N)$

~~$(\neg x_N)$~~, $(\neg x_J \vee$ ~~$x_N$~~$)$

$(\neg x_J)$

Si on veut satisfaire toutes les clauses, il faut satisfaire la clause (unitaire) $(\neg x_J)$

~~$(\neg x_J)$~~

C’est fini ! Toutes les clauses sont satisfaites

Affectation partielle : $x_P = 1$ $x_N = 0$ $x_J = 0$

# Propagation unitaire

- Une **clause unitaire** est une clause (u) qui ne contient qu'un seul littéral ; u = x ou u = ¬x
- ( ) note **la clause vide**
- Une clause et satisfaite ssi au moins un littéral est vrai, donc ( ) est toujours **fausse**
- Une formule est satisfaite ssi toutes ses clauses sont satisfaites, donc la formule vide, { }, est toujours **vraie**

# Algo PU

- ***Entrée*** : un ensemble F de clauses
- ***Sortie*** : un ensemble F′ de clauses ;
  F′ étant satisfaisable ssi F est satisfaisable

---

F′ ⟵ F

**While** F′ contient une clause unitaire (u)

  supprimer toutes ***occurrences*** de ¬u de F′

  supprimer toutes ***clauses*** qui contiennent u de F′

**Return** F′

# Exemple PU

Effectuer la propagation unitaire pour l'ensemble de clauses suivant

~~(x1)~~, ~~(x1 ∨ x2 ∨ ¬x3)~~, ~~(¬x1 ∨ x3 ∨ x4)~~, ~~(¬x1 ∨ ¬x2)~~,
~~(¬x1 ∨ ¬x2 ∨ x3)~~, (x3 ∨ ~~¬x4~~ ∨ x5), ~~(¬x1 ∨ x2 ∨ x4)~~

On supprime une occurrence de ¬u avec ( ~~¬u~~ … )

On supprime une clause qui contient u avec ~~( u )~~

Clause unitaire : (x1) (¬x2) (x4)

Affectation : x1 = 1 x2 = 0 x4 = 1

Clauses restantes : (x3 ∨ x5)

# Exemple PU

Effectuer la propagation unitaire pour l'ensemble de clauses suivant

~~(x1)~~, ~~( ¬x1 ∨ x2)~~, (¬~~x1~~ ∨ x3 ∨ x4), (¬~~x1~~ ∨ ¬~~x2~~)

On supprime une occurrence de ¬u avec ( ~~x~~ … )

On supprime une clause qui contient u avec ~~( … )~~

Clause unitaire : (x1) (x2)

Affectation : x1 = 1 x2 = 1

**Clause vide, donc l'ensemble n'est pas satisfaisable !**

Clauses restantes : (x3 ∨ x4), ( )

# A retenir

- Il peut y avoir un choix entre deux clauses unitaires différentes. *L'ordre de choix n'a pas d'importance* : le résultat sera le même.
- La PU n'affecte pas toujours toutes les variables.
- Si la PU rend une clause vide, alors l'ensemble de clauses restantes n'est pas satisfaisable (et par conséquent l'ensemble de clauses d'origine n'est pas satisfaisable).
- Donc, la PU peut :
  - trouver une affectation satisfaisante
  - trouver une affectation partielle
  - assurer que l'ensemble de clauses n'est pas satisfaisable

# Algo DPLL

- ***Entrée*** : un ensemble F de clauses
- ***Sortie*** : "SAT" ou "UNSAT"

---

F ⟵ **PU**(F)

**If** F contient la clause vide, **then**

    **Return** "UNSAT"

**If** F est vide, **then**

    **Return** "SAT"

x ⟵ une variable non-affectée

**If DPLL**(F ∪ {(x)}) == "SAT", **then**

    **Return** "SAT"

**Else**

    **Return DPLL**(F ∪ {(¬x)})

# Exemple DPLL

Résoudre la formule suivante par l'algorithme DPLL

$F = \{(x1 \vee \neg x2 \vee \neg x3), (\neg x1 \vee \neg x2 \vee \neg x3), (\neg x1 \vee x2 \vee \neg x3), (x1 \vee x3)\}$

- **Pile : vide**

  $F_1 \longleftarrow$ PU(F) ne modifie rien

  Choix de variable : x1

- **Pile : x1 = 1**

  $F_2 \longleftarrow PU(F_1 \cup \{(x1)\}) = \{$~~$(x1 \vee \neg x2 \vee \neg x3)$~~, $(\neg$~~$x1$~~$ \vee \neg x2 \vee \neg x3)$,
  $(\neg$~~$x1$~~$ \vee x2 \vee \neg x3)$, ~~$(x1 \vee x3)$~~, ~~$(x1)$~~$\}$

  Choix de variable : x2

- **Pile : x1 = 1, x2 = 1**

  $F_3 \longleftarrow PU(F_2 \cup \{(x2)\}) = \{$~~$(\neg x2 \vee \neg x3)$~~, ~~$(x2 \vee \neg x3)$~~, ~~$(x2)$~~$\}$

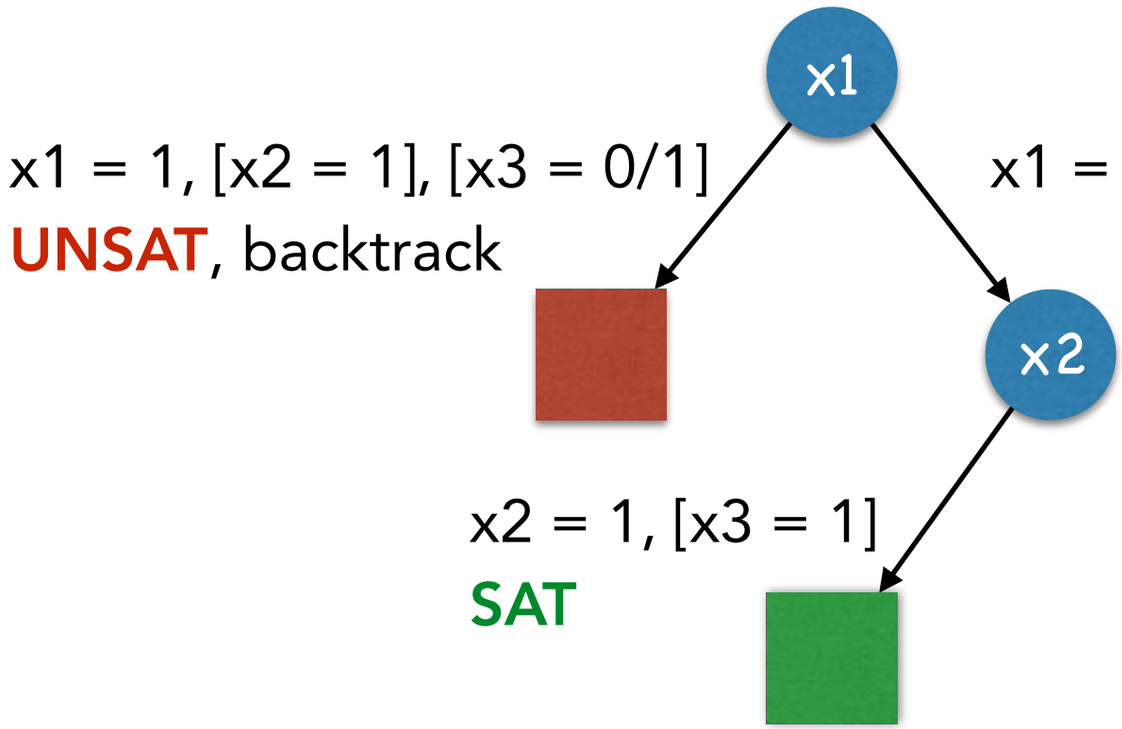
  PU affecte : x3 = 0

  $F_3 = \{\ \}$, donc la formule est satisfaite par : x1 = 1, x2 = 1, x3 = 0

# Exemple DPLL (arbre)

Résoudre la formule suivante par l'algorithme DPLL

$$F = \{ (\neg x1 \vee x2), (\neg x1 \vee \neg x2 \vee \neg x3), (\neg x2 \vee x3) \}$$



# A retenir

- Différentes critère de choix de variable possibles
  - Nous avons utilisé l'ordre lexicographique : x1, …, xn et essayé d'abord l'affectation à 1, puis à 0
- Sur la pile, on indique les affectations faites par la **PU** entre crochets :
  - **Pile : x1 = 1, [x2 = 1], x3 = 1**
- Si on trouve une clause vide dans F, on "backtrack":
  - on "renvoie" UNSAT ;
  - on dépile jusqu'à une affectation x = 1 (faite par un choix de variable, non par la **PU**) et on la remplace par x = 0 ;
  - si on vide la pile, c'est terminé

# Améliorations à DPLL

- Les solveurs d'aujourd'hui sont basés sur le DPLL mais sont beaucoup plus élaborés
- Structures de données permettant un backtrack rapid
- Conflict-Driven Clause Learning (CDCL)
  ``apprentissage de clauses guidé par des conflits"
  - quand l'algorithme DPLL trouve un conflit, il cherche **une raison concise** (exprimée en clauses) et l'utilise pour plus rapidement détecter un conflit similaire **dans le futur** !
- Back jumping (non-chronological backtracking)
- Redémarrage fréquent pour éviter de se retrouver coincé
  - **garder les clauses apprises pour le démarrage**

# CDCL et back jumping

~~(¬x1 v ¬x7)~~
~~(x7 v x11)~~
~~(x2 v x10)~~
(¬x6 v ¬x1 v x8)
(¬x6 v x7 v ¬x8)
(x6 v x7 v ¬x9)
(x6 v x9 v ¬x11)

conflit !

x1 = 1, [x7 = 0], [x11 = 1]

x1

x2

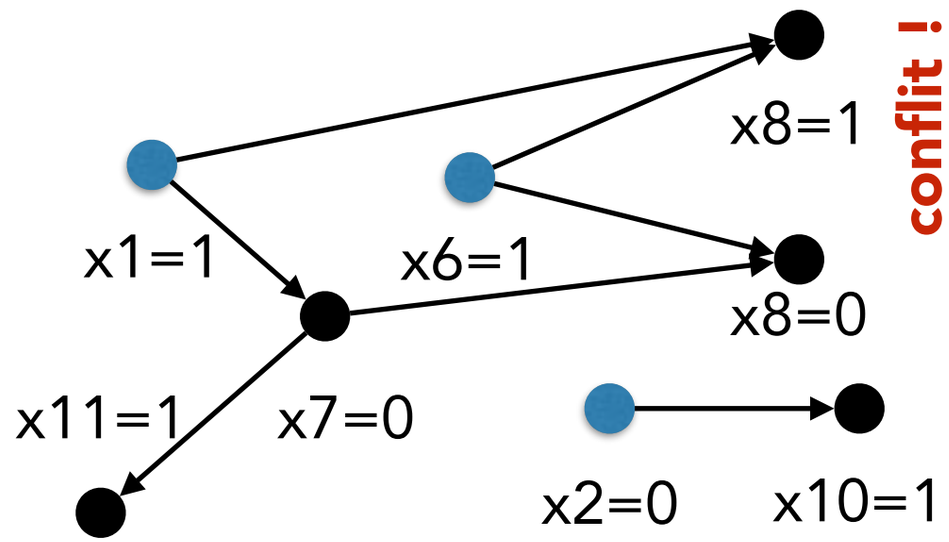
x2 = 0, [x10 = 1]

x6

x6 = 1, [x8 = 0/1]

**UNSAT**

**Graphe de conflit**



# CDCL et back jumping

~~(¬x1 ∨ ¬x7)~~
~~(x7 ∨ x11)~~
~~(x2 ∨ x10)~~
(¬x6 ∨ ¬x1 ∨ x8)
(¬x6 ∨ x7 ∨ ¬x8)
~~(x6 ∨ x7 ∨ ¬x9)~~
~~(x6 ∨ x9 ∨ ¬x11)~~

conflit !

x1 = 1, [x7 = 0], [x11 = 1]

x1

x2

x2 = 0, [x10 = 1]

x6

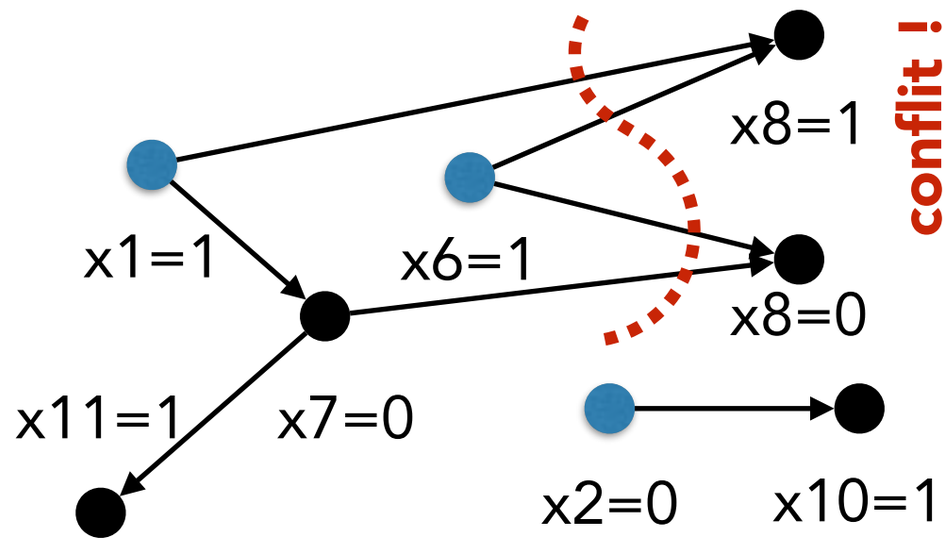
x6 = 1, [x8 = 0/1]

**UNSAT**

**Une coupe du conflit implique une nouvelle clause :**

$$\neg(x1 \wedge x6 \wedge \neg x7)$$
$$\Updownarrow$$
$$(\neg x1 \vee \neg x6 \vee x7)$$

**Graphe de conflit**

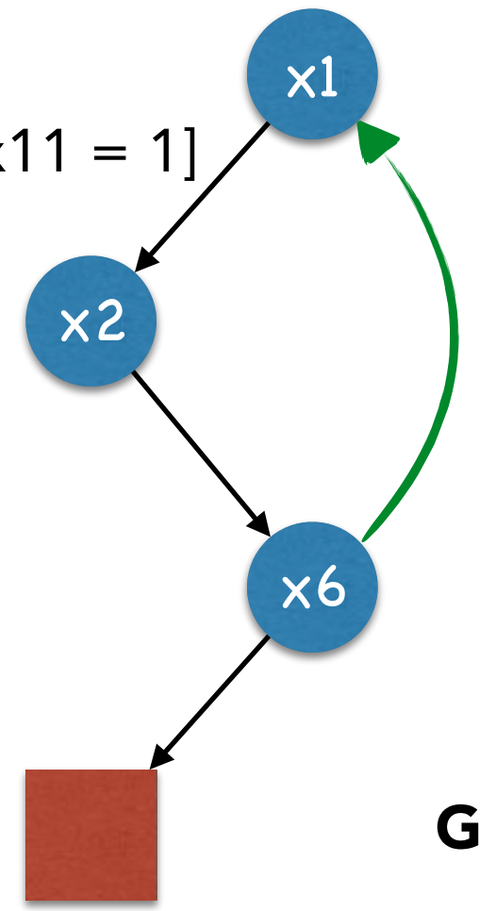


# CDCL et back jumping

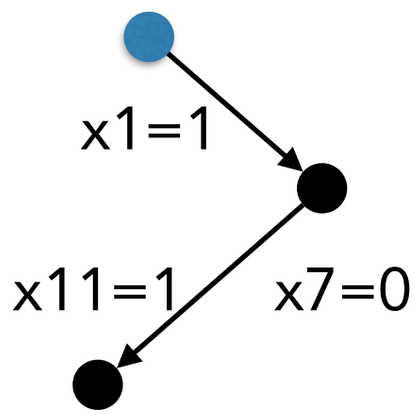
~~(¬x1 ∨ ¬x7)~~
~~(x7 ∨ x11)~~
(x2 ∨ x10)
(¬x6 ∨ ~~¬x1~~ ∨ x8)
(¬x6 ∨ ~~x7~~ ∨ ¬x8)
(x6 ∨ ~~x7~~ ∨ ¬x9)
(x6 ∨ x9 ∨ ~~¬x11~~)
(¬x1 ∨ ¬x6 ∨ x7)

**Ajouter la nouvelle clause**
**et back jump à x1=1**
**(PU donne x6=0, x9=0, x9=1, UNSAT)**

x1 = 1, [x7 = 0], [x11 = 1]




**Graphe de conflit**



# Examen le 10 janvier

- 14h00-16h00 à Rabelais A3
- Sur les notions abordées aux cours, TD et TP
- 1 feuille recto-verso manuscrite autorisée
- Calculatrice, machine, smartphone, smartwatch interdits