TP2

Un moteur de règles d'ordre 0

Objectif

L'objectif de ce TP est de construire un système à base de règles d'ordre 0. La **base de connaissances** est constituée d'une **base de règles** et d'une **base de faits**. Les **règles** sont positives et conjonctives : l'hypothèse est une conjonction d'atomes et la conclusion est composée d'un seul atome. Un **fait** est un atome. Comme nous sommes en logique propositionnelle (ou logique d'ordre 0), un atome est simplement une variable propositionnelle prenant la valeur vrai ou faux.

Le moteur d'inférence de ce système fonctionne en **chaînage avant** et en **chaînage arrière**.

On ne vous demande pas de créer une interface graphique, une exécution en mode "console" suffira.

Etape 1 : Utilisation des classes fournies

➔ Lancez votre environnement Java favori (Eclipse par exemple) et créez un projet contenant les fichiers sources java que vous trouverez sur l'ENT / Moodle / HMIN107 / Sources TP2.

Remarque : Les squelettes de classe fournis permettront par la suite de gérer des bases de connaissances en **logique du premier ordre**. En logique du premier ordre, un **atome** est de la forme $p(t_1...t_k)$ où p est un prédicat et les t_i des termes (variables ou constantes pour la partie de la logique qu'on considérera). En logique des propositions, un **atome** est simplement une variable propositionnelle, qu'on peut voir comme un prédicat sans arguments (autrement dit, $k = 0$).

Les fichiers sources java appartiennent à un package nommé "structure" et fournissent quelques classes de base :

- **Atom** (pour représenter un atome -- pour l'instant une variable propositionnelle). Cette classe possède notamment une méthode equalsP pour vérifier si deux atomes ont le même prédicat et equalsA pour vérifier si deux atomes sont identiques (même prédicat et même liste de termes). Pour des atomes propositionnels, ces deux méthodes donnent le même résultat.
- **Term** (pour représenter un terme) -- inutile pour ce TP
- **Rule** (pour représenter une règle)
- **FactBase** (pour représenter une base de faits). Cette classe possède des méthodes permettant de tester si un fait apparaît dans la base de faits, d'ajouter un fait sans vérification, et d'ajouter une liste de faits (un ArrayList) en n'ajoutant effectivement que ceux qui ne sont pas déjà présents.
- **RuleBase** (pour représenter une base de règles).

Initialisations et forme textuelle des objets

- Une base de faits est initialisée soit par une chaîne de caractères décrivant une liste d'atomes `"atome1;atome2;...atomek"`, soit comme une base vide. On peut ensuite lui ajouter des faits, soit un par un, soit en lui passant un ArrayList.
- Une règle est initialisée par 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: `Benoît;Djamel;Emma;Félix`
- Une base de règles est initialisée comme une base vide, à laquelle on peut ensuite ajouter des règles.

➔ Pour vous familiariser avec les classes fournies, écrire une méthode main qui :

- crée une base de faits (tester les différentes possibilités d'initialisation)
- affiche la base de faits obtenue (affichage console)
- crée une base de règles (vide au départ)
- lui ajoute un certain nombre de règles
- affiche la liste des règles obtenue

Vous pouvez partir du fichier Application0.java qui construit une base de faits avec 2 faits et une base de règles avec 2 règles.

Etape 2 : Classe Base de Connaissances

1) Ecrire une classe **KnowledgeBase** composée :

- d'une base de faits initiale (une instance de FactBase)
- d'une base de règles (une instance de RuleBase)
- d'une deuxième base de faits initialement vide (ce sera la base de faits saturée)

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 (voir en annexe des rappels sur les fichiers textes)
- des accesseurs aux deux bases de faits (initiale et saturée) et à la base de règles
- une méthode toString (qu'on utilisera en particulier pour afficher la base sur la console).

- *La* racine du projet *java est le répertoire du projet (si vous réduisez le chemin d'accès du fichier texte à son nom, il faut donc que le fichier soit à la racine du projet).*
- *Pensez à réutiliser (et éventuellement améliorer) les méthodes des classes fournies*

Exemple de format texte pour la base de connaissances :

```
Liste des faits (sur une seule ligne)
règle 1
…
règle n
```

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

2) Testez votre classe : écrivez une méthode main qui charge une base de connaissances à partir d'un fichier texte et affiche son contenu. Traitez en particulier les exemples suivants :

- réunion d'amis (fichier texte `reunion.txt` fourni en utilisant le format ci-dessus)
- exercices 4 et 5 du TD 4 à coder sous forme de fichier texte.

Etape 3 : Chaînage avant

1. Ajouter à la classe **KnowledgeBase** une méthode publique "forwardChaining" qui sature une base de connaissances en chaînage avant. La base de faits saturée est stockée dans l'attribut correspondant (la base de faits initiale n'étant pas modifiée).

- Rappel des deux versions de l'algorithme de chaînage avant vues en cours :

```
Algorithme FC(K) // saturation de la base K
// Données : K = (BF, BR)
// Résultat : BF saturée par application des règles de BR

Version « naïve »

Début
Fin ← faux
Pour toute règle R de BR
  Appliquée(R) ← faux
Tant que non fin
  nouvFaits ← ∅ // ensemble des nouveaux faits obtenus à cette étape
  Pour toute règle R = H → C de BR telle que Appliquée(R) = faux
    Si H est incluse dans BF // R est applicable
      Appliquée(R) ← vrai // que l'application soit utile ou pas
      Si C n'est ni dans BF ni dans nouvFaits // l'application de R est utile
        Ajouter C à nouvFaits
  FinPour
  Si nouvFaits = ∅
    Fin ← vrai
  Sinon ajouter tous les éléments de nouvFaits à BF
FinTantQue
Fin
```

```
Version avec compteurs

Début
  ATraiter ← BF
  Pour toute règle R de BR
    Compteur(R)← Nombre de symboles de l'hypothèse de R
  Tant que ATraiter n'est pas vide
    Retirer un atome A de ATraiter
    Pour toute règle de BR ayant A dans son hypothèse
      Décrémenter Compteur(R)
      Si Compteur(R) = 0 // R est applicable
           Soit C la conclusion de R
           Si C n'est ni dans BF ni dans ATraiter
                 Ajouter C à ATraiter
                 Ajouter C à BF
    FinPour
  FinTantQue
Fin
```

Si vous avez besoin de tester l'égalité de deux atomes, utilisez la méthode equalsA (ou equalsP) de la classe Atome. Vous pouvez aussi redéfinir la méthode `public boolean equals(Object o)` héritée de Object pour qu'elle fasse l'équivalent de equalsA. Ne pas utiliser equals sans la redéfinir (elle effectuerait le test == ce qui n'est pas le comportement attendu).

2. Ajouter une méthode main qui charge une base de connaissances à partir d'un fichier texte, affiche son contenu puis la base de faits saturée.

Etape 4 : Chaînage arrière

1. Ajouter à la classe **KnowledgeBase** une méthode publique "backwardChaining" qui étant donné un atome (but à prouver) et retourne vrai si et seulement si l'atome peut être prouvé.

   - L'algorithme ne doit pas boucler (cf. l'algorithme BC3 du cours).

2. Améliorer l'algorithme de chaînage arrière en mémorisant les atomes déjà prouvés ou ayant déjà mené à un échec, et en exploitant ces informations (cf. exercice 5 du TD 4).

3. Tester votre algorithme sur vos 3 bases de connaissances exemples.

Etape 5 : Application finale

Ecrire une application qui :
- charge une base de connaissances à partir d'un fichier texte
- l'affiche
- calcule la base de faits saturée et affiche sa taille
- boucle sur : saisie d'un atome à prouver, et affichage :
  - du résultat par recherche dans la base de faits saturée : oui/non
  - du résultat par recherche en chaînage arrière : oui/non.

Bien évidemment, les deux types de recherche sont censés donner le même résultat.

Annexe : lecture et affichage d'un fichier texte

- Rappels Java (reste à ajouter la gestion des exceptions d'entrée/sortie)

```
String fileName = "essai.txt" ; // nom du fichier
System.out.println("Chargement du fichier : "+
          new java.io.File( "." ).getCanonicalPath()+ "/" + fileName);
BufferedReader readFile = new BufferedReader(new FileReader (fileName));

System.out.println("Lecture du fichier" + fileName);
String s = readFile.readLine();
/* readLine() retourne :
        -  la ligne lue jusqu'au retour chariot (lu mais non retourné),
           donc une chaîne vide si la ligne ne comporte qu'un RC
        -  la valeur null s'il n'y a rien à lire (fin du flux de données)
 */
while (s!= null && s.length()!= 0)
// arrêt si fin de fichier ou ligne lue vide
     {
          System.out.println(s);
          s = lectureFichier.readLine();
     }
readFile.close();
```