LOGIQUE COMPUTATIONNELLE & PROLOG – 2

CALCUL PROPOSITIONNEL II

2.1

Algorithmes pour le calcul propositionnel

Le problème de la satisfiabilité d'une fbf A à partir des hypothèses $C_1, C_2, \ldots, C_n$ avec n fini, se décline de plusieurs façons :

- La fbf A est une conséquence valide des fbf $(C_1, C_2, \ldots, C_n)$? Cette proposition s'écrit $C_1 \wedge C_2 \wedge \ldots \wedge C_n \models A$.
- La fbf $C_1 \wedge C_2 \wedge \ldots \wedge C_n \rightarrow A$ est une formule valide ? Cette proposition s'écrit $\models (C_1 \wedge C_2 \wedge \ldots \wedge C_n \rightarrow F)$
- $C_1 \wedge C_2 \wedge \ldots \wedge C_n \wedge \neg A$ est une fbf insatisfiable ?

L'équivalence entre les trois propositions ci-dessus découle du

THÉORÈME 2.0.1 *(de réfutation)* $C_1 \wedge C_2 \wedge \ldots \wedge C_n \models C$ *si et seulement si est un ensemble inconsistant.*

Ce théorème fournit une méthode qui permet, étant donnée une base de connaissances, d'obtenir de nouvelles connaissances, qui, néanmoins, sont implicitement contenues dans la base de connaissances. De plus il est le fondement théorique du langage de programmation logique Prolog.

Cette pluralité d'approches du problème a donné naissance à une multitude d'algorithmes concernant la preuve de la validité d'une fbf.

Dans la suite nous présentons l'algorithme de résolution qui est facile à l'emploi et qui permet d'obtenir toutes les conclusions à partir d'un ensemble de prémisses et aussi de

tester la déduction proposée par un ensemble des prémisses. De plus il est utilisé par Prolog pour obtenir des déductions logiques.

2.2 Algorithme de résolution

Considérons une fnc $C = C_1 \wedge C_2 \wedge \ldots \wedge C_n$, ou C_i clause. Dans la suite on notera une fnc sous forme ensembliste $C = \{C_1, C_2, \ldots, C_n\}$, où la virgule ici est mise à la place du connecteur $\wedge$.

Si la fnc C est insatisfiable, alors il est toujours possible d'obtenir, à partir de C, une contradiction qui, sous forme clausale, équivaut à la clause vide. Ainsi si on veut automatiser la procédure de la détermination de l'insatisfiabilité, on peut envisager une méthode qui produirait des conséquences à partir de la fcn sous test et qui s'arrêterait dès que la clause vide serait engendrée.

Il est évident que cette technique peut aussi s'appliquer quand on veut établir qu'une formule est une conséquence logique d'un ensemble de formules, i.e. si on veut établir que $C \models A$. En utilisant le théorème de réfutation on sait que $C \models A$ ssi $C \cup \{\neg A\}$ est insatisfiable.

Nous avons jusqu'ici présenter la conséquence logique de manière sémantique. Mais en vue de pouvoir automatiser le processus de la conséquence logique il faut pouvoir utiliser des méthodes syntaxiques, c-à-d. des méthodes qui ne necessitent pas la connaissance des valeurs de vérité des propositions que l'on manipule. Pour ce faire nous devons utiliser des règles d'inférence. Herbrand a montré qu'en utilisant la règle de résolution, on peut obtenir le modus ponens et le modus tollens. Donc nous pouvons utiliser cette règle pour obtenir la déduction logique [1] d'une fbf à partir d'une fnc. La règle de résolution est la suivante

$$\frac{p \vee q, \neg p \vee r}{q \vee r}$$

De manière plus générale on représente une clause A sous forme ensembliste. Par exemple si $A = p \vee q$, on note $A = \{p, q\}$. On dira que deux clauses A et B sont *discordantes* si nous avons $p \in A$ et $\neg p \in B$. Dans ce cas, la règle de résolution stipule que nous pouvons remplacer ces deux clauses par leur *résolvante* $R(A,B) = \{A - \{p\}\} \cup \{B - \{\neg p\}\}$. Nous avons donc

$$\text{Règle de résolution } \frac{A, B \text{ clauses discordantes en } p}{R(A,B)}$$

Robinson en utilisant cette règle a établi un algorithme pour la deduction logique qui s'appelle *algorithme de résolution par réfutation* qui est donné ci-après :

(1) Considérons une fnc $C = C_1 \wedge C_2 \wedge \ldots \wedge C_n$ où C_i clauses disjonctives et soit la fbf $C \rightarrow A$ où A clause disjonctive.
Exemple : Soit la fbf $p \wedge (p \rightarrow q) \wedge (q \rightarrow r) \rightarrow r$ avec $C = p \wedge (p \rightarrow q) \wedge (q \rightarrow r)$ et $A = r$. On transforme C en fnc et on obtient $C = p \wedge (\neg p \vee q) \wedge (\neg q \vee r)$

(2) On écrit C sous forme ensembliste $C = \{C_1, \ldots C_n\}$.
Exemple : $C = \{p, \neg p \vee q, \neg q \vee r\}$

1. qui est l'équivalent syntaxique de la conséquence logique en sémantique

(3) On rajoute à cette ensemble la negation de la conclusion A et on obtient l'ensemble $S = \{C_1, \ldots C_n, \neg A\}$

Exemple : $S = \{p, \neg p \vee q, \neg q \vee r, \neg r\}$

(4) On cherche dans S deux clauses discordantes. S'il n'y en a pas, A n'est pas une déduction logique de C et l'algorithme s'arrête.

Sinon, soit C_i et C_j deux clauses discordantes en, par exemple, p. On forme leur résolvante $R(C_i, C_j)$.

Exemple : $C_i = p$ et $C_j = \neg p \vee q$. La résolvante est $R(C_i, C_j) = \{q\}$.

(5) On modifie C comme suit : $C = \left(C - \left\{C_i, C_j\right\}\right) \cup R(C_i, C_j)$

Si C se réduit à la clause vide, alors la déduction logique est valide.

Sinon on pose $S = C$ et on retourne à 4.

Exemple : $C = \{q, \neg q \vee r, \neg r\}$. On continue $S = \{q, \neg q \vee r, \neg r\}, C_i = q, C_j = \neg q \vee r, R(C_i, C_j) = \{r\} \Longrightarrow C = \{r, \neg r\}, S = \{r, \neg r\}, C_i = r, C_j = \neg r, R(C_i, C_j) = \{\square\} \Rightarrow C = \{\square\}$

où $\square$ représente la clause vide dont sa valeur de vérité est Faux [2]. Par conséquent la fbf $p \wedge (p \rightarrow q) \wedge (q \rightarrow r) \rightarrow r$ est valide.

Nous pouvons aussi utiliser la règle de résolution pour examiner si une fbf A est valide. Pour ce faire on nie la fbf A et on cherche à montrer que cette négation conduit à une absurdité.

Exemple : Soit la fbf $A = (p \rightarrow \neg p) \rightarrow \neg p$. On nie la proposition et on a $\neg A = \neg [(p \rightarrow \neg p) \rightarrow \neg p]$. On traduit cette formule en fnc $\neg [(p \rightarrow \neg p) \rightarrow \neg p] \leftrightarrow \neg [\neg (\neg p \vee \neg p) \vee \neg p]$. Donc $C = \{\neg p \vee \neg p, p\} \leftrightarrow C\ \{\neg p, p\} \leftrightarrow C = \{\square\}$ et donc la formule est valide dans la mesure où sa négation est absurde.

Nous pouvons encore utiliser la règle de résolution pour calculer la conséquence d'une fbf. Dans ce cas il faut appliquer l'algorithme sans nier la fbf et récupérer la conclusion.

Exemple : Reprenons l'exemple précédent en considérant uniquement les prémisses : $A = (p \rightarrow \neg p)$. On applique l'algorithme à A, ce qui donne $C = \{\neg p \vee \neg p\} \Rightarrow C = \{\neg p\}$c-à-d. cette formule est équivalente à $\neg p$.

2.3

Clauses de Horn

Les clauses de Horn sous leur forme générale sont fbf composées de l'union de la négation d'une fnc et d'une proposition atomique positive, c-à-d. elles ont la forme

$$\neg (C_1 \wedge C_2 \wedge \ldots \wedge C_n) \vee p$$

ou, encore

$$C_1 \wedge C_2 \wedge \ldots \wedge C_n \rightarrow p$$

En plus de cette forme générale, qui s'appelle règle, les clauses de Horn peuvent prendre deux formes simplifiées, à savoir

donnée : $\rightarrow p$

2. N.B. La valeur de vérité d'une clause vide $\square$ est Faux. Par contre la valeur de vérité d'un ensemble de clauses vide est Vrai. Car dans ce dernier cas, l'absence de clause signifie que cet ensemble est toujours vérifiée.

et

question : $C_1 \wedge C_2 \wedge \ldots \wedge C_n \rightarrow$

Les clauses de Horn sont particulièrement utiles en Prolog, car elles permettent un application facile de la règle de résolution.

2.4

Exercices de Logique Computationnelle

EXERCICE 2.1 *Examiner si la proposition*

$$[(p \vee q) \wedge (\neg r \rightarrow \neg q)] \rightarrow (p \vee r \vee \neg q)$$

est valide.

EXERCICE 2.2 *Soit le raisonnement d'Aristote*

– *S'il faut philosopher, alors il faut philosopher.*
– *S'il ne faut pas philosopher, alors il faut encore philosopher.*
– *Conclusion : il faut philosopher.*

Traduiser ce raisonnement et prouver-le.

EXERCICE 2.3 *Bias, un de sages grecs de l'antiquité, tenait le raisonnement suivant contre le mariage :*

(1) Si vous vous marier, votre femme sera belle ou laide

(2) Si elle belle, vous serez en proie à la jalousie

(3) Si elle est laide, vous ne la supporterez pas

Donc, il ne faut pas vous marier.

Montrer que le raisonnement n'est pas valide et trouvez les propositions à rajouter aux prémisses pour qu'il devienne valide.

2.5

Introduction à Prolog

Prolog = PROgrammation en LOGique est un langage de programmation apparu pour la première fois en 1972. Son concepteur est A. Colmerauer de l'université de Marseille. Ensuite il a été popularisé par R. Kowalski sur des ordinateurs PDP-11. Kowalski étant à l'université d'Endibourg, cette version de Prolog est connue jusqu'à nos jours comme le Prolog d'Endibourg.

Il s'agit d'un langage *déclaratif*. Il diffère donc des langages procéduraux ou impératifs qui sont Fortran, Pascal, C, Ada, etc. En effet pour ces derniers un algorithme doit être

traduit en actions que l'ordinateur doit effectué pour exécuter l'algorithme. Ainsi le programmeur, pour écrire son programme, doit se mettre à la place de l'ordinateur et agir comme lui. Par contre avec un lagage déclaratif, comme Prolog, on présente ce qui doit être calculé, non pas comment l'ordinateur doit faire les calculs. Le travail donc, que doit fournir l'informaticien est minime. Par conséquent ces programmes sont faciles à construire, faciles à comprendre, faciles à tester, faciles à maintenir et faciles à adapter pour satisfaire à d'autres buts.

La programmation procédurale, la première qui est apparue, est la conséquence du fait que les ordinateurs ont une architecture machine de type von Neumann. Les ordinateurs à architecture von Neumann ont un procédé de traitement de l'information fondé sur le cycle « fetch-execute », à savoir une boucle qui passe successivement par les trois états suivants :

- activer l'instruction prochaine ;
- décoder l'instruction ;
- exécuter l'instruction.

Comme un langage de programmation permet à l'homme d'expliquer à la machine ce qu'il souhaiterait qu'elle fasse, nous pouvons, en programmant, faire l'une de deux choses :

- soit établir la structure de l'ordinateur. Ce que nous faisons avec tous les langages procéduraux. Dans ce cas le programmeur doit obtenir une correspondance entre le modèle de l'ordinateur et le modèle du problème. Ce travail n'est pas intrinsèque au langage de programmation. Il s'agit d'une tâche qui, pour le même problème, doit se faire quasiment de la même façon pour tout langage procédural, à l'extérieur du travail de la programmation – et même avant celui-ci. Ainsi avec un langage procédural il est difficile d'écrire un programme et encore plus difficile de le maintenir.
- soit établir la structure du problème. Pour ce faire nous devons avoir un modèle du monde ou, de façon plus modeste, de la partie de l'univers (du micromonde, comme on disait jadis) dans lequel ce situe notre problème. C'est l'approche utilisée les langages déclaratifs. Par exemple, pour le `Prolog` tout problème peut se ramener à un calcul de la valeur de vérité d'une suite des prédicats.

De ce qui précède découle qu'un avantage des langages déclaratifs est le fait qu'ils obligent le programmeur d'établir une démarche algorithmique indépendante des caractéristiques technologiques et de l'architecture matérielle de l'ordinateur sur lequel le programme s'exécutera. `Prolog` en particulier utilise comme élément de base de la programmation la notion du prédicat. Sa définition inclut les arguments d'entrée et de sortie et laisse donc au programmeur seulement le soin de déterminer le résultat souhaité – la sortie – et les paramètres – les entrées – qui permettent d'obtenir ce résultat, sans qu'il soit préoccupé d'écrire des instructions détaillées concernant la démarche pour arriver au résultat.

Les caractéristiques essentielles de Prolog sont :

- l'utilisation des faits du domaine de connaissance comme des données ;
- l'utilisation des règles pour exprimer les relations entre les faits ;
- l'utilisation de la déductions pour répondre à des questions concernant le domaine de connaissance particulier.

Sa différence par rapport aux langages procéduraux ce concentre essentiellement à l'absence de contrôle du flux des instructions à exécuter. Ainsi il n'y a pas dans Prolog les différents blocs conditionnels qu'on trouve dans les autres langages. Par exemple il n'existe pas le bloc `if - then - else`, ni des boucles `while` ou `repeat`. De plus il n'y a pas des variables

globales ni des états qui se modifient globalement. De plus, Prolog n'est pas un langage typé.

Prolog est un sous-ensemble de la logique du premier ordre. Prolog considère un programme comme une théorie et pour faire des calculs utilise le modus ponens qui est un cas particulier de la règle de résolution.

Exemple de fichier source.

```
porteParapluie :- ilPleut.
ilPleut.

mortel(X) :- humain(X).
existeHumain :- humain(X).
humain(socrate).
humain(aristote).

animal(X) :- lion(X).
```

Exemple des questions :

```
?- ilPleut.
yes

?- porteParapluie.
yes

?- animal(socrate).
no

?- mortel(X).
X = socrate;
X = aristote;
no

?- existeHumain.
yes
```

2.6
Clauses de Horn en Prolog

Prolog fonctionne avec des clauses de Horn sous leurs trois formes différentes, à savoir des faits, des prédicats et des questions.

- ***Faits :*** Ils sont de la forme
  `humain(socrate).`
  On affirme le fait que Socrate est un être humain.
  On distingue
  - Entiers. Ex. `6, factoriel(6).`
  - Réels. Ex. `3.1415, pi(3.1415).`
  - Constantes. Ce sont des atomes. Ex. `socrate, toto, ... .` Notons que les nombres entiers ou flottants sont considérés comme des constantes.

- Variables. Ex. `X, Arbre`. Il y a une variable particulière, la variable anonyme notée "_", c'est-à-dire elle existe mais elle ne peut pas être utilisée par le programme.
  Bien sûr à proprement parler, une variable ne peut pas être considérée comme un fait. Mais en Prolog nous pouvons envisager d'avoir des données incomplètes. Par exemple une base de données dans laquelle pour un article donné tous les champs ne sont pas remplis. On peut palier à ce manque des données par une variable.
  N.B. Prolog n'est pas un langage typé. On doit seulement distinguer entre les constantes et les variables. Les noms des constantes commencent par une lettre minuscule. Ex. `socrate`. Les noms des variables commencent par une lettre majuscule. Ex. `Arbre`.
- Données structurées composées des foncteurs et des arguments. Par exemple `date(10, novembre, 2007)`.
- Prédicats. Ils ont un nom, par exemple `humain(X)` et une arité positive. La notation utilisée est `nomPredicat/arité`. Ex. `humain/1`.
  En particulier on peut considérer des prédicats qui contiennent comme arguments des foncteurs.
  Exemple :
  - Tester si la valeur de sinus d'un angle est positive : `plusGrand(sin(3.1415/6), 0)`.
  - Les dates anniversaires : `anniversaire(toto, date(10, novembre, 2007))`.
    Ici `date(10, novembre, 2007)` est un foncteur.
  - Description des branches gauche et droite d'un arbre binaire.
    `brancheGauche(arbre(L,R), L).`
    `brancheDroite(arbre(L,R),R).`
    Ici `brancheGauche` et `brancheDroite` sont des prédicats, `arbre` est un foncteur.

- ***Règles :*** Il s'agit des règles strictes de Horn composées par des prédicats.
  `porteParapluie :- ilPleut.`
  On lira ce morceau de programme "je porte un parapluie SI il pleut". Ici la notation " `:-`" se lit "SI".
  L'écriture d'un programme en Prolog est en fait l'écriture des règles.
  On comprend ainsi qu'un programme en Prolog est déclaratif et non pas procédural : les règles présentent (déclarent) le comportement du programme, ce qu'il doit faire le programme et on ne se préoccupe pas de la manière (procédure) avec laquelle ce qu'il doit être fait, se fera.
- ***Questions :*** Elles servent à savoir si un fait est vérifié, c'est-à-dire s'il est une implication logique du programme, considéré comme un ensemble des formules logiques.
  Ex. `?-mortel(socrate)..`
  On peut aussi avoir des questions existentielles : On cherche à savoir s'il existe un élément qui satisfait à la propriété exprimée par le prédicat de la question.
  Ex. `?-mortel(X)..`

2.7

Fonctionnement (simplifié) de Prolog

Il est fondé sur la règle de résolution.

Soit le programme E :

```
p.
q.
```

Le programme s'écrit sous forme ensembliste `E = {p, q}`. Soit maintenant la question.

Q : `?- p ∧ q`

Pour répondre à la question,Prolog applique la résolution par réfutation. Donc dans E est placée aussi la négation de la question :

`E = {p, q, ¬(p∧q)}` $= \{p, q, \neg p \vee \neg q\}$

Comme E contient maintenant la clause absurde $p \wedge \neg p$, on conclut que $E \vdash Q$.

2.8

Exercices de Prolog

EXERCICE 2.4 *Écrire en Prolog le code du §1.5. Préciser ce que chaque ligne du code représente en termes de clauses de Horn et examiner les réponses aux questions.*

EXERCICE 2.5 *Écrire en Prolog et tester la fbf* $p \wedge (p \rightarrow q) \wedge (q \rightarrow r) \rightarrow r$.

EXERCICE 2.6 *Écrire en Prolog et tester le fbf de l'exercice 2 de la logique computationnelle.*

EXERCICE 2.7 *Considérons le graphe de la figure 2.1 :*

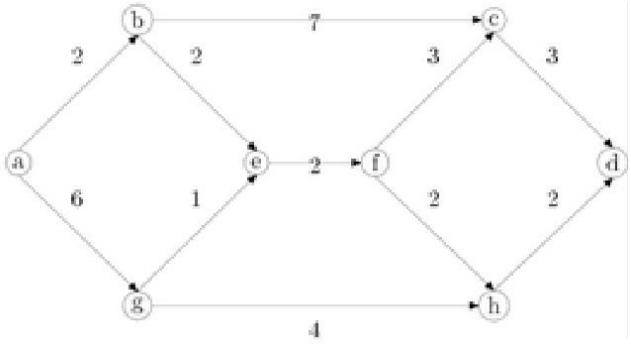

FIGURE 2.1 – Graphe pour un réseau

Si nous utilisons la logique des propositions nous pouvons représenter, en `Prolog` *ce graphe par la base de données suivante :*

```
gr(a,b,2).
gr(a,g,6).
gr(b,e,2).
gr(b,c,7).
gr(g,e,1).
gr(g,h,4).
gr(e,f,2).
gr(f,c,3).
gr(f,h,2).
gr(c,d,3).
gr(h,d,2).
```

(1) Écrire en Prolog un programme qui établit s'il y a un chemin entre deux sommets.

(2) Écrire en Prolog un programme qui calcule la distance entre deux sommets du graphe.