TP2

Un moteur de règles d'ordre 0

Objectif

L'objectif de ce TP est de construire un système à base de règles en logique d'ordre 0 (ou logique propositionnelle). 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, un atome est simplement une variable propositionnelle (ou symbole propositionnel) 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

➔ Lancer votre environnement Java favori (Eclipse par exemple) et créer un projet contenant les fichiers sources java que vous trouverez sur l'ENT / Moodle / HMIN107 / Sources TP2. Ces fichiers définissent quelques classes de base au sein d'un package nommé « structure » :

- **Atom** (pour représenter un atome)
- **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: `Benoit;Djamel;Emma;Felix`
- Une base de règles est initialisée à vide, on peut ensuite lui ajouter des règles.

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

- crée une base de faits et l'affiche ;
- crée une base de règles (vide au départ), lui ajoute un certain nombre de règles et 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) Tester votre classe : écrivez une méthode main qui charge une base de connaissances à partir d'un fichier texte et affiche son contenu. Traiter en particulier les exemples suivants :

- réunion d'amis (fichier texte `reunion.txt` fourni, utilisant le format ci-dessus)
- exercices du TD 5 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 selon l'algorithme naïf. La base de faits saturée est stockée dans l'attribut correspondant (la base de faits initiale n'étant pas modifiée).

2. Ajouter une méthode main qui charge une base de connaissances à partir d'un fichier texte, l'affiche, exécute l'algorithme de chaînage avant, puis affiche la base de faits saturée.

Etape 4 : Chaînage avant optimisé

Ajouter à la classe **KnowledgeBase** une méthode publique "forwardChainingOpt" qui sature une base de connaissances en chaînage avant selon l'algorithme avec compteurs. Utilisez des structures de données qui permettent d'effectuer **efficacement** les deux opérations suivantes :

a. Trouver toutes les règles dont l'hypothèse contient F (utilisé lorsque F est traité)
b. Tester si C est dans BF ou dans ATraiter.

Avec vos structures de données, le chargement d'une base de connaissances reste-t-il linéaire (en la taille de la base de connaissances) ? Le chaînage avant est-il effectué en temps linéaire ? Si ce

n'est pas le cas, quelles sont les complexités du chargement de la base de connaissances et du chaînage avant ?

Etape 5 : Chaînage arrière

1. Ajouter à la classe **KnowledgeBase** une méthode publique "backwardChaining" qui, étant donné un atome (but à prouver), retourne vrai si et seulement si l'atome peut être prouvé. L'algorithme ne doit pas boucler (cf. l'algorithme BC3 du cours).

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

3. Modifier votre méthode de chaînage arrière pour qu'elle puisse afficher l'arbre de recherche (à l'aide d'indentations).

Etape 6 : Chaînage arrière optimisé

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. TD).

Optimiser les structures de données. Finalement quelle est la complexité de votre algorithme de chaînage arrière (donner au moins une borne supérieure grossière) ?

Etape 7 : Application finale

Ecrire une application qui :
- charge une base de connaissances à partir d'un fichier texte
- l'affiche, ainsi que sa taille
- 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.

Etape 8 (Optionnel) : Extension à la négation par défaut

On étend les règles de façon à permettre la négation par défaut. L'hypothèse d'une règle est maintenant une conjonction de littéraux, tandis que sa conclusion reste un atome. Une règle est applicable à une base de faits BF si tous ses littéraux positifs appartiennent à BF et aucun de ses littéraux négatifs n'appartient à BF.

1. Etendre le cadre de travail : format textuel des règles (et format textuel d'une base de connaissances), classe Rule plus générale (vous pouvez étendre la notion d'atome à celle de littéral, ou considérer que l'hypothèse d'une règle a deux listes d'atomes, correspondant respectivement aux littéraux positifs et négatifs).
2. On se limite dans un premier temps à des ensembles de règles semi-positifs (un symbole qui apparaît dans un littéral négatif ne peut donc pas figurer en conclusion de règle). Etendre votre algorithme de chaînage avant.

3. On considère maintenant des ensembles de règles stratifiés. Etendre à nouveau le cadre de travail et l'algorithme de chaînage avant de façon à gérer correctement ce type d'ensemble de règles. Vous pouvez supposer que l'ensemble de règles est fourni déjà stratifié (auquel cas il faut aussi étendre le format textuel) ou bien résoudre d'abord le point 4.
4. Implémenter l'algorithme qui permet de tester si un ensemble de règles est stratifiable, et le cas échéant fournit une stratification.

Tester votre algorithme sur les exemples vus en cours et TD.

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 = readFile.readLine();
       }
readFile.close();
```