TP3 : Interrogation de bases de connaissances

Objectif du TP

On considère des bases de connaissances de la forme K = (BF, BR) où BF est une base de faits et BR une base de règles en logique du premier ordre (sans fonction). Un terme est une variable ou une constante. Un atome est de la forme $p(e_1, \ldots, e_k)$, où p est un prédicat, les e_i sont des termes et $k \geq 0$. Un fait est un atome instancié. Une règle est une règle positive et conjonctive, dont la conclusion est composée d'un seul atome.

L'objectif de ce TP est d'implémenter une application qui permettra :
- de charger une base de connaissances K = (BF, BR) à partir d'un fichier
- de calculer la saturation de BF par BR
- de calculer les réponses à des requêtes conjonctives sur K. On utilise pour cela la base de faits saturée.

Etape 1 : Classe FactBase

Nous adopterons les conventions suivantes pour la représentation textuelle des atomes et listes d'atomes (utilisées par exemple pour le chargement à partir d'un fichier texte) :

- Une constante est entourée de guillemets simples. Ex: `'Toto'`
- Un atome a la forme logique habituelle. Ex: `mange(x,'Herbe')`
- Un ensemble d'atomes est donné par une chaîne de caractères de la forme `atome1;atome2;... ;atomek`
- Il n'y a pas d'espace autour des séparateurs que sont le point-virgule ; et les parenthèses ( ).

Nous vous fournissons les classes `Term` et `Atom` pour représenter la notion de terme et d'atome.
→ voir les fichiers `Atom.java` et `Term.java` sur l'ENT.

Description des classes Atom et Term (qui font partie d'un package nommé structure)

- un **terme** possède un label (le nom du terme) et un booléen qui est vrai si le terme est une constante ; voir les méthodes de cette classe (elles sont très simples) ;

- un **atome** possède un prédicat (une chaîne de caractères) et une liste d'arguments (qui sont des termes) ; dans cette liste, un terme peut apparaître deux fois : `p(x,x,y)` a pour prédicat p et une liste de trois terme dont deux sont le même terme. Le constructeur prend en paramètre la chaîne de caractères décrivant l'atome. La méthode `main` de cette classe donne des exemples de création d'atomes.

Implémenter une classe **FactBase** selon les spécifications suivantes :

- pour un accès rapide aux atomes ayant un certain prédicat, l'ensemble des atomes est codé sous la forme d'une table de hachage dont les clés sont les prédicats et les valeurs les listes de termes des atomes ayant ce prédicat (par exemple pour les deux atomes `p(a,b,c)` et `p(b,c,d)`, on aura la clé `p` dont la valeur associée sera la liste `{(a,b,c),(b,c,d)}`.

```
private HashMap<String,ArrayList<ArrayList<Term>>> atoms;
```

- un constructeur permet de créer une base de faits à partir d'une chaîne de caractères de la forme `atome1;atome2;... ;atomek`

  Aide pour le découpage de la chaîne de caractères :

```
StringTokenizer st = new StringTokenizer(laStringALire,";");
while(st.hasMoreTokens())
{
      String s = st.nextToken(); // s represente un atome
      Atom a = new Atom(s);
}     Ajouter a à la base de faits
```

- une méthode `getTerms` retourne l'ensemble des termes (des constantes ici) apparaissant dans la base de faits. Vous pouvez ajouter un attribut qui stocke cet ensemble de termes. Cet attribut peut être initialisé dans le constructeur (auquel cas la méthode `getTerms` ne fait qu'accéder à cet attribut).
- une méthode `toString` retourne une représentation textuelle de la base de faits comportant la liste des faits, le nombre total de faits et la liste des constantes.

Testez votre classe en créant une base de faits à partir d'une chaîne de caractères et en l'affichant.

Etape 2 : Algorithme de recherche d'homomorphismes

Interroger une base de faits (saturée ou non) consiste à rechercher les homomorphismes de la requête dans la base de faits. Appliquer une règle lors du chaînage avant consiste à rechercher les homomorphismes de l'hypothèse (ou condition) de la règle dans la base de faits courante.

Dans les deux cas, on doit résoudre le problème suivant :

Données : Q un ensemble d'atomes (requête ou hypothèse de règle) et BF une base de faits
Question : trouver tous les homomorphismes de Q dans BF.

Au lieu de programmer à nouveau un algorithme de backtrack, on va réutiliser le travail fait pour CSP. On procédera donc en deux étapes :

1) transformer (Q,BF) en un réseau de contraintes P = (X,D,C)
2) appeler l'algorithme de backtrack sur P : chaque assignation solution définit un homomorphisme de Q dans BF.

On définit maintenant le squelette d'une classe appelée « `Homomorphisms` ». Cette classe possède (au minimum) les attributs suivants :

- une liste d'atomes Q (`ArrayList<Atom>`)
- une (référence sur une) base de faits BF (`FactBase`)
- l'ensemble des homomorphismes de Q dans BF (`ArrayList<Assignation>`) ; si l'ensemble des homomorphismes n'est pas calculé dans le constructeur, vous pouvez ajouter un booléen qui indique si la méthode de calcul des homomorphismes a déjà été appelée ou non.

La classe `Homomorphisms` possède au moins les méthodes suivantes :

- une méthode privée qui transforme une chaîne de caractères (représentant Q) en une liste d'atomes
- un constructeur qui prend en paramètre une chaîne de caractères (Q) et une base de faits (BF)
- un constructeur qui prend en paramètre une liste d'atomes (Q) et une base de faits (BF)
- une méthode privée qui transforme (Q, BF) en un réseau de contraintes P. Une façon simple de procéder consiste à ajouter à la classe `ConstraintExt` un constructeur qui initialise une contrainte en prenant en paramètre la liste de ses variables et (une référence sur) une liste de tuples qui constitue la définition de la contrainte, puis utiliser ce constructeur pour initialiser les contraintes obtenues par votre transformation en passant pour liste de tuples d'une contrainte correspondant à `p(…)` une référence sur la liste des arguments des atomes de prédicat `p`.
- une méthode `backtrackAll` qui calcule tous les homomorphismes de Q dans BF ; cette méthode peut être appelée dans les constructeurs ou pas (dans ce dernier cas, il faut qu'elle soit publique).

- Une méthode `toString` qui affiche Q et BF, ainsi que la liste des homomorphismes de Q dans BF (si ceux-ci ont été calculés).

Ajouter à la classe `FactBase` une méthode qui, étant donnée une requête (vue comme un ensemble d'atomes), retourne la liste des réponses à cette requête. Si la requête ne comporte pas de variables, la réponse est simplement oui ou non.

Testez vos classes.

Etape 3 : Amélioration de la réduction à CSP (option)

Si vous avez programmé « brutalement » la réduction précédente, vous avez associé à chaque variable de Q le même domaine qui est l'ensemble des constantes de la base de faits, qui est donc copié plusieurs fois. Ceci est problématique si la base de faits est grande (par rapport à la requête qui ne comporte habituellement que quelques atomes).

Pour améliorer ce point, vous pouvez considérer les positions qu'occupent les variables dans Q : on note (p,i) la position i dans un atome de prédicat p. Vous pouvez par exemple affiner le domaine initial de chaque variable en considérant les positions (p,i) qu'elle occupe dans les atomes de Q et en restreignant son domaine aux constantes qui apparaissent dans chacune de ces positions (p,i).

Etape 4 : Classes Rule et KnowledgeBase

Ecrire une classe **Rule** qui permet de représenter une règle. L'hypothèse d'une règle est représentée par une liste d'atomes, et sa conclusion par un atome.

Cette classe possède entre autres un constructeur qui permet d'initialiser une règle à partir d'une chaîne de caractères de la forme suivante : `"atome1;atome2;...atomek"`, où les (k-1) premiers atomes sont l'hypothèse et le dernier est la conclusion. Ex: `p(x,y);p(y,z);p(x,z)`

Ecrire une classe **KnowledgeBase** composée :
- d'une base de faits (`FactBase`)
- et d'une base de règles (une liste de règles – vous pouvez aussi créer une classe dédiée)

Cette classe fournit les méthodes publiques suivantes :
- un constructeur qui crée une base vide
- un constructeur qui crée une base à partir d'un fichier texte dont le chemin d'accès est passé en paramètre
- des accesseurs à la base de faits et à la base de règles
- une méthode d'ajout d'une règle
- une méthode d'ajout d'un fait
- une méthode `toString` (qu'on utilisera en particulier pour afficher la base sur la console).

On supposera que le fichier a une syntaxe conforme aux règles définies. Il ne vous est donc pas demandé de vérifier sa syntaxe.

Testez le chargement et l'affichage de votre base de connaissances. Voir par exemple la base de connaissances donnée en annexe.

Etape 5 : Saturation

Ajouter à la classe KnowledgeBase une méthode qui sature la base de faits avec l'ensemble des règles.

Rappel de l'algorithme de saturation

```
// Données : K = (BF, BR)
// Résultat : BF saturée par application des règles de BR
Début
Fin ← faux
Tant que non fin
   new ← ∅          // ensemble des nouveaux faits obtenus à cette étape
   Pour toute règle R = H → C de BR
      Pour tout nouvel homomorphisme S de H dans BF
                  // implémentation de « nouvel » optionnelle
         Si S(C) n'est ni dans BF ni dans new
               Ajouter S(C) dans new
      FinPour
   FinPour
   Si new = ∅
      Fin ← vrai
   Sinon ajouter tous les éléments de new à BF
FinTantQue
Fin
```

Etape 6 : Interrogation d'une base de connaissances

Ajouter à la classe KnowledgeBase une méthode qui, étant donnée une requête, retourne la liste des réponses à cette requête sur la base de connaissances (autrement dit sur la base de faits saturée).

Remarque : il peut être utile de noter l'état de la base de faits : est-elle saturée ou non ? Lors de la première requête, il sera nécessaire de saturer la base ; pour les suivantes, le temps de réponse sera plus court.

Ajouter une méthode main qui permet de :

- charger une base de connaissances à partir d'un fichier
- interroger cette base de connaissances en saisissant les requêtes sur la console.

Annexe : une base de connaissances

[chèvre, loup, herbe et eau seront représentés par des constantes]

Faits

La chèvre est un animal herbivore
Le loup est un animal cruel *(le loup des fables …)*

Règles

Tout animal cruel est carnivore
Tout animal herbivore mange de l'herbe
Tout animal carnivore mange n'importe quel animal herbivore
Tous les animaux carnivores et tous les animaux herbivores boivent de l'eau
Tout animal consomme ce qu'il boit ou mange

Quelques exemples de requêtes

Le loup mange-t-il la chèvre ?
Quels sont les animaux cruels ?
Quels sont les animaux cruels et que mangent-ils ?
Quels sont les animaux qui consomment la chèvre ?
Que consomme la chèvre ?