Informatique *Cours*
Logique - SAT Ipesup

Objectifs

À l'issue de cette section, l'étudiant devra être capable de :
- comprendre le problème SAT et la notion de satisfiabilité ;
- modéliser un problème combinatoire sous forme de formule logique ;
- comprendre le principe de résolution de 2-SAT par graphe d'implication.

De nombreux algorithmiques admettent une réponse binaire. Ce sont des *problèmes de décision*. Le problème SAT entre dans cette catégorie, en cherchant à déterminer si une formule est satisfiable. Il revêt une importance considérable, car il est le cadre naturel pour formaliser, et tenter de résoudre, des problèmes de satisfaction de contraintes, de planification, de *model-checking* (vérification de propriété d'un modèle) ou de cryptographie.

Le problème SAT est cependant un problème difficile, pour lequel on ne connaît que des algorithmes dont la complexité dans le pire cas est exponentielle. Et on n'a guère d'espoirs d'améliorer un jour ce pire cas, sachant que Cook et Levin ont établi, dans les années 1970, que le problème SAT est NP-complet. Malgré tout, les années 2000 ont vu émerger de nombreuses propositions de codes permettant le traitement de problèmes SAT ayant des milliers de variables propositionnelles et des millions de contraintes. Il existe même des compétitions de SAT solver (`http://www.satcompetition.org/`).

Problème SAT

Le **problème SAT** est un problème de décision qui détermine si une formule de la logique propositionnelle est satisfiable ou non.
- $\mathrm{SAT}(\varphi) = \mathsf{V}$ si φ est satisfiable.
- $\mathrm{SAT}(\varphi) = \mathsf{F}$ si φ est contradictoire.

Souvent, la recherche automatisée d'une solution se fait à partir de la forme CNF d'une formule : on parle de **problème CNF-SAT**. La restriction du problème SAT aux CNF ayant au plus k littéraux par clause est appelée **problème k-SAT**.

Dans tous les cas, une approche naïve énumère toutes les valuations possibles d'une formule pour vérifier si l'une d'entre elles satisfait la formule.
Mais si φ est une formule qui comporte n variables propositionnelles, il existe 2^n interprétations possibles pour φ. Cette première approche est donc généralement vouée à l'échec pour des formules comportant un grand nombre de variables.
Certains algorithmes permettent toutefois d'accélérer la découverte d'une solution.

Format DIMACS

Les solveurs SAT sont des programmes qui résolvent un problème SAT.
Les formules sous forme CNF ou DNF y sont décrites suivant des règles simples dans un fichier au format dit **DIMACS-CNF**.

Ce dernier n'est autre qu'un fichier texte dont

— les premières lignes sont des commentaires signalés par la présence d'un caractère `c` en début de chaque ligne.
— La ligne suivante commence par un `p` et décrit la nature du problème en précisant la forme normale codée : `cnf` ou `dnf`. Deux entiers indiquent le nombre de variables propositionnelles et le nombre de clauses.
— Viennent ensuite les descriptions de chaque clause sous la forme de suites d'entiers. Le format adopte la convention suivante :
  — une variable propositionnelle est représentée par un entier strictement positif ;
  — sa négation par l'entier opposé ;
  — la fin d'une ligne est indiquée par la présence d'un `0`.

Exemple

```
c x y z ¬x ¬y ¬z
c 1 2 3 -1 -2 -3
p cnf 3 5
-1 0
-1 -2 0
-1 3 0
-2 3 0
3 0
```

$$(\neg x) \vee (\neg x \wedge \neg y) \vee (\neg x \wedge z) \vee (\neg y \wedge z) \vee (z)$$

1 Algorithme de Quine

L'algorithme de Quine cherche à déterminer la satisfiabilité d'une formule en construisant un *arbre de décision*. L'idée est de remplacer chaque variable propositionnelle par $\top$ d'une part, et $\bot$ d'autre part, chacune de ces options représentant une branche de l'arbre. Chaque substitution, en plus de décrire un choix d'une branche dans l'arbre de décision, donne une évaluation partielle de la formule. Si à un nœud de l'arbre, on constate que l'évaluation partielle est déjà fausse, il est inutile de poursuivre l'exploration de cette branche.

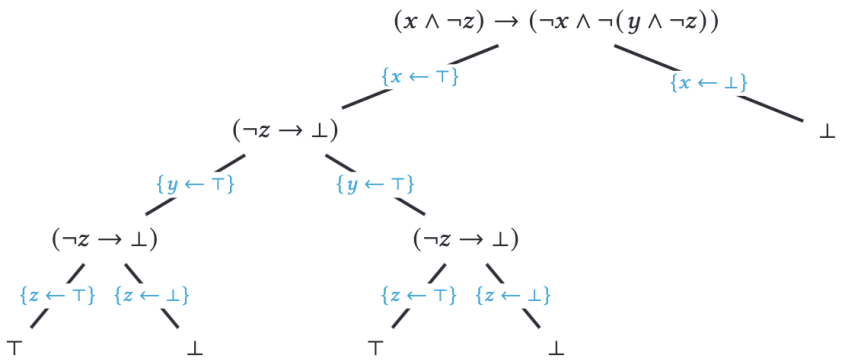
Illustrons sa mise en œuvre avec la formule

$$\varphi = (x \wedge \neg z) \to (\neg x \wedge \neg(y \wedge \neg z)).$$

On choisit d'abord de substituer x, puis, si nécessaire, de substituer y dans les formules obtenues et enfin, si nécessaire encore, de substituer z dans les dernières formules.

Après chaque substitution, on simplifie la formule à l'aide des équations suivantes :

$$\begin{array}{lll} \varphi \wedge \bot \equiv \bot & \varphi \wedge \top \equiv \varphi & \neg\top \equiv \bot \\ \varphi \vee \top \equiv \top & \varphi \vee \bot \equiv \varphi & \neg\bot \equiv \top \\ \varphi \to \top \equiv \top & \top \to \varphi \equiv \varphi & \varphi \to \bot \equiv \neg\varphi \end{array}$$



— On obtient tout d'abord avec x :

$$\varphi_1 = \varphi^{\{x \leftarrow \top\}} = \neg z \to \bot \qquad \varphi_2 = \varphi^{\{x \leftarrow \bot\}} = \top$$

On peut déjà constater que la substitution $\{x \leftarrow \bot\}$ suffit à satisfaire la formule. Mais en pratique, selon le code qui met en œuvre l'algorithme de Quine, cette solution ne sera pas forcément découverte avant d'avoir exploré la branche correspondant à φ_1. Poursuivons donc les substitutions.

— On obtient alors, en substituant y dans φ_1 :

$$\varphi_{11} = \varphi_1^{\{y \leftarrow \top\}} = (\neg z \to \bot) \qquad \varphi_{12} = \varphi_1^{\{y \leftarrow \bot\}} = (\neg z \to \bot)$$

— Une dernière substitution, de z dans φ_{11} et φ_{12}, donne :

$$\varphi_{111} = \varphi_{11}^{\{z \leftarrow \top\}} = \top \qquad \varphi_{112} = \varphi_{11}^{\{z \leftarrow \bot\}} = \bot$$

$$\varphi_{121} = \varphi_{12}^{\{z \leftarrow \top\}} = \top \qquad \varphi_{122} = \varphi_{12}^{\{z \leftarrow \bot\}} = \bot$$

Considerons les suivantes formules

Simplification de formules

```
let smart_not f = match f with
| True  -> False
| False -> True
| _     -> Not f

let smart_and f1 f2 = match f1, f2 with
| False, _ | _, False -> False
| True, f | f, True  -> f
| _, _               -> Bin (And, f1, f2)

let smart_or f1 f2 = match f1, f2 with
| True, _ | _, True  -> True
| False, f | f, False -> f
| _, _               -> Bin (Or, f1, f2)

let smart_imp f1 f2 = match f1, f2 with
| False, _ | _, True -> True
```

```
| True, f            -> f
| f, False           -> smart_not f
| _, _               -> Bin (Imp, f1, f2)

let rec simplify f = match f with
| Var _ | True | False -> f
| Not f               -> smart_not (simplify f)
| Bin (And, f1, f2)   -> smart_and (simplify f1) (simplify f2)
| Bin (Or, f1, f2)    -> smart_or  (simplify f1) (simplify f2)
| Bin (Imp, f1, f2)   -> smart_imp (simplify f1) (simplify f2)
```

La fonction `simplify` utilise des *smart constructors*, c'est-à-dire des fonctions qui se comportent comme un constructeur du type `fmla`, mais appliquent éventuellement des simplifications.

Le *smart constructor* de la conjonction, par exemple, reçoit deux formules φ_1 et φ_2 et renvoie `Bin(And,` φ_1`,` φ_2`)`, sauf s'il est certain que le résultat sera équivalent à $\top$, $\bot$ ou à l'une des deux sous-formules φ_1 ou φ_2, auquel cas il renvoie directement la formule simplifiée à la place d'une conjonction.

Alors on peut definir la fonction `quine_sat` que met en œuvre l'algorithme de Quine,

Algorithme de Quine

```
let rec quine_sat f =
  match simplify f with
  | True  -> true
  | False -> false
  | f     -> let x = varmax f in
             quine_sat (subst x True f) || quine_sat (subst x False f)
```

en substituant à chaque étape la variable de plus grand numéro encore présente.

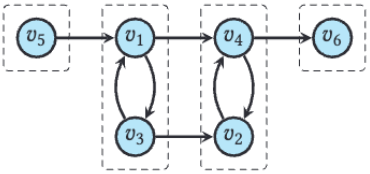
La simplification consiste alors à appliquer le *smart constructor* correspondant à chaque constructeur de la formule, en partant des feuilles pour s'assurer que les simplifications puissent s'enchaîner en cascade.

2 Graphes orientés

Définition

Un *graphe orienté* est défini par un ensemble V de *sommets* et un ensemble $E \subseteq V \times V$ de couples de sommets appelés *arcs*.

Un arc $(x, y) \in E$ est traditionnellement dessiné comme une flèche entre les sommets x et y. Voici un exemple de graphe orienté avec six sommets et sept arcs :



On a $V = \{a, b, c, d, e, f\}$ et

$$E = \{(a, b), (a, d), (b, c), (b, d), (c, d), (d, b), (e, f)\}.$$

Il est important de comprendre que le dessin importe peu. Seule la donnée des ensembles V et E définit le graphe.

Entre deux sommets, il existe au plus un arc. Dit autrement, E est un ensemble, pas un multiensemble. Il serait tout à fait possible d'autoriser de tels *multi-arcs* et on parlerait alors de *multi-graphe*. Mais cette notion plus générale n'est pas considérée ici.

Définition– adjacence dans un graphe

Si $(x, y) \in E$, on dit que y est un *successeur* de x et que x est un *prédécesseur* de y.
On note $x \to y$ la présence de cet arc.
Un arc de la forme (x, x) est appelé une *boucle*.
Pour un sommet $x \in V$, le nombre d'arcs de la forme (x, y) est appelé le *degré sortant* du sommet x et noté $d_+(x)$.
De même, le nombre d'arcs de la forme (y, x) est appelé le *degré entrant* du sommet x et noté $d_-(x)$.

Sur l'exemple ci-dessus, on a $d_-(a) = 0$ et $d_+(a) = 2$. Dans la suite, on utilisera aussi le terme de *voisins* pour désigner les successeurs d'un sommet.

Définition– chemin dans un graphe

Un *chemin* du sommet u au sommet v dans un graphe (V, E) est une séquence $x_0, \ldots, x_n$ de sommets de V tels que $x_0 = u$, $x_n = v$ et $(x_i, x_{i+1}) \in E$ pour $0 \leq i < n$:

$$u = x_0 \to x_1 \to \cdots \to x_{n-1} \to x_n = v.$$

La longueur d'un tel chemin est n, c'est-à-dire le nombre d'arcs qui le constitue.
Un *chemin simple* est un chemin sans répétition d'arc.
Un *cycle* est un chemin simple de u à u de longueur $n > 0$.
Un graphe orienté qui ne contient pas de cycle est appelé un *DAG* pour *Directed Acyclic Graph*.

On note $x_0 \to^* x_n$ la présence d'un chemin entre les sommets x_0 et x_n.

Il y a toujours un chemin de longueur 0 entre un sommet u et lui-même.

Définition– forte connexité

Un graphe orienté $G = (V, E)$ est *fortement connexe* si, pour toute paire de sommets x et y de V, il existe un chemin de x à y. Une *composante fortement connexe* de G est un sous-ensemble de sommets deux à deux reliés par des chemins, maximal pour l'inclusion.

Dans l'exemple de graphe donné plus haut, le graphe n'est pas fortement connexe car il n'y a pas de chemin de b à a. En revanche, l'ensemble $\{b, c, d\}$ est une composante fortement connexe.

3 Une modélisation SAT

3.1 Position du problème.

Un algorithme glouton de coloration de graphe ne renvoie pas toujours la meilleure solution, la logique propositionnelle constitue une approche alternative : pour ce faire, il convient d'abord de *formaliser* le problème puis de le *modéliser* sous la forme d'une formule logique.

Considérons un graphe $G = (V, E)$ où V est l'ensemble de ses n sommets ($n \in \mathbb{N}^*$) et E l'ensemble de ses arêtes. L'objectif est d'établir l'existence d'une coloration de G, c'est-à-dire une application c qui associe à chacun des sommets de G un entier de

$$C = [0, k-1], \quad k \in \mathbb{N}^*,$$

de sorte que si deux sommets u et v sont voisins alors leurs couleurs sont différentes :

$$\forall (u, v) \in E, \qquad c(u) \neq c(v).$$

Traduire ce problème en termes de logique propositionnelle requiert, dans un premier temps, la définition d'un ensemble de variables propositionnelles. Choisissons de représenter chaque sommet de G par un entier de sorte que $V = [1, n]$ et chaque couleur par un entier j de C. Un couple (i, j) définit une variable propositionnelle x_{ij} dans le sens où sa valeur de vérité est V si $j = c(i)$, F si $j \neq c(i)$.

L'ensemble des variables propositionnelles est ainsi :

$$\mathcal{V} = V \times C.$$

Parmi toutes ces variables propositionnelles, on en recherche un sous-ensemble qui satisfait aux trois contraintes suivantes :

- chaque sommet a au moins une couleur ;
- chaque sommet a au plus une couleur ;
- deux sommets adjacents n'ont pas la même couleur.

3.2 Modélisation.

La première contrainte doit spécifier qu'à tout entier $i \in V$ est associé au moins un élément de $j \in C$. Ainsi, pour un sommet i fixé, le fait qu'il ait au moins une couleur peut se traduire par la formule :

$$\bigvee_{j \in C} x_{ij}.$$

En l'appliquant à tous les sommets de G, on obtient la formule φ_1 suivante :

$$\varphi_1 = \bigwedge_{i \in V} \left(\bigvee_{j \in C} x_{ij} \right).$$

La deuxième contrainte doit spécifier qu'un sommet i du graphe ne peut pas avoir plus d'une couleur. Pour deux couleurs différentes j et j' de C, la formule $(x_{ij} \wedge x_{ij'})$ doit être fausse ; sa négation doit donc être vraie. Ce qu'on peut traduire, pour toutes paires de couleurs différentes, par :

$$\bigwedge_{(j,j') \in C^2,\, j \neq j'} \neg(x_{ij} \wedge x_{ij'}).$$

En l'appliquant à tous les sommets de G, on obtient la formule φ_2 suivante :

$$\varphi_2 = \bigwedge_{i \in V} \bigwedge_{(j,j') \in C^2,\, j \neq j'} \neg(x_{ij} \wedge x_{ij'}).$$

La troisième et dernière contrainte doit spécifier que les sommets i et i' de chaque arête (i, i') de E ne peuvent pas avoir la même couleur j. La formule $\neg(x_{ij} \wedge x_{i'j})$ doit être vraie. Appliquée à toutes les arêtes de G et pour toutes les couleurs possibles, on obtient la formule φ_3 suivante :

$$\varphi_3 = \bigwedge_{(i,i') \in E} \bigwedge_{j \in C} \neg(x_{ij} \wedge x_{i'j}).$$

La formule finale qui modélise le problème de coloration d'un graphe est :

$$\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3.$$

En transformant les négations de conjonctions en disjonctions de négations et en tenant compte des priorités des connecteurs, cette expression prend la forme suivante :

$$\varphi = \left(\bigwedge_{i \in V} \bigvee_{j \in C} x_{ij} \right) \wedge \left(\bigwedge_{i \in V} \bigwedge_{(j,j') \in C^2,\, j \neq j'} (\neg x_{ij} \vee \neg x_{ij'}) \right) \wedge \left(\bigwedge_{(i,i') \in E} \bigwedge_{j \in C} (\neg x_{ij} \vee \neg x_{i'j}) \right).$$

Cette formule étant à présent définie, le programme suivant définit la fonction `color2sat` qui renvoie une CNF associée à un graphe.

Coloration de graphe

```
let color2sat g k =
  let n = size g in
  let var s c = k * s + c + 1 in
  let clauses = ref [] in

  (* tous les sommets sont colorés *)
  for s = 0 to n - 1 do
    clauses := List.init k (fun c -> var s c) :: !clauses
  done;

  (* chaque sommet a une unique couleur *)
  for s = 0 to n - 1 do
    for c1 = 0 to k - 2 do
      for c2 = c1 + 1 to k - 1 do
        clauses := [-var s c1; -var s c2] :: !clauses
      done
    done
```

```
done;

(* deux sommets adjacents ont des couleurs différentes *)
List.iter (fun (s1, s2) ->
  for c = 0 to k - 1 do
    clauses := [-var s1 c; -var s2 c] :: !clauses
  done) (edges g);

{ kind = CNF; nbvars = n * k; clauses = !clauses }
```

Un code disponible en ligne complète ce programme en définissant notamment une fonction `color_using_sat` qui renvoie un tableau des couleurs attribuées à chaque sommet.

4 2-SAT

2-SAT occupe une position particulière puisqu'il peut être résolu avec une complexité temporelle polynomiale. Dans ce problème, une CNF comporte au plus deux littéraux par clause :

$$\bigvee_{i,j}(\ell_i \wedge \ell_j).$$

La formule suivante en est un exemple :

$$\varphi = (x \vee \neg y) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y) \wedge (x \vee \neg z).$$

La satisfiabilité de la formule est établie en déplaçant le problème vers celui de la recherche de composantes fortement connexes d'un graphe particulier appelé *graphe d'implication*. Le principe de cette transformation repose sur les équivalences sémantiques

$$(\ell_i \vee \ell_j) \equiv (\neg\ell_i) \to \ell_j \quad \text{et} \quad (\ell_i \vee \ell_j) \equiv (\neg\ell_j) \to \ell_i.$$

Si une clause ne comporte qu'un seul littéral ℓ_i, sa forme équivalente est $(\ell_i \vee \ell_i)$.

Pour une 2-CNF φ à n variables propositionnelles et m clauses, le graphe d'implication G comporte :

— $2n$ sommets, associés à chaque variable propositionnelle et chaque négation d'une variable propositionnelle ;
— $2m$ arêtes orientées : à chaque clause $(x \vee y)$, une arête est orientée du sommet associé à $\neg x$ vers le sommet associé à y ; une arête est orientée du sommet associé à $\neg y$ vers le sommet associé à x.

En termes de notations, on peut adopter les conventions suivantes.

— Les n variables propositionnelles sont désignées par $x_1, \ldots, x_n$.
— À x_i, on associe le sommet v_{2i-1} de G ; à $\neg x_i$, on associe le sommet v_{2i}.

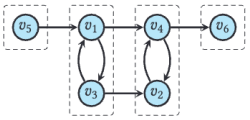
Appliquons cette procédure à la formule φ précédente en définissant les sommets et les littéraux associés à chaque variable x, y, z et à leurs négations :

littéral $\ell_1 = x$ associé au sommet v_1 littéral $\ell_2 = \neg x$ associé au sommet v_2
littéral $\ell_3 = y$ associé au sommet v_3 littéral $\ell_4 = \neg y$ associé au sommet v_4
littéral $\ell_5 = z$ associé au sommet v_5 littéral $\ell_6 = \neg z$ associé au sommet v_6

La clause $(x \vee \neg y)$ étant équivalente à $(\neg x \to \neg y)$ et $(y \to x)$, le graphe d'implication recherché comporte les deux arêtes orientées (v_2, v_4) et (v_3, v_1). En procédant de même avec les autres clauses, on définit les arêtes orientées suivantes :

$$(x \vee \neg y) : (v_2, v_4), (v_3, v_1)$$
$$(\neg x \vee y) : (v_1, v_3), (v_4, v_2)$$
$$(\neg x \vee \neg y) : (v_1, v_4), (v_3, v_2)$$
$$(x \vee \neg z) : (v_2, v_6), (v_5, v_1)$$

Ce qui mène au graphe



On peut observer que ce graphe présente une symétrie : d'une part en termes d'arêtes orientées, d'autre part en termes de littéraux présents dans chaque composante fortement connexe.

L'intérêt d'un tel graphe est que décider si φ est satisfiable équivaut à montrer que chacune des composantes fortement connexes ne contient jamais les deux sommets associés à une variable propositionnelle et à sa négation. La preuve de ce résultat fournit même une procédure de construction d'une valuation qui satisfait φ, quand cette dernière l'est effectivement.

Commençons par prouver la propriété suivante.

Propriété

Soit G le graphe d'implication d'une 2-CNF φ et v_ℓ, $v_{\ell'}$ deux sommets de G associés aux littéraux ℓ et ℓ' de φ. S'il existe un chemin de v_ℓ à $v_{\ell'}$ alors il existe un chemin de $v_{\neg\ell'}$ à $v_{\neg\ell}$.

Demonstration

Notons tout d'abord que si φ contient la clause $(\ell \vee \ell')$ alors G contient les deux arêtes orientées $(v_{\neg\ell}, v_{\ell'})$ et $(v_{\neg\ell'}, v_\ell)$.
Supposons qu'il existe un chemin $(v_{\ell_1}, v_{\ell_2}, \ldots, v_{\ell_p})$ dans G, chaque sommet v_{ℓ_i} étant associé à un littéral ℓ_i. Alors G contient les arêtes $(v_{\ell_1}, v_{\ell_2}), (v_{\ell_2}, v_{\ell_3}), \ldots, (v_{\ell_{p-1}}, v_{\ell_p})$.
Or, d'après l'observation précédente, pour chacune des arêtes $(v_{\ell_i}, v_{\ell_{i+1}})$, G contient également l'arête orientée $(v_{\neg\ell_{i+1}}, v_{\neg\ell_i})$.
Par conséquent, G contient la suite d'arêtes $(v_{\neg\ell_p}, v_{\neg\ell_{p-1}}), (v_{\neg\ell_{p-1}}, v_{\neg\ell_{p-2}}), \ldots, (v_{\neg\ell_2}, v_{\neg\ell_1})$, c'est-à-dire un chemin de $v_{\neg\ell_p}$ à $v_{\neg\ell_1}$.

Propriété

Soit G le graphe d'implication d'une formule logique φ.
La formule φ est satisfiable si et seulement si aucune composante fortement connexe de G ne contient à la fois le sommet associé à une variable propositionnelle et le sommet associé à la négation de cette variable propositionnelle.

Demonstration

Soit v_ℓ et $v_{\neg\ell}$ les sommets associés à un littéral ℓ et à sa négation.
Supposons les dans une même composante fortement connexe de G. Il existe un chemin de v_ℓ à $v_{\neg\ell}$ que l'on peut noter $(v_\ell, v_{\ell_1}, \ldots, v_{\ell_p}, v_{\neg\ell})$. Alors φ contient la sous-formule :

$$\psi = (\ell \to \ell_1) \wedge (\ell_1 \to \ell_2) \wedge \cdots \wedge (\ell_p \to \neg\ell).$$

Supposons φ satisfiable. Il existe une valuation v telle que $v(\varphi) = \mathsf{V}$ et, par conséquent, telle que $v(\psi) = \mathsf{V}$.

— Supposons $v(\ell) = \mathsf{V}$. Alors nécessairement $v(\ell_1) = \mathsf{V}$, puis $v(\ell_2) = \mathsf{V}$ et ainsi de suite jusqu'à $v(\neg\ell) = \mathsf{V}$. Ce résultat est incompatible avec l'hypothèse.
— Supposons $v(\ell) = \mathsf{F}$. Par la même analyse, on montre encore que cette hypothèse est absurde.

Donc φ n'est pas satisfiable.

Pour l'autre implication, tout d'abord, remarquons que le graphe réduit à ses composantes fortement connexes est un graphe orienté acyclique (DAG). On peut donc trier ses composantes à l'aide d'un tri topologique. Alors, il existe au moins une composante dont aucune arête n'est sortante.
En outre, comme établi plus haut, tous les littéraux des sommets d'une même composante fortement connexe ont la même valeur de vérité.
Définissons une valuation v telle que tous les littéraux de cette composante aient la valeur de vérité V. D'après la propriété précédente, les négations de ces littéraux appartiennent également à une même composante fortement connexe. On peut leur affecter la valeur de vérité F.
Puis on supprime ces deux composantes fortement connexes du graphe et on recommence la même procédure avec le DAG qui reste jusqu'à ce qu'il ne reste rien. On a ainsi construit une valuation qui satisfait φ.

L'efficacité de cet algorithme dépend en particulier de celle de la construction des composantes fortement connexes.