



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

Département Intelligence Artificielle et Sciences des Données Spécialité :

Master Systèmes Informatiques Intelligents

Module :

REPRÉSENTATION DES CONNAISSANCES ET LE RAISONNEMENT 1

Rapport TP : **Représentation des Connaissances et Raisonnement 1**

Présenté par :

Benzemrane Lydia

Ghamraci Rahil

1	TP1 : Solveur SAT	1
1.1	Étape 1	1
1.1.1	Création du répertoire et des fichiers	1
1.2	Étape 2	2
1.2.1	Exécution du fichier test1.cnf	2
1.2.2	Exécution du fichier test2.cnf	3
1.3	Étape 3	3
1.3.1	Traduction de la base de connaissances zoologiques	3
1.3.2	Transformation de la base en clauses	4
1.3.3	transformation des classes en format CNF :	5
1.3.4	Test des fichiers Benchmarks	5
1.4	Étape 4	6
1.4.1	Code source du programme	7
1.4.2	Compilation et exécution	7
1.4.3	Le fichier test.cnf avant l'exécution (ajout du littéral non a , en format cnf -1) :	8
1.4.4	Le fichier test.cnf après l'exécution de exo4.c :	8
1.4.5	résultat de l'exécution :	9
1.5	Modélisation d'un problème et le résoudre avec le solveur sat	9
1.5.1	Modélisation : Coloriage de graphe	9
1.5.2	Les clauses de la base :	10
1.5.3	les classes en format cnf :	11
1.5.4	la base de faits :	11
1.5.5	L'exécution avec le solveur SAT :	12
2	TP2 : Logique des prédicats	13
2.1	La librairie Java Tweety	13
2.2	Exemple de logique des prédicats	13
2.2.1	définition de la signature	13
2.2.2	Ajout des types	13
2.2.3	Ajout des constantes	13
2.2.4	Ajout des prédicats	13
2.2.5	Percement de connaissances	14
2.2.6	Utilisation du raisonneur	14

2.2.7	Résultat obtenu	14
2.2.8	Interprétation du Résultat	15
2.3	Implémentation de l'exemple du cours avec tweety	15
2.3.1	définition de la signature	15
2.3.2	Ajout des types	15
2.3.3	Ajout des constantes	15
2.3.4	Ajout des prédicats	15
2.3.5	Percement de connaissances	16
2.3.6	Utilisation du raisonneur	16
2.3.7	Résultat obtenu	16
2.3.8	Interprétation du Résultat	16
2.4	Implémentation d'un exemple	17
2.4.1	définition de la signature	17
2.4.2	Ajout des types	17
2.4.3	Ajout des constantes	17
2.4.4	Ajout des prédicats	17
2.4.5	Percement de connaissances	17
2.4.6	Utilisation du raisonneur	18
2.4.7	Résultat obtenu	18
2.4.8	Interprétation du Résultat	18
3	TP3 : Logique modale	19
3.1	Étude d'un exemple	19
3.1.1	Importation des bibliothèques	19
3.1.2	Parsemant de la base de connaissances	19
3.1.3	La base de connaissances	19
3.1.4	Préparation de la formule	20
3.1.5	Le raisonnement	20
3.1.6	Résultat	20
3.1.7	Interprétation du résultat	20
3.2	Modélisation d'un exemple	21
3.3	Implémentation	21
3.3.1	Parsemant de la base de connaissances	21
3.3.2	La base de connaissances	21
3.3.3	Préparation de la formule 1	21
3.3.4	Le raisonnement	22
3.3.5	Résultat	22
3.3.6	Préparation de la formule 2	22
3.3.7	Résultat	22
3.3.8	Interprétation du résultat	22
4	TP4 : Logique des défauts	23
4.1	Exemple	23
4.1.1	Explication des lignes du code	23
4.1.2	Résultat	25
4.1.3	Interprétation du résultat	26

5	TP5 : Réseaux sémantiques	27
5.1	Introduction	27
5.2	Réseaux sémantiques	27
5.3	Partie 1 : implémenter l'algorithme de propagation de marqueurs dans les réseaux sémantiques	28
5.4	Partie 2 : implémenter l'algorithme d'héritage	30
5.5	Partie 3 : implémentez un algorithme qui permet d'inhiber la propagation dans le cas des liens d'exception	31
6	TP6 : Logique de description	34
6.0.1	ensuite, on définit les concepts atomiques :	34
	Conclusion générale	38

1.1	Répertoire UBCSAT	1
1.2	– Commande d’exécution du solveur SAT pour le fichier test1.cnf	2
1.3	exécution du solveur SAT pour le fichier test1.cnf	2
1.4	exécution du solveur SAT pour le fichier test1.cnf	3
1.5	Tableau des littéraux en format cnf	4
1.6	Le fichier zoo.cnf	5
1.7	– Solution de la base de connaissances zoologiques	5
1.8	– Téléchargement du fichier benchmark	6
1.9	– – Solution du fichier benchmark	6
1.10	Programme C qui détermine si un but est inféré ou non par une base	7
1.11	Compilation et exécution	7
1.12	Le fichier test.cnf avant l’exécution	8
1.13	Le fichier test.cnf après l’exécution	8
1.14	résultat de l’exécution	9
1.15	Les clauses de la base	10
1.16	les classes en format cnf	11
1.17	la base de faits	11
1.18	résultat de l’exécution	12
2.1	résultat de l’exécution	14
2.2	résultat de l’exécution	16
2.3	résultat de l’exécution	18
3.1	résultat de l’exécution	20
3.2	résultat de l’exécution	22
3.3	résultat de l’exécution	22
4.1	Création d’un monde	24
4.2	Les formules du monde	24
4.3	Les règles du monde	24
4.4	Ensemble des règles	25
4.5	Création du raisonneur et extraction des extensions	25
4.6	résultat de l’exécution	25

5.1	Algorithme de propagation de marqueur	28
5.2	Réseau sémantique utilisé pour la partie 1	29
5.3	Résultat obtenu par l'algorithme de propagation de marqueurs	30
5.4	Algorithme d'héritage	30
5.5	Réseau sémantique utilisé pour la partie 2 du TP	31
5.6	Résultat obtenu par l'algorithme d'héritage	31
5.7	Algorithme d'inhibition de la propagation dans le cas des liens d'exception	32
5.8	Réseau sémantique utilisé pour la partie 3 du TP	32
5.9	Résultat obtenu par l'algorithme d'inhibition de la propagation dans le cas des liens d'exception	33
6.1	Importation de owlready2	34
6.2	Concepts atomiques	35
6.3	Roles	35
6.4	concepts composés	36
6.5	ABOX	36
6.6	inférence	36
6.7	fichier généré	37
6.8	inférence	37

CHAPITRE 1

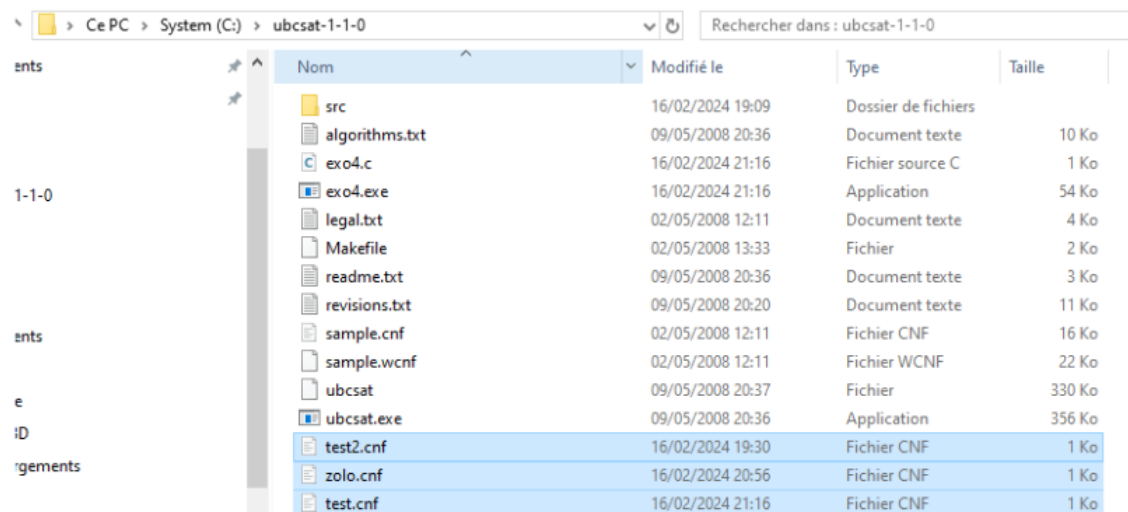
TP1 : SOLVEUR SAT

nous débuterons par l'utilisation d'un solveur SAT pour étudier la satisfaisabilité de divers ensembles de connaissances. En plus de cela, nous traduirons une base de connaissances sur la zoologie, la testerons sur des référentiels de performance, et simulerons l'inférence d'une base de connaissances à l'aide d'un algorithme.

1.1 Étape 1

1.1.1 Création du répertoire et des fichiers

On commence par créer un dossier, puis on y copie le fichier ubcsat (le solveur) ainsi que les deux fichiers test1.cnf et test2.cnf.



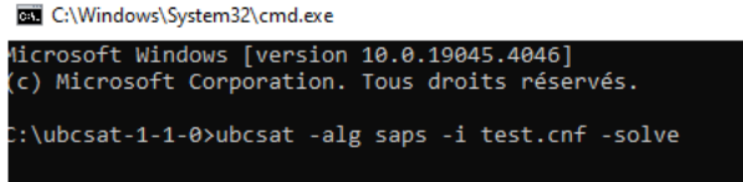
Nom	Modifié le	Type	Taille
src	16/02/2024 19:09	Dossier de fichiers	
algorithms.txt	09/05/2008 20:36	Document texte	10 Ko
exo4.c	16/02/2024 21:16	Fichier source C	1 Ko
exo4.exe	16/02/2024 21:16	Application	54 Ko
legal.txt	02/05/2008 12:11	Document texte	4 Ko
Makefile	02/05/2008 13:33	Fichier	2 Ko
readme.txt	09/05/2008 20:36	Document texte	3 Ko
revisions.txt	09/05/2008 20:20	Document texte	11 Ko
sample.cnf	02/05/2008 12:11	Fichier CNF	16 Ko
sample.wcnf	02/05/2008 12:11	Fichier WCNF	22 Ko
ubcsat	09/05/2008 20:37	Fichier	330 Ko
ubcsat.exe	09/05/2008 20:36	Application	356 Ko
test2.cnf	16/02/2024 19:30	Fichier CNF	1 Ko
zolo.cnf	16/02/2024 20:56	Fichier CNF	1 Ko
test.cnf	16/02/2024 21:16	Fichier CNF	1 Ko

FIGURE 1.1 – Répertoire UBCSAT

1.2 Étape 2

Maintenant, nous allons tester la satisfiabilité des bases de connaissances des deux fichiers test1.cnf et test2.cnf. Pour cela, on va exécuter le solveur SAT.

1.2.1 Exécution du fichier test1.cnf

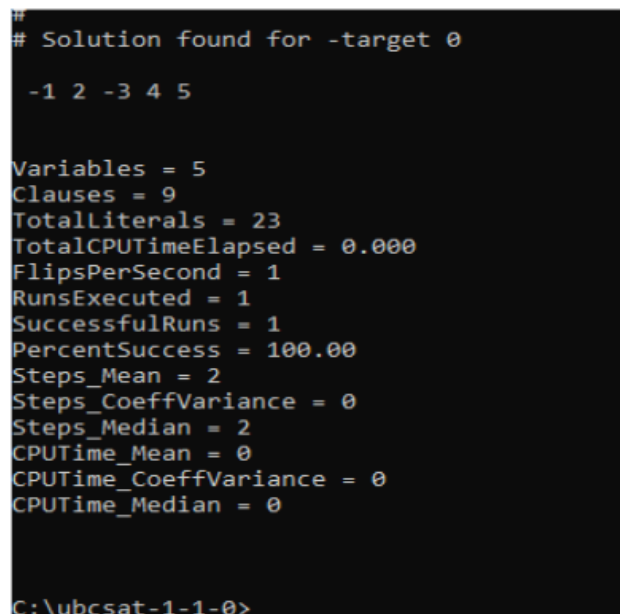


```
C:\Windows\System32\cmd.exe
Microsoft Windows [version 10.0.19045.4046]
(c) Microsoft Corporation. Tous droits réservés.

C:\ubcsat-1-1-0>ubcsat -alg saps -i test.cnf -solve
```

FIGURE 1.2 – Commande d'exécution du solveur SAT pour le fichier test1.cnf

Une fois la commande exécutée, des informations et des paramètres sur l'UBCSA2 s'affichent, ainsi qu'une solution : Le solveur a fourni le modèle non a et b et non c et d et e => il existe au



```
#
# Solution found for -target 0
-1 2 -3 4 5

Variables = 5
Clauses = 9
TotalLiterals = 23
TotalCPUtimeElapsed = 0.000
FlipsPerSecond = 1
RunsExecuted = 1
SuccessfulRuns = 1
PercentSuccess = 100.00
Steps_Mean = 2
Steps_CoeffVariance = 0
Steps_Median = 2
CPUtime_Mean = 0
CPUtime_CoeffVariance = 0
CPUtime_Median = 0

C:\ubcsat-1-1-0>
```

FIGURE 1.3 – exécution du solveur SAT pour le fichier test1.cnf

moins un modèle pour cette base de connaissance donc elle est satisfiable

1.2.2 Exécution du fichier test2.cnf

```

# No Solution found for -target 0

Variables = 5
Clauses = 12
TotalLiterals = 28
TotalCPUtimeElapsed = 0.038
FlipsPerSecond = 2631572
RunsExecuted = 1
SuccessfulRuns = 0
PercentSuccess = 0.00
Steps_Mean = 100000
Steps_CoeffVariance = 0
Steps_Median = 100000
CPUtime_Mean = 0.0380001068115
CPUtime_CoeffVariance = 0
CPUtime_Median = 0.0380001068115

C:\ubcsat-1-1-0>

```

FIGURE 1.4 – exécution du solveur SAT pour le fichier test1.cnf

Le solveur n'a fourni aucun modèle => cette base de connaissance donc elle est insatisfiable

1.3 Étape 3

1.3.1 Traduction de la base de connaissances zoologiques

La base est la suivante :

1. ((Céa Na) Coa);
2. ((Céb Nb) Cob);
3. ((Céc Nc) Coc);
4. ((Ma (Céa Na) Coa);
5. ((Mb (Céb Nb) Cob);
6. ((Mc (Céc Nc) Coc).

littéral	représentation format cnf
Na	1
Nb	2
Nc	3
Céa	4
Céb	5
Céc	6
Ma	7
Mb	8
Mc	9
Coa	10
Cob	11
Coc	12

FIGURE 1.5 – Tableau des littéraux en format cnf

1.3.2 Transformation de la base en clauses

$((Céa \vee Na) \vee Coa)$; devient alors $Céa \vee Na \vee Coa \dots$ clause 1 $((Céb \vee Nb) \vee Cob)$; devient alors $Céb \vee Nb \vee Cob \dots$ clause 2 $((Céc \vee Nc) \vee Coc)$; devient alors $Céc \vee Nc \vee Coc \dots$ clause 3

$((Ma \vee (Céa \vee Na)) \vee Coa)$; devient $(Ma \vee (Céa \vee Na)) \vee Coa$ et donc $Ma \vee Céa \vee Coa \dots$ clause 4.1 $Ma \vee Na \vee Coa \dots$ clause 4.2

$((Mb \vee (Céb \vee Nb)) \vee Cob)$; devient $(Mb \vee (Céb \vee Nb)) \vee Cob$ et donc $Mb \vee Céb \vee Cob \dots$ clause 5.1 $Mb \vee Nb \vee Cob \dots$ clause 5.2

$((Mc \vee (Céc \vee Nc)) \vee Coc)$; devient $(Mc \vee (Céc \vee Nc)) \vee Coc$ et donc $Mc \vee Céc \vee Coc \dots$ clause 6.1 $Mc \vee Nc \vee Coc \dots$ clause 6.2

1.3.3 transformation des classes en format CNF :

```
p cnf 12 9
-4 1 -10 0
-5 2 -11 0
-6 3 -12 0
-7 4 10 0
-7 -1 10 0
-8 5 11 0
-8 -2 11 0
-9 6 12 0
-9 -3 12 0
```

FIGURE 1.6 – Le fichier zoo.cnf

On exécute le solveur SAT pour ce fichier : On obtient la solution suivante : 1 2 3 4 -5 6 7 -8 9 10 -11 12 -Le solveur a fournit le modèle Na et non Nb et Nc et non Cés et non Céb et Céc et Ma

```
# Solution found for -target 0

1 -2 3 -4 -5 6 7 -8 9 10
-11 12

Variables = 12
Clauses = 9
TotalLiterals = 27
TotalCPUTimeElapsed = 0.006
FlipsPerSecond = 333
RunsExecuted = 1
SuccessfulRuns = 1
PercentSuccess = 100.00
Steps_Mean = 2
Steps_CoeffVariance = 0
Steps_Median = 2
CPUTime_Mean = 0.00599980354309
CPUTime_CoeffVariance = 0
CPUTime_Median = 0.00599980354309

C:\ubcsat-1-1-0>
```

FIGURE 1.7 – – Solution de la base de connaissances zoologiques

et non Mb et Mc et Coa et non Cob et Coc => il existe au moins un modèle pour cette base de connaissance donc elle est satisfiable

1.3.4 Test des fichiers Benchmarks

Un fichier benchmark est un fichier contenant des données ou des instructions utilisées pour tester les performances ou les fonctionnalités d'un logiciel, d'un système ou d'un appareil. Dans le contexte des problèmes de satisfiabilité booléenne (SAT), un fichier benchmark CNF (Conjunctive Normal Form) est spécifiquement conçu pour tester les performances des solveurs SAT.

Télécharger un fichier benchmark

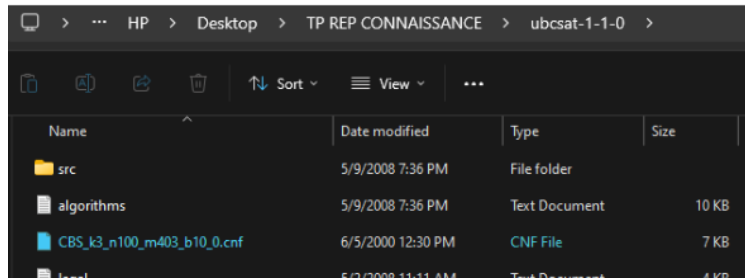


FIGURE 1.8 – Téléchargement du fichier benchmark

Tester le fichier benchmark

```
# 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

Variables = 100
Clauses = 403
TotalLiterals = 1209
TotalCPUtimeElapsed = 0.001
FlipsPerSecond = 126979
RunsExecuted = 1
SuccessfulRuns = 1
PercentSuccess = 100.00
Steps_Mean = 127
Steps_CoeffVariance = 0
Steps_Median = 127
CPUtime_Mean = 0.00100016593933
```

FIGURE 1.9 – – Solution du fichier benchmark

- Le solveur a fourni un modèle => il existe au moins un modèle pour cette base de connaissance donc elle est satisfiable

1.4 Étape 4

Soit la même base de connaissances de l'exemple 1 et soit le but a , afin de prouver a il faut ajouter sa négation à la base de connaissances et trouver une contradiction en pratique cela revient

à exécuter la base avec le solveur sat et trouver aucune solution.

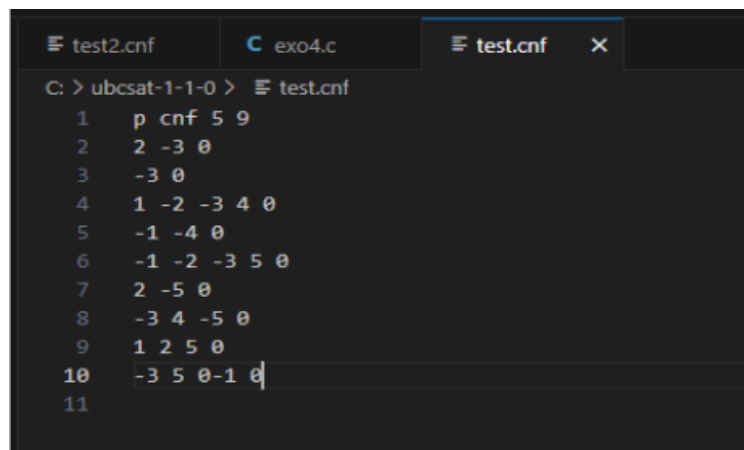
1.4.1 Code source du programme

FIGURE 1.10 – Programme C qui détermine si un but est inféré ou non par une base

1.4.2 Compilation et exécution

FIGURE 1.11 – Compilation et exécution

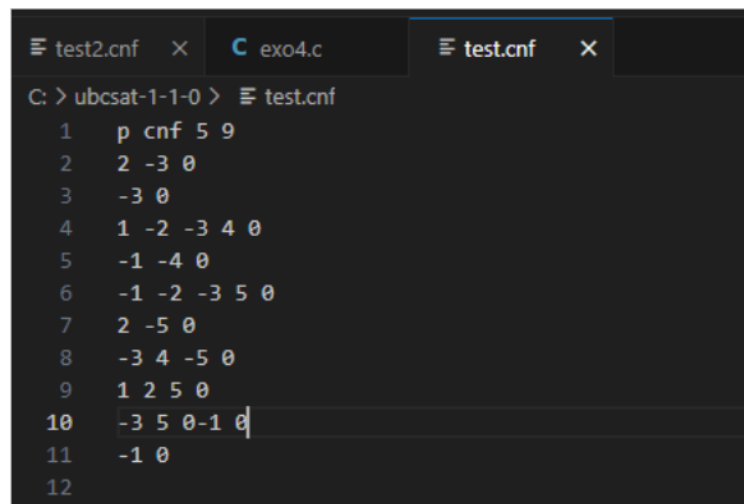
1.4.3 Le fichier test.cnf avant l'exécution (ajout du littéral non a , en format cnf -1) :



```
test2.cnf  C  exo4.c  test.cnf  X
C: > ubcsat-1-1-0 > test.cnf
1  p cnf 5 9
2  2 -3 0
3  -3 0
4  1 -2 -3 4 0
5  -1 -4 0
6  -1 -2 -3 5 0
7  2 -5 0
8  -3 4 -5 0
9  1 2 5 0
10 -3 5 0 -1 0
11
```

FIGURE 1.12 – Le fichier test.cnf avant l'exécution

1.4.4 Le fichier test.cnf après l'exécution de exo4.c :



```
test2.cnf  X  C  exo4.c  test.cnf  X
C: > ubcsat-1-1-0 > test.cnf
1  p cnf 5 9
2  2 -3 0
3  -3 0
4  1 -2 -3 4 0
5  -1 -4 0
6  -1 -2 -3 5 0
7  2 -5 0
8  -3 4 -5 0
9  1 2 5 0
10 -3 5 0 -1 0
11 -1 0
12
```

FIGURE 1.13 – Le fichier test.cnf après l'exécution

1.4.5 résultat de l'exécution :

```
#
# Solution found for -target 0

1 -2 -3 -4 -5

Variables = 5
Clauses = 9
TotalLiterals = 23
TotalCPUtimeElapsed = 0.000
FlipsPerSecond = 1
RunsExecuted = 1
SuccessfulRuns = 1
PercentSuccess = 100.00
Steps_Mean = 2
Steps_CoeffVariance = 0
Steps_Median = 2
CPUtime_Mean = 0
CPUtime_CoeffVariance = 0
CPUtime_Median = 0

C:\ubcsat-1-1-0>
```

FIGURE 1.14 – résultat de l'exécution

- Le solveur sat a pu trouver une solution et n'a pas générer une contradiction donc le fait a ne peut pas être prouvé

1.5 Modélisation d'un problème et le résoudre avec le solveur sat

1.5.1 Modélisation : Coloriage de graphe

Etant donné un graphe non-orienté, colorier une couleur à un sommet de manière que deux sommets adjacents ne peuvent pas recevoir une de même couleur. Le problème consiste à trouver un nombre minimal de couleurs.

Exemple : Notre graphe contient 4 etats P1,P2,P3 et P4 et on a 3 couleurs :

Représenter les éléments du problème en variables propositionnelles $4 \times 3 = 12$ variables propositionnelles :

$x_{ij} = 1$, si le noeud i est colorié de couleur j

$x_{ij} = 0$, sinon 3 couleurs : violet (1), blue (2), rouge (3)

Le noeud 1 reçoit une seule couleur :

- $x_{11} \vee x_{12} \vee x_{13}$
- $X_{11} \vee X_{12}$
- $X_{11} \vee X_{13}$

- $x_{12} \vee x_{13}$

Le noeud 1 et le noeud 2 ne peuvent pas recevoir une même couleur :

- $X_{11} \vee X_{21}$
- $X_{12} \vee X_{22}$
- $X_{13} \vee X_{23}$
- etc ..

1.5.2 Les clauses de la base :

```

1. Les clauses
(X11 V X12 V x13)
(211 V ¬312)
(¬x11 V ¬x13)
(¬x12 V ¬x13)
(¬x11 V ¬x21)
(¬X12 V ¬X22)
(¬x13 V ¬x23)
(¬x21 V ¬x22)
(¬x21 V ¬x23)
(¬x22 V ¬x23)
(x21 V x22 V x23)
(¬x21 V ¬x31)
(¬x22 V ¬X32)
(¬x23 V ¬X33)
(¬x31 V ¬X32)
(¬X31 V ¬x33)
(¬X32 V ¬X33)
(X31 V X32 V X33)
(¬x32 V ¬x42)
(¬X33 V ¬x43)
(¬X41 V ¬X42)
(¬x41 V ¬X43)
(¬x42 V ¬x43)
(X41 V X42 V X43)
    
```

FIGURE 1.15 – Les clauses de la base

1.5.3 les classes en format cnf :

```
en format CNF :
x11 : 1
x12 : 2
x13 : 3
x21 : 4
x22 : 5
x23 : 6
x31 : 7
x32 : 8
x33 : 9
x41 : 10
x42 : 11
x43 : 12
```

FIGURE 1.16 – les classes en format cnf

1.5.4 la base de faits :

```
p cnf 12 24
1 2 3 0
-1 -2 0
-1 -3 0
-2 -3 0
-1 -4 0
-2 -5 0
-3 -6 0
-4 -5 0
-4 -6 0
-5 -6 0
4 5 6 0
-4 -7 0
-5 -8 0
-6 -9 0
-7 -8 0
-7 -9 0
-8 -9 0
7 8 9 0
-8 -10 0
-9 -11 0
-10 -11 0
-10 -12 0
-11 -12 0
10 11 12 0
```

FIGURE 1.17 – la base de faits

1.5.5 L'exécution avec le solveur SAT :

```
# Solution found for -target 0

 1 -2 -3 -4 5 -6 7 -8 -9 -10
11 -12

Variables = 12
Clauses = 24
TotalLiterals = 52
TotalCPUTimeElapsed = 0.000
FlipsPerSecond = 1
RunsExecuted = 1
SuccessfulRuns = 1
PercentSuccess = 100.00
Steps_Mean = 6
Steps_CoeffVariance = 0
Steps_Median = 6
CPUTime_Mean = 0
CPUTime_CoeffVariance = 0
CPUTime_Median = 0
```

FIGURE 1.18 – résultat de l'exécution

- Le solveur a fournit un modèle => il existe au moins un modèle pour cette base de connaissance donc elle est satisfiable

Dans ce TP2, nous allons exploiter la librairie Java Tweety pour la modélisation des connaissances en logique des prédicats.

2.1 La librairie Java Tweety

La librairie Tweety est une bibliothèque Java conçue pour la manipulation et la raison sur des structures de données logiques telles que les ontologies, les réseaux bayésiens, les logiques de description, etc. Elle fournit des fonctionnalités pour la représentation et le raisonnement sur ces données, facilitant ainsi le développement d'applications intelligentes dans des domaines tels que l'intelligence artificielle, la logique formelle et la gestion des connaissances.

2.2 Exemple de logique des prédicats

Elle contient des informations sur les symboles utilisés dans les formules, tels que les sorts (types), les constantes et les prédicats.

2.2.1 définition de la signature

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

2.2.2 Ajout des types

```
Sort sortAnimal = new Sort("Animal");  
sig.add(sortAnimal);
```

2.2.3 Ajout des constantes

```
Constant constantPenguin = new Constant("penguin", sortAnimal);  
Constant constantKiwi = new Constant("kiwi", sortAnimal);  
sig.add(constantPenguin, constantKiwi);
```

2.2.4 Ajout des prédicats

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

predicateList.add(sortAnimal);
Predicate p = new Predicate("Flies", predicateList);
List<Sort> predicateList2 = new ArrayList<Sort>();
predicateList2.add(sortAnimal);
predicateList2.add(sortAnimal);
Predicate p2 = new Predicate("Knows", predicateList2);

//Add Predicate Knows(Animal, Animal)
sig.add(p, p2);
```

2.2.5 Percement de connaissances

```
FolParser parser = new FolParser();

parser.setSignature(sig); //Use the signature defined above
FolBeliefSet bs = new FolBeliefSet();

FolFormula f1 = (FolFormula) parser.parseFormula("! Flies (kiwi)");
FolFormula f2 = (FolFormula) parser.parseFormula(" Flies (penguin)");
FolFormula f3 = (FolFormula) parser.parseFormula("!Knows(penguin , kiwi)");
FolFormula f4 = (FolFormula) parser.parseFormula("/==(penguin , kiwi)");
FolFormula f5 = (FolFormula) parser.parseFormula("kiwi_==_kiwi");

bs.add(f1 , f2 , f3 , f4 , f5);
System.out.println("\nParsed_BeliefBase:_ " + bs);
```

2.2.6 Utilisation du raisonneur

Des requêtes sont effectuées en utilisant le raisonneur sur la base de croyances pour vérifier si certaines formules peuvent être inférées à partir des connaissances existantes.

```
(FolFormula) parser.parseFormula(" Flies (kiwi)");
(FolFormula) parser.parseFormula(" forall_X:
(exists_Y:_ ( Flies (X) _&&_ Flies (Y) _&&_X/==Y))");
(FolFormula) parser.parseFormula("kiwi_==_kiwi");
(FolFormula) parser.parseFormula("kiwi_/_==_kiwi");
(FolFormula) parser.parseFormula("penguin_/_==_kiwi");
```

2.2.7 Résultat obtenu

```
ANSWER 1: false
ANSWER 2: false
ANSWER 3: true
ANSWER 4: false
ANSWER 5: true
```

FIGURE 2.1 – résultat de l'exécution

2.2.8 Interprétation du Résultat

Ce résultat signifi que nous avons pu inférer les connaissances `kiwi == kiwi` et `penguin /== kiwi` depuis la base connaissances :

- `!Flies(kiwi)`
- `Flies(penguin)`
- `!Knows(penguin,kiwi)`
- `/==(penguin,kiwi)`
- `kiwi == kiwi`

En revanche les connaissances `Flies(kiwi)` , forall `X : (exists Y : (Flies(X) Flies(Y) X/==Y))` et `kiwi /== kiwi` n'ont pas pu être inférées

2.3 Implémentation de l'exemple du cours avec tweety

Elle contient des informations sur les symboles utilisés dans les formules, tels que les sorts (types), les constantes et les prédicats.

2.3.1 définition de la signature

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

2.3.2 Ajout des types

```
Sort sortAnimal = new Sort("Animal");
sig.add(sortAnimal);
```

2.3.3 Ajout des constantes

```
Constant constantB = new Constant("b", sortAnimal);
Constant constantC = new Constant("c", sortAnimal);
Constant constantA = new Constant("a", sortAnimal);
sig.add(constantB, constantC, constantA);
```

2.3.4 Ajout des prédicats

```
List<Sort> predicateListCeph = new ArrayList<Sort>();
predicateListCeph.add(sortAnimal);
Predicate cephPredicate = new Predicate("ceph", predicateListCeph);
sig.add(cephPredicate);

List<Sort> predicateListNaut = new ArrayList<Sort>();
predicateListNaut.add(sortAnimal);
Predicate nautPredicate = new Predicate("naut", predicateListNaut);
sig.add(nautPredicate);
```

```

List<Sort> predicateListMol = new ArrayList<Sort>();
predicateListMol.add(sortAnimal);
Predicate molPredicate = new Predicate("mol", predicateListMol);
sig.add(molPredicate);

List<Sort> predicateListAcoq = new ArrayList<Sort>();
predicateListAcoq.add(sortAnimal);
Predicate acoqPredicate = new Predicate("acoq", predicateListAcoq);
sig.add(acoqPredicate);

```

2.3.5 Percement de connaissances

```

FolFormula f1 = (FolFormula) parser.parseFormula("forall X: (naut(X) => cephalopode(X))");
FolFormula f2 = (FolFormula) parser.parseFormula("forall X: (cephalopode(X) => mol(X))");
FolFormula f3 = (FolFormula) parser.parseFormula("forall X: ((mol(X) && ! (cephalopode(X) && ! naut(X))) => acoq(X))");
FolFormula f4 = (FolFormula) parser.parseFormula("forall X: ((cephalopode(X) && ! naut(X)) => ! acoq(X))");
FolFormula f5 = (FolFormula) parser.parseFormula("forall X: (naut(X) => acoq(X))");
FolFormula f6 = (FolFormula) parser.parseFormula("naut(a)");
FolFormula f7 = (FolFormula) parser.parseFormula("cephalopode(b)");
FolFormula f8 = (FolFormula) parser.parseFormula("mol(c)");
bs.add(f1, f2, f3, f4, f5, f6, f7, f8);
System.out.println("\nParsed BeliefBase: " + bs);

```

2.3.6 Utilisation du raisonneur

Des requêtes sont effectuées en utilisant le raisonneur sur la base de croyances pour vérifier si certaines formules peuvent être inférées à partir des connaissances existantes.

```

"ANSWER 1: " + prover.query(bs, (FolFormula) parser.parseFormula("acoq(a)"));
"ANSWER 2: " + prover.query(bs, (FolFormula) parser.parseFormula("acoq(b)"));
"ANSWER 3: " + prover.query(bs, (FolFormula) parser.parseFormula("!acoq(c)"));

```

2.3.7 Résultat obtenu

```

ANSWER 1: true
ANSWER 2: false
ANSWER 3: false

```

FIGURE 2.2 – résultat de l'exécution

2.3.8 Interprétation du Résultat

Chaque réponse est une affirmation logique basée sur la base de croyances analysée.

"ANSWER 1 : true" : La requête "acoq(a)" est vraie, ce qui signifie que a a une coquille.

"ANSWER 2 : false" : La requête "acoq(b)" est fausse, ce qui signifie que b n'a pas de coquille.

"ANSWER 3 : false" : La requête "!acoq(c)" est fausse, ce qui signifie que c n'a pas de coquille.

Le résultat montre les inférences effectuées à partir de la base de croyances FOL, ainsi que les réponses aux requêtes posées sur ces croyances.

2.4 Implémentation d'un exemple

Supposons que nous ayons une base de connaissances contenant des déclarations sur des légumes. Nous voulons utiliser la logique du premier ordre (FOL) pour inférer des informations à partir de cette base de connaissances.

2.4.1 définition de la signature

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

2.4.2 Ajout des types

```
Sort sortVegetable = new Sort("Vegetable");
sig.add(sortVegetable);
```

2.4.3 Ajout des constantes

```
Constant constantTomato = new Constant("tomato", sortVegetable);
Constant constantCarrot = new Constant("carrot", sortVegetable);
Constant constantBroccoli = new Constant("broccoli", sortVegetable);
sig.add(constantTomato, constantCarrot, constantBroccoli);
```

2.4.4 Ajout des prédicats

```
List<Sort> predicateListEdible = new ArrayList<>();
predicateListEdible.add(sortVegetable);
Predicate ediblePredicate = new Predicate("edible", predicateListEdible);
sig.add(ediblePredicate);

List<Sort> predicateListRed = new ArrayList<>();
predicateListRed.add(sortVegetable);
Predicate redPredicate = new Predicate("red", predicateListRed);
sig.add(redPredicate);

List<Sort> predicateListGreen = new ArrayList<>();
predicateListGreen.add(sortVegetable);
Predicate greenPredicate = new Predicate("green", predicateListGreen);
sig.add(greenPredicate);
```

2.4.5 Percement de connaissances

```
FolFormula f1 = (FolFormula) parser.parseFormula("forall _X: _ (red(_X) => _ edible(_X))");
FolFormula f2 = (FolFormula) parser.parseFormula("forall _X: _ (green(_X) => _ edible(_X))");
FolFormula f3 = (FolFormula) parser.parseFormula("forall _X: _ (edible(_X) => _ (red(_X) || _ green(_X)))");
FolFormula f4 = (FolFormula) parser.parseFormula("forall _X: _ (red(_X) && _ green(_X))");
FolFormula f5 = (FolFormula) parser.parseFormula("edible(tomato)");
FolFormula f6 = (FolFormula) parser.parseFormula("edible(carrot)");
FolFormula f7 = (FolFormula) parser.parseFormula("!edible(broccoli)");
bs.add(f1, f2, f3, f4, f5, f6, f7);
```

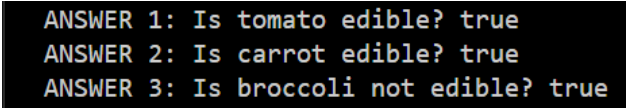
```
System.out.println("\nParsed_BeliefBase:_ " + bs);
```

2.4.6 Utilisation du raisonneur

Des requêtes sont effectuées en utilisant le raisonneur sur la base de croyances pour vérifier si certaines formules peuvent être inférées à partir des connaissances existantes.

```
FolReasoner.setDefaultReasoner(new SimpleFolReasoner());
FolReasoner prover = FolReasoner.getDefaultReasoner();
System.out.println("ANSWER_1:_Is_tomato_edible?_");
+ prover.query(bs, (FolFormula) parser.parseFormula("edible(tomato)"));
System.out.println("ANSWER_2:_Is_carrot_edible?_");
+ prover.query(bs, (FolFormula) parser.parseFormula("edible(carrot)"));
System.out.println("ANSWER_3:_Is_broccoli_not_edible?_");
+ prover.query(bs, (FolFormula) parser.parseFormula("!edible(broccoli)"));
System.out.println("ANSWER_4:_Is_there_any_vegetable_that_is_both_red_and_green?_");
+ prover.query(bs, (FolFormula) parser.parseFormula("exists_X:_(red(X)&&green(X))"));
```

2.4.7 Résultat obtenu



```
ANSWER 1: Is tomato edible? true
ANSWER 2: Is carrot edible? true
ANSWER 3: Is broccoli not edible? true
```

FIGURE 2.3 – résultat de l'exécution

2.4.8 Interprétation du Résultat

Ce résultat indique que nous avons pu inférer les connaissances suivantes depuis la base de connaissances :

Tomate est comestible : edible(tomato) Carotte est comestible : edible(carrot) Brocoli n'est pas comestible : !edible(broccoli)

Dans ce TP3, nous allons exploiter la librairie Java Tweety pour la modélisation des connaissances en logique modale.

3.1 Étude d'un exemple

Afin d'exploiter la librairie Tweety, on va écrire un programme qui utilise cette bibliothèque pour effectuer des raisonnements sur des formules d'un modèle de logique modale.

3.1.1 Importation des bibliothèques

```
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;
```

3.1.2 Parsemant de la base de connaissances

```
MlParser parser = new MlParser();
MlBeliefSet b1 = parser.parseBeliefBaseFromFile("examplebeliefbase2.mlogic");
```

3.1.3 La base de connaissances

```
type(A)
type(B)
(<>(A || B)) <=> ((<>(A) || <>(B)))
<>(¬)
<>(A) && (|| (B))
```

$\langle \rangle (A \parallel B) \Leftrightarrow ((\langle \rangle (A)) \parallel (\langle \rangle (B)))$: Cette formule établit une équivalence entre deux assertions modales. Elle exprime que dans chaque monde possible, la disjonction de A et B est possible si et seulement si au moins une des propositions A ou B est possible dans chaque monde possible. Cela reflète la propriété de la disjonction modale. Si A ou B est possible dans chaque monde, alors leur disjonction est également possible dans chaque monde.

$\langle \rangle (-)$: Cette formule indique que quelque chose est possible. Cela ne spécifie pas quoi, mais simplement que quelque chose est possible dans au moins un monde possible.

$\langle \rangle (A) \ (\langle \rangle (B))$: Cette formule signifie que dans au moins un monde possible, A est possible et dans au moins un monde possible, B est possible. Cela reflète la conjonction de deux assertions modales.

3.1.4 Préparation de la formule

```
FolFormula f1 = (FolFormula) parser.parseFormula("<>(A&&B)");
System.out.println("Parsed_belief_base:" + b1 +
"\nSignature_of_belief_base:" + b1.getMinimalSignature());
System.out.println("Parsed_formula:" + f1);
```

3.1.5 Le raisonnement

```
SimpleMlReasoner reasoner = new SimpleMlReasoner();
System.out.println("Answer_to_query:_ " + reasoner.query(b1, f1));
```

3.1.6 Résultat

```
Parsed belief base:{ <>(A)&&[](B), (<>(A||B)<=><>(A)||<>(B)), <>(-) }
Signature of belief base:[], [A, B], []
Parsed formula:<>(A&&B)
Parsed belief base:{ Flies(eagle) }
Parsed formula:Flies(penguin)||!Flies(penguin)
Parsed belief base:{ Eats(eagle,worm), <>(Flies(eagle)), [](forall X: ([
(Flies(X))||<>(!HasWings(X)))), )), !<>(Eats(worm,p
enguin)) }
Signature of belief base:[Animal = {eagle, penguin, worm}, Plant = {cactu
s}], [HasWings(Animal), Eats(Animal,Animal), Stings(Plant), Flies(Animal)
], []
Answer to query: true
Answer to query: true
```

FIGURE 3.1 – résultat de l'exécution

Donc la formule possible de (A et B) a été prouvée

3.1.7 Interprétation du résultat

La formule $\langle \rangle (-)$ indique que quelque chose est possible. Cela ne spécifie pas quoi, mais simplement que quelque chose est possible dans au moins un monde possible.

Donc pour n'importe quelle formule le possible de cette formule est vrai selon cette base de connaissance et donc le possible de $A \rightarrow B$ est vrai.

3.2 Modélisation d'un exemple

Soit un système de sécurité d'une maison. Notre système de sécurité peut avoir différents états et comportements .

États du système :

- Alarme activée (A)
- Présence détectée à l'intérieur de la maison (P)

contraintes modales : $A \rightarrow P \Rightarrow \Diamond(A \wedge P)$ Si le système a l'alarme activée et une présence est détectée alors le possible de cette formule sachant que la relation est réflexive.

$A \Rightarrow \Diamond(P)$

si l'alarme est activée alors, il est possible qu'une présence ait détectée

Formule 1 à inférer : Il est possible que l'alarme soit activée et qu'une présence soit détectée simultanément.

Formulation de la formule : $\Diamond(A \wedge P)$

Formule 2 à inférer : Il est possible qu'une présence soit détectée.

Formulation de la formule : $\Diamond(P)$

3.3 Implémentation

3.3.1 Parsemant de la base de connaissances

```

MlParser parser = new MlParser();
MlBeliefSet b1 = parser.parseBeliefBaseFromFile("beliefbase.mlogic");

```

3.3.2 La base de connaissances

```

type(A)
type(P)
(A && P)
(A && P) => Diamond(A && P)
A => Diamond(P)

```

3.3.3 Préparation de la formule 1

```

FolFormula f1 = (FolFormula) parser.parseFormula("<>(A&&P)");
System.out.println("Parsed_belief_base:" + b1 +
"\nSignature_of_belief_base:" + b1.getMinimalSignature());
System.out.println("Parsed_formula:" + f1);

```

3.3.4 Le raisonnement

```
SimpleMlReasoner reasoner = new SimpleMlReasoner();
System.out.println("Answer_to_query:_" + reasoner.query(b1, f1));
```

3.3.5 Résultat

```
] --- exec:3.1.0:exec (default-cli) @ mavenproject1 ---
  Parsed belief base:{ A&&P, (A&&P=><>(A&&P)), (A=><>(P)) }
  Signature of belief base:[], [P, A], []
  Parsed formula:<>(A&&P)
- Answer to query: true
```

FIGURE 3.2 – résultat de l'exécution

3.3.6 Préparation de la formule 2

```
FolFormula f2 = (FolFormula) parser.parseFormula("<>(A&&P)");
System.out.println("Parsed_belief_base:" + b1 +
"\nSignature_of_belief_base:" + b1.getMinimalSignature());
System.out.println("Parsed_formula:" + f2);
```

3.3.7 Résultat

```
] --- exec:3.1.0:exec (default-cli) @ mavenproject1 ---
  Parsed belief base:{ A&&P, (A&&P=><>(A&&P)), (A=><>(P)) }
  Signature of belief base:[], [P, A], []
  Parsed formula:<>(P)
- Answer to query: true

-----
BUILD SUCCESS
```

FIGURE 3.3 – résultat de l'exécution

3.3.8 Interprétation du résultat

pour la formule $\langle \rangle (AP)$ nous avons pu l'inférer grâce à la connaissance $A \ P \Rightarrow \langle \rangle (A \ P)$ et le fait que A et B soit vrai.

pour la formule $\langle \rangle (P)$ nous avons pu l'inférer grâce à la connaissance $A \Rightarrow \langle \rangle (P)$ et le fait que A est vrai.

La logique des défauts, aussi connue sous le nom de logique non monotone, se concentre sur la représentation des connaissances qui sont soit incomplètes soit incertaines. Contrairement à la logique classique, qui est monotone, c'est-à-dire que l'ajout de nouvelles informations ne peut que renforcer les conclusions précédentes, la logique des défauts permet la révision ou la modification des conclusions précédentes lorsque de nouvelles informations sont introduites.

En d'autres termes, la logique des défauts représente les connaissances sous forme de règles avec des exceptions ou des conditions spéciales qui peuvent être activées ou désactivées en fonction du contexte. Cette approche permet de modéliser des situations où une règle est généralement valide, mais peut présenter des exceptions

Dans ce TP, nous allons explorer la logique des défauts en implémentant un exemple.

4.1 Exemple

Nous avons employé l'outil DefaultLogic pour ce TP, un puissant instrument utilisé en logique formelle pour le raisonnement par défaut. Il offre une structure pour la représentation des connaissances et le raisonnement dans des contextes où l'information est incomplète ou incertaine.

4.1.1 Explication des lignes du code

un défaut est constitué de trois parties :

- Un prérequis (prerequisite)
- Une Justification (Justificatoin).
- Une conséquence (Consequence).

Un défaut se produit dans un monde , un monde contient des formules.

tous d'abord, on commence par créer un monde, et on met les formules au début a nulle :

```
WorldSet myWorld = new WorldSet();  
myWorld.addFormula(a.e.EMPTY_EFFECT);
```

FIGURE 4.1 – Création d'un monde

puis, on rajoute les formules

```
myWorld.addFormula(_wff: "B"); // B est vrai  
myWorld.addFormula(_wff: "(A -> Z)"); // Règle supplémentaire pour tester la clôture  
myWorld.addFormula(_wff: "(C -> (D | A))"); // C implique soit D, soit A, soit les deux  
// D et A  
myWorld.addFormula(_wff: "((A & C) -> ~E)"); // A et C impliquent non E
```

FIGURE 4.2 – Les formules du monde

ensuite, on ajoute les règles

```
DefaultRule rule1 = new DefaultRule();  
rule1.setPrerequisite(a.e.EMPTY_EFFECT);  
rule1.setJustificatoin("A");  
rule1.setConsequence("A");  
  
DefaultRule rule2 = new DefaultRule();  
rule2.setPrerequisite("B");  
rule2.setJustificatoin("C");  
rule2.setConsequence("C");  
  
DefaultRule rule3 = new DefaultRule();  
rule3.setPrerequisite("(D & A)");  
rule3.setJustificatoin("E");  
rule3.setConsequence("E");  
  
DefaultRule rule4 = new DefaultRule();  
rule4.setPrerequisite("C & E");  
rule4.setJustificatoin("(~A) & (D | A)");  
rule4.setConsequence("F");
```

FIGURE 4.3 – Les règles du monde

Par la suite, nous ajoutons toutes les règles à l'ensemble des règles.

```
RuleSet myRules = new RuleSet();
myRules.addRule(rule1);
myRules.addRule(rule2);
myRules.addRule(rule3);
myRules.addRule(rule4);
```

FIGURE 4.4 – Ensemble des règles

Nous allons maintenant résoudre le problème à l'aide d'un raisonneur et nous chargeons ensuite le monde avec les règles qui l'impliquent et nous extrayons toutes les extensions placées dans un hashSet

```
DefaultReasoner loader = new DefaultReasoner(myWorld, myRules);
HashSet<String> extensions = loader.getPossibleScenarios();
```

FIGURE 4.5 – Création du raisonneur et extraction des extensions

4.1.2 Résultat

```
Trying B & (A -> Z) & (C -> (D | A)) & ((A & C) -> ~E)
Trying B & (A -> Z) & (C -> (D | A)) & ((A & C) -> ~E)
Trying B & (A -> Z) & (C -> (D | A)) & ((A & C) -> ~E)
Trying B & (A -> Z) & (C -> (D | A)) & ((A & C) -> ~E)
Trying B & (A -> Z) & (C -> (D | A)) & ((A & C) -> ~E)
Trying B & (A -> Z) & (C -> (D | A)) & ((A & C) -> ~E)
Trying B & (A -> Z) & (C -> (D | A)) & ((A & C) -> ~E)
Trying B & (A -> Z) & (C -> (D | A)) & ((A & C) -> ~E)
Étant donné le monde :
    B & (A -> Z) & (C -> (D | A)) & ((A & C) -> ~E)
Et les règles :
    [([]):(A) ==> (A)] , [(B):(C) ==> (C)] , [((D & A)):(E) ==> (E)] , [(C & E):(((~A) & (D | A))) ==> (F)]
Extensions possibles :
    Ext: Th(W U (C & A))
    = B & (~A | ~C | ~E) & C & ~E & Z & (Z | ~A) & (A | ~C) & (D | A | ~C) & A
-----
Le temps d'exécution était de 8352 ms.
```

FIGURE 4.6 – résultat de l'exécution

4.1.3 Interprétation du résultat

1. Le programme essaie différentes combinaisons de formules logiques basées sur le monde initial et les règles fournies. Ces combinaisons sont générées pour déterminer les extensions possibles du système.
2. Après plusieurs essais, le programme imprime le monde initial et les règles qui lui sont associées.
3. Ensuite, il affiche les extensions possibles du système, où chaque extension est une combinaison de formules logiques qui peuvent être vraies dans un certain contexte.
4. Pour chaque extension, le programme imprime une représentation logique de celle-ci, utilisant les opérateurs logiques et les formules du monde initial et des règles.
5. Enfin, le programme affiche le temps d'exécution total.

5.1 Introduction

Au sein de ce TP, notre focalisation se dirige vers l'implémentation des réseaux sémantiques, en mettant en avant la réalisation pratique de divers algorithmes qui leur sont associés.

5.2 Réseaux sémantiques

Les réseaux sémantiques sont des structures de données utilisées en intelligence artificielle et en traitement automatique du langage naturel pour représenter des connaissances sous forme de graphes. Chaque nœud dans un réseau sémantique représente un concept, et les arcs entre les nœuds représentent les relations sémantiques entre ces concepts. Ces relations peuvent inclure des liens tels que "est-un", "a", "fait-partie-de", etc. Les réseaux sémantiques permettent de modéliser des informations complexes de manière organisée, ce qui les rend utiles pour diverses tâches telles que la recherche d'informations, la compréhension de texte, la résolution de problèmes, etc.

5.3 Partie 1 : implémenter l'algorithme de propagation de marqueurs dans les réseaux sémantiques

Dans cette partie, nous allons implémenter l'algorithme de propagation de marqueurs vu en cours :

```

semantic_networks > algorithms > propagation.py > ...
1 import json
2
3 def get_label(reseau_semantique, node, relation):
4     node_relation_edges = [edge["from"] for edge in reseau_semantique["edges"] if (edge["to"] == node["id"] and edge["label"] == relation)]
5     node_relation_edges_label = [node["label"] for node in reseau_semantique["nodes"] if node["id"] in node_relation_edges]
6     reponse = "il y a un lien entre les 2 noeuds : " + ", ".join(node_relation_edges_label)
7     return reponse
8
9 def propagation_de_marqueurs(reseau_semantique, node1, node2, relation):
10     nodes = reseau_semantique["nodes"]
11
12     solutions_found = []
13
14     for i in range(min(len(node1), len(node2))):
15         solution_found = False
16
17         try:
18             M1 = [node for node in nodes if node["label"] == node1[i][0]]
19             M2 = [node for node in nodes if node["label"] == node2[i][0]]
20
21             edges = reseau_semantique["edges"]
22             propagation_edges = [edge for edge in edges if (edge["to"] == M1["id"] and edge["label"] == "is a")]
23             while len(propagation_edges) != 0 and not solution_found:
24                 temp_node = propagation_edges.pop()
25                 temp_node_contient_edges = [edge for edge in edges if (edge["from"] == temp_node["from"] and edge["label"] == relation)]
26                 solution_found = any(d["to"] == M2["id"] for d in temp_node_contient_edges)
27                 if not solution_found:
28                     temp_node_is_a_edges = [edge for edge in edges if (edge["to"] == temp_node["from"] and edge["label"] == "is a")]
29                     propagation_edges.extend(temp_node_is_a_edges)
30
31             solutions_found.append(get_label(reseau_semantique, M2, relation) if solution_found else "il n'y a pas un lien entre les 2 noeuds")
32         except IndexError:
33             solutions_found.append("Aucune reponse n'est fournie par manque de connaissances.")
34
35     return(solutions_found)

```

FIGURE 5.1 – Algorithme de propagation de marqueur

Pour tester l'algorithme, nous allons utiliser le réseau sémantique fourni par le lien qui se trouvent dans la série du TP.

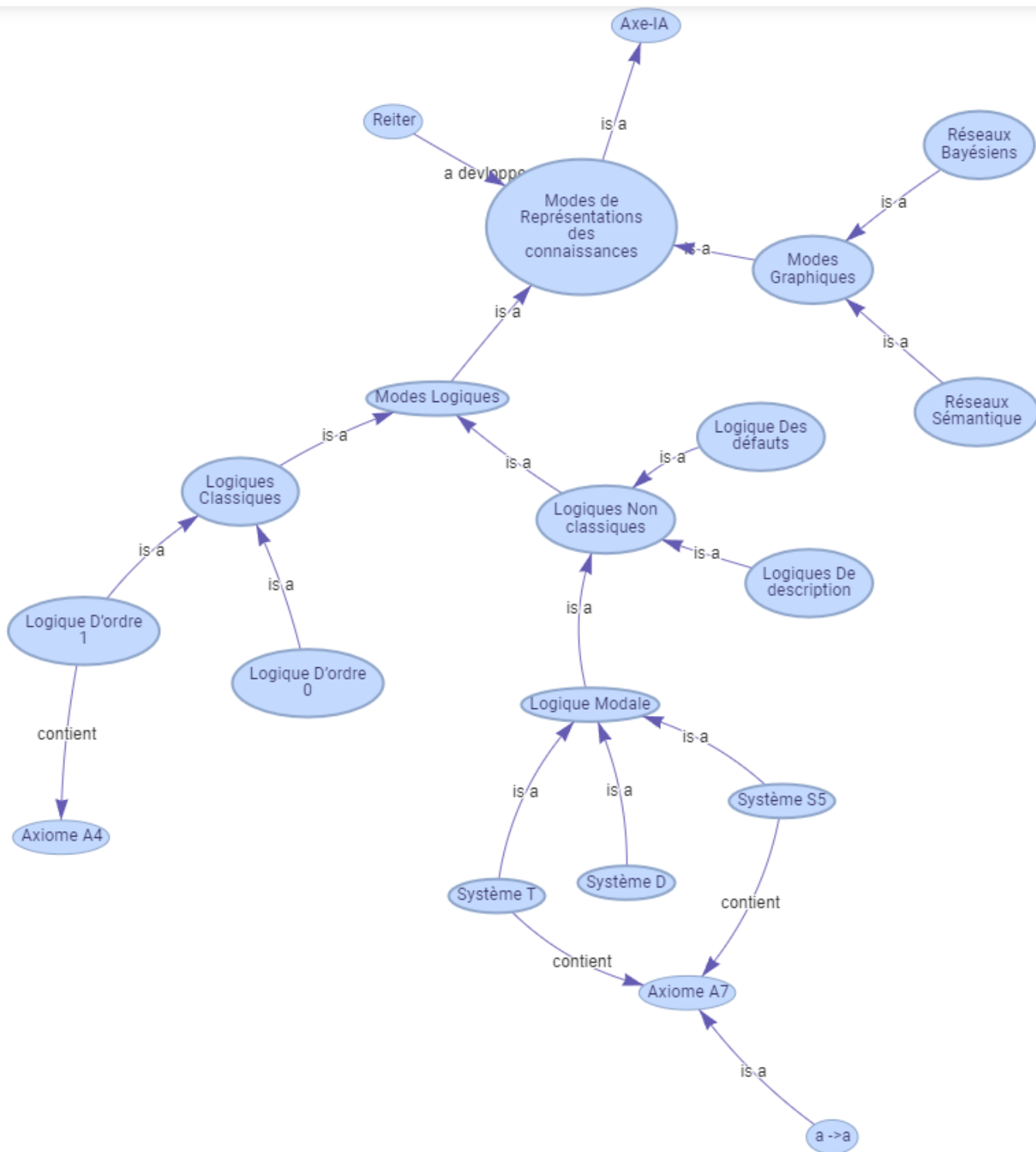


FIGURE 5.2 – Réseau sémantique utilisé pour la partie 1

on teste en utilisant plusieurs nœuds en entrée, et on visualise le résultat suivant :

```

Partie 1: l'algorithm de propagation de marqueurs
Modes de Representations des connaissances contient Axiome A7
il y a un lien entre les 2 noeuds : Systeme T, Systeme S5
Modes de Representations des connaissances contient Axiome A4
il y a un lien entre les 2 noeuds : Logique D ordre 1
Modes de Representations des connaissances contient Axe-IA
il n'y a pas un lien entre les 2 noeuds
Modes de Representations des connaissances contient Axiome A9
Aucune reponse n'est fournie par manque de connaissances.

```

FIGURE 5.3 – Résultat obtenu par l'algorithme de propagation de marqueurs

on remarque que l'algorithme a bien trouvé tous les liens entre les nœuds entrés.

5.4 Partie 2 : implémenter l'algorithme d'héritage

Dans cette partie, nous allons implémenter l'algorithme d'héritage vu en cours :

```

import json

def get_label(reseau_semantique, node_id):
    node = next((node for node in reseau_semantique["nodes"] if node["id"] == node_id), None)
    if node:
        return node.get("label")
    else:
        return None

def heritage(reseau_semantique, name):
    the_end = False

    nodes = reseau_semantique["nodes"]
    edges = reseau_semantique["edges"]

    node = next((node for node in nodes if get_label(reseau_semantique, node["id"]) == name), None)
    if not node:
        print("Node with label '{}' not found.".format(name))
        return [], []

    node_id = node["id"]
    legacy_edges = [edge["to"] for edge in edges if (edge["from"] == node_id and edge["label"] == "is a ")]
    legacy = []
    properties = set() # Using a set to store unique properties
    while not the_end:
        n = legacy_edges.pop(0)
        legacy.append(get_label(reseau_semantique, n))
        legacy_edges.extend([edge["to"] for edge in edges if (edge["from"] == n and edge["label"] == "is a ")])
        # Check for properties inherited from ancestors
        properties_nodes = [edge for edge in edges if (edge["from"] == n and edge["label"] != "is a ")]
        for pn in properties_nodes:
            if pn["label"] in ["vit dans", "a"]:
                properties.add(": ".join([pn["label"], get_label(reseau_semantique, pn["to"])]))
        # Check for properties belonging directly to the node itself
        own_properties = [edge for edge in edges if (edge["from"] == node_id and edge["label"] != "is a ")]
        for op in own_properties:
            if op["label"] in ["vit dans", "a"]:
                properties.add(": ".join([op["label"], get_label(reseau_semantique, op["to"])]))

        if not legacy_edges:
            the_end = True

    return legacy, list(properties) # Convert set back to list for consistent output

```

FIGURE 5.4 – Algorithme d'héritage

Pour cet Algorithme, nous allons utiliser un autre réseau sémantique exploitant mieux les propriétés de cet algorithme

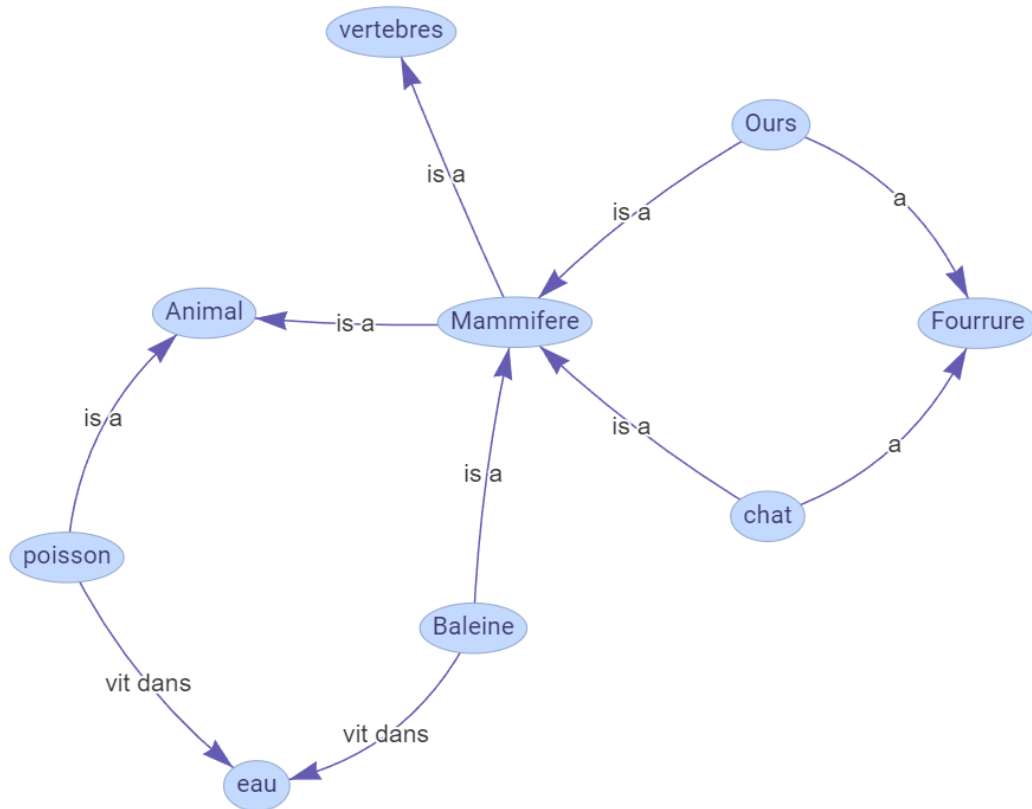


FIGURE 5.5 – Réseau sémantique utilisé pour la partie 2 du TP

En utilisant le nœud “chat” en entrée, on obtient le résultat suivant :

```

Partie 2: l'algorithme d'heritage
Resultat de l'inference utiliser:
chat
Inheritance chain for chat:
['Mammifere', 'Animal', 'vertebres']
Properties for chat:
['a: Fourrure']
  
```

FIGURE 5.6 – Résultat obtenu par l'algorithme d'héritage

5.5 Partie 3 : implémentez un algorithme qui permet d'inhiber la propagation dans le cas des liens d'exception

Cet algorithme est très similaire à celui de la partie 1 la seule différence est dans le fait qu'on ne prend pas en compte les arcs de types exception, l'algorithme est plus détaillé ci-dessous :

5.5. PARTIE 3 : IMPLÉMENTEZ UN ALGORITHME QUI PERMET D'INHIBER LA PROPAGATION DANS LE CAS DES LIENS D'EXCEPTION

```
import json

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[1]])
    return reponse

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 – Algorithme d'inhibition de la propagation dans le cas des liens d'exception

Pour cet Algorithme, nous allons utiliser le même réseau sémantique avec l'ajout d'un lien d'exception pour voir la différence avec le premier algorithme. Le réseau est détaillé dans l'image suivante :

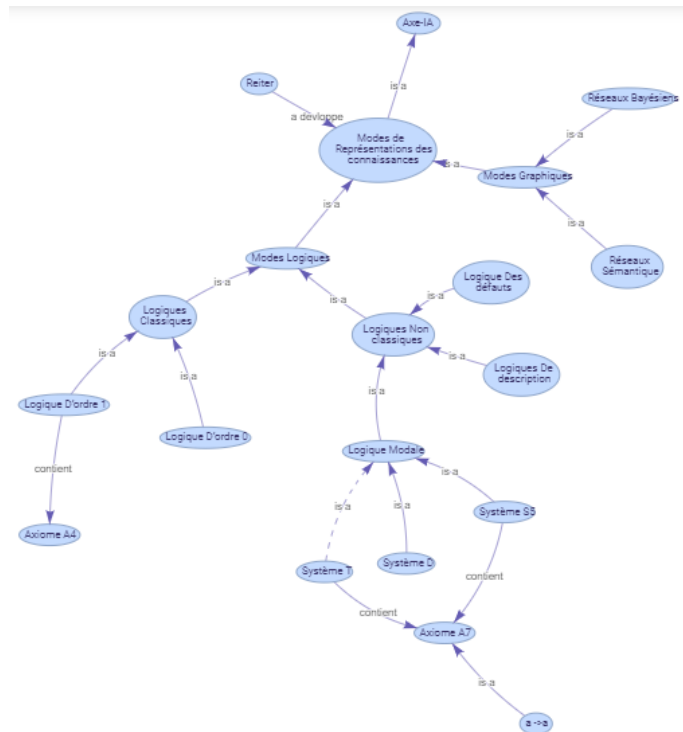


FIGURE 5.8 – Réseau sémantique utilisé pour la partie 3 du TP

5.5. PARTIE 3 : IMPLÉMENTEZ UN ALGORITHME QUI PERMET D'INHIBER LA PROPAGATION DANS LE CAS DES LIENS CHAÎNÉS

Utilisant la question “Mode de représentation des connaissances contient Axiome A7” on obtient la réponse suivante :

```
Choisissez la partie que vous voulez tester
1) Partie 1
2) Partie 2
3) Partie 3
3
Partie 3: l'algorithme de propagation de marqueurs avec exception
Modes de Représentations des connaissances contient Axiome A7
il y a un lien entre les 2 noeuds : Systeme S5
```

FIGURE 5.9 – Résultat obtenu par l'algorithme d'inhibition de la propagation dans le cas des liens d'exception

CHAPITRE 6

TP6 : LOGIQUE DE DESCRIPTION

La modélisation d'exemples de logique descriptive est une étape cruciale dans la compréhension et l'application des ontologies. Dans ce rapport, nous explorerons la modélisation d'un exemple de logique descriptive en utilisant la bibliothèque Python Owlready2. Owlready2 offre une interface conviviale pour la création, la manipulation et le raisonnement sur les ontologies basées sur le langage de description Web Ontology Language (OWL). Notre objectif est de démontrer comment Owlready2 peut être utilisé pour représenter des concepts, des relations et des individus, ainsi que pour effectuer un raisonnement sur ces ontologies.

tous d'abord, on commence par importer la bib owlready2 et on crée une ontologie :

```
from owlready2 import *  
  
onto = get_ontology("http://testxyz.org/onto.owl") #create ontology using iri
```

FIGURE 6.1 – Importation de owlready2

6.0.1 ensuite, on définit les concepts atomiques :


```
with onto: #defining our ontology

    ##Defining concepts##
    class Personne(Thing):
        pass

    class Femelle(Thing):
        pass

    class Male(Thing):
        pass

    class Entreprise(Thing):
        pass

    AllDisjoint([Male, Femelle])
```

FIGURE 6.2 – Concepts atomiques

on définit les roles atomiques :

```
##Defining roles##
class travaille(Personne >> Entreprise):
    pass

class derige(Personne >> Entreprise):
    pass

class derige_par(ObjectProperty):
    inverse_property = derige
```

FIGURE 6.3 – Roles

on définit les concepts composés :

```

    ##Defining composed entities##
class Femme(Thing):
    equivalent_to = [Personne & Femelle]

class Homme(Thing):
    equivalent_to = [Personne & Male]

class Employee(Thing):
    equivalent_to = [Femme & travaille.some(Entreprise)]

class Employe(Thing):
    equivalent_to = [Homme & travaille.some(Entreprise)]

class Directeur(Thing):
    equivalent_to = [Personne & dirige.some(Entreprise)]

```

FIGURE 6.4 – concepts composés

on procède à la définition de la ABOX :

```

##defining instances ABOX##

Rahil = Employee("Rahil")
Lydia = Employee("Lydia")
Mohamed = Directeur("Mohamed")
SONATRACH = Entreprise("SONATRACH")

```

FIGURE 6.5 – ABOX

et enfin, on invoque le raisonneur Pellet pour effectuer un raisonnement sur l'ontologie courante. Le raisonnement est une opération logique qui permet d'inférer de nouvelles informations à partir des axiomes et des faits déjà présents dans l'ontologie

```

sync_reasoner_pellet(infer_property_values=True)
onto.save(file = "tp_rc1.owl", format = "rdxml")

```

FIGURE 6.6 – inférence

Appercu sur le fichier généré

```

128 </owl:Class>
129
130 <owl:Class rdf:about="#Rahil">
131   <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
132   <owl:equivalentClass rdf:resource="http://www.w3.org/2002/07/owl#Nothing"/>
133   <owl:equivalentClass rdf:resource="#Employee"/>
134   <owl:equivalentClass rdf:resource="#Lydia"/>
135   <owl:disjointWith rdf:resource="#Lydia"/>
136 </owl:Class>
137
138 <owl:Class rdf:about="#Lydia">
139   <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
140   <owl:equivalentClass rdf:resource="http://www.w3.org/2002/07/owl#Nothing"/>
141   <owl:equivalentClass rdf:resource="#Employee"/>
142   <owl:equivalentClass rdf:resource="#Rahil"/>
143 </owl:Class>
144
145 <owl:Class rdf:about="#SONATRACH">
146   <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
147   <owl:equivalentClass rdf:resource="#Entreprise"/>
148 </owl:Class>
149
150 <rdf:Description rdf:about="http://www.w3.org/2002/07/owl#Nothing">
151   <owl:equivalentClass rdf:resource="#Employee"/>
152   <owl:equivalentClass rdf:resource="#Rahil"/>
153   <owl:equivalentClass rdf:resource="#Lydia"/>
154 </rdf:Description>
155
156

```

FIGURE 6.7 – fichier généré

Vérification des assertions

```

* Owlready2 * Pellet took 6.736842155456543 seconds
* Owlready * Reparenting onto.Femme: {owl.Thing} => {onto.Femme, onto.Personne}
* Owlready * Reparenting onto.Employee: {owl.Thing} => {onto.Femme}
* Owlready * Reparenting onto.Homme: {owl.Thing} => {onto.Male, onto.Personne}
* Owlready * Reparenting onto.Employee: {owl.Thing} => {onto.Homme}
* Owlready * Reparenting onto.Directeur: {owl.Thing} => {onto.Personne}
* Owlready * (NB: only changes on entities loaded in Python are shown, other changes are done but not listed)
Rahil est une employee: True
Lydia une employee: True

```

FIGURE 6.8 – inférence

En conclusion, ce rapport a illustré le processus de modélisation d'un exemple de logique descriptive à l'aide de la bibliothèque Python Owlready2. Nous avons présenté les étapes de création des concepts, des relations et des individus, ainsi que la définition d'ontologies composées et de règles d'inférence. En utilisant Owlready2, nous avons pu définir un exemple d'ontologie de manière intuitive et efficace, et effectuer un raisonnement pour inférer de nouvelles informations à partir des axiomes et des faits existants. Cette approche de modélisation basée sur Owlready2 offre une solution flexible et puissante pour la création et la manipulation d'ontologies dans divers domaines d'application.

Dans ce rapport, nous avons exploré et testé divers outils et bibliothèques pour représenter plusieurs exemples dans différentes logiques : la logique des prédicats, la logique des défauts, la logique modale, les réseaux sémantiques et la logique des descriptions. Les outils examinés comprennent le solveur SAT UBCSAT, ainsi que les bibliothèques Tweety et Owlready2.

Le solveur SAT UBCSAT s'est révélé efficace, offrant des solutions optimisées et performantes pour des problèmes de satisfaction de contraintes. Son utilisation a permis de démontrer la puissance des solveurs SAT dans la résolution de problèmes complexes.

La bibliothèque Tweety a été utilisée pour la logique des prédicats et la logique modale. Tweety a montré une grande flexibilité et des capacités étendues pour la modélisation et l'inférence dans ces logiques. Sa compatibilité avec divers paradigmes logiques en fait un outil polyvalent pour les chercheurs et les développeurs travaillant dans le domaine de l'intelligence artificielle et de la logique formelle.

Enfin, Owlready2 s'est avéré être un outil robuste pour la manipulation des ontologies dans la logique des descriptions. Cette bibliothèque facilite la création, la manipulation et l'interrogation d'ontologies basées sur le Web Ontology Language (OWL). Son intégration avec des bases de données comme SQL permet une gestion efficace des connaissances et une interopérabilité avec d'autres systèmes.

En conclusion, chaque outil et bibliothèque testé possède ses propres avantages et inconvénients, et le choix de l'un ou l'autre dépendra fortement des besoins spécifiques de l'application envisagée. Le solveur SAT UBCSAT excelle dans la logique des prépositions, Tweety est idéal pour les logiques des prédicats et modale, tandis qu'Owlready2 est incontournable pour la logique des descriptions. Cette diversité d'outils enrichit le champ des possibles pour les chercheurs et les praticiens, leur permettant de sélectionner les technologies les plus adaptées à leurs projets de modélisation et d'inférence logiques.