République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Informatique

Spécialité : Systèmes Informatiques Intelligents

Module : Représentation des Connaissances et Raisonnement

Rapport TP RCR

Réalisé par :
RAGOUB Ahmed Abdelouadoud
BOULAHLIB Ali

Introduction

Depuis les débuts de l'intelligence artificielle dans les années 1950, la question fondamentale de comment représenter et manipuler les connaissances a constitué un défi majeur pour les chercheurs. Des premiers systèmes experts aux ontologies modernes, l'évolution des formalismes de représentation des connaissances a façonné le développement de l'IA. C'est dans cette riche tradition historique que s'inscrivent nos travaux pratiques en Représentation des Connaissances et Raisonnement.

Nous avons exploré à travers six chapitres distincts les différentes techniques et formalismes logiques permettant de modéliser, de représenter et de raisonner sur des connaissances. La représentation des connaissances constitue un domaine fondamental de l'intelligence artificielle, nous offrant les outils nécessaires pour formaliser l'information de manière structurée et exploitable par les systèmes informatiques. Le raisonnement, quant à lui, nous permet d'inférer de nouvelles connaissances à partir de celles déjà acquises, simulant ainsi certains aspects du raisonnement humain.

Dans notre rapport, nous abordons successivement l'inférence logique basée sur SAT, la logique des prédicats, la logique modale, la logique des défauts, les réseaux sémantiques et les logiques de description. Pour chaque sujet, nous présentons les concepts théoriques fondamentaux suivis de leur mise en pratique à travers des exemples concrets et des implémentations informatiques.

Ces travaux pratiques nous ont permis d'acquérir une compréhension approfondie des mécanismes de représentation et de raisonnement, compétences essentielles pour le développement de systèmes intelligents.

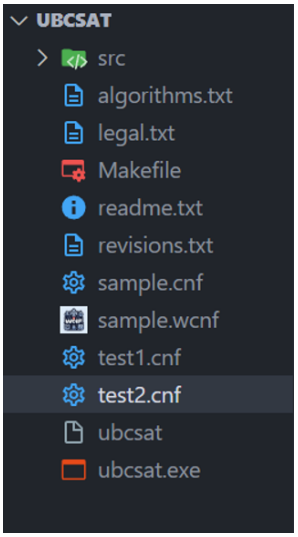
Chapitre 1 : TP1 - Inférence logique basée sur SAT

Introduction

L'inférence logique consiste à déduire de nouvelles informations à partir d'une base de connaissances. Les solveurs SAT, qui évaluent la satisfiabilité des formules logiques, sont des outils fondamentaux dans ce domaine. Dans ce TP, nous avons utilisé un solveur SAT pour illustrer ce processus à travers plusieurs étapes.

Étape 1 : Préparation de l'environnement

Nous avons débuté par la mise en place d'un environnement de travail en créant un répertoire dédié et en y intégrant les fichiers nécessaires pour exécuter le solveur SAT, ainsi que des fichiers de test.



Étape 2 : Exécution du solveur SAT

Nous avons lancé le solveur sur des fichiers de test pour analyser leur satisfiabilité. Cette étape nous a permis de comprendre comment interpréter les réponses du solveur et de confirmer la validité des formules testées.

Pour exécuter le solveur SAT, nous ouvrons l'invite de commande (Touche Windows > Accessoires > Invite de commande).
La commande d'exécution est la suivante :

```
C:\Users\ahmedrag\Desktop\Tp Rcr\UBCSAT>ubcsat -alg saps -i nom_fichier.cnf -solve
```

Commande pour test 1 :

```
C:\Users\ahmedrag\Desktop\Tp Rcr\UBCSAT>ubcsat -alg saps -i test1.cnf -solve
```

Résultat Test1.cnf :

```
# UBCSAT default output:
#    'ubcsat -r out null' to suppress, 'ubcsat -hc' for customization help
#
#
# Output Columns: |run|found|best|beststep|steps|
#
# run: Run Number
# found: Target Solution Quality Found? (1 => yes)
# best: Best (Lowest) # of False Clauses Found
# beststep: Step of Best (Lowest) # of False Clauses Found
# steps: Total Number of Search Steps
#
#       F  Best       Step      Total
#   Run N Sol'n         of     Search
#   No. D Found       Best      Steps
#
      1 1     0          1          1
#
# Solution found for -target 0

 -1 2 -3 -4 -5
```

Commande pour test 2 :

```
C:\Users\ahmedrag\Desktop\Tp Rcr\UBCSAT>ubcsat -alg saps -i test2.cnf -solve
```

Résultat Test2.cnf:

```
# UBCSAT default output:
#    'ubcsat -r out null' to suppress, 'ubcsat -hc' for customization help
#
#
# Output Columns: |run|found|best|beststep|steps|
#
# run: Run Number
# found: Target Solution Quality Found? (1 => yes)
# best: Best (Lowest) # of False Clauses Found
# beststep: Step of Best (Lowest) # of False Clauses Found
# steps: Total Number of Search Steps
#
#       F  Best       Step      Total
#   Run N Sol'n         of     Search
#   No. D Found       Best      Steps
#
#     1 1     0          7          7
#
# Solution found for -target 0

1 2 -3 -4 5
```

Étape 3 : Traduction de la base de connaissances

Nous supposons trois individus : a, b et c. Pour chacun, nous introduisons les prédicats suivants : Na, Nb, Nc pour indiquer que a, b et c sont des nautiles ; Cea, Ceb, Cec pour indiquer qu'ils sont des céphalopodes ; Ma, Mb, Mc pour signifier qu'ils sont des mollusques ; et Coa, Cob, Coc pour indiquer qu'ils ont une coquille. En logique des prédicats, nous traduisons ensuite les connaissances suivantes en clauses en forme normale conjonctive (CNF) : les nautiles sont des céphalopodes ((¬Na ∨ Cea), (¬Nb ∨ Ceb), (¬Nc ∨ Cec)), les céphalopodes sont des mollusques ((¬Cea ∨ Ma), (¬Ceb ∨ Mb), (¬Cec ∨ Mc)), les mollusques ont une coquille ((¬Ma ∨ Coa), (¬Mb ∨ Cob), (¬Mc ∨ Coc)), les céphalopodes n'ont pas de coquille ((¬Cea ∨ ¬Coa), (¬Ceb ∨ ¬Cob), (¬Cec ∨ ¬Coc)), et les nautiles ont une coquille ((¬Na ∨ Coa), (¬Nb ∨ Cob), (¬Nc ∨ Coc)). Nous ajoutons enfin les faits spécifiques : Na (a est un nautile), Ceb (b est un céphalopode), et Mc (c est un mollusque). Pour coder cela dans un fichier au format DIMACS (nommé zoo.cnf), nous associons à chaque prédicat un entier allant de 1 à 12, et traduisons les 18 clauses dans le fichier en commençant par l'en-tête p cnf 12 18. Nous avons ainsi traduit une base de connaissances sur des animaux en un format logique adapté au solveur SAT. En définissant des prédicats et des règles, nous avons créé un fichier logique et testé sa cohérence. Nous avons également expérimenté avec des fichiers de référence pour approfondir notre compréhension du fonctionnement du solveur.

Résultat pour zoo.cnf :

```
# UBCSAT default output:
#    'ubcsat -r out null' to suppress, 'ubcsat -hc' for customization help
#
#
# Output Columns: |run|found|best|beststep|steps|
#
# run: Run Number
# found: Target Solution Quality Found? (1 => yes)
# best: Best (Lowest) # of False Clauses Found
# beststep: Step of Best (Lowest) # of False Clauses Found
# steps: Total Number of Search Steps
#
#       F  Best       Step      Total
#   Run N Sol'n         of     Search
#   No. D Found       Best      Steps
#
      1 0     2          4     100000
# No Solution found for -target 0
```

Résultat pour fichier de benchmarking 1 :

```
# Output Columns: |run|found|best|beststep|steps|
#
# run: Run Number
# found: Target Solution Quality Found? (1 => yes)
# best: Best (Lowest) # of False Clauses Found
# beststep: Step of Best (Lowest) # of False Clauses Found
# steps: Total Number of Search Steps
#
#       F  Best       Step      Total
#   Run N Sol'n         of     Search
#   No. D Found       Best      Steps
#
      1 1     0         36         36
#
# Solution found for -target 0

 -1 2 -3 4 5 6 7 8 9 -10
 -11 12 -13 14 15 -16 -17 -18 19 20
 -21 -22 23 -24 -25 -26 27 -28 -29 -30
 -31 -32 -33 -34 35 36 37 38 39 -40
 -41 42 -43 -44 -45 -46 47 48 49 -50


Variables = 50
```

Résultat pour fichier de benchmarking 2:

```
# UBCSAT default output:
#    'ubcsat -r out null' to suppress, 'ubcsat -hc' for customization help
#
#
# Output Columns: |run|found|best|beststep|steps|
#
# run: Run Number
# found: Target Solution Quality Found? (1 => yes)
# best: Best (Lowest) # of False Clauses Found
# beststep: Step of Best (Lowest) # of False Clauses Found
# steps: Total Number of Search Steps
#
#       F  Best       Step      Total
#   Run N Sol'n         of     Search
#   No. D Found       Best      Steps
#
      1 0     1          8     100000
# No Solution found for -target 0


Variables = 50
```

Étape 4 : Simulation de l'inférence

Notre programme simule l'inférence dans une base de connaissances en utilisant le raisonnement par l'absurde. L'idée est de tester si $BC \models \phi$ en vérifiant si $BC \cup \{\neg\phi\} \models \bot$ (c'est-à-dire si cela dérive une contradiction). On utilise un **solveur SAT** pour déterminer si la base de connaissances avec la négation de la formule est satisfiable. Une base de connaissances est initialisée sous forme CNF et permet d'ajouter des **clauses**, qui sont des listes de littéraux (entiers non nuls), ou des **formules** complètes déjà sous forme CNF. Pour tester si une base de connaissances infère une formule, on crée une copie de cette base, puis on y ajoute la **négation** de la formule à tester : si la formule est une clause comme $(a \vee b \vee c)$, sa négation est représentée par $(\neg a \wedge \neg b \wedge \neg c)$, donc chaque littéral est ajouté comme **clause unitaire** négée. La base étendue est ensuite convertie en CNF pour le solveur SAT. Si la formule est **non satisfiable**, cela signifie que la base de connaissances **infère** la formule. Enfin, un exemple montre l'utilisation de l'algorithme avec des littéraux $A = 1$, $B = 2$, $C = 3$, et teste si la base infère certaines formules comme $(A \vee C)$ ou C.

Code source :

```python
from pysat.formula import CNF

from pysat.solvers import Solver



class KnowledgeBase:

    def __init__(self):

        self.clauses = []
```

```
    def add_clause(self, clause):
        self.clauses.append(clause)

    def add_formula(self, formula):
        for clause in formula:
            self.add_clause(clause)

    def to_cnf(self):
        cnf = CNF()
        for clause in self.clauses:
            cnf.append(clause)
        return cnf

def is_satisfiable(cnf):
    solver = Solver(name='g3')
    solver.append_formula(cnf)
    result = solver.solve()
    solver.delete()
    return result

def infers(kb, formula):
    kb_copy = KnowledgeBase()
    for clause in kb.clauses:
        kb_copy.add_clause(clause)

    if isinstance(formula, list):
        for literal in formula:
            kb_copy.add_clause([-literal])
    else:
        kb_copy.add_clause([-formula])
```

```
    cnf = kb_copy.to_cnf()

    satisfiable = is_satisfiable(cnf)

    return not satisfiable


def main():

    kb = KnowledgeBase()

    kb.add_clause([1, 2])

    kb.add_clause([-2, 3])


    formula = [1, 3]

    result = infers(kb, formula)

    print(f"La base de connaissances infère-t-elle (A ∨ C) ? {result}")


    result = infers(kb, 3)

    print(f"La base de connaissances infère-t-elle C ? {result}")


if __name__ == "__main__":

    main()
```

Résultat de l'exécution du programme :

```
La base de connaissances infère-t-elle (A ∨ C) ? True
La base de connaissances infère-t-elle C ? False
```

Chapitre 2 : TP2 - Logique des prédicats

La logique des prédicats, ou logique du premier ordre, est un système puissant pour représenter et raisonner sur des assertions complexes. Dans ce TP, nous avons utilisé une bibliothèque Java pour explorer ses concepts fondamentaux.

Exemple 1 : Introduire une base de croyance

Nous avons créé un programme Java qui utilise la bibliothèque Tweety pour modéliser une base de croyances en logique du premier ordre. Nous commençons par définir une signature contenant deux sortes (animal et plant), trois constantes (anna, bob, emma), deux prédicats (flies et eats) et un foncteur (fatherOf). Nous créons ensuite une base de croyances (FolBeliefSet) et y ajoute plusieurs formules logiques exprimant des relations comme : si un animal vole alors il mange quelque chose, ou s'il vole alors son père aussi. La base contient aussi des faits comme "Anna vole" et "Bob mange Emma". Le contenu de la base est affiché à la console, puis sauvegardé dans un fichier .fol et relu depuis ce fichier pour être affiché à nouveau.

Code source:

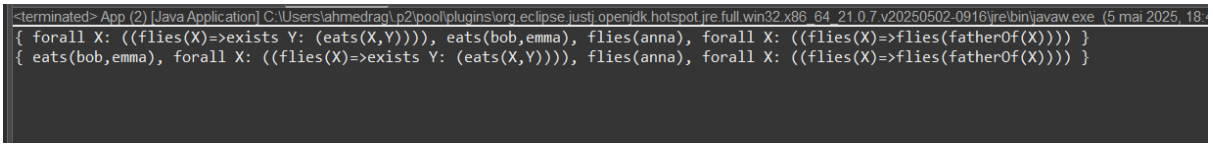
```
package defaultlogique;


import java.io.FileWriter;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;


import org.tweetyproject.logics.fol.parser.FolParser;

import org.tweetyproject.logics.fol.reasoner.FolReasoner;

import org.tweetyproject.logics.fol.syntax.FolBeliefSet;

import org.tweetyproject.logics.fol.syntax.FolFormula;

import org.tweetyproject.logics.fol.syntax.FolSignature;

import org.tweetyproject.logics.fol.writer.StandardFolWriter;

import org.tweetyproject.commons.ParserException;
```

```java
import org.tweetyproject.commons.Signature;

import org.tweetyproject.logics.commons.syntax.Constant;

import org.tweetyproject.logics.commons.syntax.Predicate;

import org.tweetyproject.logics.commons.syntax.Sort;

import org.tweetyproject.logics.commons.syntax.Functor;



public class App {

    public static void main(String[] args) throws ParserException,
IOException {

        String filename = "src\\main\\resources\\test.fol";



        FolSignature sig = new FolSignature();

        Sort animal = new Sort("animal");

        sig.add(animal);

        Constant anna = new Constant("anna", animal);

        sig.add(anna);

        Constant bob = new Constant("bob", animal);

        sig.add(bob);

        Sort plant = new Sort("plant");

        sig.add(plant);

        Constant emma = new Constant("emma", plant);

        sig.add(emma);



        List<Sort> l = new ArrayList<Sort>();

        l.add(animal);

        Predicate flies = new Predicate("flies", l);

        sig.add(flies);



        l = new ArrayList<Sort>();

        l.add(animal);
```

```
        l.add(plant);

        Predicate eats = new Predicate("eats", l);

        sig.add(eats);



        l = new ArrayList<Sort>();

        l.add(animal);

        Functor fatherOf = new Functor("fatherOf", l, animal);

        sig.add(fatherOf);



        FolBeliefSet b = new FolBeliefSet();

        b.setSignature(sig);

        FolParser parser = new FolParser();

        parser.setSignature(sig);



        b.add(parser.parseFormula("forall X:(flies(X) => (exists
Y:(eats(X,Y))))"));

        b.add(parser.parseFormula("forall X:(flies(X) =>
flies(fatherOf(X)))"));

        b.add(parser.parseFormula("flies(anna)"));

        b.add(parser.parseFormula("eats(bob,emma)"));



        System.out.println(b);



        StandardFolWriter writer = new StandardFolWriter(new
FileWriter(filename));

        writer.printBase(b);

        writer.close();



        parser = new FolParser();

        System.out.println(parser.parseBeliefBaseFromFile(filename));

    }
```

```
}
```

Résultat de l'affichage de la base de croyance:



Exemple 2 : L'inférence

Le programme Java qu'on a écrit utilise la bibliothèque *TweetyProject* pour illustrer comment créer, parser et raisonner sur des formules de logique du premier ordre. Il commence par définir une signature logique incluant un sort nommé Animal, deux constantes penguin et kiwi, et deux prédicats : Flies à un argument et Knows à deux arguments. Ensuite, il construit un ensemble de croyances (FolBeliefSet) en y ajoutant des formules logiques telles que !Flies(kiwi) et penguin /== kiwi, en utilisant un parseur configuré avec la signature précédente. Puis, il étend la signature en ajoutant une nouvelle constante archaeopteryx et affiche la signature minimale requise par l'ensemble de croyances. Enfin, il utilise un raisonneur simple (SimpleFolReasoner) pour vérifier si certaines formules sont déductibles à partir de cet ensemble de croyances, telles que Flies(kiwi), kiwi == kiwi, ou encore des formules quantifiées avec des différences d'identité.

code source:

```
package defaultlogique;


import java.io.IOException;

import java.util.ArrayList;

import java.util.List;


import org.tweetyproject.commons.ParserException;

import org.tweetyproject.logics.commons.syntax.Constant;

import org.tweetyproject.logics.commons.syntax.Predicate;

import org.tweetyproject.logics.commons.syntax.Sort;

import org.tweetyproject.logics.fol.parser.FolParser;
```

```
import org.tweetyproject.logics.fol.reasoner.FolReasoner;

import org.tweetyproject.logics.fol.reasoner.SimpleFolReasoner;

import org.tweetyproject.logics.fol.syntax.FolBeliefSet;

import org.tweetyproject.logics.fol.syntax.FolFormula;

import org.tweetyproject.logics.fol.syntax.FolSignature;


public class App {

    public static void main(String[] args) throws ParserException,
IOException {

        FolSignature sig = new FolSignature(true);


        Sort sortAnimal = new Sort("Animal");

        sig.add(sortAnimal);


        Constant constantPenguin = new Constant("penguin", sortAnimal);

        Constant constantKiwi = new Constant("kiwi", sortAnimal);

        sig.add(constantPenguin, constantKiwi);


        List<Sort> predicateList = new ArrayList<>();

        predicateList.add(sortAnimal);

        Predicate p = new Predicate("Flies", predicateList);


        List<Sort> predicateList2 = new ArrayList<>();

        predicateList2.add(sortAnimal);

        predicateList2.add(sortAnimal);

        Predicate p2 = new Predicate("Knows", predicateList2);


        sig.add(p, p2);

        System.out.println("Signature: " + sig);
```

```
        FolParser parser = new FolParser();

        parser.setSignature(sig);


        FolBeliefSet bs = new FolBeliefSet();

        bs.add(

            (FolFormula) parser.parseFormula("!Flies(kiwi)"),

            (FolFormula) parser.parseFormula("Flies(penguin)"),

            (FolFormula) parser.parseFormula("!Knows(penguin,kiwi)"),

            (FolFormula) parser.parseFormula("/==(penguin,kiwi)"),

            (FolFormula) parser.parseFormula("kiwi == kiwi")

        );


        System.out.println("\nParsed BeliefBase: " + bs);


        FolSignature sigLarger = bs.getSignature();

        sigLarger.add(new Constant("archaeopteryx", sortAnimal));

        bs.setSignature(sigLarger);


        System.out.println(bs);

        System.out.println("Minimal signature: " +
bs.getMinimalSignature());


        FolReasoner.setDefaultReasoner(new SimpleFolReasoner());

        FolReasoner prover = FolReasoner.getDefaultReasoner();


        System.out.println("ANSWER 1: " + prover.query(bs, (FolFormula)
parser.parseFormula("Flies(kiwi)")));

        System.out.println("ANSWER 2: " + prover.query(bs, (FolFormula)
parser.parseFormula("forall X: (exists Y: (Flies(X) && Flies(Y) &&
X/==Y))")));

        System.out.println("ANSWER 3: " + prover.query(bs, (FolFormula)
parser.parseFormula("kiwi == kiwi")));
```

```
        System.out.println("ANSWER 4: " + prover.query(bs, (FolFormula)
parser.parseFormula("kiwi /== kiwi")));

        System.out.println("ANSWER 5: " + prover.query(bs, (FolFormula)
parser.parseFormula("penguin /== kiwi")));

    }

}
```

Résultat des requêtes:

```
Signature: [_Any = {}, Animal = {kiwi, penguin}], [Knows(Animal,Animal), ==(_Any,_Any), /==(_Any,_Any), Flies(Animal)], []

Parsed BeliefBase: { !Knows(penguin,kiwi), (kiwi==kiwi), !Flies(kiwi), (penguin/==kiwi), Flies(penguin) }
{ !Knows(penguin,kiwi), (kiwi==kiwi), !Flies(kiwi), (penguin/==kiwi), Flies(penguin) }
Minimal signature: [_Any = {}, Animal = {kiwi, penguin}], [Knows(Animal,Animal), ==(_Any,_Any), /==(_Any,_Any), Flies(Animal)], []
ANSWER 1: false
ANSWER 2: false
ANSWER 3: true
ANSWER 4: false
ANSWER 5: true
```

Chapitre 3 : TP3 - Logique modale

La logique modale enrichit la logique classique avec des notions comme la nécessité et la possibilité. Dans ce TP, nous avons étudié ses principes et ses applications pratiques.

Exemple 1

Nous utilisons la bibliothèque TweetyProject en Java pour parser et raisonner sur des bases de croyances en logique modale**.** Nous commençons par lire une base de croyances (b1) depuis un fichier .mlogic, puis analysons une formule modale <>(A&&B). Ensuite, nous construisons une autre base de croyances (b2) à partir d'une chaîne de caractères qui décrit des individus (penguin, eagle), un prédicat Flies, et une assertion (Flies(eagle)), puis nous analysons une formule tautologique portant sur le vol du pingouin. Nous chargeons également une troisième base (b3) depuis un autre fichier et affichons sa signature minimale. Enfin, nous utilisons un raisonneur modal simple (SimpleMlReasoner) pour interroger les bases b1 et b2 avec leurs formules respectives, afin de vérifier si ces formules sont déductibles à partir des croyances définies.

Code source:

```
package defaultlogique;



import java.io.IOException;

import org.tweetyproject.commons.ParserException;

import org.tweetyproject.logics.ml.reasoner.SimpleMlReasoner;

import org.tweetyproject.logics.ml.syntax.MlBeliefSet;

import org.tweetyproject.logics.fol.syntax.FolFormula;

import org.tweetyproject.logics.ml.parser.MlParser;



public class App {

    public static void main(String[] args) throws ParserException,
IOException {

        MlParser parser = new MlParser();
```

```
        MlBeliefSet b1 =
parser.parseBeliefBaseFromFile("src/main/resources/examplebeliefbase2.m
logic");

        FolFormula f1 = (FolFormula) parser.parseFormula("<>(A&&B)");

        System.out.println("Parsed formula:" + f1);



        parser = new MlParser();

        MlBeliefSet b2 = parser.parseBeliefBase("Animal =
{penguin,eagle} \n type(Flies(Animal)) \n (Flies(eagle))");

        FolFormula f2 = (FolFormula)
parser.parseFormula("(Flies(penguin)) || (!(Flies(penguin)))");

        System.out.println("Parsed belief base:" + b2);

        System.out.println("Parsed formula:" + f2);



        parser = new MlParser();

        MlBeliefSet b3 =
parser.parseBeliefBaseFromFile("src/main/resources/examplebeliefbase.ml
ogic");

        System.out.println("Parsed belief base:" + b3 + "\nSignature of
belief base:" + b3.getMinimalSignature());



        SimpleMlReasoner reasoner = new SimpleMlReasoner();

        System.out.println("Answer to query: " + reasoner.query(b1,
f1));

        System.out.println("Answer to query: " + reasoner.query(b2,
f2));

    }

}
```

Résultat des requêtes modales:

```
Parsed formula:<>(A&&B)
Parsed belief base:{ Flies(eagle) }
Parsed formula:Flies(penguin)||!Flies(penguin)
Parsed belief base:{ (Knows(alice,bob)=>[](Knows(alice,bob))), [](Knows(alice,bob)), Knows(alice,bob), Knows(bob,charlie), Likes(alice,charlie), <>(Likes(bob,alice)), <>(forall X: ((Knows(X,alice)=>Likes(a
Signature of belief base:[Person = {charlie, alice, bob}], [Knows(Person,Person), Likes(Person,Person)], []
Answer to query: true
Answer to query: true
```

Chapitre 4 : TP4 - Logique des défauts

La logique des défauts permet de raisonner malgré des informations partielles en appliquant des règles par défaut. Nous avons abordé ses bases à travers des exemples pratiques.

Outil :

Nous avons utilisé **DLV** pour représenter un système de raisonnement non-monotone combinant des connaissances de base, des règles de défaut et des contraintes de cohérence, afin de générer des extensions stables reflétant des hypothèses sous incertitude. Le **premier fichier .dlv** contient des règles de défaut générales utilisées pour raisonner en l'absence d'information complète. Il inclut des règles comme a :- not -a et b :- a, not -b qui permettent d'accepter une conclusion par défaut, ainsi qu'une contrainte de cohérence (:- a, -a.) empêchant qu'une proposition et sa négation soient vraies en même temps. Le **deuxième fichier .dlv** modélise un monde de connaissances (W) avec des implications classiques (p → q, p → r, etc.) et un ensemble de règles de défaut spécifiques (D) qui expriment des hypothèses raisonnables sous incertitude. Il inclut également des contraintes supplémentaires pour garantir la cohérence logique des solutions générées.

Commande:

```
C:\Users\ahmedrag\Downloads\Logique des defauts>dlv -silent theorie_1.dlv
```

Exécution:

```
C:\Users\ahmedrag\Downloads\Logique des defauts>dlv -silent theorie_1.dlv
{b, a}
{-a}
```

Commande 2 :

```
C:\Users\ahmedrag\Downloads\Logique des defauts>dlv -silent theorie_2.dlv
```

Exécution:

```
C:\Users\ahmedrag\Downloads\Logique des defauts>dlv -silent theorie_2.dlv
{q, p, r, -s, u, v}
{q, p, r, -s, -v}
```

On a utiliser le raisonneur dlv sur l'exo 3 sur la theorie Delta et Delta prime également :

Exercice 3 : (non monotonie du raisonnement par défaut)
Quelles sont les extensions des théories $\Delta=<W,D>$ et $\Delta'=<W',D>$ telles que ;
$W=\{A,B\}$,
$W'=\{A,B,C\}$ et
$D=\{A \wedge B : \neg C / \neg C\}$.

pour Delta : le fichier.dlv est comme suit :

% Fichier DLV pour la théorie Delta = <W, D>

% W = {A, B}

% D = { A /\ B : ~C / ~C }

% Base de connaissances (W)

a.

b.

% Règle par défaut (D)

% La règle est "Si A et B sont vrais, ET si C n'est PAS prouvé (not c),

% ALORS nous pouvons conclure not_c (pour représenter ~C)."

% 'not c' en DLV est la négation par l'échec, signifiant 'c' n'est pas dans le modèle stable.

not_c :- a, b, not c.

Exécution :

```
C:\Users\ahmedrag\Downloads\Logique des defauts>dlv -silent exo3.dlv
{a, b, not_c}
```

Pour Delta prime : le fichier.dlv est comme suit :

% Fichier DLV pour la théorie Delta' = <W', D>

% W' = {A, B, C}

% D = { A /\ B : ~C / ~C }

% Base de connaissances (W')

a.

b.

c. % Ajout du fait C

% Règle par défaut (D) - La même règle que précédemment

% "Si A et B sont vrais, ET si C n'est PAS prouvé (not c),

% ALORS nous pouvons conclure not_c (pour représenter ~C)."

not_c :- a, b, not c.

Exécution :

```
C:\Users\ahmedrag\Downloads\Logique des defauts>dlv -silent exo3.dlv
{a, b, c}
```

Chapitre 5 : TP5 - Les Réseaux Sémantiques

Les réseaux sémantiques sont des structures graphiques représentant les connaissances. Dans ce TP, nous avons travaillé sur leur mise en œuvre avec des outils en Python.

Modélisation du réseau:

Nous avons créé une classe SemanticNetwork pour modéliser un réseau sémantique où nous stockons des nœuds et des liens entre ces nœuds. Nous maintenons un dictionnaire nodes qui associe chaque nœud à ses propriétés, ainsi qu'un dictionnaire links qui garde en mémoire les relations entre nœuds sous forme de triplets (source, relation, cible) avec un type de lien qui peut être "normal" ou "exception". Nous pouvons ajouter des nœuds avec leurs propriétés, créer des liens généraux ou spécifiques de type "is-a" (est-un), et ajouter des propriétés à un nœud existant ou nouvellement créé. Nous sommes capables de récupérer la valeur d'une propriété pour un nœud donné, ainsi que tous les liens sortant d'un nœud ou entrant vers un nœud, avec la possibilité de filtrer selon le type de relation. Pour visualiser notre réseau, nous affichons simplement la liste des nœuds avec leurs propriétés et les liens entre eux, en indiquant clairement si un lien est une exception. Cette structure nous permet de modéliser et d'explorer les relations sémantiques entre concepts avec des propriétés et des hiérarchies précises.

Code source :

```
class SemanticNetwork:
    def __init__(self):
        self.nodes = {}
        self.links = {}

    def add_node(self, node_name, properties=None):
        if properties is None:
            properties = {}
        self.nodes[node_name] = properties
        return self

    def add_link(self, source, relation, target, link_type="normal"):
        self.links[(source, relation, target)] = link_type
        return self

    def add_is_a_link(self, source, target, link_type="normal"):
        return self.add_link(source, "is-a", target, link_type)

    def add_property(self, node_name, property_name, property_value):
        if node_name not in self.nodes:
```

```
            self.add_node(node_name)
        self.nodes[node_name][property_name] = property_value
        return self

    def get_property(self, node_name, property_name):
        if node_name in self.nodes and property_name in
self.nodes[node_name]:
            return self.nodes[node_name][property_name]
        return None

    def get_links_from(self, source, relation=None):
        result = []
        for (src, rel, target), link_type in self.links.items():
            if src == source and (relation is None or rel == relation):
                result.append((rel, target, link_type))
        return result

    def get_links_to(self, target, relation=None):
        result = []
        for (src, rel, tgt), link_type in self.links.items():
            if tgt == target and (relation is None or rel == relation):
                result.append((src, rel, link_type))
        return result

    def visualize(self):
        print("Nœuds:")
        for node, props in self.nodes.items():
            print(f"  {node}: {props}")

        print("\nLiens:")
        for (src, rel, tgt), link_type in self.links.items():
            exception_mark = " [EXCEPTION]" if link_type == "exception"
else ""
            print(f"  {src} --({rel})--> {tgt}{exception_mark}")
```

Exemple d'utilisation

Le code inclut un exemple d'utilisation avec un réseau sémantique représentant des animaux (mammifères, oiseaux) avec leurs propriétés et une exception (les pingouins ne peuvent pas voler bien qu'ils soient des oiseaux).

Code Source :

```
if __name__ == "__main__":

    network = SemanticNetwork()



    network.add_node("Animal", {"vivant": True})
```

```
network.add_node("Oiseau")
network.add_node("Pingouin")
network.add_node("Aigle")
network.add_node("Mammifère")
network.add_node("Chien")
network.add_node("Labrador")

network.add_is_a_link("Oiseau", "Animal")
network.add_is_a_link("Mammifère", "Animal")
network.add_is_a_link("Pingouin", "Oiseau")
network.add_is_a_link("Aigle", "Oiseau")
network.add_is_a_link("Chien", "Mammifère")
network.add_is_a_link("Labrador", "Chien")

network.add_property("Oiseau", "peut_voler", True)
network.add_property("Oiseau", "a_des_plumes", True)
network.add_property("Mammifère", "allaite", True)
network.add_property("Mammifère", "a_des_poils", True)

network.add_link("Pingouin", "peut_voler", True, "exception")
network.add_property("Pingouin", "peut_voler", False)

print("Réseau sémantique:")
network.visualize()
```

Partie 1 : Implémenter l'algorithme de propagation de marqueurs

Nous avons implémenté un algorithme de propagation de marqueurs dans notre réseau sémantique qui, à partir d'une liste de nœuds marqués initialement, explore récursivement les liens selon certaines relations spécifiées tout en évitant les boucles

infinies grâce à une profondeur maximale. Notre méthode garde la trace de tous les chemins possibles vers les nœuds atteints, en excluant les nœuds initiaux pour ne conserver que ceux atteints par propagation. Ensuite, nous avons notre fonction d'interrogation du réseau qui, en utilisant notre algorithme de propagation, permet de rechercher tous les chemins possibles entre les nœuds sources marqués et, si demandé, vers un nœud cible précis, en suivant les relations spécifiées ou toutes les relations disponibles par défaut. Ainsi, nous pouvons analyser la structure et les relations de notre réseau pour en extraire les chemins pertinents.

Code source:

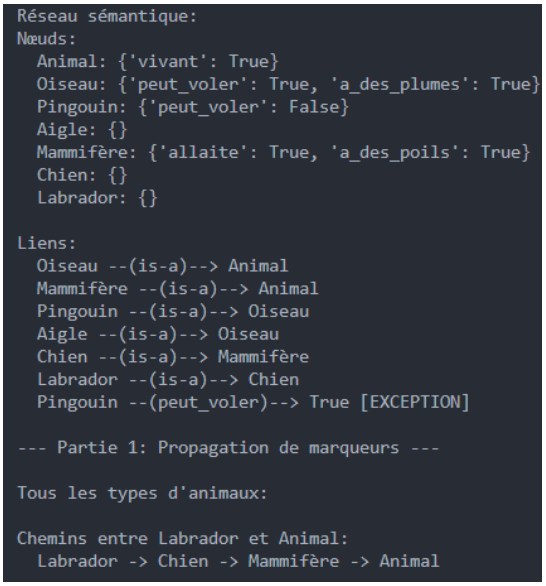
```
def marker_propagation(network, marked_nodes, relations, max_depth=10):
    reached_nodes = {node: [[node]] for node in marked_nodes}
    queue = [(node, 0) for node in marked_nodes]
    while queue:
        current_node, depth = queue.pop(0)
        if depth >= max_depth:
            continue
        for relation, target, link_type in
network.get_links_from(current_node):
            if relation in relations and link_type == "normal":
                if target not in reached_nodes:
                    reached_nodes[target] = []
                    queue.append((target, depth + 1))
                for path in reached_nodes[current_node]:
                    new_path = path + [target]
                    if new_path not in reached_nodes[target]:
                        reached_nodes[target].append(new_path)
    for node in marked_nodes:
        if node in reached_nodes:
            del reached_nodes[node]
    return reached_nodes
```

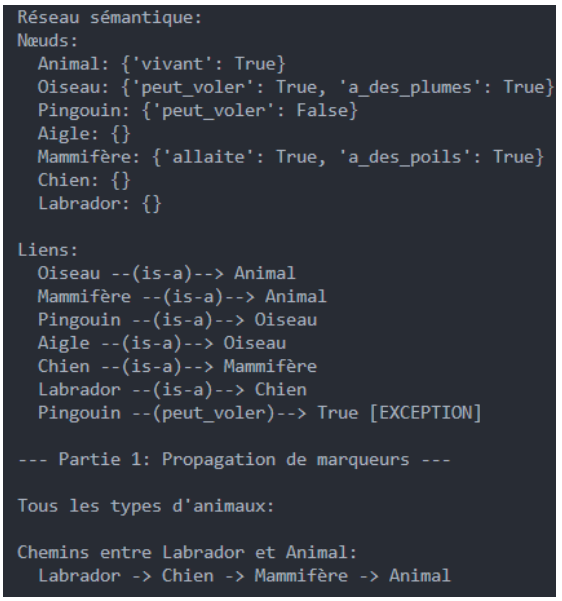
```
def query_network(network, source_nodes, target_node=None,
relations=None):
    if relations is None:
        relations = set()
        for src, rel, tgt in network.links.keys():
            relations.add(rel)
        relations = list(relations)
    result = marker_propagation(network, source_nodes, relations)
    if target_node is not None:
        if target_node in result:
            return result[target_node]
        else:
            return []
    return result
```

Appel de l'algorithme:

```
print("\n--- Partie 1: Propagation de marqueurs ---")
# Exemple 1: Quels sont tous les types d'animaux?
result1 = query_network(network, ["Animal"], relations=["is-a"])
print("\nTous les types d'animaux:")
for node, paths in result1.items():
    print(f"  {node}:")
    for path in paths:
        print(f"    {' -> '.join(path)}")

# Exemple 2: Trouve le chemin entre Labrador et Animal
result2 = query_network(network, ["Labrador"], "Animal", ["is-a"])
print("\nChemins entre Labrador et Animal:")
if result2:
    for path in result2:
        print(f"  {' -> '.join(path)}")
else:
    print("  Aucun chemin trouvé")
```

Captures d'exécution :





Partie 2 : Implémenter l'algorithme d'héritage

Nous avons conçu une fonction qui, dans notre réseau sémantique, déduit toutes les propriétés d'un nœud en héritant celles de ses parents via les liens « is-a ». Nous récupérons d'abord les propriétés directes du nœud, puis, pour chaque parent connecté par un lien normal « is-a », nous appelons récursivement notre fonction pour récupérer ses propriétés et les intégrer à celles du nœud, sans écraser les propriétés déjà définies localement. Ensuite, nous avons notre fonction de saturation du réseau qui crée une copie de notre réseau initial, puis pour chaque nœud, elle calcule toutes les propriétés héritées grâce à notre fonction précédente et les ajoute au nœud dans ce nouveau réseau saturé. Cela nous permet ainsi d'avoir un réseau enrichi où chaque nœud contient l'ensemble complet de ses propriétés directes et héritées.

Code source :

```
def inherit_properties(network, node_name):

    if node_name not in network.nodes:

        return {}

    properties = network.nodes[node_name].copy()

    is_a_links = []

    for relation, target, link_type in
network.get_links_from(node_name):
```

```
        if relation == "is-a" and link_type == "normal":
            is_a_links.append(target)
    for parent in is_a_links:
        parent_properties = inherit_properties(network, parent)
        for prop, value in parent_properties.items():
            if prop not in properties:
                properties[prop] = value
    return properties
def saturate_network(network):
    saturated_network = SemanticNetwork()
    for node, props in network.nodes.items():
        saturated_network.add_node(node, props.copy())
    for (src, rel, tgt), link_type in network.links.items():
        saturated_network.add_link(src, rel, tgt, link_type)
    for node in network.nodes:
        inherited_props = inherit_properties(network, node)
        for prop, value in inherited_props.items():
            saturated_network.add_property(node, prop, value)
    return saturated_network
```

Appel de l'algorithme:

```
print("\n--- Partie 2: Héritage de propriétés ---")
# Hérite les propriétés pour Labrador
lab_props = inherit_properties(network, "Labrador")
print("\nPropriétés héritées pour Labrador:")
for prop, value in lab_props.items():
    print(f"  {prop}: {value}")

# Hérite les propriétés pour Pingouin
pingouin_props = inherit_properties(network, "Pingouin")
print("\nPropriétés héritées pour Pingouin (sans gestion des exceptions):")
for prop, value in pingouin_props.items():
    print(f"  {prop}: {value}")
```

Capture d'exécution:

```
--- Partie 2: Héritage de propriétés ---

Propriétés héritées pour Labrador:
  allaite: True
  a_des_poils: True
  vivant: True

Propriétés héritées pour Pingouin (sans gestion des exceptions):


Propriétés héritées pour Pingouin (sans gestion des exceptions):
  peut_voler: False
  a_des_plumes: True
  vivant: True
```

Partie 3 : Implémenter l'algorithme d'exception

Nous avons ici notre algorithme de propagation de marqueurs adapté à notre réseau sémantique, prenant en compte les exceptions. Lors de la propagation, nous vérifions si un lien est marqué comme une exception, ce qui bloque la propagation de ce chemin. Nous vérifions également si des exceptions sont héritées via des relations « is-a » entre nœuds. Cela nous permet de ne pas propager certaines propriétés ou relations si une exception est présente, ce qui correspond à notre gestion fine des règles dans le réseau.

Ensuite, notre fonction d'héritage des propriétés est adaptée pour gérer les exceptions. Nous identifions les propriétés du nœud qui sont explicitement exclues de l'héritage via des liens d'exception. Lors de la collecte des propriétés héritées de nos parents, nous excluons celles qui sont dans notre liste d'exceptions pour ne pas les intégrer au nœud courant. Grâce à ces mécanismes, nous pouvons modéliser notre réseau sémantique avec des règles d'héritage et des exceptions précises, ce qui reflète mieux la complexité du monde réel.

Code source:

```
def marker_propagation_with_exceptions(network, marked_nodes,
relations, max_depth=10):

    reached_nodes = {node: [[node]] for node in marked_nodes}
```

```
    queue = [(node, 0) for node in marked_nodes]

    exception_links = {(src, rel, tgt) for (src, rel, tgt), link_type
in network.links.items() if link_type == "exception"}



    while queue:

        current_node, depth = queue.pop(0)

        if depth >= max_depth:

            continue

        for relation, target, link_type in
network.get_links_from(current_node):

            if relation in relations:

                blocked = False

                if (current_node, relation, target) in exception_links:

                    blocked = True

                for is_a_source, _, parent in network.links.items():

                    if is_a_source[0] == current_node and
is_a_source[1] == "is-a":

                        if (is_a_source[2], relation, target) in
exception_links:

                            blocked = True

                            break

                if not blocked:

                    if target not in reached_nodes:

                        reached_nodes[target] = []

                        queue.append((target, depth + 1))

                    for path in reached_nodes[current_node]:

                        new_path = path + [target]

                        if new_path not in reached_nodes[target]:

                            reached_nodes[target].append(new_path)



    for node in marked_nodes:
```

```
        if node in reached_nodes:

            del reached_nodes[node]

    return reached_nodes




def inherit_properties_with_exceptions(network, node_name):

    if node_name not in network.nodes:

        return {}

    properties = network.nodes[node_name].copy()

    exception_properties = set()

    for (src, rel, tgt), link_type in network.links.items():

        if src == node_name and link_type == "exception" and
rel.startswith("has_"):

            exception_properties.add(rel[4:])

    is_a_links = []

    for relation, target, link_type in
network.get_links_from(node_name):

        if relation == "is-a" and link_type == "normal":

            is_a_links.append(target)

    for parent in is_a_links:

        parent_properties = inherit_properties_with_exceptions(network,
parent)

        for prop, value in parent_properties.items():

            if prop not in properties and prop not in
exception_properties:

                properties[prop] = value

    return properties
```

Appel de l’algorithme:

```python
print("\n--- Partie 3: Gestion des exceptions ---")
# Hérite les propriétés pour Pingouin avec gestion des exceptions
pingouin_props_exc = inherit_properties_with_exceptions(network, "Pingouin")
print("\nPropriétés héritées pour Pingouin (avec gestion des exceptions):")
for prop, value in pingouin_props_exc.items():
    print(f"  {prop}: {value}")

# Exemple de propagation avec exceptions
print("\nPropagation avec exceptions:")
result_exc = marker_propagation_with_exceptions(network, ["Animal"], ["is-a"])
for node, paths in result_exc.items():
    print(f"  {node}:")
    for path in paths:
        print(f"    {' -> '.join(path)}")
```

Capture d’exécution:

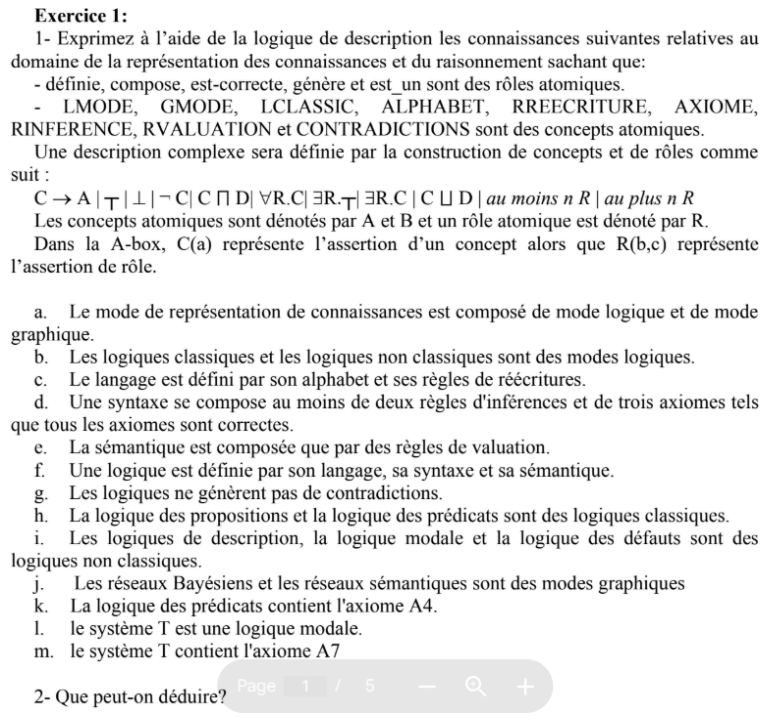
```
--- Partie 3: Gestion des exceptions ---

Propriétés héritées pour Pingouin (avec gestion des exceptions):
  peut_voler: False
  a_des_plumes: True
  vivant: True

Propagation avec exceptions:
```

Chapitre 6 : TP6 - Les Logiques de description

Les logiques de description structurent les connaissances, notamment dans les ontologies. Nous avons utilisé un outil dédié pour explorer ce domaine.

TBox et ABox

Celles de l'exo 1 dans la série TD.



Exercice 1:

1- Exprimez à l'aide de la logique de description les connaissances suivantes relatives au domaine de la représentation des connaissances et du raisonnement sachant que:

- définie, compose, est-correcte, génère et est_un sont des rôles atomiques.
- LMODE, GMODE, LCLASSIC, ALPHABET, RREECRITURE, AXIOME, RINFERENCE, RVALUATION et CONTRADICTIONS sont des concepts atomiques.

Une description complexe sera définie par la construction de concepts et de rôles comme suit :

$C \rightarrow A \mid \top \mid \bot \mid \neg C \mid C \sqcap D \mid \forall R.C \mid \exists R.\top \mid \exists R.C \mid C \sqcup D \mid$ *au moins n R* | *au plus n R*

Les concepts atomiques sont dénotés par A et B et un rôle atomique est dénoté par R.

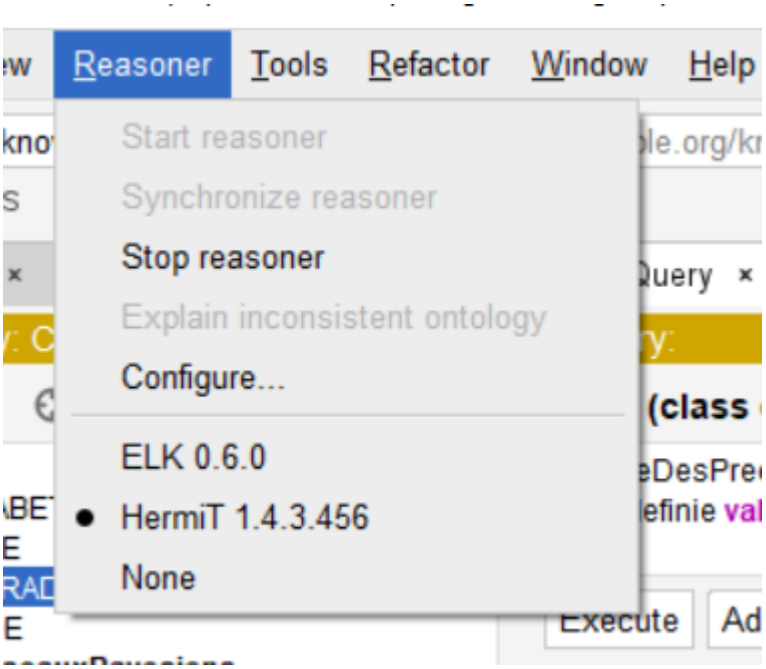
Dans la A-box, C(a) représente l'assertion d'un concept alors que R(b,c) représente l'assertion de rôle.

a. Le mode de représentation de connaissances est composé de mode logique et de mode graphique.
b. Les logiques classiques et les logiques non classiques sont des modes logiques.
c. Le langage est défini par son alphabet et ses règles de réécritures.
d. Une syntaxe se compose au moins de deux règles d'inférences et de trois axiomes tels que tous les axiomes sont correctes.
e. La sémantique est composée que par des règles de valuation.
f. Une logique est définie par son langage, sa syntaxe et sa sémantique.
g. Les logiques ne génèrent pas de contradictions.
h. La logique des propositions et la logique des prédicats sont des logiques classiques.
i. Les logiques de description, la logique modale et la logique des défauts sont des logiques non classiques.
j. Les réseaux Bayésiens et les réseaux sémantiques sont des modes graphiques
k. La logique des prédicats contient l'axiome A4.
l. le système T est une logique modale.
m. le système T contient l'axiome A7

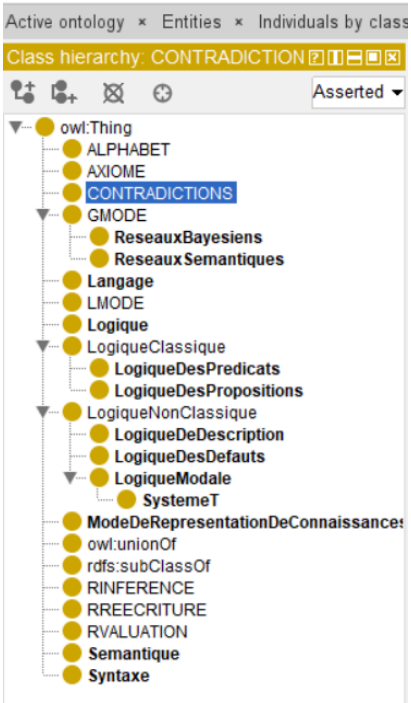
2- Que peut-on déduire?

Raisonneur utilisé:

nous avons utilisé le raisonneur **HermiT Reasoner** comme étant une integration dans un éditeur ontologique nommé "**protégé**".



On a construit un fichier exo1.owl pour démontrer le fonctionnement sur l'exo 1. il continent la TBox sous forme de classes , et la ABox sous forme d'instances , il sont affichés sous forme de hiérarchie.



Structure du fichier :

Syntaxe de Base des Déclarations OWL

Structure du Document

- **Déclaration XML**: <?xml version="1.0"?>
- **Racine RDF**: <rdf:RDF xmlns="..." xmlns:owl="..." xmlns:rdf="..." xmlns:rdfs="...">

Déclarations de Base

- **Ontologie**: <owl:Ontology rdf:about="URI"/>
- **Classe**: <owl:Class rdf:about="URI"/>
- **Sous-Classe**: <rdfs:subClassOf rdf:resource="URI"/>
- **Propriété d'Objet**: <owl:ObjectProperty rdf:about="URI"/>
- **Propriété de Type de Données**: <owl:DatatypeProperty rdf:about="URI"/>
- **Individu**: <owl:NamedIndividual rdf:about="URI">

Relations entre Classes

- **Disjointe Avec**: <owl:disjointWith rdf:resource="URI"/>
- **Classe Équivalente**: <owl:equivalentClass rdf:resource="URI"/>

Caractéristiques des Propriétés

- **Domaine**: <rdfs:domain rdf:resource="URI"/>
- **Codomaine**: <rdfs:range rdf:resource="URI"/>
- **Sous-Propriété**: <rdfs:subPropertyOf rdf:resource="URI"/>

Expressions de Classes

- **Union**: <owl:unionOf rdf:parseType="Collection">...</owl:unionOf>
- **Intersection**: <owl:intersectionOf rdf:parseType="Collection">...</owl:intersectionOf>
- **Complément**: <owl:complementOf rdf:resource="URI"/>

Restrictions

- **Quelques Valeurs**: <owl:Restriction><owl:onProperty rdf:resource="URI"/><owl:someValuesFrom rdf:resource="URI"/></owl:Restriction>
- **Toutes les Valeurs**: <owl:Restriction><owl:onProperty rdf:resource="URI"/><owl:allValuesFrom rdf:resource="URI"/></owl:Restriction>
- **A comme Valeur**: <owl:Restriction><owl:onProperty rdf:resource="URI"/><owl:hasValue rdf:datatype="...">valeur</owl:hasValue></owl:Restriction>
- **Cardinalité Minimale**: <owl:Restriction><owl:onProperty rdf:resource="URI"/><owl:minCardinality rdf:datatype="...">n</owl:minCardinality></owl:Restriction>
- **Cardinalité Maximale**: <owl:Restriction><owl:onProperty rdf:resource="URI"/><owl:maxCardinality rdf:datatype="...">n</owl:maxCardinality></owl:Restriction>
- **Cardinalité Exacte**: <owl:Restriction><owl:onProperty rdf:resource="URI"/><owl:cardinality rdf:datatype="...">n</owl:cardinality></owl:Restriction>

Définitions d'Instances

- **Type**: <rdf:type rdf:resource="URI"/>
- **Assertion de Propriété**: <nomPropriété rdf:resource="URI"/> ou <nomPropriété>valeur littérale</nomPropriété>

Démonstration par requêtes :

Syntaxe des requêtes :

- **Restriction des propriété :**

<property> some <Class>

<property> only <Class>

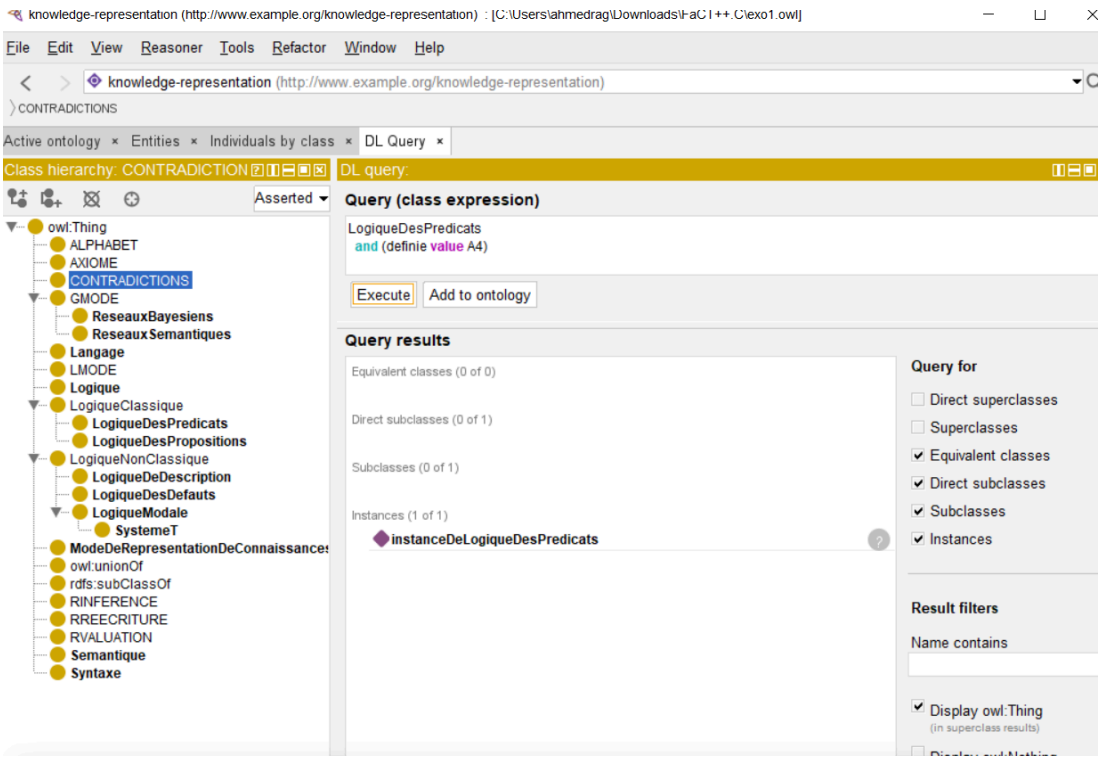
<property> min <n> <Class>

<property> max <n> <Class>

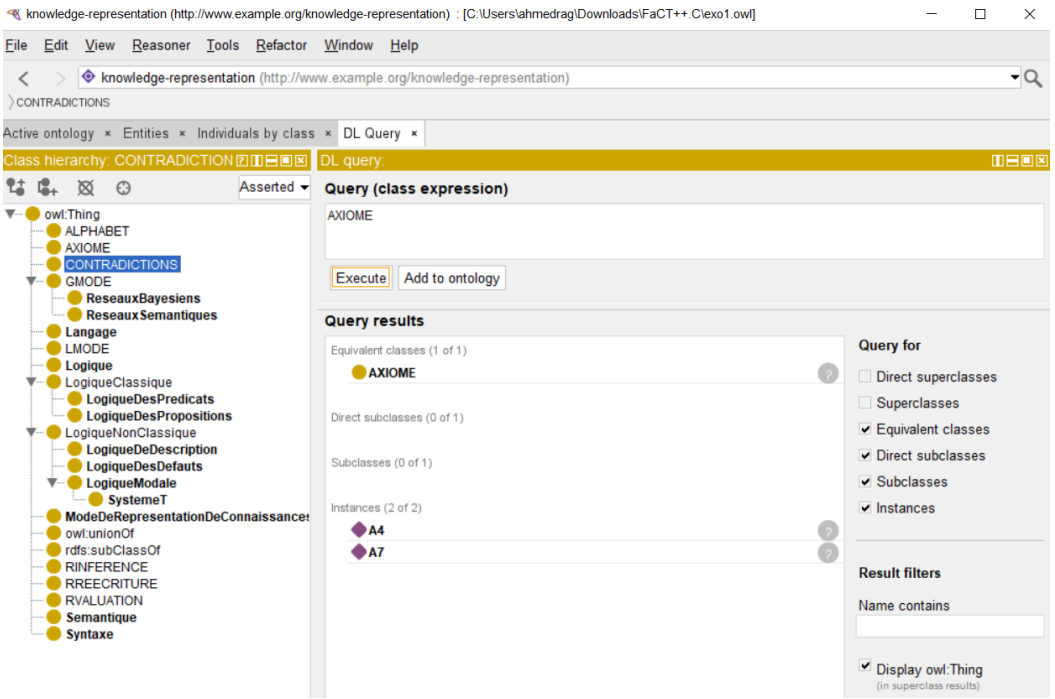
- **Individus :** <property> value <Individual>

on combine avec and, or, not pour construire des requêtes complexes.

Requête 1 : Trouver chaque individu de LogiqueDesPredicats défini par l'axiome A4.



Requête 2 : Listez tous les individus d'AXIOME



Conclusion et perspectives

Ces travaux pratiques nous ont offert une vue d’ensemble des logiques utilisées en informatique pour représenter et raisonner sur les connaissances. Nous avons appris à modéliser des situations réelles, à générer des conclusions logiques, à évaluer des assertions et à représenter visuellement des informations. Ces compétences constituent une fondation solide pour appliquer ces techniques dans divers contextes futurs.