TP3 : Méthode de résolution

Outils Logiques (OL3)

28 novembre 2006

Introduction

La *méthode de résolution* est une technique qui permet de montrer systématiquement et facilement qu'une formule est insatisfiable. Ici nous allons l'utiliser sur des formules en CNF.

L'idée est la suivante. On considère les deux formules suivantes

$$\phi = A \wedge (x \vee C) \wedge (\neg x \vee C') \qquad \phi' = A \wedge (x \vee C) \wedge (\neg x \vee C') \wedge (C \vee C')$$

Il est clair que ϕ' implique ϕ, étant donné qu'elle contient plus de clauses. Mais on peut aussi montrer facilement (par disjonction de cas sur x) que ϕ implique ϕ'. Par conséquent,

$$\phi \equiv \phi'$$

L'idée de la méthode de résolution est d'utiliser l'équivalence ci-dessus pour ajouter les clauses de la forme $C \vee C'$ et tenter de faire apparaitre la clause vide.

Donnons quelques définitions :

Définition 1. *Étant donné deux clauses C et C', on dit que lq clause $C \vee C'$ est un* résolvant *de $x \vee C$ et $\neg x \vee C'$, ce qu'on note $C \vee C' \in Res(x \vee C, \neg x \vee C')$.*

La méthode de résolution consiste à ajouter itérativement à une formule les résolvants de toutes ses clauses. Si la clause vide apparait, cela signifie que la formule est insatisfiable. Dans le cas contraire, elle est satisfiable.

Comme pour la fonction de Davis-Putnam, on considère que les formules en CNF sont des ensembles de clauses. Le pseudo-algorithme présenté Figure 1 implémente la méthode de résolution.

On peut montrer que la méthode de résolution est complète pour la réfutation :

Théorème 1. *L'algorithme* **Resolution**(A) *renvoie "A n'est pas satisfiable" si et seulement si A n'est pas satisfiable.*

Le but de ce TP est de programmer la méthode de résolution. On s'efforcera de réutiliser les structures de données du TP 1.

Resolution(A)
Répéter
  $A' \leftarrow A \cup (\bigcup_{C,C' \in A} Res(C, C'))$
Si A' contient la clause vide,
  retourner "A n'est pas satisfiable"
Sinon, si $A = A'$
  retourner "A est satisfiable"
Sinon
  $A \leftarrow A'$

FIG. 1 – Algorithme pour la méthode de résolution

1 Résultats préliminaires

On commencera par traiter la question suivante sur papier.

Question 1. *Appliquer la méthode de résolution sur les formules suivantes :*

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

2 Calcul du résolvant

Le résolvant de deux clauses peut être une clause, plusieurs clauses, ou l'ensemble vide. Par exemple,

$$\begin{aligned} Res(x \vee y,\ y \vee z) &= \emptyset \\ Res(x \vee y, \neg x \vee \neg z) &= \{y \vee \neg z\} \\ Res(x \vee y, \neg x \vee \neg y) &= \{y \vee \neg y, x \vee \neg x\} \end{aligned}$$

Question 2. *Ajouter à la classe clause une méthode* `resolvant` *qui prend en argument une clause et telle que* `c1.resolvant(c2)` *renvoie la liste des clauses correspondant au résolvant de* c_1 *et* c_2*.*

Question 3. *Ajouter à la classe conjonction une méthode* `ajouter_resolvant`*. Le résultat de* `c.ajouter_resolvant` *doit être* `c` *dans laquelle on a ajouté le résolvant de toutes les clauses de* `c` *prises 2 à 2.*

À l'aide de ces deux questions, il est possible d'écrire un algorithme qui termine si la formule est non satisfiable (et qui boucle si ce n'est pas le cas). Pour cela, on commence par tester si la formule contient la clause vide ; si c'est le cas on renvoie `true`. Sinon, on appelle `ajouter_resultant`, et on se rapppelle récursivement.

Question 4. *Écrire la méthode* `non_satisfiable` *décrite ci-dessus. La tester sur les formules*

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

et

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

Le reste du TP est en bonus. Les 2 parties sont indépendantes.

3 Détection du point fixe

L'algorithme de la question précédente boucle sur toutes les formules qui sont satisfiables ; en effet, il ne verifie pas que les résolvants ajoutés ne sont pas déjà dans la formule. Cette section a pour but de corriger ce problème.

Question 5. *Ajouter à la classe conjonction une méthode* `contient_clause` *telle que* `c.contient_clause(cl)` *renvoie* true *si et seulement si la clause* `cl` *est déjà dans* `c`*.*

Question 6. *Modifier la méthode* `ajouter_resolvant` *pour qu'elle n'ajoute que les résolvants qui ne sont pas encore présents, et qu'elle renvoie un booléen indiquant si un résolvant a été ajouté. Modifier* `non_satisfiable` *pour qu'elle prenne en compte ces modification et termine dans tous les cas.*

On peut proposer une autre méthode plus efficace en nombre de calculs. Elle consiste à noter les résolvants deja calculés.

Question 7 (Long)**.** *Ajouter à la classe clause un champ* `num` *permettant de numéroter les clauses ;* `c1.num` *et* `c2.num` *doivent être égaux si et seulement si* `c1` *et* `c2` *sont égales. On pourra utiliser un compteur statique interne.*

Ajouter dans la classe conjonction un champ contenant la liste des couples (`c1`, `c2`) *pour lesquels les résolvants de* `c1` *et* `c2` *ont déja ete calculés. Modifier la méthode* `ajouter_resolvant` *pour qu'elle exploite et mette à jour cette information.*

L'algorithme ainsi modifié doit toujours terminer (et, bien sûr, renvoyer une réponse correcte).

Question 8. *Tester le ou les algorithme(s) écrits dans cette partie sur les formules suivantes, et comparer aux solutions "papier".*

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

4 Clauses de Horn

On rappelle (cf. planches de TD) que les clauses de Horn sont les clauses de la forme $\neg x_1 \vee \cdots \vee \neg x_n$ ou $\neg x_1 \vee \cdots \vee \neg x_n \vee x_{n+1}$, c'est à dire qu'elles contiennent au plus un littéral positif.

Donnons quelques definitions :

Définition 2. *On appelle* unitaire *une clause qui est réduite à un seul littéral (positif ou négatif). On note URes la restriction de Res à deux clauses dont au moins une est unitaire : $URes(l, C) = URes(C, l) = res(l, C)$ et URes n'est pas défini si une des clauses n'est pas réduite à un littéral. On appelle* **UResolution** *l'algorithme obtenu en utilisant URes au lieu de Res dans* **Resolution**.

L'algorithme **UResolution** permet de réfuter les clauses de Horn.

Théorème 2. *Étant donné une clause de Horn A,* **UResolution**(A) *renvoie "A n'est pas satisfiable" si et seulement si A n'est pas satisfiable.*

Il est très intéressant d'utiliser **UResolution** sur des clauses de Horn car la taille des clauses générées pendant la resolution "n'explose" jamais.

Question 9. *Implémenter aussi efficacement que possible (par exemple en gardant une liste des clauses unitaires) l'algorithme* **UResolution**. *Testez-le sur des clauses de Horn générées aléatoirement.*