



Rapport des Travaux

Pratiques - RCR1

Master 1 Systèmes Informatiques Intelligents

Enseignantes : Mme F. KHELLAF

Mme H. MOULAI

SAYOUD Maïssa

191931040670

BOULKABOUL Amira

202031043294

Mai 2024

Table des Matières

TP N° 1 : Inférence Logique Basée sur un Solveur SAT	3
Introduction	3
1. Etapes 1 et 2: Préparation de l'Environnement & Exécution du Solveur SAT	3
2. Etape 3 : Tester la Satisfiabilité.....	6
3. Etape 4 : Algorithme de Simulation de l'Inférence d'une Base de Connaissances.....	10
TP N° 2 : Logique du Premier Ordre	14
Introduction	14
1. Logique du Premier Ordre.....	14
2. Préparation de l'Environnement	14
3. Modélisation des connaissances en logique des prédicats	16
4. Implémentation de la modélisation des connaissances en logique des prédicats.....	16
5. Résultats	18
Conclusion	18
TP N° 3 : Logique Modale	20
Introduction	20
1. Définition	20
2. Implémentation avec Tweety	20
Résultats.....	22
Conclusion	23
TP N° 4 : Logique des Défauts.....	25
Introduction	25
1. Utilisation de la Librairie Orbital.....	25
2. Résultats	27
Conclusion	27
TP N° 5 : Réseaux Sémantiques.....	29

Introduction	29
1. Algorithme de propagation de marqueurs dans les réseaux sémantiques	29
2. Algorithme d'Héritage	31
3. Algorithme d'Exception	32
Conclusion	33
Conclusion Générale	34

TP N° 1 :Inférence Logique
Basée sur un Solveur SAT

TP N° 1 :Inférence Logique Basée sur un Solveur SAT

Introduction

Dans ce premier TP nous allons explorer l'application du raisonnement logique basé sur le solveur SAT. Nous aborderons les différentes étapes du processus, depuis la préparation de l'environnement jusqu'à la résolution du problème de satisfaction des contraintes et nous utiliserons des exemples concrets pour illustrer l'importance de modéliser avec précision un problème et d'évaluer sa satisfiabilité.

1. Etapes 1 et 2: Préparation de l'Environnement & Exécution du Solveur SAT

Tout d'abord, un répertoire appelé UBCSAT a été créé pour stocker les fichiers nécessaires au processus de résolution dont tous les fichiers essentiels y sont présents.

Après avoir préparé l'environnement, le solveur SAT a été lancé à l'aide de l'exécution de la commande suivante dans l'invite de commande :

```
C:\UBCSAT> ubcsat -alg saps -i test1.cnf -solve
```

Grâce à l'algorithme de résolution **saps**, le solveur SAT commence à chercher une solution au problème spécifié dans le fichier test1.cnf qui contient la base donnée dans l'exemple 1 du TP1.

```
C:\UBCSAT>ubcsat -alg saps -i test1.cnf -solve
# UBCSAT version 1.1.0 (Sea to Sky Release)
#
# http://www.satlib.org/ubcsat
#
# ubcsat -h for help
#
# -alg saps
# -runs 1
# -cutoff 100000
# -timeout 0
# -gtimeout 0
# -noimprove 0
# -target 0
# -wtargset 0
# -seed 188019076
# -solve 1
# -find,-numsol 1
# -findunique 0
# -srestart 0
# -prestart 0
# -drestart 0
#
# -alpha 1.3
# -rho 0.8
# -ps 0.05
# -wp 0.01
# -sapsthresh -0.1
#
# 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      2      2
#
# Solution found for -target 0
-1 2 -3 4 5

Variables = 5
Clauses = 9
TotalLiterals = 23
TotalCPUtimeElapsed = 0.001
FlipsPerSecond = 2000
RunsExecuted = 1
SuccessfulRuns = 1
PercentSuccess = 100.00
Steps_Mean = 2
Steps_CoeffVariance = 0
Steps_Median = 2
CPUtime_Mean = 0.000999927520752
CPUtime_CoeffVariance = 0
CPUtime_Median = 0.000999927520752
```

Figure 1 : Exécution du premier exemple satisfiable

À partir des résultats de l'exécution, nous pouvons observer que le solveur a trouvé une solution satisfaisante à l'étape 2 de la recherche qui est :

$$-1 \ 2 \ -3 \ 4 \ 5 \quad \neg a \wedge b \wedge \neg c \wedge d \wedge e$$

Cela confirme que le problème défini dans le fichier test1.cnf est satisfiable.

- Un deuxième exemple, **test2.cnf**, représente une autre base de connaissances :

```
C:\UBCSAT>ubcsat -alg saps -i test2.cnf -solve
#
# UBCSAT version 1.1.0 (Sea to Sky Release)
#
# http://www.satlib.org/ubcsat
#
# ubcsat -h for help
#
# -alg saps
# -runs 1
# -cutoff 100000
# -timeout 0
# -gtimeout 0
# -noimprove 0
# -target 0
# -wtarg 0
# -seed 189763678
# -solve 1
# -find,-numsol 1
# -findunique 0
# -srestart 0
# -prestart 0
# -drestart 0
#
# -alpha 1.3
# -rho 0.8
# -ps 0.05
# -wp 0.01
# -sapsthresh -0.1
#
# 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      2      100000
# No Solution found for -target 0
#
Variables = 5
Clauses = 12
TotalLiterals = 28
TotalCPUtimeElapsed = 0.004
FlipsPerSecond = 25000322
RunsExecuted = 1
SuccessfulRuns = 0
PercentSuccess = 0.00
Steps_Mean = 100000
Steps_CoeffVariance = 0
Steps_Median = 100000
CPUtime_Mean = 0.00399994850159
CPUtime_CoeffVariance = 0
CPUtime_Median = 0.00399994850159
```

Figure 2. Exécution du deuxième exemple insatisfiable

Après avoir exécuté le solveur SAT avec le deuxième fichier test2.cnf, nous avons remarqué qu'aucune solution n'a été trouvée (aucune combinaison de valeurs ne pouvait satisfaire toutes les clauses en même temps). Le problème est alors insatisfiable.

2. Etape 3 : Tester la Satisfiabilité

Dans cette étape, nous avons traduit les connaissances zoologiques relatives aux céphalopodes en forme CNF, puis nous avons testé la satisfiabilité de cette base. Nous avons aussi téléchargé des fichiers Benchmarks sous forme CNF pour les tester également en utilisant le solveur SAT.

3.1. Traduction des connaissances zoologiques en forme CNF

Soient les connaissances zoologiques suivantes modélisée en utilisant la logique propositionnelle :

1 - Les nautes sont des céphalopodes :

$$(Na \supset Cea); (Nb \supset Ceb); (Nc \supset Cec)$$

2 - Les céphalopodes sont des mollusques :

$$(Cea \supset Ma); (Ceb \supset Mb); (Cec \supset Mc)$$

*3- Les mollusques ont une coquille :

$$(Ma \supset Coa); (Mb \supset Cob); (Mc \supset Coc)$$

*4- Les céphalopodes n'en ont pas :

$$(Cea \supset \neg Coa); (Ceb \supset \neg Cob); (Cec \supset \neg Coc)$$

5- Les nautes en ont une :

$$(Na \supset Coa); (Nb \supset Cob); (Nc \supset Coc)$$

6- a est un naute, b est un céphalopode, c est un mollusque :

$$Na; Ceb; Mc$$

Remarque

Le système est devenu **incohérent** après avoir ignoré totalement le mot "généralement" car cela impliquerait que nous traitons les énoncés comme des tautologies et non comme des généralisations et ceci va probablement conduire à des contradictions et à des conclusions fausses.

Pour éviter les incohérences, nous interprétons les clauses incohérentes, nous obtenons la base suivante:

$(\neg Na \vee Cea); (\neg Nb \vee Ceb); (\neg Nc \vee Cec)$

$(\neg Cea \vee Ma); (\neg Ceb \vee Mb); (\neg Cec \vee Mc)$

$(\neg Na \vee Coa); (\neg Nb \vee Cob); (\neg Nc \vee Coc)$

$(\neg Ma \vee Cea \vee Coa); (\neg Ma \vee \neg Na \vee Coa)$

$(\neg Mb \vee Ceb \vee Cob); (\neg Mb \vee \neg Nb \vee Cob)$

$(\neg Mc \vee Cec \vee Coc); (\neg Mc \vee \neg Nc \vee Coc)$

$(\neg Cea \vee Na \vee \neg Coa)$

$(\neg Ceb \vee Nb \vee \neg Cob)$

$(\neg Cec \vee Nc \vee \neg Coc)$

Na; Ceb; Mc

Sachant que :

1 : Na ; 2 : Nb ; 3 : Nc ; 4 : Cea ; 5 : Ceb ; 6 : Cec ;
7 : Coa ; 8 : Cob ; 9 : Coc ; 10 : Ma ; 11 : Mb ; 12 : Mc.

Nous avons obtenu le résultat suivant après exécution :

```
C:\UBCSAT>ubcsat -alg saps -i zoo.cnf -solve
#
# UBCSAT version 1.1.0 (Sea to Sky Release)
#
# http://www.satlib.org/ubcsat
#
```

```

# 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      4      4
#
# Solution found for -target 0
#
# 1 -2 3 4 5 6 7 -8 9 10
# 11 12
#
Variables = 12
Clauses = 21
TotalLiterals = 48
TotalCPUtimeElapsed = 0.000
FlipsPerSecond = 1
RunsExecuted = 1
SuccessfulRuns = 1
PercentSuccess = 100.00
Steps_Mean = 4
Steps_CoeffVariance = 0
Steps_Median = 4
CPUtime_Mean = 0
CPUtime_CoeffVariance = 0
CPUtime_Median = 0

```

Figure 3. Résultat du Test de Satisfiabilité de la Base Zoologique

La solution trouvée est :

$$Na \wedge \neg Nb \wedge Nc \wedge Cea \wedge Ceb \wedge CeC \wedge Coa \wedge \neg Cob \wedge CoC \wedge Ma \wedge Mb \wedge Mc$$

3.2. Téléchargement et Test des Fichiers Benchmarks

Nous avons téléchargé deux fichiers Benchmarks prêts à être utilisés pour tester leur satisfiabilité. Ces fichiers contiennent des problèmes de satisfaction de contraintes sous forme CNF, conçus pour évaluer les performances des solveurs SAT.

Les deux fichiers choisis sont :

- [uf250-1065](#) contient 250 variables, 1065 clauses et 100 instances satisfiables.
- [uuf250-1065](#) contient 250 variables, 1065 clauses et 100 instances insatisfiables.

```

C:\UBCSAT>ubcsat -alg saps -i uf250-080.cnf -solve
#
# UBCSAT version 1.1.0 (Sea to Sky Release)
#
# http://www.satlib.org/ubcsat
#

```

```
#
# 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
51 52 -53 -54 -55 -56 -57 -58 -59 60
61 -62 63 64 -65 -66 -67 68 69 -70
71 -72 -73 74 75 -76 77 -78 -79 -80
-81 82 83 84 85 -86 -87 88 89 -90
-91 92 -93 94 -95 96 97 98 99 100
101 102 -103 -104 105 -106 -107 108 109 -110
111 112 -113 -114 -115 -116 -117 -118 119 -120
-121 -122 -123 -124 -125 -126 127 -128 129 130
131 132 133 -134 135 -136 -137 138 139 -140
141 -142 143 144 -145 -146 147 148 -149 150
-151 152 -153 154 155 -156 -157 -158 159 -160
161 -162 -163 164 -165 -166 167 -168 -169 170
-171 -172 -173 174 -175 -176 177 -178 179 180
181 -182 183 -184 185 186 -187 188 -189 -190
191 -192 193 194 -195 -196 -197 -198 -199 200
201 -202 -203 204 205 -206 207 208 -209 -210
211 212 213 214 -215 -216 217 218 219 220
-221 -222 -223 224 -225 -226 227 -228 -229 -230
231 232 233 234 -235 236 237 -238 239 -240
241 -242 -243 -244 -245 246 247 248 249 250

Variables = 250
Clauses = 1065
TotalLiterals = 3195
TotalCPUTimeElapsed = 0.008
FlipsPerSecond = 2174778
RunsExecuted = 1
SuccessfulRuns = 1
PercentSuccess = 100.00
Steps_Mean = 17398
Steps_CoeffVariance = 0
Steps_Median = 17398
CPUTime_Mean = 0.00799989700317
CPUTime_CoeffVariance = 0
CPUTime_Median = 0.00799989700317
```

Figure 4. Résultat du Test de Satisfiabilité d'un Fichier Benchmark satisfiable uf250-080.cnf

```
C:\UBCSAT>ubcsat -alg saps -i uuf250-080.cnf -solve
#
# UBCSAT version 1.1.0 (Sea to Sky Release)
#
# http://www.satlib.org/ubcsat
#
#
#      F Best      Step      Total
#      Run N Sol'n    of      Search
#      No. D Found    Best    Steps
#
#      1 0      1    19752    100000
# No Solution found for -target 0

Variables = 250
Clauses = 1065
TotalLiterals = 3195
TotalCPUTimeElapsed = 0.041
FlipsPerSecond = 2439017
RunsExecuted = 1
SuccessfulRuns = 0
PercentSuccess = 0.00
Steps_Mean = 100000
Steps_CoeffVariance = 0
Steps_Median = 100000
CPUTime_Mean = 0.0410001277924
CPUTime_CoeffVariance = 0
CPUTime_Median = 0.0410001277924
```

Figure 5. Résultat du Test de Satisfiabilité d'un Fichier Benchmark insatisfiable uf250-080.cnf

3. Etape 4 : Algorithme de Simulation de l'Inférence d'une Base de Connaissances

L'algorithme suivant simule l'inférence d'une base de connaissances (BC) en utilisant le raisonnement par l'absurde pour tester si la base de connaissances BC infère une formule φ donnée. L'idée est de vérifier si BC union $\{\text{non } \varphi\}$ infère une contradiction (représentée par \perp), ce qui équivaut à vérifier si φ est inférée par BC.

Algorithme RaisonnementParAbsurde(BC, φ):

Étape 1 : Ajouter la négation de φ à la base de connaissances

BC \leftarrow BC union $\{\text{non } \varphi\}$

Étape 2 : Appel au solveur SAT pour vérifier la satisfiabilité de BC union $\{\text{non } \varphi\}$

si SAT(BC) est non satisfiable alors

retourner "BC infère φ " # φ est inféré par BC

sinon

retourner "BC n'infère pas φ " # φ n'est pas inféré par BC

fin si

➤ Explication de l'Algorithme

1. Nous ajoutons la négation de la formule φ à la base de connaissances BC pour considérer l'ensemble des connaissances de BC ainsi que la négation de φ dans le raisonnement.

2. Nous appelons le solveur SAT pour vérifier si la base de connaissances BC union $\{\text{non } \varphi\}$ est satisfiable. Si elle est non satisfiable, cela signifie que l'ajout de la négation de φ conduit à une contradiction, ce qui implique que φ est inféré par BC. Sinon, φ n'est pas inféré par BC.

L'algorithme utilise le raisonnement par l'absurde pour tester l'inférence de φ par la base de connaissances BC. Si l'ajout de la négation de φ conduit à une contradiction, alors φ est nécessairement inféré par BC. Sinon, φ n'est pas inféré par BC.

Cet algorithme peut être implémenté dans différents langages de programmation en utilisant un solveur SAT pour la vérification de la satisfiabilité.

```

TP1_exo4.py > ...
1  from pysat.solvers import Solver
2
3  def raisonnement_par_absurde(BC, phi):
4      # Ajout de la négation à la base :
5      neg_phi = [-literal for literal in phi]
6      BC_with_neg_phi = BC + [neg_phi]
7
8      # Appel du solveur SAT :
9      with Solver(name='g3') as solver:
10         for clause in BC_with_neg_phi:
11             solver.add_clause(clause)
12
13         if not solver.solve():
14             return "BC infère  $\phi$ "
15         else:
16             return "BC n'infère pas  $\phi$ "
17
18  BC = [[1, -2], [-1, 3], [2, 3]]
19  phi = [3]
20
21  resultat = raisonnement_par_absurde(BC, phi)
22  print(resultat)
23

```

Figure 6. Algorithme du raisonnement par l'absurde implémenté en python

Nous avons tester l'algorithme sur la base suivante qui représente par exemple des règles de jeu:

$$BC = [[1, 2, -3], [2, 3, -4], [-1, -2, 5], [3, 4, -5]]$$

Et nous avons trouvé :

```

PS C:\VSCODE\s2\PYTHON\RCR> & C:/Users/ADMIN/AppData/Local/Programs/Python/Python39/python.exe c:/VSCODE/s2/PYTHON/RCR/TP1_exo4.py
Raisonnement par l'Absurde :
BC = [[1, -2], [-1, 3], [2, 3]]
 $\phi$  = [3]

resultat : BC infère  $\phi$ 
PS C:\VSCODE\s2\PYTHON\RCR>

```

Figure 7. Exemple d'exécution d'un exemple des règles de jeu sur l'algorithme de raisonnement par l'absurde

Conclusion

En conclusion, ce premier travail pratique nous a permis de bien maîtriser l'inférence logique basée sur un solveur SAT. Nous avons appris à modéliser des problèmes sous forme CNF, à tester la satisfiabilité de bases de connaissances, et à évaluer l'inférence de formules à l'aide du raisonnement par l'absurde.

**TP N° 2 : Logique du Premier
Ordre**

TP N° 2 : Logique du Premier Ordre

Introduction

Dans ce TP nous visons à explorer la bibliothèque Tweety pour la modélisation des connaissances en logique des prédicats. Pour cela nous verrons les étapes qui nous permettent d'exploiter cette bibliothèque et nous explorons ses classes prédéfinies pour la modélisation de la logique du premier ordre.

1. Logique du Premier Ordre

La logique du premier ordre, également connue sous le nom de logique des prédicats, est un formalisme logique qui étend la logique propositionnelle en permettant de quantifier sur les variables et de définir des prédicats pour représenter les relations entre les objets. Elle est utilisée pour décrire des structures plus complexes que ce qui est possible en logique propositionnelle.

Les caractéristiques distinctives de la logique du premier ordre incluent :

- L'utilisation de **variables** telles que x , y , etc., pour représenter des éléments du domaine d'interprétation.
- L'utilisation de **prédicats** (ou relations) pour exprimer des propriétés ou des relations entre les éléments.
- L'utilisation **d'opérations logiques** telles que ET, OU, IMPLIQUE, etc., pour construire des formules complexes à partir de propositions simples.
- L'utilisation de **quantificateurs**, notamment le quantificateur universel \forall et le quantificateur existentiel \exists pour exprimer des affirmations sur un ensemble d'éléments

Ces caractéristiques permettent à la logique du premier ordre d'exprimer des propositions plus complexes et plus riches que ce qui est possible en logique propositionnelle.

2. Préparation de l'Environnement

Pour l'environnement du travail, nous avons besoin d'Eclipse et de la bibliothèque Tweety.

- 1- Tout d'abord, nous téléchargeons la bibliothèque Tweety à partir de son site officiel [TweetyProject - Home](#). Nous avons opté pour la version : TweetyProject Full (with all dependencies) 1.25, 18.01.2024.

- 2- Nous créons un projet Maven dans Eclipse; en suivant la séquence d'action : "File" -> "New" -> "Maven Project" -> "Next".
- 3- Nous créons ensuite un simple projet.
- 4- Pour ajouter la bibliothèque Tweety, nous accédons au fichier pom.xml.

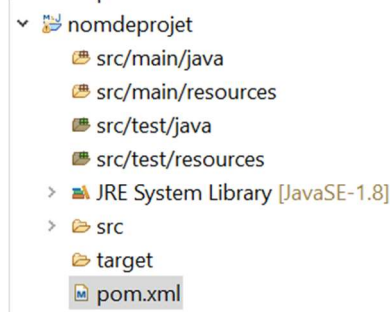


Figure 2.1. Emplacement du fichier pom.xml

- 5- Nous ajoutons le bout de code suivant :

```
<dependencies>

<dependency>

<groupId>org.tweetyproject</groupId>

<artifactId>tweety-full</artifactId>

<version>1.25</version>

</dependency>

</dependencies>
```

```
1<?xml version="1.0" encoding="UTF-8"?>
2<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3  <modelVersion>4.0.0</modelVersion>
4  <groupId>mytweetyproject</groupId>
5  <artifactId>mytweetyproject</artifactId>
6  <version>0.0.1-SNAPSHOT</version>
7  <dependencies>
8    <dependency>
9      <groupId>org.tweetyproject</groupId>
10     <artifactId>tweety-full</artifactId>
11     <version>1.25</version>
12   </dependency>
13 </dependencies>
14 </project>
```

Figure 2.2. Configuration du fichier pom.xml

7_ Enfin, nous créons les classes et importons les modules nécessaires pour le traitement depuis la bibliothèque Tweety, telles que :

```
import org.tweetyproject.arg.dung.causal.syntax.KnowledgeBase;
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.syntax.FolBeliefSet;
import org.tweetyproject.logics.fol.syntax.FolFormula;
import org.tweetyproject.logics.fol.syntax.FolSignature;
```

Figure 2.3. Exploitation de la bibliothèque Tweety

3. Modélisation des connaissances en logique des prédicats

- Définir une signature :

Créez une signature pour votre domaine d'interprétation, en définissant des sorts, des constantes et des prédicats pour spécifier les éléments de base qui seront utilisés pour décrire les connaissances et les relations dans un domaine donné.

ce qui facilite la modélisation et la manipulation de ces concepts en logique du premier ordre.

_ création de signature:

```
FolSignature sig = new FolSignature(true);
```

_ création de sort :

Les sorts définissent les catégories auxquelles appartiennent les constantes et les variables dans les formules logiques(les types d'objets ou d'entités).

```
Sort sortPerson = new Sort("Person");
```

"Person" qui sera utilisé pour catégoriser les constantes et les variables représentant des personnes.

4. Implémentation de la modélisation des connaissances en logique des prédicats

Notre exemple se concentre sur la modélisation des connaissances en logique des prédicats concernant les plantes, en mettant l'accent sur les fleurs et les couleurs associées à ces plantes.

Nous utilisons des prédicats pour déterminer si une plante est une fleur et pour spécifier la couleur d'une plante donnée. Ensuite, nous utilisons ces prédicats pour créer des croyances et vérifier différentes requêtes sur ces croyances à l'aide d'un prouveur.

importer les classes nécessaires de la bibliothèque Tweety ainsi que d'autres classes standard de Java

```
import java.io.IOException;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import org.tweetyproject.arg.dung.causal.syntax.KnowledgeBase;
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.syntax.FolBeliefSet;
import org.tweetyproject.logics.fol.syntax.FolFormula;
import org.tweetyproject.logics.fol.syntax.FolSignature;
```

Figure 2.4.

La classe principale est définie. À l'intérieur de cette classe, dans la méthode main, nous commençons par créer une signature **sig** car elle établit les symboles d'un langage formel, y compris les sorts (types d'objets), les constantes (objets spécifiques) et les prédicats (relations entre les objets). Dans notre exemple, nous définissons des constantes spécifiques telles que "rose", "rouge", ainsi que des prédicats comme "Fleur" et "Couleur". La signature est essentielle pour que l'analyseur syntaxique puisse reconnaître et valider les symboles utilisés dans nos formules logiques.

```
FolSignature sig = new FolSignature(true);
```

Puis nous définissons les sorts définissent les types d'objets que nous utilisons dans notre modèle. En définissant un sort "Plante" et un sort "Couleur", nous indiquons que nous allons parler d'objets qui appartiennent à ces catégories. Dans notre exemple, nous définissons des constantes spécifiques telles que "rose", "rouge", ainsi que des prédicats comme "Fleur" et "Couleur".

```
Sort sortPlante = new Sort("Plante");
Sort sortCouleur = new Sort("Couleur");
sig.add(sortPlante, sortCouleur);
```

Ensuite, nous définissons deux prédicats :

- Un prédicat Fleur pour décrire si une plante est une fleur $\text{Fleur}(x)$.
- Un prédicat Couleur pour décrire la couleur d'une plante $\text{Couleur}(x,y)$.

```
Predicate p = new Predicate("Fleur", predicateList);
```

```
Predicate p2 = new Predicate("Couleur", predicateList2);
```

Voici notre base de connaissances :

```
{ Couleur(menthe,vert), Couleur(tournesol,jaune), !Fleur(menthe), Couleur(rose,rouge),  
Fleur(tournesol), Fleur(rose) }
```

```
FolFormula f1 = (FolFormula)parser.parseFormula("Fleur(rose)");  
FolFormula f2 = (FolFormula)parser.parseFormula("Couleur(rose, rouge)");  
FolFormula f3 = (FolFormula)parser.parseFormula("Fleur(tournesol)");  
FolFormula f4 = (FolFormula)parser.parseFormula("Couleur(tournesol, jaune)");  
FolFormula f5 = (FolFormula)parser.parseFormula("!Fleur(menthe)");  
FolFormula f6 = (FolFormula)parser.parseFormula("Couleur(menthe, vert)");  
bs.add(f1, f2, f3, f4, f5, f6);
```

Enfin, nous avons utilisé un prouveur de la bibliothèque Tweety pour vérifier différentes requêtes sur notre base de connaissances

```
FolReasoner prover = FolReasoner.getDefaultReasoner();
```

5. Résultats

```
rose est une fleur : true  
la couleur de tournesol est jaune : true  
menthe est une fleur : false  
la couleur de menthe est rouge : false
```

Nous avons réussi à modéliser efficacement les connaissances sur les plantes et les couleurs en utilisant la logique des prédicats avec la bibliothèque Tweety. Nous avons pu vérifier avec succès diverses propriétés des plantes et de leurs couleurs à l'aide du prouveur intégré.

Conclusion

Ce travail nous a permis de mettre en pratique la modélisation des connaissances en logique des prédicats à l'aide de la bibliothèque Tweety. Nous avons appris à définir des sorts, des constantes, des prédicats et des formules pour représenter des connaissances complexes et à les utiliser pour répondre à des requêtes spécifiques.

TP N° 3 : Logique Modale

TP N° 3 : Logique Modale

Introduction

Dans le cadre du TP N°3 sur la logique modale, nous explorons l'utilisation de la librairie Tweety pour modéliser des connaissances en logique modale. Cette librairie Java dédiée aux modes logiques nous offre une des outils pour la représentation des connaissances couvrant diverses branches.

1. Définition

La logique modale constitue une extension de la logique classique permettant de raisonner sur des propositions modales, exprimant des notions telles que la possibilité, la nécessité, et la contingence. Elle trouve des applications dans divers domaines tels que la philosophie, l'informatique, et l'intelligence artificielle.

Dans cette partie du TP nous allons implémenter et vérifier les la véracité de quelques formules en partant de la base de connaissance suivante :

$\Diamond(\text{Fleur}(\text{rose}))$: Il est possible que la rose soit une fleur.

$\Box(\text{Couleur}(\text{tournesol}, \text{jaune}))$: Le tournesol est jaune dans tous les mondes possibles.

$\Diamond(\text{Fleur}(\text{tournesol}))$: Il est possible que le tournesol soit une fleur.

$\Box(\text{Couleur}(\text{menthe}, \text{vert}))$: La menthe est verte dans tous les mondes possibles.

$\Diamond(\neg \text{Fleur}(\text{menthe}))$: Il est possible que la menthe ne soit pas une fleur.

$\Box(\text{Avoir}(\text{menthe}, \text{rose}))$: La menthe a des roses dans tous les mondes possibles.

2. Implémentation avec Tweety

1. Création de la Signature : Nous avons défini une signature logique pour notre système, en spécifiant les sorts (types d'objets) et les prédicats utilisés dans nos formules modales.

```

// Création d'une signature de logique modale
FolSignature sig = new FolSignature(true);

// Ajout de tris (sorts) pour les plantes et les couleurs
Sort sortPlante = new Sort("Plante");
Sort sortCouleur = new Sort("Couleur");
sig.add(sortPlante, sortCouleur);

// Ajout de constantes pour les plantes
Constant rose = new Constant("rose", sortPlante);
Constant tournesol = new Constant("tournesol", sortPlante);
Constant menthe = new Constant("menthe", sortPlante);

// Ajout de constantes pour les couleurs
Constant rouge = new Constant("rouge", sortCouleur);
Constant jaune = new Constant("jaune", sortCouleur);
Constant vert = new Constant("vert", sortCouleur);

sig.add(rose, tournesol, menthe, rouge, jaune, vert);

// Ajout de prédicats
List<Sort> fleurSorts = new ArrayList<>();
fleurSorts.add(sortPlante);
Predicate Fleur = new Predicate("Fleur", fleurSorts);

List<Sort> couleurSorts = new ArrayList<>();
couleurSorts.add(sortPlante);
couleurSorts.add(sortCouleur);
Predicate Couleur = new Predicate("Couleur", couleurSorts);

sig.add(Fleur, Couleur);

// Création d'un ensemble de croyances modales
MLBeliefSet bs = new MLBeliefSet();

// Création d'un parseur pour analyser les formules modales
MLParser parser = new MLParser();
parser.setSignature(sig);

```

Figure 3.1. Initialisation et préparation des données

2. Construction des Formules Modales : Nous avons utilisé le parser de Tweety pour analyser et construire des formules modales à partir de chaînes de caractères. Cela nous permet de représenter des connaissances sous forme de formules logiques.

```

// Création d'un parseur pour analyser les formules modales
MLParser parser = new MLParser();
parser.setSignature(sig);

// Ajout de formules modales à l'ensemble de croyances
bs.add((RelationalFormula) parser.parseFormula("<(Fleur(rose))")); // Il est possible que la rose soit une fleur
bs.add((RelationalFormula) parser.parseFormula("[](Couleur(tournesol, jaune))")); // Le tournesol est jaune dans tous les mondes possibles
bs.add((RelationalFormula) parser.parseFormula("<(Fleur(tournesol))")); // Il est possible que le tournesol soit une fleur
bs.add((RelationalFormula) parser.parseFormula("[](Couleur(menthe, vert))")); // La menthe est verte dans tous les mondes possibles
bs.add((RelationalFormula) parser.parseFormula("<(Fleur(menthe))")); // Il est possible que la menthe ne soit pas une fleur

```

Figure 3.2. Parser et construction des formules modales

3. Manipulation des Ensembles de Croyances Modales : Nous avons créé des ensembles de croyances modales pour stocker et raisonner sur des ensembles de formules modales. Cela nous permet de vérifier la cohérence des connaissances et d'effectuer des déductions modales.


```
// Affichage de l'ensemble de croyances modales
System.out.println("Modal knowledge base: " + bs);
```

Figure 3.3. Ensemble des croyances

4. Raisonnement Modal : Nous avons implémenté un raisonneur modal capable de vérifier la validité de propositions modales par rapport à un ensemble de croyances donné. Cela nous permet de répondre à des requêtes modales telles que la possibilité, la nécessité, etc.

```
// Création d'un prouveur modale
SimpleMLReasoner reasoner = new SimpleMLReasoner();

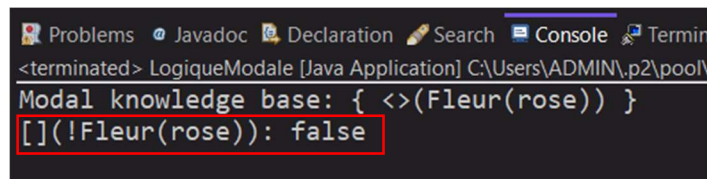
// Vérification des requêtes modales
System.out.println("[](!Fleur(rose)): " + reasoner.query(bs, (FolFormula) parser.parseFormula("[](!Fleur(rose))")) + "\n"); // La rose n'est pas une fleur dans tous les mondes possibles
System.out.println("<>(Fleur(tournesol) && Couleur(tournesol, jaune)): " + reasoner.query(bs, (FolFormula) parser.parseFormula("<>(Fleur(tournesol) && Couleur(tournesol, jaune))")) + "\n");
System.out.println("[](Fleur(rose) || Fleur(menthe)): " + reasoner.query(bs, (FolFormula) parser.parseFormula("[](Fleur(rose) || Fleur(menthe))")) + "\n"); // La rose ou la menthe est une fl
System.out.println("<>(Couleur(rose, rouge)): " + reasoner.query(bs, (FolFormula) parser.parseFormula("<>(Couleur(rose, rouge))")) + "\n"); // Il est possible que la rose ne soit pas rouge
```

Figure 3.4. Raisonnement modal

Résultats

En ce qui suit nous affichons le résultats de l'exécution de certaines formules.

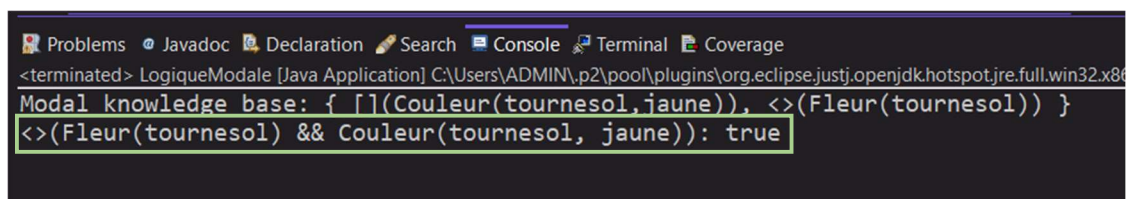
Première formule :



```
Problems Javadoc Declaration Search Console Termin
<terminated> LogiqueModale [Java Application] C:\Users\ADMIN\p2\pool\
Modal knowledge base: { <>(Fleur(rose)) }
[](!Fleur(rose)): false
```

Figures 3.5. Véracité de la première formule

Deuxième formule :



```
Problems Javadoc Declaration Search Console Terminal Coverage
<terminated> LogiqueModale [Java Application] C:\Users\ADMIN\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86
Modal knowledge base: { [](Couleur(tournesol,jaune)), <>(Fleur(tournesol)) }
<>(Fleur(tournesol) && Couleur(tournesol, jaune)): true
```

Figures 3.6. Véracité de la deuxième formule

Remarque

Nous ne pouvons pas vérifier la véracité de toutes les formules tous a la fois car ceci prend énormément du temps et des fois mène a des dépassements de capacité !

Conclusion

En conclusion, notre implémentation de la logique modale avec Tweety nous permet de modéliser et de raisonner sur des connaissances modales de manière efficace. Cette approche offre une base solide pour le développement de systèmes intelligents capables de raisonner sur des situations complexes impliquant des notions de possibilité, de nécessité et d'autres modalités.

TP N° 4 : Logique des Défauts

TP N° 4 : Logique des Défauts

Introduction

La logique des défauts est une approche puissante pour traiter l'incertitude et l'imperfection des connaissances en introduisant des règles par défaut et des exceptions. Dans ce rapport, nous explorons l'utilisation de la bibliothèque Java Orbital pour implémenter et exploiter des systèmes basés sur la logique des défauts. Orbital fournit des outils avancés pour la modélisation et le raisonnement, permettant de gérer efficacement des informations incomplètes ou incertaines.

1. Utilisation de la Librairie Orbital

- Création de la Signature

Nous avons d'abord créé une signature qui comprend les termes et les prédicats utilisés dans notre base de connaissances.

```
try {  
    // Création d'une signature  
    Signature signature = new Signature();  
    signature.add("rose", "tournesol", "menthe", "rouge", "jaune", "vert");  
    signature.add("Fleur", "Couleur");  
}
```

Figure 4.1. Création des signatures

- Création de la Base de Connaissances par Défaut

Ensuite, nous avons créé une base de connaissances par défaut (DefaultKnowledgeBase) en utilisant cette signature. Nous avons ajouté des défauts, chacun étant constitué d'un prérequis, d'exceptions (qui sont vides dans notre cas), et d'une conclusion.

```

// Création d'une base de connaissances par défaut
DefaultKnowledgeBase kb = new DefaultKnowledgeBase(signature);

// Ajout des défauts (pré-requis : exceptions / conclusion)
kb.add(new Default(
    "Fleur(rose)", // Pré-requis
    "",           // Exceptions (vide dans ce cas)
    "Fleur(rose)" // Conclusion
));
kb.add(new Default(
    "Couleur(rose, rouge)", // Pré-requis
    "",                     // Exceptions (vide dans ce cas)
    "Couleur(rose, rouge)" // Conclusion
));

kb.add(new Default(
    "Fleur(tournesol)",
    "",
    "Fleur(tournesol)"
));
kb.add(new Default(
    "Couleur(tournesol, jaune)",
    "",
    "Couleur(tournesol, jaune)"
));

kb.add(new Default(
    "Fleur(menthe)",
    "",
    "Fleur(menthe)"
));
kb.add(new Default(
    "Couleur(menthe, vert)",
    "",
    "Couleur(menthe, vert)"
));

// Affichage de la base de connaissances
System.out.println("Base de connaissances par défaut:");
System.out.println(kb);

```

Figure 4.2. Création de la base de connaissance

- Vérification des Requêtes

Enfin, nous avons utilisé un `DefaultReasoner` pour interroger la base de connaissances et vérifier certaines assertions concernant les plantes et leurs couleurs.

```

// Vérification des requêtes
DefaultReasoner reasoner = new DefaultReasoner();

System.out.println("Fleur(rose) ? " + reasoner.query(kb, "Fleur(rose)"));
System.out.println("Couleur(rose, rouge) ? " + reasoner.query(kb, "Couleur(rose, rouge)"));
System.out.println("Fleur(tournesol) ? " + reasoner.query(kb, "Fleur(tournesol)"));
System.out.println("Couleur(tournesol, jaune) ? " + reasoner.query(kb, "Couleur(tournesol, jaune)"));
System.out.println("Fleur(menthe) ? " + reasoner.query(kb, "Fleur(menthe)"));
System.out.println("Couleur(menthe, vert) ? " + reasoner.query(kb, "Couleur(menthe, vert)"));

```

Figure 4.3. Vérification et traitement de la base de connaissance

2. Résultats

Le programme interroge la base de connaissances par défaut pour vérifier si certaines plantes sont des fleurs et si elles ont des couleurs spécifiques. Les résultats des requêtes sont affichés pour montrer si les assertions sont vraies ou fausses selon les défauts définis.

Remarque

Pour des raisons d'incompatibilité d'Eclipse avec la bibliothèque Orbital, l'exécution ne s'est pas passée comme convenue.

Conclusion

Cette implémentation démontre comment utiliser la bibliothèque Orbital pour manipuler une base de connaissances par défaut et effectuer des raisonnements sur cette base. La logique des défauts est une approche puissante pour traiter les exceptions et les incertitudes dans les systèmes de connaissances, et la bibliothèque Orbital fournit donc des outils efficaces même si cela engendre des erreurs de compatibilité.

TP N° 5 : Réseaux Sémantiques

TP N° 5 : Réseaux Sémantiques

Introduction

Les réseaux sémantiques sont des structures de données sophistiquées qui permettent de représenter et de manipuler des connaissances sous forme de graphes, où les nœuds représentent des concepts ou des entités, et les arêtes représentent les relations entre ces concepts. Ils sont largement utilisés dans divers domaines de l'intelligence artificielle, tels que le traitement du langage naturel, les systèmes experts et la modélisation des connaissances. Dans cette partie du travail demandé nous explorons trois algorithmes fondamentaux utilisés dans les réseaux sémantiques : la propagation de marqueurs, l'héritage et la gestion des liens d'exception.

Voici notre réseau sémantique:

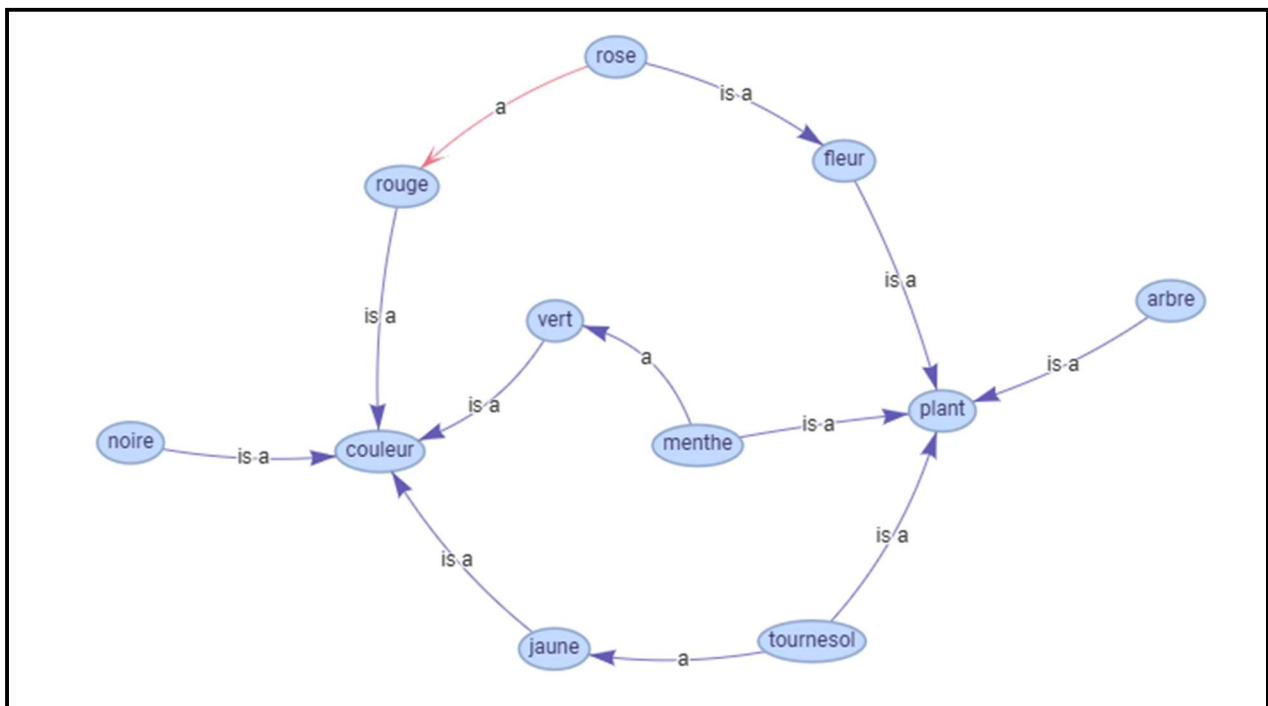


Figure 5.1. Réseau sémantique de la base de connaissances des plantes

1. Algorithme de propagation de marqueurs dans les réseaux sémantiques

Cet algorithme a pour objectif d'inférer des relations entre différents nœuds marqués dans un réseau sémantique. Nous utilisons cette technique pour parcourir les liens et découvrir les relations indirectes entre les concepts.

➤ Composants de l'Algorithme

Cet algorithme se compose principalement de deux fonctions : **get_label** et **propagation_de_marqueurs**.

- ❖ La fonction **get_label** est utilisée pour vérifier et formuler une réponse qui indique s'il existe un lien spécifique entre un nœud donné et d'autres nœuds dans un réseau sémantique.

```
def get_label(reseau_semantique, node, relation):
    node_relation_edges = [edge["from"] for edge in reseau_semantique["edges"] if (edge["to"] == node["id"] and edge["label"] == relation)]
    node_relation_edges_label = [node["label"] for node in reseau_semantique["nodes"] if node["id"] in node_relation_edges]
    reponse = "il y a un lien entre les 2 noeuds : " + ", ".join(node_relation_edges_label)
    return reponse
```

Figure 5.2. Code de la fonction *get_label*

Pour cela nous procédons comme suit :

- Filtrage des Arêtes : Elle identifie les arêtes du réseau où l'arête va vers le nœud donné (node) et a une étiquette spécifique (relation).
 - Filtrage des Nœuds : Elle récupère les étiquettes des nœuds qui sont à l'origine des arêtes filtrées.
 - Formation de la Réponse : Elle génère une réponse en indiquant les nœuds connectés par la relation spécifiée.
- ❖ La fonction **propagation_de_marqueurs** détermine s'il existe une relation spécifique entre deux nœuds marqués en parcourant le réseau.

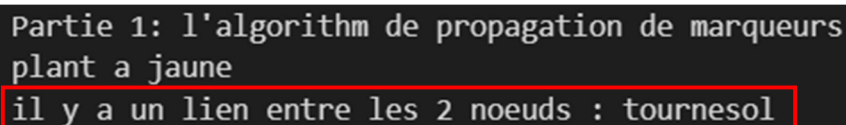
```
def propagation_de_marqueurs(reseau_semantique, node1, node2, relation):
    nodes = reseau_semantique["nodes"]
    solutions_found = []
    for i in range(min(len(node1), len(node2))):
        solution_found = False
        try:
            M1 = [node for node in nodes if node["label"] == node1[i]][0]
            M2 = [node for node in nodes if node["label"] == node2[i]][0]
            edges = reseau_semantique["edges"]
            propagation_edges = [edge for edge in edges if (edge["to"] == M1["id"] and edge["label"] == "is a")]
            while len(propagation_edges) != 0 and not solution_found:
                temp_node = propagation_edges.pop()
                temp_node_contient_edges = [edge for edge in edges if (edge["from"] == temp_node["from"] and edge["label"] == relation)]
                solution_found = any(d["to"] == M2["id"] for d in temp_node_contient_edges)
                if not solution_found:
                    temp_node_is_a_edges = [edge for edge in edges if (edge["to"] == temp_node["from"] and edge["label"] == "is a")]
                    propagation_edges.extend(temp_node_is_a_edges)
            solutions_found.append(get_label(reseau_semantique, M2, relation) if solution_found else "il n'y a pas un lien entre les 2 noeuds")
        except IndexError:
            solutions_found.append("Aucune reponse n'est fournie par manque de connaissances.")
    return(solutions_found)
```

Figure 5.3. code de la fonction *Propagation_de_marqueurs*

Nous avons procédé comme suit :

- Initialisation d'une liste `propagation_edges` contenant les arêtes sortantes du nœud M1 avec l'étiquette "is a".
- Une boucle `while` est utilisée pour propager les marqueurs à travers le réseau :
- Extraction d'une arête de `propagation_edges`.
- Nous cherchons les arêtes sortantes de ce nœud avec l'étiquette spécifiée par relation.
- Si l'une de ces arêtes mène à M2, `solution_found` est mise à `True`. Sinon, nous continuons la propagation en ajoutant les arêtes "is a" du nœud courant à `propagation_edges`.
- Si une solution est trouvée, nous utilisons la fonction `get_label` pour formuler une réponse. Sinon, nous ajoutons une réponse qui indique qu'il n'y a pas de lien entre les deux nœuds.

Résultat



```
Partie 1: l'algorithm de propagation de marqueurs
plant a jaune
il y a un lien entre les 2 noeuds : tournesol
```

Figure 5.4. Exemple d'exécution de l'algorithme de propagation des marqueurs

2. Algorithme d'Héritage

Cet algorithme permet de déduire toutes les propriétés relatives à un nœud en exploitant les relations d'héritage ("is a") pour saturer le réseau et inférer toutes les informations possibles.

```
def heritage(reseau_semantique, name):
    nodes = reseau_semantique["nodes"]
    edges = reseau_semantique["edges"]

    # Trouver le nœud initial par son nom
    node = [node for node in nodes if node["label"] == name][0]

    # Initialisation
    legacy = [node["label"]]
    properties = []
    legacy_edges = [edge["to"] for edge in edges if (edge["from"] == node["id"] and edge["label"] == "is a")]

    while legacy_edges:
        n = legacy_edges.pop()
        legacy.append(get_label(reseau_semantique, n))

        # Ajouter les nouvelles relations "is a" à traiter
        legacy_edges.extend([edge["to"] for edge in edges if (edge["from"] == n and edge["label"] == "is a")])

        # Récupérer et ajouter les propriétés des nœuds
        properties_nodes = [edge for edge in edges if (edge["from"] == n and edge["label"] != "is a")]
        for pn in properties_nodes:
            properties.append(":".join([pn["label"], get_label(reseau_semantique, pn["to"])]))

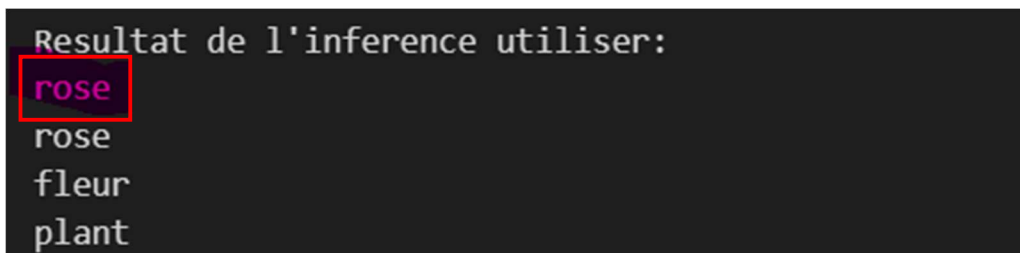
    return legacy, properties
```

Figure 5.5. implémentation de l'algorithme d'héritage

Nous procédons dans un algorithme comme suit :

- Tant que legacy_edges n'est pas vide, nous continuons la propagation de l'héritage.
- Extraction d'un nœud de legacy_edges.
- Nous ajoutons l'étiquette du nœud extrait à legacy.
- Nous ajoutons les nœuds hérités de ce nœud via les relations "is a" à legacy_edges pour traitement ultérieur.
- Nous récupérons les propriétés (arêtes avec des étiquettes autres que "is a") du nœud extrait et nous les ajoutons à properties.

Résultat

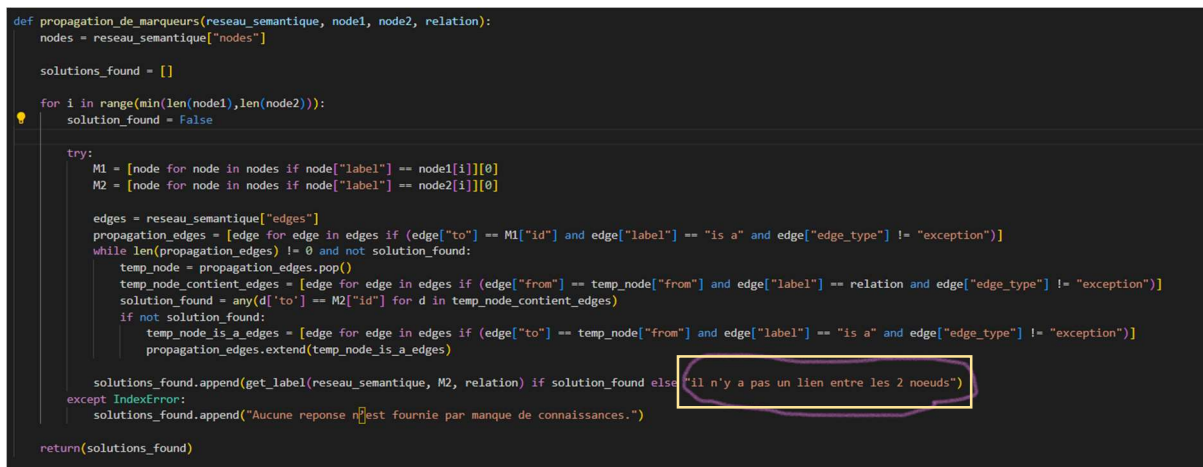


```
Resultat de l'inference utiliser:
rose
rose
fleur
plant
```

Figure 5.6. Propriétés du nœud rose

3. Algorithme d'Exception

l'algorithme d'inhibition de la propagation dans le cas des liens d'exception est important pour maintenir des inférences exactes et pertinentes dans un réseau sémantique, en respectant les exceptions spécifiées.



```
def propagation_de_marqueurs(reseau_semantique, node1, node2, relation):
    nodes = reseau_semantique["nodes"]

    solutions_found = []

    for i in range(min(len(node1), len(node2))):
        solution_found = False

        try:
            M1 = [node for node in nodes if node["label"] == node1[i]][0]
            M2 = [node for node in nodes if node["label"] == node2[i]][0]

            edges = reseau_semantique["edges"]
            propagation_edges = [edge for edge in edges if (edge["to"] == M1["id"] and edge["label"] == "is a" and edge["edge_type"] != "exception")]
            while len(propagation_edges) != 0 and not solution_found:
                temp_node = propagation_edges.pop()
                temp_node_contient_edges = [edge for edge in edges if (edge["from"] == temp_node["from"] and edge["label"] == relation and edge["edge_type"] != "exception")]
                solution_found = any(d["to"] == M2["id"] for d in temp_node_contient_edges)
            if not solution_found:
                temp_node_is_a_edges = [edge for edge in edges if (edge["to"] == temp_node["from"] and edge["label"] == "is a" and edge["edge_type"] != "exception")]
                propagation_edges.extend(temp_node_is_a_edges)

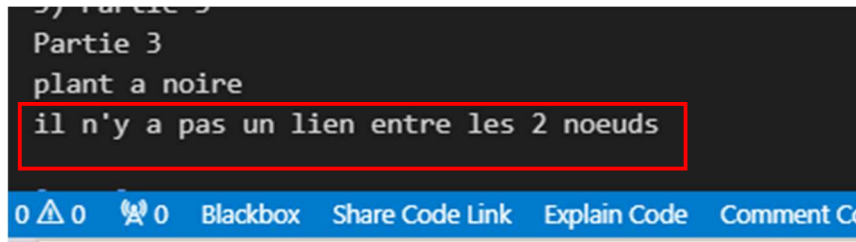
            solutions_found.append(get_label(reseau_semantique, M2, relation) if solution_found else "il n'y a pas un lien entre les 2 noeuds")
        except IndexError:
            solutions_found.append("Aucune reponse n'est fournie par manque de connaissances.")

    return(solutions_found)
```

Figure 5.7. Implémentation de l'algorithme d'Exception en python

Résultat

D'après notre réseau sémantique, il n'y a pas une relation entre le nœud plante et la couleur noire, nous vérifions ceci avec l'algorithme d'exception :



```
Partie 3  
plant a noire  
il n'y a pas un lien entre les 2 noeuds
```

Figure 5.8. Résultat de l'exécution d'un exemple avec l'algorithme d'exception

Conclusion

En conclusion, les réseaux sémantiques et les algorithmes associés sont des outils puissants pour représenter et manipuler des connaissances de manière structurée et efficace en intelligence artificielle. Ils offrent un cadre flexible et expressif pour la modélisation des connaissances et la résolution de problèmes complexes.

Conclusion Générale

Dans ces travaux pratiques réalisé dans le cadre du cours Représentation des Connaissances et Raisonnement, nous avons exploré diverses techniques de modélisation et d'inférence logique.

Nous avons commencé par l'utilisation d'un solveur SAT pour l'inférence logique, où nous avons appris à modéliser des problèmes sous forme CNF et à évaluer leur satisfiabilité. Ensuite, nous avons exploré la logique du premier ordre avec la bibliothèque Tweety, nous permettant de représenter des connaissances complexes et de répondre à des requêtes spécifiques. Nous avons également été initiés à la logique modale et aux réseaux sémantiques. Les réseaux sémantiques nous ont permis de manipuler des connaissances sous forme de graphes, en utilisant des algorithmes tels que la propagation de marqueurs et l'héritage.

Enfin, nous avons abordé la logique de description pour représenter et raisonner sur des connaissances complexes. Ce projet nous a offert une vue d'ensemble des différents domaines de la représentation de connaissance et nous a permis d'approfondir nos compétences en modélisation et en inférence logique.