4.5 Le problème SAT

4.5.1 Définition du problème

Définition 4.39 (Problème SAT)**.** Le problème SAT est le problème de décision qui consiste à déterminer si une formule propositionnelle φ donnée en entrée admet, ou non, un modèle.

Le plus souvent, on suppose que la formule φ en entrée est en forme clausale.

Exemple 4.40. Donnée : Une formule propositionnelle mise sous forme FNC :

$$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_1)\,.$$

Question : Est-ce que la formule φ admet au moins un modèle ?
Réponse : Pour cet exemple, la réponse est oui : la valuation $v(x_1) = 0$, $v(x_2) = 1$, $v(x_3) = 1$ satisfait la formule φ, c'est-à-dire $v \in \mathsf{mod}(\varphi)$.

Théorème 4.41. *Le problème SAT est décidable : cela signifie qu'il existe un algorithme capable de déterminer pour toute formule φ si elle admet un modèle.*

Démonstration. Nous donnons ici un algorithme naif, et nous étudierons dans cette section des solutions plus efficaces.

```
entrée: une formule F du calcul propositionnel
sortie: vrai si F admet un modèle, faux sinon
début:
    P = l'ensemble des variables propositionnelles de F
    V= l'ensemble des valuations de P dans {0,1}
    tant que V non vide
           choisir v dans V
           V = V - {v}
           calculer b= v(F)
            si b = 1 retourner vrai
     fintantque
     retourner faux
fin
```

Notez que la complexité de l'algorithme dépend de la taille n de l'ensemble `P`. En effet, il a alors 2^n valuations à tester, donc au pire cas, 2^n tours de boucles. □

La plupart des algorithmes de résolution de SAT ne se contentent pas de répondre par oui ou par non, il peuvent fournir aussi un modèle, ou même l'ensemble des modèles.

Exercice 4.42. L'algorithme donné ci-dessus suppose que l'on dispose d'un algorithme pour calculer `v(F)` : il est donné Définition 5.27 ; ainsi qu'un algorithme permettant de calculer l'ensemble des variables propositionnelles d'une formule.
Donnez ce dernier algorithme sous forme d'une fonction définie inductivement.

Le problème SAT joue un rôle important en théorie de la complexité. C'est un problème NP (Non déterministe temps Polynomial), c'est à dire qu'étant donné une éventuelle solution (pour SAT, une valuation), on sait déterminer en temps polynomial si cette solution satisfait le problème (pour SAT, si c'est un modèle). En plus d'être NP, le problème est NP-complet : il fait parti des problèmes les plus « durs » de la classe NP. Cela signifie que sa complexité majore la complexité de tous les autres problèmes NP.

Il existe des algorithmes plus performants que celui présenté, mais ces améliorations ne changent pas fondamentalement la difficulté du problème. On est devant la situation suivante. Étant donnée une formule φ, on se demande si φ admet un modèle ou non :

— une recherche exhaustive comme dans l'algorithme précédent peut demander jusqu'à 2^n vérifications si φ possède n variables propositionnelles. Cette démarche est dite **déterministe**, mais son temps de calcul est exponentiel.

— d'un autre côté, si φ est satisfiable, il suffit d'une vérification à faire, à savoir tester précisément la valuation qui satisfait φ. Cette vérification demande un simple calcul booléen, qui se fait en temps polynomial (essentiellement linéaire en fait). Le temps de calcul cesse donc d'être exponentiel, à condition de savoir quelle valuation tester. Celle-ci pourrait par exemple être donnée par un être omniscient auquel on ne ferait pas totalement confiance. Une telle démarche est dite **non déterministe**.

La question de la satisfiabilité de φ, ainsi que tous les problèmes qui se résolvent suivant la méthode que nous venons d'esquisser, sont dits NP (pour polynomial non déterministe). Par exemple, tester si la formule φ est une tautologie équivaut, par des calculs très simples en temps polynomial, à tester que sa négation n'est pas satisfaisable (par la Proposition 4.22).

Le problème SAT joue un rôle fondamental en théorie de la complexité, puisqu'on peut montrer que la découverte d'un algorithme déterministe en temps polynomial pour ce problème permettrait d'en déduire des algorithmes déterministes en temps polynomial pour tous les problèmes de type NP (théorème de Cook). On dit que SAT (et donc également le problème de la non-démontrabilité d'une proposition) est un problème NP-complet.

4.5.2 Modélisation - Réduction à SAT

Bien que le problème soit très difficile, nous verrons que nous disposons de logiciels très performants permettant de résoudre le problème de satisfaction d'une formule. Il est donc important pour tout informaticien de savoir profiter de ces outils. L'étape préalable est la suivante : étant donné un problème qui peut-être apparemment complètement dissocié de la logique, réduire la résolution de ce problème à la satisfaction d'une formule du calcul propositionnel. Il est bien sûr nécessaire que la réduction elle-même soit réalisable en un temps raisonnable (en temps polynomial).

Nous détaillons ici un exemple de modélisation :

Formation d'un jury

Un juge doit former un jury de 3 personnes. Personne dans le jury ne doit connaître un autre membre. Il y a 6 candidats et ils se connaissent comme suit :

candidat	connaissances
1	2,3
2	1,3,4
3	1,2,6
4	2,5
5	4,6
6	3,5

Modélisons ce problème : nous utilisons les variables propositionnelles $x_1, x_2, \ldots, x_6$ auxquelles nous donnons l'interprétation suivante : x_i signifie que le candidat i est choisi pour le jury.

Formalisons maintenant l'ensemble des contraintes du problème :

1. Le jury doit être formé de 3 membres : il existe $i, j, k \in \{1, 6\}$ tels que $i \neq j$, $j \neq k$ et $i \neq k$ et x_i, x_j et x_k sont choisis pour le jury, et pour tout $\ell \neq i, j, k$, x_ℓ n'est pas choisi.

$$\varphi := \bigvee_{\substack{i,j,k\in\{1,6\}\\ i\neq j,k\\ j\neq k}} ((x_i \wedge x_j \wedge x_k) \wedge \bigwedge_{\substack{\ell\in\{1,6\}\\ \ell\neq i,j,k}} \neg x_\ell)$$

   Notez que la formule $\bigvee_{\substack{i,j,k\in\{1,6\}\\ i\neq j,k\\ j\neq k}} (x_i \wedge x_j \wedge x_k)$ aurait été insuffisante car elle formalise simplement le fait que le jury est composé d'au moins 3 membres.

2. Personne dans le jury ne doit connaître un autre membre. On formalise une contrainte pour chaque membre :

   $\psi_1 := (x_1 \Rightarrow (\neg x_2 \wedge \neg x_3))$

   $\psi_2 := (x_2 \Rightarrow (\neg x_1 \wedge \neg x_3 \wedge \neg x_4))$

$\psi_3 := (x_3 \Rightarrow (\neg x_1 \wedge \neg x_2 \wedge \neg x_6))$

$\psi_4 := (x_4 \Rightarrow (\neg x_2 \wedge \neg x_5))$

$\psi_5 := (x_5 \Rightarrow (\neg x_4 \wedge \neg x_6))$

$\psi_6 := (x_6 \Rightarrow (\neg x_3 \wedge \neg x_5))$

Finalement, considérons la formule

$$S = \varphi \wedge \psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4 \wedge \psi_5 \wedge \psi_6.$$

Elle satisfait les propriétés suivantes :

1. S est satisfaisable, si et seulement si il est possible de constituer un jury avec ces candidats.

2. Tout modèle de S défini un jury possible. Prenons par exemple la valuation v suivante :

x_1	x_2	x_3	x_4	x_5	x_6
1	0	0	1	0	1

   On peut vérifier qu'elle est modèle de chacune des formules φ, $\psi_1, \ldots \psi_6$, et donc modèle de S. Il est donc possible de constituer un jury valide en choisissant les candidats 1, 4 et 6.

4.5.3 Algorithmes de résolution de SAT

De nombreux algorithmes ont été proposés pour résoudre SAT, nous en présentons quelques-uns, en nous focalisant sur le cas où la formule dont il faut décider la satisfaisabilité est déjà en forme normale conjonctive.

4.5.3.1 Algorithme de Quine

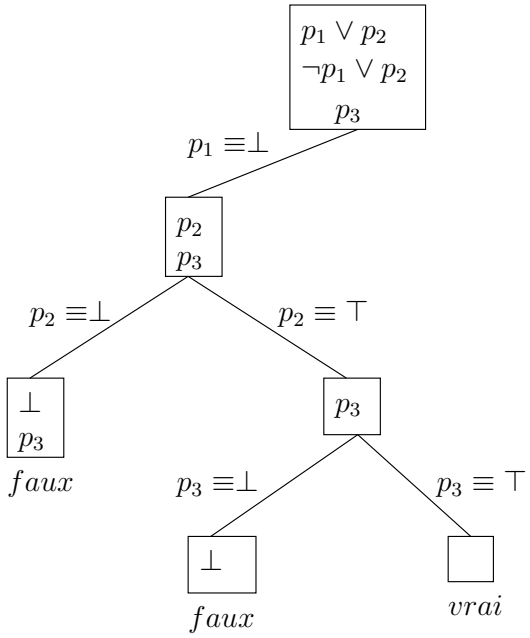
Nous présentons d'abord la méthode de Quine dans le cas restreint de formules mises au préalable sous forme normale conjonctive, assimilées donc à des ensembles de clauses. La discussion de la méthode de Quine nous aidera à comprendre de près le fonctionnement de l'algorithme DPLL présenté ensuite.

La méthode ne fait rien d'autre que parcourir l'arbre de toutes les solutions, selon un ordre définit initialement sur l'ensemble des proposition.

```
Algorithme de Quine
  input: un ensemble de clauses C sur PROP suppos'e ordonn'e
  output: vrai si C est satisfaisable ou faux sinon

  debut
      simplifier(C)
      si C = ∅ retourner vrai
      si C contient la clause ⊥ retourner faux
      choisir p ∈ PROP minimal
      si Quine(C[p ← ⊥]) = vrai
           retourner vrai
      retourner Quine(C[p ← ⊤])
  fin
```

Exemple 4.43. Considérons $\mathcal{C} = \{ p_1 \vee p_2, \neg p_1 \vee p_2, p_3 \}$. L'algorithme explore l'arbre sémantique selon un parcours en profondeur gauche comme suit :



L'ensemble de clauses est donc satisfaisable. Notez qu'on peut extraire un modèle de l'arbre construit en suivant une branche du sommet vers la feuille $vrai$: la valuation $p_1 \mapsto 0, p_2 \mapsto 1, p_3 \mapsto 1$ est un modèle. On peut aussi voir en suivant la première branche, que toute valuation satisfaisant $p_1 \mapsto 0$ et $p_2 \mapsto 0$ ne sera pas modèle. En particulier, $p_1 \mapsto 0, p_2 \mapsto 0, p_3 \mapsto 1$ et $p_1 \mapsto 0, p_2 \mapsto 0, p_3 \mapsto 0$ ne sont pas des modèles de $\mathcal{C}$.

Notez que l'algorithme ne fait aucun choix pertinent ni sur l'ordre dans lequel les variables sont choisies (en général elles sont ordonnées dans une liste, puis choisies selon cet ordre), ni sur l'ordre dans lequel on les teste (on essaie d'abord $\bot$ puis $\top$). Or, des constations simples sur l'ensemble de clause aurait permis d'accélérer le parcours. Par exemple, on voit clairement qu'il faut choisir $p_3 \mapsto \top$ si on veut un modèle de $\mathcal{C}$. L'algorithme suivant corrige ce défaut.

4.5.3.2 Algorithme de Davis-Putnam-Logemann-Loveland

L'algorithme DPLL est considéré, à ce jour, comme l'une des méthodes les plus efficaces parmi celles permettant de résoudre le problème SAT ; la plupart des outils de satisfaction de contraintes (« SAT solvers », en anglais) implémentent une variante de cet algorithme (les meilleurs complexités atteintes pour ces implémentation oscillent entre $O(1.5^n)$ et $O(1.3^n)$). Il peut être vu comme un raffinement de la méthode de Quine, apportant

— la réduction du branchement via la propagation des clauses unitaires (une clause est unitaire si elle ne contient qu'un seul littéral) ;
— l'utilisation d'heuristiques pour accélérer le parcours des solutions.

```
Algorithme DPLL
  input: un ensemble de clauses C
  output: vrai si C est satisfaisable ou faux sinon

  debut
      simplifier(C)
      si C = ∅ retourner vrai
      si C contient la clause ⊥ retourner faux
      si C contient la unitaire clause ℓ
              retourner DPLL(C[ℓ ← ⊤])
      litteral ℓ = choix(C)
      si DPLL(C[ℓ ← ⊥]) = vrai
```

```
            retourner vrai
        retourner DPLL(C[ℓ ← ⊤])
    fin
```

Il y a deux différences majeures avec l'algorithme de Quine :

1. on va choisir en priorité la substitution des clauses unitaires : par exemple, si $\mathcal{C}$ contient une clause p, alors on va explorer en priorité la branche $p \mapsto 1$ et on n'explorera jamais la branche $p \mapsto 0$, puisqu'on sait par avance qu'elle ne peut donner une solution
2. si il n'y a pas de clause unitaire, la branche à explorer va être choisie à l'aide de la fonction `choix`. Cette fonction est différentes selon les implémentations, elle consiste à utiliser des méthodes qui peuvent raisonnablement faire penser qu'elles accélèreront la procédure, sans qu'il n'y en ait en fait aucune garantie (heuristiques).

Heuristiques. Les heuristiques sont très importantes car elles permettent de réduire rapidement la taille de l'arbre de recherche.

— **Littéraux purs.** Si une variable apparaît seulement sous forme positive ou seulement sous forme négative alors ses littéraux sont dits purs. Soit ℓ un des littéraux purs apparaissant dans le plus de clauses : si $\ell = p$ alors choisir $\neg p$ et si $\ell = \neg p$, alors choisir p.
On peut raisonnablement penser que ce choix accélèrera la procédure car il supprimera toutes les clauses contenant p, diminuant ainsi de 1 la hauteur de l'arbre d'exploration.

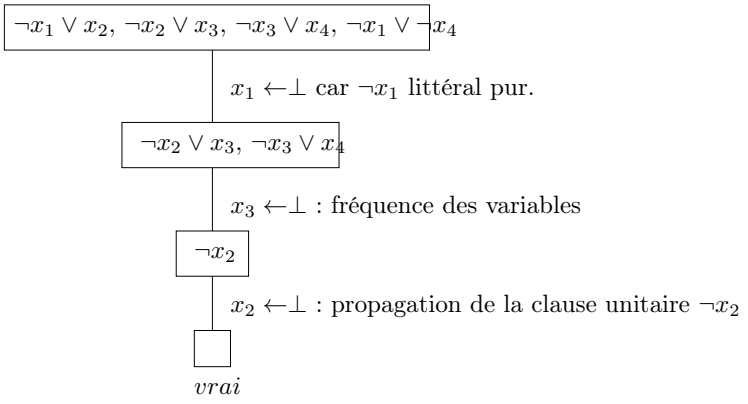
— **Fréquence des variables.** Choisir une variable p parmi celles qui apparaissent le plus, dans les clauses les plus courtes ;
Ceci à pour effet de réduire les clauses les plus courtes pour obtenir plus rapidement des clauses unitaires.

— **Fréquence des Littéraux.** Choisir un littéral parmi ceux apparaissant les plus, dans les clauses les plus courtes.
Ceci à pour effet de réduire les clauses les plus courtes pour obtenir plus rapidement des clauses unitaires.

Pour nos exemples, on considérera que la fonction `choix` cherche un littéral candidat en cherchant un littéral pur, puis si il ne le trouve pas, choisi un littéral parmi ceux permettant la meilleur réduction par les heuristiques fréquence des variables, et fréquences des littéraux.

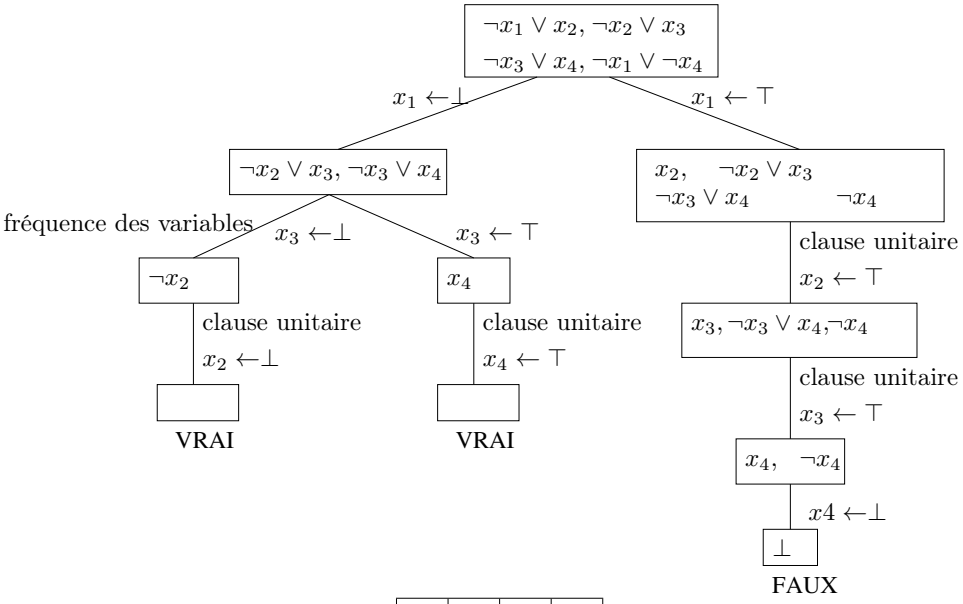
Notez qu'il existe beaucoup d'autres heuristiques et que toutes les implémentations n'utilisent pas forcément celles présentées ici.

Exemple 4.44. Considérons l'ensemble de clauses $\mathcal{C} = \{ \neg x_1 \vee x_2 \, , \, \neg x_2 \vee x_3 \, , \, \neg x_3 \vee x_4 \, , \, \neg x_1 \vee \neg x_4 \}$. L'algorithme DPLL construit l'arbre suivant :



Notez que puisque la valeur de x_4 n'a pas été choisie, cette branche nous donne en fait deux modèles de $\mathcal{C}$: v_0 tel que $v_0(x_1) = v_0(x_3) = v_0(x_2) = v_0(x_4) = 0$ et v_1 tel que $v_1(x_1) = v_1(x_3) = v_1(x_2) = 0$ et $v_1(x_4) = 1$.

Exemple 4.45. Nous pouvons modifier l'algorithme DPLL (et bien sur, l'algorithme de Quine aussi), afin qu'il trouve tous les modèles d'un ensemble de clauses. Avec le même $\mathcal{C}$ de l'exemple précédent, l'algorithme produira le parcours suivant :



Les modèles de cette formule sont :

x_1	x_3	x_2	x_4
0	0	0	0
0	0	0	1
0	1	0	1
0	1	1	1

On remarque dans cet exemple que :

— dans la branche de droite, la propagation des clauses unitaires permet de ne suivre qu'un seul chemin ;

— dans le sous-arbre de de gauche, l'heuristique, faisant choisir x_3 plutôt que x_2 ou x_4, réduit l'exploration car elle produit plus vite des clauses unitaires.

L'exécution est meilleure que si on avait fait une simple exploration de toutes les solutions (Algorithme de Quine).

4.5.4 Sous-classes de SAT

4.5.4.1 2-SAT

Le problème 2-SAT est celui de la satisfaisabilité d'une formule sous forme clausale dont les clauses sont d'ordre 2 (i.e., chaque clause est une disjonction d'au plus deux littéraux).

Exemple : $\varphi = (p_1 \vee p_2) \wedge (\neg p_1 \vee p_3) \wedge (\neg p_2 \vee p_1) \wedge (\neg p_2 \vee p_3) \wedge (\neg p_1 \vee \neg p_3)$ est sous forme clausale d'ordre deux.

Une clause d'ordre deux $\ell_1 \vee \ell_2$ est équivalente à $(\neg\ell_1 \Rightarrow \ell_2)$ et à $(\neg\ell_2 \Rightarrow \ell_1)$.

Contrairement à SAT, ce problème est solvable en temps polynomial. Pour le résoudre, on construit le graphe orienté $G = (V, E)$ dual (appelé graphe 2-SAT) selon les deux règles suivantes :

— l'ensemble V des sommets est l'ensemble des littéraux sur $Prop$: $S = \bigcup_{p \in Prop} \{\neg p, p\}$

— l'ensemble E des arcs est $E = \{(\ell_1, \ell_2) \mid \varphi$ contient une clause équivalente à $\ell_1 \Rightarrow \ell_2\}$.

Chaque clause $\ell_1 \vee \ell_2$ est donc associée à deux arcs $(\neg\ell_1, \ell_2)$ et $(\neg\ell_2, \ell_1)$.

Proposition 4.46. *La formule φ est insatisfaisable ssi il existe une variable p pour laquelle qu'il existe un cycle dans G passant par p et par $\neg p$.*

Notez que fait d'avoir un cycle passant par p et $\neg p$ implique que $\varphi \models (p \Rightarrow \neg p) \wedge (\neg p \Rightarrow p)$ ce qui est équivalent à $\bot$, et donc φ est insatisfaisable.

L'algorithme de Tarjan que nous ne détaillons pas ici permet déterminer si il existe un tel cycle en temps polynomial (si il n'en existe pas, il permet de déterminer un modèle de la formule). Le problème 2-SAT est donc solvable en temps polynomial.

Exemple 4.47. Soit $\varphi = (p_1 \vee p_2) \wedge (\neg p_1 \vee p_3) \wedge (\neg p_2 \vee p_1) \wedge (\neg p_2 \vee p_3) \wedge (\neg p_1 \vee \neg p_3)$. On a :

$$\begin{aligned}\varphi \quad \equiv \quad & (\neg p_1 \Rightarrow p_2) \wedge (\neg p_2 \Rightarrow p_1) \wedge (p_1 \Rightarrow p_3) \wedge (\neg p_3 \Rightarrow \neg p_1) \wedge (p_2 \Rightarrow p_1) \wedge \\ & (\neg p_1 \Rightarrow \neg p_2) \wedge (p_2 \Rightarrow p_3) \wedge (\neg p_3 \Rightarrow \neg p_2) \wedge (p_1 \Rightarrow \neg p_3) \wedge (p_3 \Rightarrow \neg p_1)\end{aligned}$$

ce qui nous donne le graphe donné Figure 4.3.

On remarque qu'il existe un cycle passant par p_1 et $\neg p_1$, donc la formule est insatisfaisable.

Exemple 4.48. $\varphi = (\neg p_1 \vee p_3) \wedge (\neg p_2 \vee p_1) \wedge (\neg p_2 \vee p_3) \wedge (\neg p_1 \vee \neg p_3)$. Le graphe est alors le suivant celui donné Figure 4.4.

Il n'y a pas de cycles passant par une variable et sa négation, donc la formule est satisfaisable. On peut remarquer que :

— $\varphi \models p_1 \Rightarrow \neg p_1$, donc on doit avoir $p_1 \mapsto 0$;
— $\varphi \models p_2 \Rightarrow \neg p_2$, donc on doit avoir $p_2 \mapsto 0$;
— $\varphi \models p_3 \Rightarrow \neg p_3$, donc on doit avoir $p_3 \mapsto 0$

Finalement, il n'y a qu'un seul modèle : $v(p_1) = v(p_2) = v(p_3) = 0$.

4.5.4.2 3-SAT

Le problème 3-SAT est lui aussi NP-complet (c'est à dire qu'il est aussi difficile que le problème SAT). Pour le démontrer, il suffit de prouver que le problème SAT est polynomialement réductible à 3-SAT : cela signifie que répondre à la question SAT revient à répondre à 3-SAT et que le temps nécessaire à transformer SAT en 3-SAT est polynomial.

Preuve : On remarque que toute clause $\varphi_n = \ell_1 \vee \ell_2 \vee \ldots \vee \ell_n$ avec $n \geq 3$ peut se mettre sous la forme :

$\psi_n = (\ell_1 \vee \ell_2 \vee q_1) \wedge (\ell_3 \vee \neg q_1 \vee q_2) \wedge \ldots \wedge (\ell_{n-2} \vee \neg q_{n-4} \vee q_{n-3}) \wedge (\ell_{n-1} \vee \ell_n \vee \neg q_{n-3})$ où les q_i sont de nouveaux symboles propositionnels.

Donc le problème SAT est polynomialement réductible à 3-SAT. On conclut que 3-SAT est NP-complet.

Tous les problèmes n-SAT avec n supérieur ou égal à 3 sont également des problèmes NP-complets.

4.5.4.3 Horn-SAT

Une *clause de Horn* est une clause comportant au plus un littéral positif, donc de l'une des trois formes suivantes :

$$\neg p_1 \vee \ldots \vee \neg p_n \vee p\,; \qquad \neg p_1 \vee \ldots \vee \neg p_n\,; \qquad p\,; \qquad \bot.$$

Le problème Horn-SAT consiste à déterminer si un ensemble de clauses de Horn est satisfaisable. Ce problème est solvable en temps polynomial.

Dans l'algorithme suivant, on appelle *fait* une clause réduite à une variable propositionnelle.

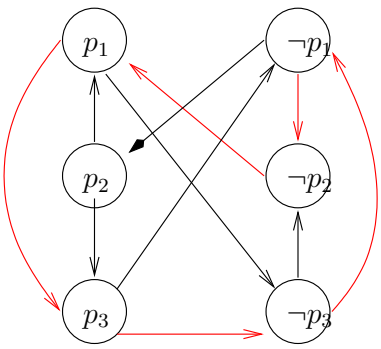


FIGURE 4.3 – Le graphe de l'exemple 4.47

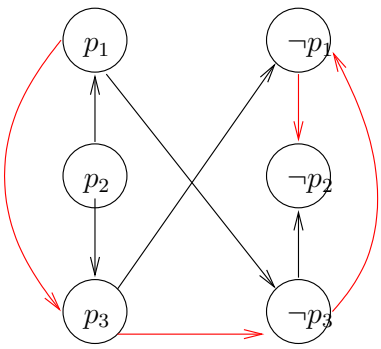


FIGURE 4.4 – Le graphe de l'exemple 4.48

```
Algorithme HornSat
  input: un ensemble de clauses de Horn C
  output: vrai si C est satisfaisable ou faux sinon

  debut
      tant que C contient des faits
           choisir un fait p
           C = C[p ← ⊤]
      fintq
      si C contient la clause ⊥ retourner faux
      sinon retourner vrai
  fin
```

La validité de l'algorithme repose sur les points suivants :
— si p est un fait, alors $\mathsf{mod}(\mathcal{C}) = \mathsf{mod}(\mathcal{C}[p \leftarrow \top])$ donc à la fin de la boucle, on n'a pas changé l'ensemble des modèles.
— si $\mathcal{C}$ ne contient ni faits, ni la clause $\bot$, il est satisfaisable : il suffit de mettre à 0 toutes les variables apparaissant dans les clauses.

Exemple 4.49. Soit $\mathcal{C} = \{\neg x_1 \vee \neg x_2 \vee x_3, \neg x_2 \vee x_4, x_4 \vee \neg x_5, x_5\}$.
Appliquons l'algorithme :
— $\mathcal{C}[x_5 \leftarrow \top] = \{\neg x_1 \vee \neg x_2 \vee x_3, \neg x_2 \vee x_4, x_4\}$
— $\mathcal{C}[x_5 \leftarrow \top, x_4 \leftarrow \top] = \{\neg x_1 \vee \neg x_2 \vee x_3\}$
Donc $\mathcal{C}$ est satisfaisable et admet $x_1 \mapsto 0, x_2 \mapsto 0, x_3 \mapsto 0$ pour modèle.

Exemple 4.50. Soit $\mathcal{C} = \{\neg x_1 \vee \neg x_2 \vee x_3, \neg x_4 \vee \neg x_5, x_4 \vee \neg x_5, x_5\}$.
Appliquons l'algorithme :
— $\mathcal{C}[x_5 \leftarrow \top] = \{\neg x_1 \vee \neg x_2 \vee x_3, \neg x_4, x_4\}$
— $\mathcal{C}[x_5 \leftarrow \top, x_4 \leftarrow \top] = \{\neg x_1 \vee \neg x_2 \vee x_3, \bot\}$
Donc $\mathcal{C}$ est insatisfaisable.

4.5.5 Les SAT-solvers

Puisque le problème SAT est présent dans de nombreux domaines de l'informatique, le développement de SAT-solvers efficaces est un des défis majeur de l'informatique. De nouveaux solvers sont constamment développés pour répondre à des besoins spécifiques. Des concours mondiaux de SAT-solvers sont d'ailleurs organisés chaque année (voir http://www.satcompetition.org).

On peut citer parmi ces nombreux solvers : zChaff datant de 2001 qui utilise l'algorithme de Chaff qui est une amélioration de DP, Siege en 2003, MiniSAT en 2005 logiciel open-source, picoSAT dont les méthodes s'inspirent de l'algorithme de Chaff, SARzilla en 2009 qui a gagné de nombreux prix et Glucose développé au LaBRI à Bordeaux

Enfin remarquons que les solvers modernes exploitent les techniques les plus récentes de l'informatique, comme l'apprentissage, la programmation parallèle exploitant l'architecture multicoeur des processeurs récents, etc.