Connaissances et Raisonnement - TD3

17 janvier 2021

Pour ce TD, nous vous demanderons de rendre l'exercice 6.

1 Rappels Prolog

Prolog travaille avec les termes de la logique des prédicats du premier ordre.

- Signature $\mathcal{F} = (\mathbb{F}_0, \mathbb{F}_1, \mathbb{F}_2, ...)$: symboles de fonctions, suite finie d'ensembles finis de symbole, ensembles disjoints deux à deux
- Ensemble V de variables
- $T_{\mathcal{F}(V)}$: ensemble des termes avec variables défini inductivement
- $\mathcal{R} = (\mathbb{R}_0, \mathbb{R}_1, \mathbb{R}_2, ...)$: symboles de relations, suite finie d'ensembles finis de symbole, ensembles disjoints deux à deux
- Littéral : $r_i(t_1, \dots, t_i)$ ou $¬r_i(t_1, ..., t_i) $ tel que $r_i \in \mathbb{R}_i$, $t_1 \dots t_i \in T_{\mathcal{F}(V)}$
- Les littéraux sont des formules du langage de la logique des prédicats
- Inductivement : si A et B sont des formules de la logique des prédicats, alors $\neg A$, $A \vee B$, $A \wedge B$, $A \Rightarrow B$, $A \Leftrightarrow B$, (A), $\forall x\ A$, $\exists x\ A$ sont des formules de la logique des prédicats

En prolog, les symboles de $\mathcal{F}$ et de $\mathcal{R}$ sont constitués par toute suite de symbole commençant par une minuscule. Les variables sont les symboles qui commencent par une majuscule ou par le symbole '_'. Le symbole '_' seul est la variable anonyme : 2 occurences de ce symbole désignent 2 variables différentes.

Les clauses s'écrivent :

```
q :- p_1(...), ..., p_n(...).
```

Si q est absent, on a une question. Si les p sont absents, on a un fait.

Lorsqu'on lui soumet une question, Prolog effectue une recherche de preuve par chainage arrière, par ordre linéaire des buts, et avec bactracking. La première réponse est retournée si il y en a une. L'utilisation du ';' permet d'obtenir les autres réponses. (Se reporter aux slides du cours.)

Les listes s'écrivent :

```
[a_1, ..., a_n]
```

où les `a_i` désignent tout terme de $T_{\mathcal{F}(V)}$, y compris d'autres listes.

On peut "séparer" une liste non-vide en unifiant deux variables : `[a_1, ..., a_n] = [X | Y]` La variable `X` est unifiée avec `a_1` et la variable `Y` avec `[a_2, ..., a_n]` (qui peut être vide).

Prolog connaît les nombres mais l'évaluation des expressions arithmétiques nécessite l'utilisation du prédicat 'is'. En Prolog les termes '2+3' et '3+2' sont différents et ne sont pas réduits automatiquement par évaluation.

2 Effets de l'ordre entre les prédicats et dans la définition des prédicats

Reprenons l'exemple de `ancestor_of` et la base de faits suivante :

```
female(helen).
female(ruth).
female(petunia).
female(lili).

male(paul).
male(albert).
male(vernon).
male(james).
male(dudley).
male(harry).

parent_of(paul,petunia).
parent_of(helen,petunia).
parent_of(paul,lili).
parent_of(helen,lili).
parent_of(albert,james).
parent_of(ruth,james).
parent_of(petunia,dudley).
parent_of(vernon,dudley).
parent_of(lili,harry).
parent_of(james,harry).
```

Comparer ce qui se passe avec les définitions suivantes :

```
% Définition 1
ancestor_of(X,Y) :- parent_of(X,Y).
ancestor_of(X,Y) :- parent_of(X,Z),
                    ancestor_of(Z,Y).

% Définition 2
ancestor_of(X,Y) :- parent_of(X,Z),
                    ancestor_of(Z,Y).
ancestor_of(X,Y) :- parent_of(X,Y).

% Définition 3
ancestor_of(X,Y) :- parent_of(X,Y).
ancestor_of(X,Y) :- ancestor_of(Z,Y),
                    parent_of(X,Z).

% Définition 4
ancestor_of(X,Y) :- ancestor_of(Z,Y),
                    parent_of(X,Z).
ancestor_of(X,Y) :- parent_of(X,Y).
```

3 Listes et arithmétique

Écrire un prédicat `appartient(Elt, Liste)` qui sera vrai uniquement lorsque l'élément appartient à la liste. Ce prédicat permettra aussi d'énumerer les éléments de la liste.

Écrire un prédicat `longueur(L, N)` qui associe sa longueur à une liste.

4 Contrôle du retour arrière : le "cut"

Soit un but `h` qui possède des clauses

```
h1 :- p1, ... pi, !, p(i+1),  ... pn
h2 :- ...
```

Lorsque Prolog essaie de satisfaire le but `h`, il commencera par la clause `h1 :- ...` et entrera donc dans les but `p1, ..., pi` qu'il essaiera de satisfaire successivement par retour-arrière éventuellement. Tant que le `!` n'est pas franchi, toutes les options pour le but `h` restent ouvertes. En particulier, celle de passer à la clause `h2 :- ...` si la partie `p1, ..., pi` n'est pas satisfiable. Dès que le `!` est franchi en revanche, il est encore possible de chercher une solution par retour arrière dans la partie `p(i+1), ..., pn` mais le choix pour le but `h` de la clause `h1 :- ...` ne peut plus être remis en cause. Si aucune solution n'est trouvée dans la partie `p(i+1), ..., pn`, le but `h` aura échoué.

Soit le prédicat `max(X, Y, Z)` qui retourne dans `Z` le plus grand nombre entre `X` et `Y` :

```
max(X, Y, X) :- X >= Y.
max(X, Y, Y).
```

En quoi cette définition pose-t-elle problème ? Corrigez cette définition.

En quoi l'utilisation de `!` permet-elle d'optimiser l'exécution de ce prédicat ?

Traduire en Prolog l'assertion suivante : *Marie aime tous les animaux, sauf les araignées.*. Aidez-vous de la coupure `!` et des prédicats `true` et `fail`. Vous pouvez aussi recourrir à l'alternative `;` pour exprimer cette assertion sous une forme plus compacte.

5 Cryptarithmétique

Soit le puzzle classique :

```
  S E N D
+ M O R E
---------
M O N E Y
```

où chaque lettre doit être remplacée par un chiffre entre 0 et 9, l'addition devant être correcte et chaque lettre remplacée par un chiffre différent.

Écrire un programme Prolog pour résoudre ce puzzle :

```
crypt([S, E, N, D], [M, O, R, E], [M, O, N, E, Y]) :- ...
```

Toutes les lettres désignent des chiffres différents. Les lettes S et M sont différentes de 0.

Aide : vous pouvez utiliser le prédicat between.

6 Enigme 1

Il y a 4 étudiants : Alice, Bob, Craig et Eve. Chacun bénéficie d'une bourse et étudie une matière principale. L'objectif est de savoir quel étudiant a quelle bourse et étudie quelle matière (toutes les bourses et les matières principales étant différentes les unes des autres) à partir des indications fournies. Les bourses disponibles sont les suivantes : 25000€, 30000€, 35000€ et 40000€. Les matières disponibles sont : Astronomie, Anglais, Philosophie, Physique. Les indices suivants sont donnés pour résoudre le problème :

- L'étudiant qui étudie l'astronomie reçoit une bourse inférieur à celle de Craig.
- Craig est soit celui qui étudie l'anglais, soit celui qui étudie la philosophie.
- L'étudiant qui étudie la physique a une bourse plus importante de 5000€ que celle de Alice.
- Bob a une bourse de 10000€ plus importante que celle de Alice.
- Eve a une bourse plus importante que celle de l'étudiant qui étudie l'anglais.

Résoudre cette énigme en la modélisant en Prolog.

7 Système expert

7.1 Exemple d'une base de connaissances simple

On part de l'exemple suivant exprimé en Prolog :

```
leak_in_bathroom :- hall_wet, kitchen_dry.
problem_in_kitchen :- hall_wet, bathroom_dry.
no_water_from_outside :- window_closed ; no_rain.
leak_in_kitchen :- problem_in_kitchen, no_water_from_outside.
```

Les observations sont :

```
hall_wet.
bathroom_dry.
window_closed.
```

Nous pouvons vérifier l'hypothèse :

```
?- leak_in_kitchen.
```

Inconvénients : il faut connaître Prolog et sa syntaxe pour écrire de telles règles.

7.2 Modification de la syntaxe

Prolog est un langage dont l'analyseur syntaxique est paramétrable ! On peut donc introduire une notation plus sympathique pour les règles :

```
:- op( 800, fx, if).
:- op( 700, xfx, then).
:- op( 300, xfy, or).
:- op( 200, xfy, and).
```

```
if
   hall_wet and kitchen_dry
then
   leak_in_bathroom.
if
   hall_wet and bathroom_dry
then
   problem_in_kitchen.
if
   window_closed or no_rain
then
   no_water_from_outside.

fact(hall_wet).
fact(bathroom_dry).
fact(window_closed).
```

Écrire un interpréteur `is_true` de règles en chaînage arrière pour les règles if-then.

7.3 Chaînage avant

Prolog permet de modifier dynamiquement sa base de faits/règles à l'aide des prédicats `assert()` et `retract()`.

Écrire un mécanisme d'inférence en chaînage avant qui vérifie chaque règle et qui ajoute tous les nouveaux `fact()` possibles.

```
forward :- new_derived_fact(P), !, write( 'Derived:'), write( P), nl,
   assert(fact(P)), forward
   ;
   write('No more facts').

new_derived_fact(P) :- ...
```

Note : vous devrez ajouter la directive `:- dynamic(fact)` en tête de votre programme pour autoriser Prolog à ajouter et retirer dynamiquement les prédicats `fact()` de la base de faits.

Vous devez obtenir :

```
?- forward.
Derived: problem_in_kitchen
Derived: no_water_from_outside
Derived: leak_in_kitchen
No more facts
```

7.4 Générer des explications

Une explication est un arbre de preuve pour la question posée. Ajouter un opérateur :

```
:- op( 800, xfx, <=).
```

qui servira à conserver une preuve de la véracité d'un fait. Le terme `P <= Proof` indiquera que `Proof` est une preuve de `P`.

7.5 Poser des questions

Un des désavantages majeurs à notre système expert pour l'instant réside dans la nécessité de fournir tous les faits à l'avance. Nous préférerions les demander interactivement, et si possible ne pas demander plusieurs fois la même chose à l'utilisateur.

```
?- is_true( leak_in_kitchen, How_explanation).
Is it true: hall_wet?
Please answer yes, no, or why
|: yes.
Is it true: bathroom_dry?
Please answer yes, no, or why
|: yes.
Is it true: window_closed?
Please answer yes, no, or why
|: no.
Is it true: no_rain?
Please answer yes, no, or why
|: why. % L'utilisateur ne sait pas si il a plu, alors il veut savoir pourquoi on lui demande ça
To explore whether no_water_from_outside is true,
using rule
   if window_closed or no_rain then no_water_from_outside
To explore whether leak_in_kitchen is true,
using rule
   if problem_in_kitchen and no_water_from_outside then leak_in_kitchen
Is it true:no_rain?
Please answer yes, no, or why
|: yes.
How_explanation =
   (leak_in_kitchen <=
      (problem_in_kitchen <= (hall_wet <= was_told)
                     and (bathroom_dry <= was_told))
                     and (no_water_from_outside <= (no_rain <= was_told)))
```

Pour implémenter ce dialogue, vous vous reposerez sur le squelette de code `expert-system-ask.pl`.

7.6 Identification des fromages

On veut mettre en place un petit système expert permettant d'identifier un fromage à partir de la description de certaines de ces caractéristiques : gras, mou, diamètre, couleur, etc.

Les règles dont on dispose sont fournies dans le fichier `fromage-base.pl` disponible dans l'équipe Teams du cours.

Ces règles sont de la forme :

```
rule1 :: if Fromage est mou
et Fromage est fleuri
et Fromage est vache
et diametre(Fromage, 10)
alors Fromage est camembert.
```

```
rule2 :: si Fromage est mou
et Fromage est jaune
et Fromage 'n est pas' cuit
et diametre(Fromage, DT)
et DT > 5
alors Fromage est st_nectaire.
```

Les déductions se feront à partir de deux méta-règles :

- si p et $p \Rightarrow q$ alors q;
- si $\neg q$ et $p \Rightarrow q$ alors $\neg p$.

Dans un premier temps, écrire un prédicat qui effectue une recherche en mode déductif n'utilisant que la première des deux méta-règles. Ce prédicat fonctionnera par saturation : il devra effectuer toutes les déductions possibles à partir d'une liste de faits donnée. Vous ne vous occuperez pas des contraintes numériques sur les variables.

Augmentez votre système déductif pour garder une trace de votre raisonnement. Cette trace pourra servir à :

- donner une explication à l'utilisateur concernant la déduction d'un certain fait
- vérifier qu'on ne boucle pas dans le cas de règles cycliques.

Modifier votre système de déduction de façon à ce qu'il fonctionne en mode consultation : le système vous pose des questions parmi une liste de questions admissibles correspondant aux observations qui peuvent être faites de manière objective. Il mémorise les réponses à vos questions de manière à ne pas demander deux fois la même chose.

Modifier votre programme pour qu'il donne des explications :

- lorsqu'il a identifié un fromage possible, l'utilisateur pourra demander `pourquoi` ce fromage a été identifié,
- lorsqu'il pose une question, l'utilisateur pourra demander `pourquoi` cette question est posée.

Si la réponse à une question du type `Est-ce que Fromage est jaune` est négative, alors considérer explicitement que le fait prouvé est `Fromage 'n est pas' jaune`.

On veut utiliser les deux méta-règles données plus haut. On peut coder le raisonnement de la manière suivante. Soit `bf` la base de faits courante, `br` la base de règles courante et `b` la liste de règles restant à tester :

- Si `b` est vide, il n'y a plus de déductions possibles, on retourne la liste des faits déduits;
- Soit `r` la première règle de `b` la règle candidate. Si toutes ses prémisses sont vérifiées, on relance le moteur en augmentant la base de faits avec la conclusion de `r`, avec `br` $\setminus$ {`r`} comme base de règles et comme règles restant à tester;
- Si `r` n'a qu'une prémisse et que la négation de sa conclusion est dans `bf`, alors on relance le moteur en augmentant `bf` de la négation de la prémisse de `r` et avec `br` $\setminus$ {`r`} comme base de règles et comme ensemble de règles restant à tester;
- Enfin si les cas précédents ont échoués, c'est que la règle candidate `r` n'est pas déclenchable et on relance le moteur avec `bf`, `br` et `b` $\setminus$ {`r`}.

Modifier votre système expert pour utiliser les deux méta-règles selon les lignes explicitées ci-dessus.

Intégrer la gestion des contraintes numériques : il se peut qu'un fait comme `Fromage est gras` résulte dans une contrainte du type `mt(Fromage, MT) et MT > 60`. Est-ce que vous pouvez trouver une solution pour prendre en compte ces contraintes ?

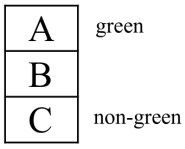
8 Enigme 2

Retour sur l'énigme suivante :

Three blocks stacked.

Top one is green.

Bottom one is not green.



Question : existe-il une boîte verte posée directement sur une boîte non-verte?
Pouvez-vous modéliser ce problème en Prolog ? Si oui, comment ? Et si non, pourquoi ?