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 (ou symboles) et la conclusion est composée d'un seul atome. Un **fait** est un atome.

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.

Étape 1 : Utilisation des classes fournies

➔ Lancez 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 pour l'instant « structureToBeCompleted » :

- **Atom** (atome).
- **Rule** (règle).
- **FactBase** (base de faits).
- **RuleBase** (base de règles).
- **KnowledgeBase** (base de connaissances) composée d'une base de faits (une instance de FactBase) et d'une base de règles (une instance de RuleBase). Un troisième attribut permet de stocker la base de faits saturée (initialement vide).

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.
- Une base de connaissances est initialisée à vide ou par lecture d'un fichier dont le chemin d'accès est passé en paramètres sous forme d'une String.

Format attendu du fichier texte pour la base de connaissances :

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

Il n'y a pas de vérification que le fichier a une syntaxe conforme aux règles définies.

➔ Lire chacun de ces fichiers java pour vous familiariser avec la structure de chacune de ces classes.

➔ Créer un fichier texte correspondant à l'exemple de la réunion d'amis. Écrire une classe Application qui crée une base de connaissances à partir de ce fichier et l'affiche (il vous suffit de réutiliser les méthodes existantes).

Rappel : *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).*

Étape 2 : Chaînage avant

La classe **KnowledgeBase** contient déjà une méthode publique "forwardChainingBasic" qui sature une base de connaissances en chaînage avant selon l'algorithme naïf en largeur. La base de faits saturée est stockée dans l'attribut correspondant (la base de faits initiale n'étant pas modifiée).

➔ Comprendre le code de cette méthode (qui suit de très près l'algorithme de base vu en cours).

➔ Ajouter la saturation de la base de connaissances à la méthode main de votre classe Application.

Prévoir de tester vos classes sur les autres bases de connaissances du TD 4.

Étape 3 : 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**. Utiliser 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.

Avec ces structures de données, vous devriez obtenir une implémentation de l'algorithme en temps linéaire (ou quasi-linéaire) en la taille de la base de connaissances.

Étape 4 : Chaînage arrière

➔ 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).

Tester votre algorithme sur vos bases de connaissances exemples.

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

Exemple de format d'affichage, que vous pouvez adapter à votre guise

(preuve de U avec la base de connaissances du TD4, exercice 1) :

```
U
--- R4
--- C
------ R1
------ B
------ Échec
--- R5
--- S
--- BF
--- T
------ R2
------ A
------ BF
U est prouvé
```

Étape 5 : 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).

Pour **pouvoir effectuer plusieurs essais indépendants** les uns des autres, on ne mémorisera pas les listes « déjà prouvé » et « déjà échec » d'un chaînage arrière à l'autre (ou bien, on réinitialisera ces listes à vide à chaque test). Étendez aussi votre affichage pour mentionner « déjà prouvé / déjà échec »).

➔ Utiliser des structures de données qui permettent d'effectuer **efficacement** les opérations suivantes :

a. Tester si Q est dans un ensemble d'atomes (celui de la BF, ainsi que les listes que vous utilisez).
b. Trouver toutes les règles dont la conclusion est Q.

Avec ces structures de données, vous devriez avoir une implémentation de l'algorithme en temps linéaire (ou quasi-linéaire) en la taille de la base de connaissance.

Étape 6 : Application

➔ Écrire 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.

Étape 7 : Extension à la négation par défaut (cas semi-positif)

On étend les règles de façon à permettre la **négation par défaut (ou : négation par l'absence, négation par l'échec)**. 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.

➔ Étendre 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).

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).

➔ Ajouter une méthode **estSemiPos** qui retourne vrai si et seulement si l'ensemble de règles est semi-positif

➔ Étendre votre algorithme de **chaînage avant** de façon à respecter les spécifications suivantes :

- si l'ensemble de règles n'est pas semi-positif, le signaler par un affichage sur la console d'erreur (`System.err`) ou par la levée d'une exception d'un type que vous créerez.
- sinon calculer la base de faits saturée.

Étape 8 [Optionnel] : Extension à la négation par défaut (cas stratifié)

1. On considère maintenant des ensembles de règles **stratifiés**. Étendre 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 le point 2.

2. 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.