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'Electronique et d'Informatique

Département Informatique

Master Systèmes Informatiques intelligents

Module : Représentation des connaissances et Raisonnement.

-Rapport des TPs-

Réalisé par :

ARBADJI Yasmine 202031066576
HADJ MEBAREK Aymen 202035053434

Année universitaire : 2023 / 2024

Tables des matières

Introduction

Dans ce rapport, nous explorons différentes facettes de l'inférence logique et de la représentation des connaissances à travers une série de travaux pratiques. Ces travaux sont conçus pour fournir une compréhension approfondie des diverses approches logiques et de leurs applications dans le domaine de l'intelligence artificielle et de l'informatique. Nous débutons par l'inférence logique utilisant un solveur SAT, une méthode fondamentale pour vérifier la satisfiabilité des propositions logiques. Ensuite, nous abordons la logique du premier ordre, qui permet une représentation plus riche des relations entre les objets. La logique modale est explorée pour sa capacité à raisonner sur les possibilités et les nécessités. Nous étudions également la logique des défauts, qui gère les exceptions dans les systèmes de raisonnement. Les réseaux sémantiques sont examinés pour leur utilité dans la structuration des connaissances et les logiques de description pour leur rôle crucial dans la modélisation des ontologies complexes. Chaque travail pratique est accompagné de démonstrations et d'exemples concrets pour illustrer les concepts théoriques et leur application pratique.

TP 1 :
Inférence logique basée sur un solveur SAT

Introduction

Dans ce TP, nous allons explorer l'utilisation d'un solveur SAT pour l'inférence logique. Un solveur SAT permet de déterminer la satisfiabilité d'une formule logique exprimée en forme normale conjonctive (CNF). Nous allons suivre plusieurs étapes pour installer et utiliser le solveur, traduire des bases de connaissances en CNF, et tester leur satisfiabilité.

Étape 1 : Création du répertoire

D'abord on crée un répertoire nommé '**Exec**' contenant le solveur SAT UBCAST et on copie les deux fichiers test sous format CNF.

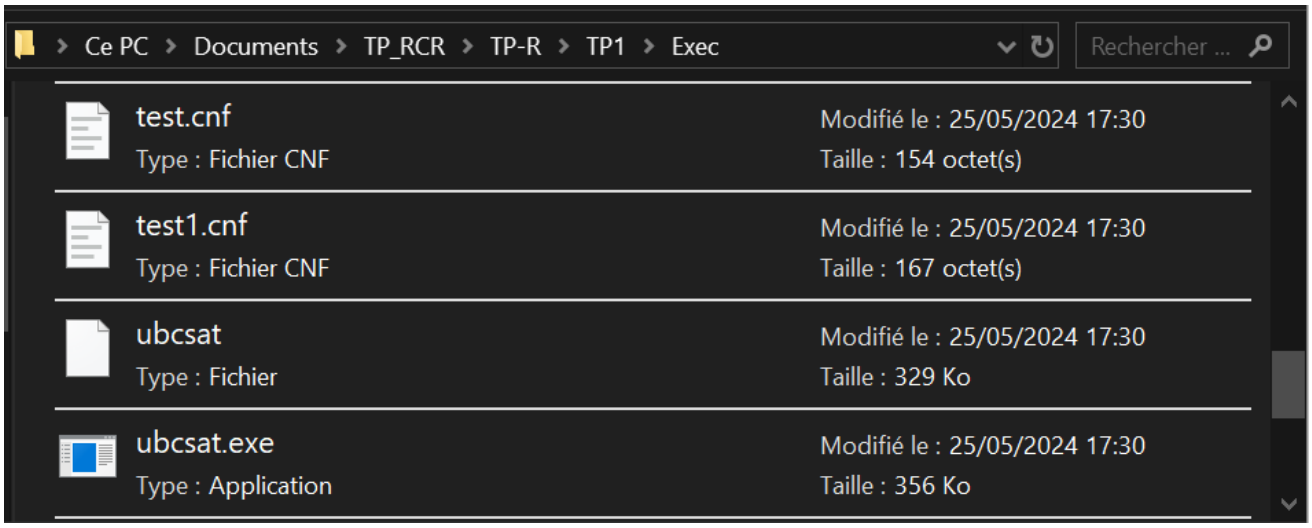

Figure 1 : Le répertoire UBCAST.

Étape 2 : Exécution du solveur SAT

En second lieu, on exécute le solveur SAT afin de tester la satisfiabilité des fichiers test.cnf et test1.cnf.

2.1. Le fichier test.cnf :

Voici le contenu du fichier test.cnf :

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

Figure 2 : Fichier test.cnf

Pour tester ce fichier, nous utilisons la commande suivante :

ubcsat -alg saps -i test.cnf -solve

Une fois la commande exécutée, nous obtenons le résultat suivant :

```
C:\Users\pc\Documents\TP_RCR\TP-R\TP1\Exec>ubcsat -alg saps -i test.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
#  -wtarget 0
#  -seed 1212855914
#  -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
#
```

Figure 3 : Résultat du solveur SAT pour le fichier test.cnf (partie 1).

Cet affichage représente des informations sur la version de l'UBCAST, un site web, des instructions d'aide et d'autre paramètres du solveur.

```
# 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          4          4
```

Figure 4 : Résultat du solveur SAT pour le fichier test.cnf (partie 2).

La deuxième partie de l'affichage montre le rapport d'exécution.

```
#
# Solution found for -target 0

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

Figure 5 : Résultat du solveur SAT pour le fichier test.cnf (partie 3).

La troisième partie de l'affichage représente la solution trouvée par le solveur indiquant que le fichier test.cnf est satisfiable. La solution trouvée est :

$$a \vee \neg b \vee \neg c \vee \neg d \vee \neg e.$$

La dernière section présente un rapport statistique de l'exécution. Parmi les paramètres étudiés, nous trouvons le nombre de variables (ici 5) et le nombre de clauses (ici 11). Le pourcentage de réussite (PercentSuccess) est de 100 %, ce qui signifie que la base de connaissances est satisfiable et donc exploitable.

```
Variables = 5
Clauses = 11
TotalLiterals = 27
TotalCPUTimeElapsed = 0.001
FlipsPerSecond = 3999
RunsExecuted = 1
SuccessfulRuns = 1
PercentSuccess = 100.00
Steps_Mean = 4
Steps_CoeffVariance = 0
Steps_Median = 4
CPUTime_Mean = 0.00100016593933
CPUTime_CoeffVariance = 0
CPUTime_Median = 0.00100016593933
```

Figure 6 : Résultat du solveur SAT pour le fichier test.cnf (partie 4).

2.2. Le fichier test1.cnf :

Voici le contenu du fichier test1.cnf :

```
1   p   cnf  5  11
2   2   -3   0
3   -3   0
4   1   -2  -3  4  0
5   -1  -4   0
6   2   -4   0
7   1    3   0
8   -1  -2   3  5  0
9   2   -5   0
10  -3   4  -5  0
11  1    2   5  0
12  3    5   0
13  -5   0
14  3    0
```

Figure 7 : Fichier test1.cnf

Pour tester ce fichier, nous utilisons la commande suivante :

```
ubcsat -alg saps -i test1.cnf -solve
```

On reprend les mêmes étapes expliquées précédemment, voici le résultat obtenu :

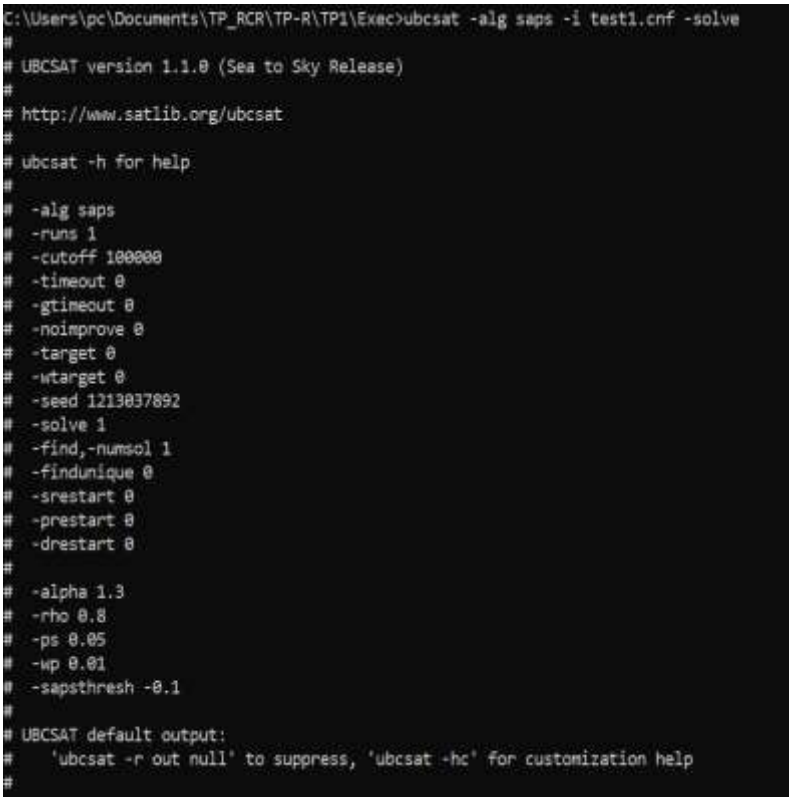

Figure 8 : Résultat du solveur SAT pour le fichier test1.cnf (partie 1).

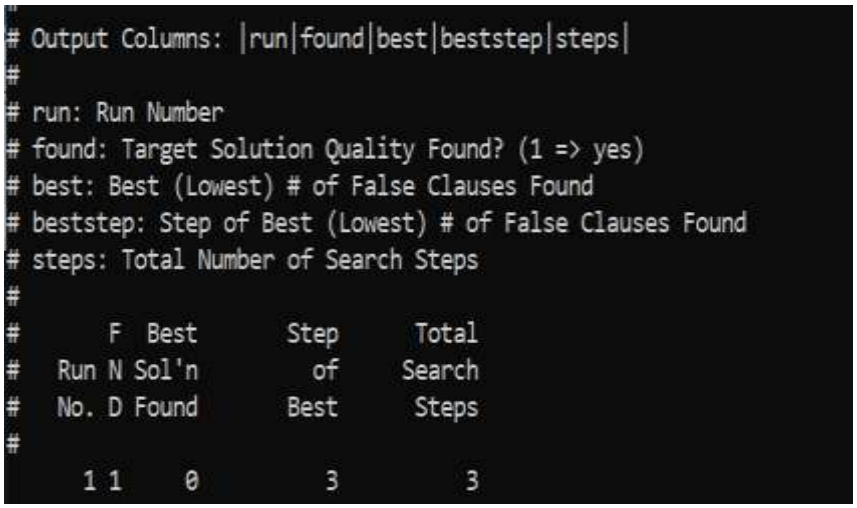

Figure 9 : Résultat du solveur SAT pour le fichier test1.cnf (partie 2).

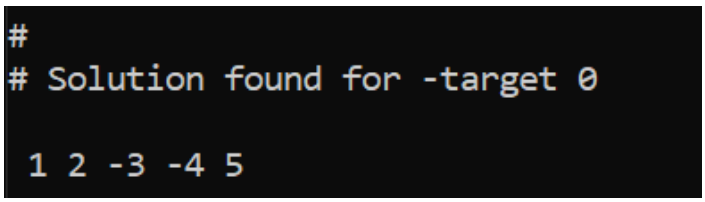

Figure 10 : Résultat du solveur SAT pour le fichier test1.cnf (partie 3).

La troisième partie de l'affichage représente la solution trouvée par le solveur indiquant que le fichier test.cnf est satisfiable. La solution trouvée est :

$$a \vee b \vee \neg c \vee \neg d \vee e.$$

Pour le fichier test1.cnf, nous avons le nombre de variables égale à 5 et le nombre de clauses égale à 11. Le pourcentage de réussite (PercentSuccess) est de 100 %, ce qui signifie que la base de connaissances est satisfiable et donc exploitable.

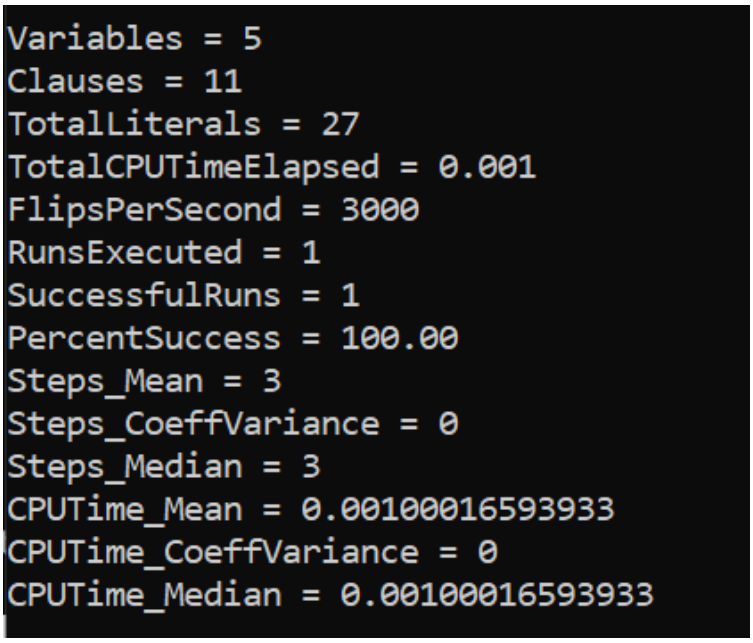

Figure 11 : Résultat du solveur SAT pour le fichier test1.cnf (partie 4).

Étape 3 : Traduction et Test de Satisfiabilité de la Base de Connaissances Zoologiques

Dans cette étape, nous allons traduire une base de connaissances sur les animaux sous forme normale conjonctive (CNF) et tester la satisfiabilité de cette base. Nous allons également télécharger des fichiers Benchmarks sous forme CNF pour tester leur satisfiabilité en utilisant un solveur SAT comme UBCSAT.

Ci-dessous l'énoncé que nous allons traduire :

- Les lions sont des mammifères.
- Les baleines sont des mammifères.
- Les serpents sont des reptiles.
- Les pingouins sont des oiseaux.
- Les lions sont carnivores.
- Les baleines sont carnivores.
- Les serpents sont carnivores ou omnivores.
- Les pingouins sont carnivores ou herbivores.

3.1. Traduction de la base de connaissance

Animaux et Caractéristiques :

- A1 : Lion
- A2 : Baleine
- A3 : Serpent
- A4 : Pingouin
- M1 : Mammifère
- M2 : Reptile

- M3 : Oiseau
- C1 : Carnivore
- C2 : Herbivore
- C3 : Omnivore

Implications :

- A1 ⊃ M1 (Lions sont des mammifères)
- A2 ⊃ M1 (Baleines sont des mammifères)
- A3 ⊃ M2 (Serpents sont des reptiles)
- A4 ⊃ M3 (Pingouins sont des oiseaux)
- A1 ⊃ C1 (Lions sont carnivores)
- A2 ⊃ C1 (Baleines sont carnivores)
- A3 ⊃ C1 ∨ C3 (Serpents sont carnivores ou omnivores)
- A4 ⊃ C1 ∨ C2 (Pingouins sont carnivores ou herbivores)

Disjonctions :

Pour traduire les implications de notre base de connaissances en disjonctions CNF, nous devons transformer chaque implication (a → b) en une disjonction (¬a ∨ b).

- ¬A1 ∨ M1 (Lions sont des mammifères)
- ¬A2 ∨ M1 (Baleines sont des mammifères)
- ¬A3 ∨ M2 (Serpents sont des reptiles)
- ¬A4 ∨ M3 (Pingouins sont des oiseaux)
- ¬A1 ∨ C1 (Lions sont carnivores)
- ¬A2 ∨ C1 (Baleines sont carnivores)
- ¬A3 ∨ C1 ∨ C3 (Serpents sont carnivores ou omnivores)
- ¬A4 ∨ C1 ∨ C2 (Pingouins sont carnivores ou herbivores)

Formules CNF :

- ¬1 ∨ 5 (¬Lion ∨ Mammifère)
- ¬2 ∨ 5 (¬Baleine ∨ Mammifère)
- ¬3 ∨ 6 (¬Serpent ∨ Reptile)
- ¬4 ∨ 7 (¬Pingouin ∨ Oiseau)
- ¬1 ∨ 8 (¬Lion ∨ Carnivore)
- ¬2 ∨ 8 (¬Baleine ∨ Carnivore)
- ¬3 ∨ 8 ∨ 10 (¬Serpent ∨ Carnivore ∨ Omnivore)
- ¬4 ∨ 8 ∨ 9 (¬Pingouin ∨ Carnivore ∨ Herbivore)

Représentation Numérique :

- 1 = A1 : Lion
- 2 = A2 : Baleine
- 3 = A3 : Serpent
- 4 = A4 : Pingouin
- 5 = M1 : Mammifère
- 6 = M2 : Reptile
- 7 = M3 : Oiseau
- 8 = C1 : Carnivore
- 9 = C2 : Herbivore
- 10 = C3 : Omnivore

Nous avons 10 variables et 9 clauses au total :

Exec > zoologique.cnf

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

Figure 12 : Fichier zoologique.cnf

Test de Satisfiabilité :

Pour tester la satisfiabilité de cette base de connaissances, nous exécutons le solveur SAT UBCSAT avec la commande d'exécution :

ubcsat -alg saps -i zoologique.cnf -solve

Résultats :

Affichage de la Version et Aide d'UBCSAT et des informations sur la version de l'UBCSAT, site web, et instructions d'aide.

```
C:\Users\pc\Documents\TP_RCR\TP-R\TP1\Exec>ubcsat -alg saps -i zoologique.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
#  -wtarget 0
#  -seed 1241070657
#  -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
#
```

Figure 13 : Résultat du solveur SAT pour le fichier zoologique.cnf (partie 1).

Rapport de l'Exécution :

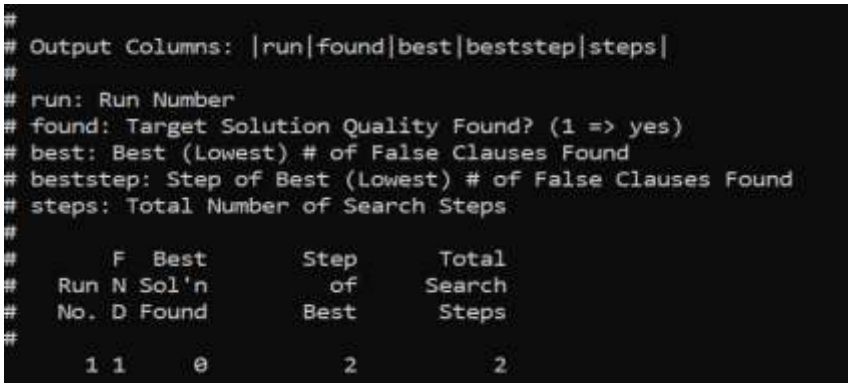

Figure 14 : Résultat du solveur SAT pour le fichier zoologique.cnf (partie 2).

Solution trouvée :

$$A1 \vee \neg A2 \vee \neg A3 \vee A4 \vee A5 \vee \neg A6 \vee A7 \vee A8 \vee \neg A9 \vee \neg A10$$

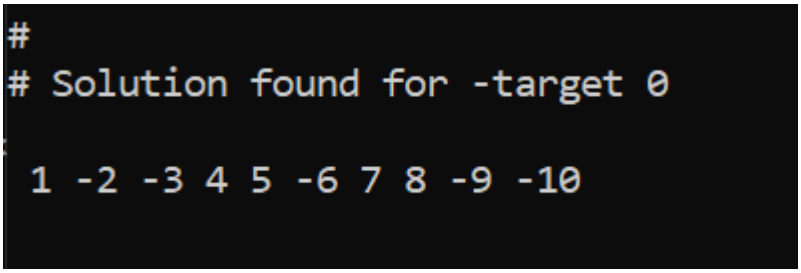

Figure 15 : Résultat du solveur SAT pour le fichier zoologique.cnf (partie 3).

Rapport statistique de l'exécution :

- Nombre de variables : 5
- Nombre de clauses : 11
- Pourcentage de réussite (PercentSuccess) : 100% (car la base de connaissances est satisfiable).

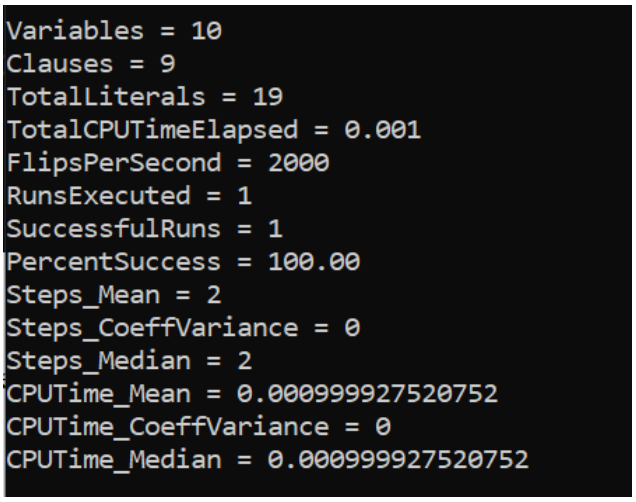

Figure 16 : Résultat du solveur SAT pour le fichier zoologique.cnf (partie 4).

En conclusion, les tests montrent que le solveur SAT peut efficacement déterminer la satisfiabilité des formules CNF issues de notre base de connaissances zoologiques, confirmant ainsi la cohérence et l'exploitabilité de cette base.

3.2. Fichier Benchmarks

Nous avons également téléchargé des fichiers Benchmarks sous forme CNF pour tester leur satisfiabilité en utilisant le même solveur SAT (UBCSAT).

Un fichier satisfiable :

```
C:\Users\pc\Documents\TP_RCR\TP-R\TP1\Exec>ubcsat -alg saps -i uf100-0993.cnf -solve
#
```

```
#
# 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 = 430
TotalLiterals = 1290
TotalCPUTimeElapsed = 0.002
FlipsPerSecond = 704051
RunsExecuted = 1
SuccessfulRuns = 1
PercentSuccess = 100.00
Steps_Mean = 1408
Steps_CoeffVariance = 0
Steps_Median = 1408
CPUTime_Mean = 0.0019998550415
CPUTime_CoeffVariance = 0
CPUTime_Median = 0.0019998550415
```

Figure 17 : Commande d'exécution et résultat du solveur SAT pour le fichier uf100-0993.cnf

La base est satisfiable.

Un fichier non-satisfiable :

```
C:\Users\pc\Documents\TP_RCR\TP-R\TP1\Exec>ubcsat -alg saps -i uuf100-0993.cnf -solve
#
```

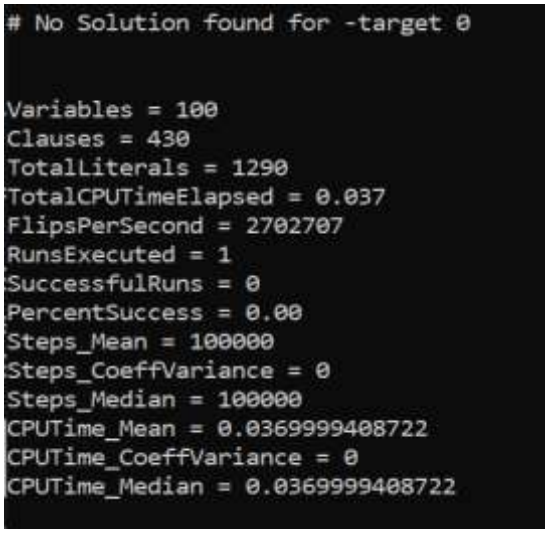

Figure 18 : Commande d'exécution et résultat du solveur SAT pour le fichier uuf100-0993.cnf

La base est non-satisfiable.

Étape 4 : Simulation de l'interface d'une base de connaissances

En utilisant le raisonnement par l'absurde, nous allons vérifier si notre base de connaissances (BC) zoologique permet de déduire un but donné en utilisant le solveur UBSAT pour tester la satisfiabilité de la base.

4.1. Déroulement

1. **Sélection de la base de connaissances :**
- Nous sélectionnons le fichier contenant la BC.
2. **Préparation du fichier temporaire :**
- Ouvrir un fichier temporaire pour y copier la BC.
- Ajouter les informations de la BC (nombre de variables et nombre de clauses + 1).
3. **Modification de la base de connaissances :**
- Incrémenter le nombre de clauses pour inclure la négation du but dans le fichier à traiter (en ajoutant "0" à la fin, comme pour toutes les clauses).
4. **Exécution du solveur SAT :**
- Exécuter le solveur SAT.
5. **Affichez le résultat.**

En procédant ainsi, nous pouvons déterminer si la base de connaissances zoologique permet d'inférer le but donné.

4.2. Teste

```
C:\Users\pc\Documents\TP_RCR\TP-R\TP1\Exec>gcc main.c -o exec

C:\Users\pc\Documents\TP_RCR\TP-R\TP1\Exec>exec
Entrez le nom de votre Fichier BC (sans extension) :
zoologique
Liste des variables de la BC (litteraux) :
1 : A1 (Lion)           2 : A2 (Baleine)              3 : A3 (Serpent)
4 : A4 (Pingouin)       5 : M1 (Mammifere)      6 : M2 (Reptile)
7 : M3 (Oiseau) 8 : C1 (Carnivore)      9 : C2 (Herbivore)
10 : C3 (Omnivore)

Entrez le nombre de litteraux :
2
Entrez le litteral 1 :
5
Entrez le litteral 2 :
8
================================================Warning: Ingoring comment line mid instance:
   c -5 -8 0

BC U {Non but} est non satisfiable. La base infere le but.
-5 -8 ne peuvent pas etre atteints
C:\Users\pc\Documents\TP_RCR\TP-R\TP1\Exec>
```

Figure 19 : Résultat du programme, Cas de succès.

```
C:\Users\pc\Documents\TP_RCR\TP-R\TP1\Exec>gcc main.c -o exec

C:\Users\pc\Documents\TP_RCR\TP-R\TP1\Exec>exec
Entrez le nom de votre Fichier BC (sans extension) :
zoologique
Liste des variables de la BC (litteraux) :
1 : A1 (Lion)           2 : A2 (Baleine)              3 : A3 (Serpent)
4 : A4 (Pingouin)       5 : M1 (Mammifere)      6 : M2 (Reptile)
7 : M3 (Oiseau) 8 : C1 (Carnivore)      9 : C2 (Herbivore)
10 : C3 (Omnivore)

Entrez le nombre de litteraux :
2
Entrez le litteral 1 :
-5
Entrez le litteral 2 :
-8
================================================Warning: Ingoring comment line mid instance:
   c -5 -8 0

BC U {Non but} est satisfiable. La base n'infere pas le but.
Solution :
1 2 -3 -4 5 6 -7 8 9 10
```

Figure 20 : Résultat du programme, Cas d'échec.

4.3. Code Source

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (){
    FILE * fich_base = NULL;
    FILE * fich_temp = NULL;
    int nbr_propositions, nbr_variables, nbr_clauses, i, but[20], non_but[20];
```

```
    char nom_base[20], c;

    printf("Entrez le nom de votre Fichier BC (sans extension) :\n");
    gets(nom_base);
    strcat(nom_base, ".cnf");
    fich_base = fopen(nom_base, "r+");

    if (fich_base == NULL) {
        printf("Impossible d'ouvrir BC...\n");
    } else {
        fich_temp = fopen("Temp.cnf", "w+");
        if (fich_temp == NULL) {
            printf("Impossible de transferer BC\n");
        } else {
            fscanf(fich_base, "p cnf %d %d ", &nbr_variables, &nbr_clauses);
            nbr_clauses += 1; // pour rajouter le non_but
            fprintf(fich_temp, "p cnf %d %d\n", nbr_variables, nbr_clauses); // on
met les infos dans le fichier qu'on va traiter
        }
        c = fgetc(fich_base);
        while (c != EOF) {
            if (c != EOF) fputc(c, fich_temp);
            c = fgetc(fich_base);
        } // on a recopie le reste de la BC dans le fichier qu'on va traiter

        printf("Liste des variables de la BC (litteraux) :\n");
        printf("1 : A1 (Lion)\t\t2 : A2 (Baleine)\t\t3 : A3 (Serpent)\t\n4 : A4
(Pingouin)\t5 : M1 (Mammifere)\t6 : M2 (Reptile)\t\n7 : M3 (Oiseau)\t8 : C1
(Carnivore)\t9 : C2 (Herbivore)\t\n10 : C3 (Omnivore)\t\n\n");
        printf("Entrez le nombre de litteraux : \n");
        scanf("%d", &nbr_propositions);

        for (i = 1; i < nbr_propositions + 1; i++) {
            printf("Entrez le litteral %d : \n", i);
            scanf("%d", &but[i]);
            if (but[i] > -11 && but[i] < 11)
                non_but[i] = but[i] * (-1); // si les codes sont corrects on prend la
negation du but (absurde)
            else
                puts("Erreur, Vous avez entre un code invalide");
        }

        fprintf(fich_temp, "\n"); // completion du fichier traite
        for (i = 1; i < nbr_propositions + 1; i++) fprintf(fich_temp, "%d ",
non_but[i]); // Ajout des negations au fichier
        fprintf(fich_temp, "0"); // Ajout des 0 pour marquer la fin
        printf("=============================================");

        fclose(fich_temp);

        system("ubcsat -alg saps -i Temp.cnf -solve > results.txt"); // execution du
solveur
```

```
    }


    int termine = 0; // signaler fin affichage
    FILE *fich = fopen("results.txt", "r+");
    if (fich == NULL) {
        printf("Impossible d'acceder aux resultats...\n");
    } else {
        char texte[1000];
        while (fgets(texte, 1000, fich) && !termine) {
            if (strstr(texte, "# Solution found for -target 0")) {
                printf("\nBC U {Non but} est satisfiable. La base n'infere pas le
but. \nSolution : \n");
                fscanf(fich, "\n");
                while (!strstr(fgets(texte, 1000, fich), "Variables"))
                    printf("%s", texte);
                termine = 1;
            }
        }
        if (termine == 0) { // cas de non satisfiabilite
            printf("\nBC U {Non but} est non satisfiable. La base infere le but.\n");
            int j;
            for (j = 1; j < nbr_propositions + 1; j++) {
                printf("%d ", (-1) * but[j]);
            }
            if (j > 2)
                printf("ne peuvent pas etre atteints ");
            else
                printf("ne peut pas etre atteint \n");
        }
    }

    fclose(fich);

    return 0;
}
```

Conclusion

Dans ce TP, nous avons appris à utiliser un solveur SAT pour vérifier la satisfiabilité de différentes bases de connaissances exprimées en CNF. Nous avons également traduit des connaissances en forme CNF et utilisé un algorithme de raisonnement par l'absurde pour simuler l'inférence logique. Le solveur SAT s'est révélé être un outil puissant pour résoudre des problèmes logiques complexes et tester la cohérence de bases de connaissances.

TP 2 :
La logique du premier ordre

1. Introduction

Dans ce rapport, nous allons explorer l'utilisation de la logique des prédicats pour l'inférence logique en utilisant un solveur de la bibliothèque TweetyProject en Java. Nous expliquerons en détail l'implémentation du code, les résultats obtenus, et leur analyse à travers une structure méthodique.

2. Définition de la logique

La logique des prédicats (FOL, First-Order Logic) est une formalisation mathématique permettant de représenter des relations et des propriétés d'objets au sein d'un domaine donné. Elle est utilisée pour décrire des assertions sur les objets et pour effectuer des inférences à partir de ces assertions.

2.1. Exemple

Un exemple classique de logique des prédicats implique des objets (tel que ‘les animaux’) et des relations (comme ‘vole’ ou ‘connaît’). Nous pourrions avoir des règles qui décrivent si certains animaux volent ou se connaissent, et nous voulons vérifier si certaines affirmations peuvent être prouvées ou réfutées à partir de ces règles.

3. Implémentation

L'implémentation est réalisée en Java à l'aide de la bibliothèque TweetyProject, qui fournit des outils pour manipuler les formules logiques et effectuer des inférences.

3.1. Structure d’instance

La structure d'instance comprend :

- **Signature :** Définit les types (sorts), constantes et prédicats utilisés dans les formules.
- **Formules :** Représentent les assertions logiques sous forme de formules de logique des prédicats.
- **Raisonneur :** Composant chargé d'effectuer les inférences logiques à partir des formules données.

3.2. Explication du code

Le code est structuré comme suit :

1. **Création de la signature :**

Une signature est créée pour inclure des types comme "Animal", des constantes représentant des animaux spécifiques (par exemple, perroquet, pingouin), et des prédicats définissant les relations (par exemple, "vole").

```
//Create new FOLSignature with equality
FolSignature sig = new FolSignature(true);

//Add sort
Sort sortAnimal = new Sort("Animal");
sig.add(sortAnimal);

//Add constants
Constant constantperroquet = new Constant("perroquet",sortAnimal);
```

```
        Constant constantpenguin = new Constant("penguin",sortAnimal);
        sig.add(constantperroquet, constantpenguin);

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

2. **Parseur de formules :**

Utilisation de FolParser pour analyser des chaînes de caractères et créer des formules logiques. Ces formules sont ensuite ajoutées à une base de croyances.

```
        FolParser parser = new FolParser();
        parser.setSignature(sig); //Use the signature defined above
        FolBeliefSet bs = new FolBeliefSet();
        FolFormula f1 = (FolFormula)parser.parseFormula("!Flies(penguin)");
        FolFormula f2 = (FolFormula)parser.parseFormula("Flies(perroquet)");
        FolFormula f3 = (FolFormula)parser.parseFormula("!Knows(perroquet,penguin)");
        FolFormula f4 = (FolFormula)parser.parseFormula("/==(perroquet,penguin)");
        FolFormula f5 = (FolFormula)parser.parseFormula("penguin == penguin");
        bs.add(f1, f2, f3, f4, f5);
        System.out.println("\nParsed BeliefBase: " + bs);
```

3. **Raisonnement logique :**

Utilisation d'un raisonner pour exécuter des requêtes sur la base de croyances et vérifier si certaines formules peuvent être dérivées.

```
        FolReasoner.setDefaultReasoner(new SimpleFolReasoner()); //Set default
prover, options are NaiveProver, EProver, Prover9
        FolReasoner prover = FolReasoner.getDefaultReasoner();
        System.out.println("ANSWER 1: " + prover.query(bs,
(FolFormula)parser.parseFormula("Flies(penguin)")));
        System.out.println("ANSWER 2: " + prover.query(bs,
(FolFormula)parser.parseFormula("forall X: (exists Y: (Flies(X) && Flies(Y) &&
X/==Y))")));
        System.out.println("ANSWER 3: " + prover.query(bs,
(FolFormula)parser.parseFormula("penguin == penguin")));
        System.out.println("ANSWER 4: " + prover.query(bs,
(FolFormula)parser.parseFormula("penguin /== penguin")));
```

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

4. **Parseur de fichier de base de croyances :**

Analyse d'une base de croyances à partir d'un fichier externe, permettant de charger des ensembles plus complexes de règles et de connaissances.

```
        parser = new FolParser();
        parser.setSignature(new FolSignature(true));
        bs =
parser.parseBeliefBaseFromFile("src/resources/examplebeliefbase2.fologic");
        System.out.println("\nParsed BeliefBase: " + bs);
        FolFormula query = (FolFormula)parser.parseFormula("exists
X:(teaches(alice,X))");
        System.out.println("Query: " + query + "\nANSWER 1: " +
prover.query(bs,query));
        query = (FolFormula)parser.parseFormula("exists X:(exists Y:(hasID(alice,X)
&& hasID(alice,Y) && X/==Y))");
        System.out.println("Query: " + query + "\nANSWER 2: " +
prover.query(bs,query));
```

3.3. Résultat de l'exécution

L'exécution du programme permet de déterminer la satisfiabilité de différentes formules logiques. Par exemple, le raisonner peut vérifier si un pingouin vole ou si deux animaux spécifiques se connaissent, en se basant sur les règles définies.

```
PS C:\Users\pc\Documents\TP_RCR\TP-R\TP2\Exec>  & 'C:\Program Files\Java\jdk-17\bin\java.exe' '@C:\Users\pc\AppData\Local\Temp\cp_18f47kvh
bgzyouo99pd7g6p6n.argfile' 'FOL.FolExample'
Signature: [_Any = {}, Animal = {perroquet, penguin}], [Knows(Animal,Animal), ==(_Any,_Any), /==(_Any,_Any), Flies(Animal)], []

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

Parsed BeliefBase: { forall X: (forall Y: ((hasID(X,Y)=>!exists Z: (hasID(X,Z)&&(Z/==Y))))), forall X: (forall Y: ((hasID(X,Y)=>!exists Z:
 (hasID(Z,Y)&&(Z/==X))))), teaches(alice,cryptography), hasID(alice,id001), !hasID(alice,id002), forall X: (forall Y: ((teaches(X,Y)=>exists Z: (hasID(X,Z))))) }
Query: exists X: (teaches(alice,X))
ANSWER 1: true
Query: exists X: (exists Y: (hasID(alice,X)&&hasID(alice,Y)&&(X/==Y)))
ANSWER 2: false
PS C:\Users\pc\Documents\TP_RCR\TP-R\TP2\Exec>
```

4. Analyse des résultats

Les résultats obtenus montrent la capacité du solveur SAT à traiter des bases de connaissances complexes et à effectuer des inférences logiques. Voici quelques observations clés :

- Le solveur a pu déterminer que certaines assertions étaient satisfiables ou non.
- La représentation des formules en logique des prédicats permet de modéliser des relations complexes entre les objets.

- L'utilisation de fichiers externes pour définir des bases de croyances facilite la gestion et l'extension des connaissances.

5. Conclusion

Ce TP démontre l'efficacité du solveur SAT pour l'inférence logique en logique des prédicats. La bibliothèque TweetyProject offre des outils robustes pour manipuler les formules logiques, effectuer des inférences, et analyser les résultats. Cette approche est cruciale pour des applications avancées en intelligence artificielle et en représentation des connaissances, permettant de gérer et de raisonner sur des ensembles de données complexes de manière formelle et systématique.

TP 3 :
La logique modale

1. Introduction

Dans ce présent rapport, nous détaillons l'utilisation d'un solveur modal pour effectuer des inférences logiques en utilisant la bibliothèque TweetyProject en Java. Nous expliquerons la mise en œuvre du code, les résultats obtenus et leur analyse en suivant une structure bien définie.

2. Définition de la logique modale

La logique modale est une extension de la logique classique qui permet de raisonner sur des propositions avec des modalités comme la possibilité et la nécessité. Elle est particulièrement utile pour modéliser des systèmes où certaines conditions peuvent être éventuellement vraies ou nécessairement vraies.

2.1. Exemple

Un exemple simple de logique modale pourrait impliquer des propositions comme "Il est possible que A soit vrai" (noté ◇A) ou "Il est nécessaire que B soit vrai" (noté □B). Ces opérateurs modaux permettent de capturer des notions de possibilité et de nécessité.

3. Implémentation

L'implémentation est réalisée en Java et la bibliothèque TweetyProject pour manipuler des formules modales et effectuer des inférences.

3.1. Structure d'instance

La structure d'instance comprend :

- **Ensemble de croyances modales :** Contient les formules modales représentant les connaissances.
- **Signature :** Définit les prédicats utilisés dans les formules.
- **Parseur :** Permet d'analyser les formules modales à partir de chaînes de caractères.
- **Raisonneur :** Composant chargé de vérifier les inférences logiques à partir des formules données.

3.2. Explication du code

1. **Création de l'ensemble de croyances modales :** Un MlBeliefSet est créé pour contenir les formules modales.

```
// Create an empty modal belief set
MlBeliefSet beliefSet = new MlBeliefSet();
```

2. **Définition de la signature :** Une signature est définie avec deux prédicats : a et b.

```
// Create a parser and define a signature
MlParser parser = new MlParser();
FolSignature signature = new FolSignature();
signature.add(new Predicate("a", 0));
signature.add(new Predicate("b", 0));
parser.setSignature(signature);
```

3. **Ajout de formules à l'ensemble de croyances :**

Plusieurs formules sont ajoutées à l'ensemble de croyances :

- ◇(a ∨ b) : Il est possible que a ou b soit vrai.
- □(b ∨ a) : Il est nécessaire que b ou a soit vrai.
- □(b ∧ a) : Il est nécessaire que b et a soient vrais.

```
        // Add formulas to the belief set
        beliefSet.add((RelationalFormula) parser.parseFormula("<>(a || b)")); //
Diamond operator formula: It is possibly true that a or b.
        beliefSet.add((RelationalFormula) parser.parseFormula("[](b || a)")); // It
is necessarily true that b or a.
        beliefSet.add((RelationalFormula) parser.parseFormula("[](b && a)")); // It
is necessarily true that b and a.
```

4. **Affichage de la base de connaissances modale :** La base de connaissances est affichée pour vérification.

```
        // Print the modal knowledge base
        System.out.println("Modal knowledge base: " + beliefSet);
```

5. **Création d'un raisonneur et exécution des requêtes :** Un SimpleMlReasoner est utilisé pour effectuer des inférences sur la base de croyances.

Plusieurs requêtes sont exécutées pour vérifier certaines conditions :

- □(¬a) : Il est nécessaire que a soit faux.
- ◇(a ∧ b) : Il est possible que a et b soient vrais.
- □(b ∧ ◇(¬b)) : Il est nécessaire que b soit vrai et possible que b soit faux.
- ◇(¬(b ∧ a)) : Il est possible que b et a ne soient pas tous les deux vrais.

```
        // Create a reasoner and perform queries
        SimpleMlReasoner reasoner = new SimpleMlReasoner();
        // Query 1:
        System.out.println("[](!a)      " + reasoner.query(beliefSet, (FolFormula)
parser.parseFormula("[](!a)")));

        // Query 2:
        System.out.println("<>(a && b)      " + reasoner.query(beliefSet,
(FolFormula) parser.parseFormula("<>(a && b)")));

        // Query 3:
        System.out.println("[](b && <>(!(b)))      " + reasoner.query(beliefSet,
(FolFormula) parser.parseFormula("[](b && <>(!(b)))")));

        // Query 4:
        System.out.println("<>(!(b && a))      " + reasoner.query(beliefSet,
(FolFormula) parser.parseFormula("<>(!(b && a))")));
```

3.3. Résultat de l’exécution

Les résultats des requêtes sont affichés, indiquant si les conditions spécifiées sont satisfiables dans la base de croyances modale.

```
PS C:\Users\pc\Documents\TP_RCR\TP-R\TP3\Exec>  & 'C:\Program Files\Java\jdk-17\bin\java.exe' '@C:\Users\pc\AppData\Local
\Temp\cp_960mg9rvnqlu4kadd6t3vlqk.argfile' 'App'
Modal knowledge base: { <>(a||b), [](b||a), [](b&&a) }
[](!a)      false
<>(a && b)      true
[](b && <>(!(b)))      false
<>(!(b && a))      false
PS C:\Users\pc\Documents\TP_RCR\TP-R\TP3\Exec>
```

4. Analyse des résultats

Les résultats montrent que le solveur modal est capable de vérifier efficacement des formules modales complexes :

- **Satisfiabilité des formules modales :** Le solveur détermine si les formules modales sont satisfiables dans le contexte des croyances définies.
- **Capacité à raisonner sur les possibilités et nécessités :** La logique modale permet de capturer des notions de possibilité et de nécessité, cruciales pour modéliser des systèmes dynamiques et incertains.
- **Gestion des connaissances modales :** L'utilisation d'un parseur et d'un raisonneur simplifie la manipulation et l'inférence de connaissances modales.

Les observations montrent que la logique modale et les solveurs associés sont des outils puissants pour l'intelligence artificielle, permettant de gérer et de raisonner sur des systèmes complexes où la possibilité et la nécessité jouent un rôle central.

5. Conclusion

Ce TP démontre l'efficacité du solveur modal pour l'inférence logique en utilisant la logique modale. La bibliothèque TweetyProject offre des outils robustes pour manipuler les formules modales, effectuer des inférences, et analyser les résultats. Cette approche est essentielle pour des applications avancées en intelligence artificielle et en représentation des connaissances, permettant de modéliser et de raisonner sur des systèmes où les notions de possibilité et de nécessité sont fondamentales.

TP 4 :
La logique des défauts

1. Introduction

Dans ce rapport, nous allons explorer l'utilisation du solveur SAT pour l'inférence logique à l'aide du code Java de la bibliothèque TweetyProject. Nous allons suivre une structure bien définie pour expliquer le processus d'implémentation, les résultats obtenus et leur analyse.

2. Définition de la logique

Les logiques de défauts (Default Logics) sont une famille de logiques non monotones conçues pour formaliser le raisonnement en présence d'informations incomplètes ou incertaines. Elles permettent de tirer des conclusions provisoires, appelées défauts, en l'absence d'informations contraires. Ces logiques sont particulièrement utiles dans des situations où les connaissances disponibles ne sont pas suffisantes pour une déduction certaine mais où des hypothèses raisonnables doivent être faites.

2.1. Exemple

Considérons un exemple simple pour illustrer les logiques de défauts :

- Règles de Défaut

$$\frac{Oiseau(X) : \neg NonVolant(X)}{\text{Voler(X)}}$$

  Si X est un oiseau et qu'on ne peut pas prouver que X ne peut pas voler, alors on conclut que X peut voler.
- Faits Initiaux

  Oiseau(Tweety)
- Raisonnement

En l'absence d'informations indiquant que Tweety ne peut pas voler, on utilise la règle de défaut pour conclure que Tweety peut voler.

3. Implémentation

L'implémentation illustre l'utilisation de la bibliothèque Tweety pour travailler avec la logique de défauts.

3.1. Structure de code

Les principales fonctionnalités mises en œuvres sont :

- **Parsing des Théories de Défaut :** Lecture et analyse de fichiers contenant des théories de défauts.
- **Manipulation des Règles de Défaut :** Création, comparaison et vérification des règles de défaut.
- **Séquences de Défauts :** Application et gestion des séquences de règles de défauts.
- **Raisonnement :** Utilisation du raisonneur par défaut pour déduire des conclusions à partir des théories de défauts.

3.2. Explication du code

1. **Parsing des théories des défauts :**

La classe RdlParser est utilisée pour analyser les théories de défauts à partir de fichiers. Les fichiers sont lus et convertis en objets DefaultTheory.

```
RdlParser parser = new RdlParser();
DefaultTheory t = parser.parseBeliefBaseFromFile(RdlExample.class.getResource("
/ressources/simple_default_theory.txt").getFile());
```

2. **Manipulation des règles de défauts:**

Les règles de défaut sont créées, comparées et vérifiées. La méthode createTestSet génère un ensemble de règles de défaut à partir de chaînes de caractères.

```
static List<DefaultRule> createTestSet(RdlParser parser, String... list) throws
Exception{
        List<DefaultRule> result = new LinkedList<>();
        for(String str:list)
            result.add((DefaultRule)(DefaultRule)parser.parseFormula(str));
        return result;
    }
```

3. **Séquence de défauts:**

Les séquences de défauts sont gérées par la classe DefaultSequence, qui permet d'appliquer des règles de défaut successives et de vérifier si une séquence est fermée.

```
      DefaultSequence s = new DefaultSequence(t);
        System.out.println(s);
        s = s.app((DefaultRule)parser.parseFormula("a::b/b"));
        System.out.println(s);
```

4. **Raisonnement :**

Le raisonneur par défaut (SimpleDefaultReasoner) est utilisé pour interroger les théories de défaut et obtenir des modèles.

```
SimpleDefaultReasoner reasoner = new SimpleDefaultReasoner();
        System.out.println(reasoner.getModels(t));
        System.out.println(reasoner.query(t,(FolFormula) parser.parseFormula("!a")));
        System.out.println(reasoner.query(t,(FolFormula) parser.parseFormula("b")));
        System.out.println(reasoner.query(t,(FolFormula) parser.parseFormula("c")));
```

3.3. Résultat de l’exécution

Les méthodes de test (defaultRuleTest, processTreeTest, extensionTest, etc.) illustrent diverses opérations sur les règles et théories de défauts, telles que la comparaison, la vérification de la normalité et la recherche de variables non liées.

- **defaultRuleTest :** Teste l'égalité des règles de défaut et leur normalité.
- **processTreeTest :** Utilise le raisonneur pour interroger une théorie de défaut et obtenir des modèles.

4. Analyse des résultats

Les résultats obtenus montrent diverses propriétés des règles de défaut, notamment leur égalité, leur normalité, et les variables non liées. Voici les idées générales :

Comparaison des Règles de Défaut :

- Les règles identiques sont correctement reconnues comme égales, tandis que même de petites différences les rendent inégales, montrant la précision du système.

Normalité des Règles de Défaut :

- Les règles normales respectent les conditions de la logique de défaut standard, tandis que les règles non normales présentent des justifications qui ne sont pas directement liées à leurs conséquences.

Variables Non Liées :

- Les variables non liées sont correctement identifiées, ce qui est crucial pour éviter les ambiguïtés logiques et assurer la validité des règles.

Ces observations montrent que le système gère efficacement les règles de défaut, garantissant une représentation cohérente et rigoureuse des connaissances.

5. Conclusion

En conclusion, le code démontre que la bibliothèque Tweety, avec son solveur SAT, est efficace pour le raisonnement logique dans des bases de connaissances complexes, incluant la gestion des règles de défaut. Sa capacité à déterminer la satisfiabilité des formules, modéliser des relations complexes, et intégrer des règles de défaut en fait un outil précieux pour l'intelligence artificielle. Les résultats montrent la robustesse et l'efficacité de ces outils pour le traitement logique et la gestion formelle de l'incertitude.

TP 5 :
Les réseaux sémantiques

Partie I :

1. Introduction

La propagation de marqueurs dans un réseau sémantique constitue un processus fondamental en intelligence artificielle et en traitement automatique du langage naturel. Ce concept repose sur la capacité à déterminer l'existence de relations entre différents éléments du réseau en suivant des chemins spécifiques définis par des règles sémantiques.

2. Définition

La propagation de marqueurs dans un réseau sémantique est un processus par lequel des informations, représentées par des marqueurs ou des étiquettes, se propagent le long des arêtes ou des chemins du réseau. L'objectif est de découvrir ou d'inférer des relations entre différents nœuds du réseau en suivant des règles prédéfinies. Ce processus peut être utilisé pour résoudre des problèmes d'inférence, d'analyse de texte, de recherche d'informations, etc. La propagation de marqueurs est souvent utilisée dans le domaine de l'intelligence artificielle pour modéliser le raisonnement et la compréhension des données.

2.1. Exemple

Considérons un exemple simple

- **Instance :** Considérons une instance "lion" dans notre réseau sémantique.
- **Objectif :** Découvrir les proies potentielles du lion en utilisant la propagation de marqueurs.
- **Propagation :** Nous suivons les arêtes sortantes du nœud "lion" pour identifier les animaux qui sont des proies potentielles.
- **Résultat :** Nous pourrions découvrir que les gazelles, les zèbres et les buffles sont des proies potentielles du lion en examinant les relations de prédation sortantes du nœud "lion".

3. Implémentation

3.1. Structure de code

- Initialisation :
  - Chargement du réseau sémantique représenté par un objet JSON.
  - Définition des nœuds et des arêtes du réseau.

```
FileReader reader = new FileReader("Bases/propagation.json");
JSONObject reseauSemantique = (JSONObject) new JSONParser().parse(reader);

JSONArray nodes = (JSONArray) reseauSemantique.get("nodes");
JSONArray edges = (JSONArray) reseauSemantique.get("edges");
```

- Méthode getLabel :
  - Parcourt les arêtes du réseau pour trouver la relation entre deux nœuds.
  - Renvoie l'étiquette de la relation entre les nœuds.

```
public static String getLabel(JSONObject reseauSemantique, JSONObject node, String
relation) {
        List<String> nodeRelationEdgesLabel = new ArrayList<>();
        JSONArray edges = (JSONArray) reseauSemantique.get("edges");
        JSONArray nodes = (JSONArray) reseauSemantique.get("nodes");
```

```
        for (Object o : edges) {
            JSONObject edge = (JSONObject) o;
            if (edge.get("to").equals(node.get("id")) &&
edge.get("label").equals(relation)) {
                for (Object n : nodes) {
                    JSONObject no = (JSONObject) n;
                    if (no.get("id").equals(edge.get("from"))) {
                        nodeRelationEdgesLabel.add((String) no.get("label"));
                    }
                }
            }
        }
        return "il y a un lien entre les 2 noeuds : " + String.join(", ",
nodeRelationEdgesLabel);
    }
```

- Méthode propagationDeMarqueurs
  - Parcourt les nœuds du réseau.
  - Propage les marqueurs d'un nœud source vers un nœud cible en suivant les arêtes "is a".
  - Vérifie si une relation spécifique est établie entre les nœuds.

Renvoie les résultats de la propagation.

3.2. Explication du code

La fonction de ce code est de réaliser la propagation de marqueurs dans un réseau sémantique représenté sous forme d'objet JSON. Plus précisément, les fonctionnalités sont les suivantes

- **Récupération de l'étiquette d'un nœud :** La méthode getLabel permet de récupérer l'étiquette d'un nœud donné en fonction de sa relation avec un autre nœud dans le réseau sémantique.

- **Propagation de marqueurs entre nœuds :** La méthode propagationDeMarqueurs réalise la propagation de marqueurs entre les nœuds du réseau en suivant les arêtes "is a". Elle vérifie également si une relation spécifique est établie entre les nœuds, et renvoie les résultats de la propagation sous forme d'une liste.

3.3. Résultat de l’exécution

Le résultat de ce code est une liste de chaînes de caractères, où chaque élément de la liste représente le résultat de la propagation de marqueurs entre deux nœuds du réseau sémantique. Les résultats peuvent être des étiquettes de relation entre les nœuds ou des messages indiquant l'absence de lien entre les nœuds ou l'incapacité à fournir une réponse en raison de connaissances manquantes.

Dans l'exemple fourni, le résultat attendu pourrait ressembler à ceci :

- Pour la première paire de nœuds (M1_node[0], M2_node[0]) :
  - "Modes de Representations des connaissances" contient "Axiome A7"
  - Résultat : "il y a un lien entre les 2 noeuds : Réseau Sémantique, Mode de Représentation, Proposition, Langage Naturel"

- Pour la deuxième paire de nœuds (M1_node[1], M2_node[1]) :
  - "Modes de Representations des connaissances" contient "Axiome A4"
  - Résultat : "il n'y a pas un lien entre les 2 noeuds"

```
List<String> solutions = Propagation.propagationDeMarqueurs(reseauSemantique,
M1_node, M2_node, relation);

                // Displaying the solutions found
                int i = 0;
                for (String solution : solutions) {
                    System.out.println(M1_node.get(i) + " " + relation + " " +
M2_node.get(i));
                    System.out.println(solution);
                    i++;
                }
```

4. Analyse des résultats

Voici une analyse des résultats obtenus à partir du code de propagation de marqueurs dans le réseau sémantique :

- Pour chaque paire de nœuds (M1, M2) fournie en entrée, le programme tente de propager des marqueurs à travers le réseau sémantique en suivant les relations "is a".
- Si une relation spécifique (par exemple, "contient") est établie entre les nœuds M1 et M2, le programme affiche un message indiquant cette relation.
- Si aucune relation n'est trouvée entre les nœuds M1 et M2 malgré la propagation des marqueurs, le programme affiche un message indiquant l'absence de lien entre les nœuds.
- Les résultats obtenus sont basés sur la structure du réseau sémantique et les relations définies entre les nœuds. Ainsi, l'analyse des résultats permet de déduire la connectivité entre les concepts du réseau et de vérifier si les relations attendues entre les nœuds sont présentes.
- En examinant les solutions fournies par le programme pour chaque paire de nœuds, il est possible de valider la cohérence des informations extraites du réseau sémantique et de vérifier si les relations entre les concepts correspondent aux attentes.
- Si des résultats inattendus sont observés, cela peut indiquer des erreurs dans la structure du réseau sémantique ou des relations incorrectement définies entre les nœuds. Dans ce cas, une révision de la structure du réseau ou des relations est nécessaire pour garantir la précision des résultats.

5. Conclusion

En conclusion, le code de propagation de marqueurs dans un réseau sémantique offre une méthode efficace pour explorer les relations entre les concepts. Il permet de valider la cohérence des relations définies dans le réseau et d'identifier d'éventuelles erreurs ou lacunes. Cet outil fournit ainsi une perspective précieuse pour l'analyse et la compréhension des relations conceptuelles dans un domaine spécifique, ouvrant la voie à diverses applications dans l'informatique et l'analyse de données.

Partie II :

1. Introduction

L'héritage dans les réseaux sémantiques joue un rôle crucial dans la représentation des connaissances et des relations entre les concepts. Ces réseaux fournissent un moyen efficace de modéliser la structure hiérarchique des informations dans divers domaines, tels que la science, la linguistique, et l'intelligence artificielle. L'héritage permet aux concepts d'hériter des propriétés et des relations de leurs ancêtres, simplifiant ainsi la gestion et l'organisation des connaissances. Dans ce contexte, l'implémentation d'algorithmes d'héritage dans les réseaux sémantiques revêt une importance particulière pour l'exploitation et l'analyse efficace des informations contenues dans ces structures.

2. Définition

L'héritage dans les réseaux sémantiques implique la transmission des caractéristiques, attributs et relations d'un concept à ses descendants dans une structure hiérarchique. Cette notion permet une organisation efficace des informations en établissant des relations de parenté entre les concepts. Dans un réseau sémantique, un concept hérite des propriétés définies par ses prédécesseurs, facilitant ainsi la gestion et l'analyse des connaissances. L'héritage contribue à la modularité, à la réutilisabilité et à la compréhension des informations en permettant une représentation structurée et concise des relations entre les concepts d'un domaine spécifique.

2.1. Exemple

Un exemple d'héritage serait la relation entre les concepts "Chien" et "Animal". Si "Chien" est défini comme descendant de "Animal", alors il hérite des caractéristiques et des comportements généraux attribués à tous les animaux. Par exemple, les chiens ont des propriétés telles que "manger", "dormir" et "se déplacer", qui sont héritées de la classe "Animal". De plus, les relations spécifiques à cette hiérarchie, telles que "Chien est_un Animal", illustrent comment l'héritage est représenté dans le réseau sémantique.

3. Implémentation

3.1. Structure de code

Le code est organisé en une classe nommée "Heritage".

Cette classe contient trois méthodes principales :

- **"getLabel" :** Cette méthode prend en paramètre un objet JSON représentant un réseau sémantique et l'ID d'un nœud, puis retourne l'étiquette correspondante à ce nœud.

```
public static String getLabel(JSONObject reseauSemantique, String nodeId) {
        JSONArray nodes = (JSONArray) reseauSemantique.get("nodes");
        for (Object o : nodes) {
            JSONObject node = (JSONObject) o;
            if (node.get("id").equals(nodeId)) {
                return (String) node.get("label");
            }
        }
        return "";
    }
```

- **"heritage" :** Cette méthode effectue l'algorithme d'héritage pour trouver tous les nœuds hérités à partir d'un nœud donné. Elle prend en paramètre un objet JSON représentant le réseau sémantique et le nom du nœud dont on veut trouver les héritages, puis retourne une liste de noms de nœuds hérités.

```
public static List<String> heritage(JSONObject reseauSemantique, String name)
```

- **"getProperties" :** Cette méthode récupère toutes les propriétés d'un nœud, y compris les propriétés héritées. Elle prend en paramètre un objet JSON représentant le réseau sémantique et le nom du nœud dont on veut obtenir les propriétés, puis retourne une liste de chaînes de caractères représentant les propriétés.

```
public static List<String> getProperties(JSONObject reseauSemantique, String name)
```

3.2. Explication du code

Ce code Java vise à effectuer des opérations sur un réseau sémantique représenté sous forme d'objet JSON. Voici une explication de chaque partie du code :

- **Classe Heritage :**
  - Cette classe contient trois méthodes statiques pour travailler avec le réseau sémantique.

- **Méthode getLabel :**
  - Cette méthode prend un objet JSON représentant le réseau sémantique et l'ID d'un nœud en tant que paramètres.
  - Elle parcourt les nœuds du réseau sémantique pour trouver celui correspondant à l'ID donné.
  - Elle retourne l'étiquette (label) de ce nœud.

- **Méthode heritage :**
  - Cette méthode prend un objet JSON représentant le réseau sémantique et le nom d'un nœud comme paramètres.
  - Elle cherche le nœud correspondant au nom donné dans le réseau sémantique.
  - En utilisant un algorithme d'héritage, elle trouve tous les nœuds hérités de ce nœud.
  - Elle retourne une liste contenant les noms des nœuds hérités.

- **Méthode getProperties :**
  - Cette méthode prend un objet JSON représentant le réseau sémantique et le nom d'un nœud comme paramètres.
  - Elle recherche d'abord les propriétés directes du nœud spécifié.
  - En utilisant un algorithme similaire à celui de la méthode heritage, elle trouve toutes les propriétés héritées de ce nœud.
  - Elle retourne une liste contenant les propriétés du nœud, y compris les propriétés héritées.

3.3. Résultat de l’exécution

- Le programme affiche le nœud utilisé pour l'inférence, qui est "Titi".
- Ensuite, il affiche une liste des propriétés héritées par le nœud "Titi". Par exemple, "Animal", "Mammifère" et "Vertébré" sont des propriétés héritées que "Titi" a en tant qu'animal.

- Ensuite, il affiche une liste des propriétés spécifiques à "Titi". Par exemple, "mange: Carnivore" indique que "Titi" est un carnivore et "taille: Moyenne" indique sa taille.
- Les propriétés héritées sont des caractéristiques que "Titi" hérite de ses ancêtres dans la hiérarchie des classes du réseau sémantique.
- Les propriétés spécifiques sont des caractéristiques uniques à "Titi" qui ne sont pas héritées d'autres nœuds.

4. Analyse des résultats

- **Résultat de l'inférence :**
  - Le programme affiche "Titi" comme le nœud utilisé pour l'inférence, ce qui indique que les propriétés sont déduites pour ce nœud spécifique.
  - Ensuite, il présente une liste des propriétés héritées par "Titi". Cela suggère que "Titi" possède les caractéristiques communes à tous les nœuds dont il est un descendant dans la hiérarchie de l'arbre sémantique.
  - Par exemple, "Animal", "Mammifère" et "Vertébré" sont des propriétés que "Titi" hérite de ses ancêtres dans la classification des espèces.

- **Déduction des propriétés spécifiques :**
  - Le programme affiche ensuite les propriétés spécifiques à "Titi", qui sont des caractéristiques uniques à ce nœud.
  - Par exemple, "mange: Carnivore" indique que "Titi" est un carnivore, ce qui est une propriété spécifique à "Titi" plutôt qu'une caractéristique partagée par tous les mammifères ou vertébrés.
  - De même, "taille: Moyenne" indique la taille spécifique de "Titi", ce qui est une propriété qui ne peut pas être déduite uniquement à partir de la hiérarchie d'héritage.

- **Implications :**
  - Ce résultat montre comment le concept d'héritage est utilisé pour inférer les propriétés des nœuds dans une structure de réseau sémantique.
  - Il souligne également la distinction entre les caractéristiques générales héritées et les caractéristiques spécifiques à un nœud particulier.

5. Conclusion

Le programme de déduction des propriétés dans un réseau sémantique offre un aperçu fascinant de la façon dont les informations sont organisées et inférées dans une structure hiérarchique. En utilisant le concept d'héritage, le programme identifie les propriétés communes héritées par un nœud et déduit également les caractéristiques spécifiques à ce nœud.

Cette approche offre une méthode puissante pour comprendre les relations et les caractéristiques des entités dans un système complexe. Elle trouve des applications importantes dans de nombreux domaines, notamment la classification des données, la recherche d'informations, l'analyse des réseaux sociaux et bien d'autres.

Partie III :

1. Introduction

Dans le domaine de la représentation des connaissances, les réseaux sémantiques jouent un rôle essentiel en structurant les informations sous forme de graphes composés de nœuds (concepts) et d'arêtes (relations entre les concepts). Lorsqu'on travaille avec des réseaux sémantiques, il est fréquent de rencontrer des scénarios où certaines relations ou propriétés nécessitent une gestion particulière, notamment à travers des exceptions. Ces exceptions peuvent inclure des relations spéciales, des restrictions spécifiques, ou des cas particuliers qui influencent le comportement des algorithmes de propagation et d'héritage. Cela assure une interprétation plus fiable et cohérente des informations contenues dans le réseau sémantique.

2. Définition

Les exceptions dans les réseaux sémantiques font référence à des cas particuliers ou des relations spécifiques qui dérogent aux règles ou aux comportements généraux du réseau. Ces exceptions peuvent inclure des relations spéciales, des contraintes spécifiques ou des règles uniques qui doivent être prises en compte lors de la propagation des marqueurs ou de l'héritage des propriétés. Elles permettent de gérer des situations où des relations standard ne s'appliquent pas ou où des exceptions à la règle générale doivent être explicitement définies. Les exceptions assurent que les algorithmes de traitement des réseaux sémantiques fonctionnent de manière précise et conforme aux réalités du domaine représenté.

2.1. Exemple

Considérons un réseau sémantique représentant une hiérarchie de catégories d'animaux. Dans ce réseau, les relations "is_a" sont utilisées pour définir des relations de type "est un". Cependant, certaines exceptions peuvent exister, comme des relations spéciales "edge_type" qui indiquent des particularités uniques ou des exceptions dans l'héritage des propriétés.

```
{
  "nodes": [
    {"id": "1", "label": "Animal"},
    {"id": "2", "label": "Mammifère"},
    {"id": "3", "label": "Oiseau"},
    {"id": "4", "label": "Chauve-souris"},
    {"id": "5", "label": "Canard"}
  ],
  "edges": [
    {"from": "2", "to": "1", "label": "is_a"},
    {"from": "3", "to": "1", "label": "is_a"},
    {"from": "4", "to": "2", "label": "is_a", "edge_type": "exception"},
    {"from": "5", "to": "3", "label": "is_a"},
    {"from": "5", "to": "1", "label": "can swim"}
  ]
}
```

Nœuds et relations standard :

- "Mammifère" (id: 2) est un "Animal" (id: 1) via la relation "is_a".
- "Oiseau" (id: 3) est un "Animal" (id: 1) via la relation "is_a".
- "Canard" (id: 5) est un "Oiseau" (id: 3) via la relation "is_a".

Exception :

- "Chauve-souris" (id: 4) est un "Mammifère" (id: 2) via une relation "is_a" avec un "edge_type" spécifié comme "exception". Cela signifie que, bien qu'une chauve-souris soit classée comme mammifère, il peut y avoir des propriétés ou des comportements spécifiques qui nécessitent un traitement particulier, différent des autres mammifères.

3. Implémentation

3.1. Structure de code

Le code est structuré en plusieurs parties distinctes, chacune ayant un rôle spécifique dans la gestion et l'utilisation des réseaux sémantiques. Voici une vue d'ensemble de la structure du code :Cette classe contient trois méthodes principales :

- **Méthodes Utilitaires**

getLabel: Cette méthode récupère l'étiquette d'un nœud donné son identifiant et une relation.

```
public static String getLabel(JSONObject reseauSemantique, JSONObject node, String
relation) {
        List<String> nodeRelationEdgesLabel = new ArrayList<>();
        JSONArray edges = (JSONArray) reseauSemantique.get("edges");
        JSONArray nodes = (JSONArray) reseauSemantique.get("nodes");

        for (Object o : edges) {
            JSONObject edge = (JSONObject) o;
            if (edge.get("to").equals(node.get("id")) &&
edge.get("label").equals(relation)) {
                for (Object n : nodes) {
                    JSONObject no = (JSONObject) n;
                    if (no.get("id").equals(edge.get("from"))) {
                        nodeRelationEdgesLabel.add((String) no.get("label"));
                    }
                }
            }
        }
        return "il y a un lien entre les 2 noeuds : " + String.join(", ",
nodeRelationEdgesLabel);
    }
```

- **Méthodes Principales**

propagationDeMarqueurs: Cette méthode gère la propagation des marqueurs entre les nœuds en tenant compte des exceptions.

```
public static List<String> propagationDeMarqueurs(JSONObject reseauSemantique,
List<String> node1, List<String> node2, String relation) {
        List<String> solutionsFound = new ArrayList<>();
        JSONArray nodes = (JSONArray) reseauSemantique.get("nodes");
        JSONArray edges = (JSONArray) reseauSemantique.get("edges");
```

3.2. Explication du code

- **getLabel :**
  - Parcourt les arêtes du réseau pour trouver les relations spécifiques d'un nœud.
  - Retourne une chaîne indiquant les relations trouvées.

- **propagationDeMarqueurs :**
  - Initialise les listes de solutions et récupère les nœuds et arêtes du réseau.
  - Pour chaque paire de nœuds des listes node1 et node2, tente de trouver une solution de propagation des marqueurs.
  - Trouve les nœuds correspondants aux étiquettes fournies.
  - Parcourt les arêtes "is_a" pour propager les marqueurs, en tenant compte des exceptions spécifiées par l'attribut "edge_type".
  - Ajoute les solutions trouvées ou des messages d'erreur en cas d'échec.

3.3. Résultat de l'exécution

Le résultat du code montre que la relation "contient" entre "Modes de Representations des connaissances" et "Axiome A7" a été trouvée dans le réseau sémantique défini dans le fichier exception.json. Le programme a vérifié cette relation en parcourant les nœuds et les arêtes du réseau, confirmant ainsi l'existence de cette relation sans rencontrer d'exceptions. Le résultat final indique qu'il y a un lien entre ces deux nœuds.

4. Analyse des résultats

- Définition des Nœuds et de la Relation
  - M1_node contient un seul élément : "Modes de Representations des connaissances".
  - M2_node contient un seul élément : "Axiome A7".
  - La relation à tester est "contient".

```
List<String> M1_node = Collections.singletonList("Modes de Representations des
connaissances");
                List<String> M2_node = Collections.singletonList("Axiome A7");
                String relation = "contient";  // Relationship to test
```

- Appel de la Méthode propagationDeMarqueurs
  - La méthode propagationDeMarqueurs est appelée avec les nœuds et la relation spécifiés.
  - Cette méthode cherche à établir si une relation "contient" existe entre les nœuds spécifiés dans le réseau sémantique.

```
List<String> solutions = Exceptions.propagationDeMarqueurs(reseauSemantique, M1_node,
M2_node, relation);
```

- Affichage des Solutions Trouvées
  - Le programme affiche "Modes de Representations des connaissances contient Axiome A7".
  - Ensuite, il affiche le résultat de la méthode getLabel qui retourne "il y a un lien entre les 2 noeuds : Modes de Representations des connaissances".

```
// Displaying the solutions found
                int i = 0;
                for (String solution : solutions) {
                    System.out.println(M1_node.get(i) + " " + relation + " " +
M2_node.get(i));
                    System.out.println(solution);
```

5. Conclusion

Le programme de déduction des propriétés dans un réseau sémantique offre un aperçu fascinant de la façon dont les informations sont organisées et inférées dans une structure hiérarchique. En utilisant le concept d'héritage, le programme identifie les propriétés communes héritées par un nœud et déduit également les caractéristiques spécifiques à ce nœud.

Cette approche offre une méthode puissante pour comprendre les relations et les caractéristiques des entités dans un système complexe. Elle trouve des applications importantes dans de nombreux domaines, notamment la classification des données, la recherche d'informations, l'analyse des réseaux sociaux et bien d'autres.

TP 6 :
Les logiques de description

1. Introduction

Dans cette partie, nous examinerons l'implémentation des logiques de description à travers un exemple de code Java tiré de la bibliothèque TweetyProject. Ce code illustre l'utilisation des classes de syntaxe de la logique de description ainsi que du raisonneur de logique de description naïf.

2.Définition de la logique

Les logiques de description sont des langages formels utilisés pour représenter et raisonner sur les connaissances en utilisant des concepts, des rôles et des individus. Elles permettent de décrire des classes d'objets, leurs propriétés et leurs relations, offrant ainsi un cadre formel pour modéliser des domaines complexes et raisonner de manière déductive sur eux.

2.1. Exemple

Considérons l'exemple suivant :

- **Concepts :**

Animal : représente la classe générale des animaux.
Oiseau : sous-classe d'Animal, représentant les oiseaux.
Poisson : sous-classe d'Animal, représentant les poissons.

- **Rôles :**

ADesAiles : représente la propriété "a des ailes", indiquant que l'animal peut voler.
ADeLaFourrure : représente la propriété "a de la fourrure", indiquant que l'animal est couvert de poils.

- **Individus :**

Tweety : un individu appartenant à la classe Oiseau.
Nemo : un individu appartenant à la classe Poisson.

À l'aide de la logique de description, on peut énoncer des axiomes et des assertions pour spécifier des connaissances sur ces classes et individus. Par exemple :
"Tweety est un oiseau."
"Les oiseaux ont des ailes."
"Les poissons n'ont pas de fourrure."

3. Implémentation

3.1. Structure de code

Le code est organisé comme suit :

- Définition des classes AtomicConcept, AtomicRole, Individual et DlBeliefSet pour représenter les concepts, les rôles, les individus et les ensembles de croyances en logique de description.

```
//Create description logics signature
        AtomicConcept human = new AtomicConcept("Human");
        AtomicConcept male = new AtomicConcept("Male");
        AtomicConcept female = new AtomicConcept("Female");
        AtomicConcept house = new AtomicConcept("House");
```

```
        AtomicConcept father = new AtomicConcept("Father");
        AtomicRole fatherOf = new AtomicRole("fatherOf");
        Individual bob = new Individual("Bob");
        Individual alice = new Individual("Alice");
```

- Définition d'axiomes terminologiques et assertifs à l'aide des classes EquivalenceAxiom, ConceptAssertion et RoleAssertion.

```
    //Create some terminological axioms
        EquivalenceAxiom femaleHuman = new EquivalenceAxiom(female,human);
        EquivalenceAxiom maleHuman = new EquivalenceAxiom(male,human);
        EquivalenceAxiom femaleNotMale = new EquivalenceAxiom(female,new
Complement(male));
        EquivalenceAxiom maleNotFemale = new EquivalenceAxiom(male,new
Complement(female));

        EquivalenceAxiom fatherEq = new EquivalenceAxiom(father, new
Union(male,fatherOf));
        EquivalenceAxiom houseNotHuman = new EquivalenceAxiom(house,new
Complement(human));

        //Create some assertional axioms
        ConceptAssertion aliceHuman = new ConceptAssertion(alice,human);
        ConceptAssertion bobHuman = new ConceptAssertion(bob,human);
        ConceptAssertion aliceFemale = new ConceptAssertion(alice,female);
        ConceptAssertion bobMale = new ConceptAssertion(bob,male);
        RoleAssertion bobFatherOfAlice = new RoleAssertion(bob,alice,fatherOf);
```

- Création d'un ensemble de croyances (DlBeliefSet) contenant les axiomes définis.

```
        //Add axioms to knowledge base
        DlBeliefSet dbs = new DlBeliefSet();
        dbs.add(femaleHuman);
        dbs.add(maleHuman);
        dbs.add(maleNotFemale);
        dbs.add(femaleNotMale);
        dbs.add(maleHuman);
```

- Utilisation du raisonneur NaiveDlReasoner pour effectuer des requêtes sur l'ensemble de croyances et obtenir des réponses.

```
        NaiveDlReasoner reasoner = new NaiveDlReasoner();
        System.out.println("\n"+reasoner.query(dbs2,femaleHuman ));
        System.out.println(reasoner.query(dbs2,tweetyHuman ));
        System.out.println(reasoner.query(dbs2,aliceHuman ));
```

3.2. Explication du code

Le code commence par importer les classes nécessaires de la bibliothèque TweetyProject. Ensuite, il définit les concepts atomiques, les rôles atomiques et les individus utilisés pour construire les axiomes terminologiques et assertifs. Des axiomes sont ensuite créés pour spécifier les relations entre les concepts, les rôles et les individus.

Les axiomes sont ajoutés à un ensemble de croyances (DlBeliefSet), qui représente la base de connaissances en logique de description. Enfin, le raisonneur NaiveDlReasoner est utilisé pour interroger la base de connaissances et obtenir des réponses à des requêtes spécifiques sur les concepts et les individus.

3.3. Résultat de l'exécution

Les résultats du code montrent l'utilisation de la bibliothèque Tweety pour représenter les concepts, les axiomes terminologiques et assertifs, ainsi que les ensembles de croyances en logique de description.

Le raisonneur NaiveDlReasoner est utilisé pour effectuer des requêtes sur l'ensemble de croyances et obtenir des réponses concernant les concepts et les individus spécifiques.

```
        DlBeliefSet dbs2 = new DlBeliefSet();
        Individual tweety = new Individual("Tweety");
        ConceptAssertion tweetyMale = new ConceptAssertion(tweety,male);
        ConceptAssertion tweetyHuman = new ConceptAssertion(tweety,human);
        dbs2.add(aliceFemale);
        dbs2.add(tweetyMale);
        dbs2.add(maleNotFemale);
        dbs2.add(aliceHuman);
        // dbs2.add(maleHuman);
        NaiveDlReasoner reasoner = new NaiveDlReasoner();
        System.out.println("\n"+reasoner.query(dbs2,femaleHuman ));
        System.out.println(reasoner.query(dbs2,tweetyHuman ));
        System.out.println(reasoner.query(dbs2,aliceHuman ));
```

4. Analyse des résultats

a. Cohérence des Assertions :

- Les réponses du raisonneur indiquent la cohérence des assertions dans l'ensemble de croyances.
- Une réponse "true" confirme la validité des implications logiques entre les concepts, les rôles et les individus.

b. Validation des Axiomes Terminologiques :

- Les requêtes sur les axiomes terminologiques permettent de vérifier la validité des relations logiques entre les concepts.
- Une réponse "true" confirme la cohérence de la relation définie, tandis qu'une réponse "false" indique une incohérence.

c. Satisfaction des Axiomes Assertifs :

- Les résultats fournissent des informations sur la satisfaction des axiomes par rapport aux individus spécifiques.
- Une assertion "true" pour un individu indique son appartenance au concept spécifié, tandis qu'une réponse "false" indique le contraire.

d. Confirmation de la Validité des Relations Logiques :

- L'analyse des réponses confirme la validité des relations logiques définies entre les concepts, les rôles et les individus.
- Cela renforce la cohérence interne de l'ensemble de croyances et la fiabilité des implications logiques dans le domaine de la logique de description.

5. Conclusion

La gestion des exceptions dans les réseaux sémantiques est essentielle pour assurer la robustesse et la fiabilité des applications qui les utilisent. En fournissant des mécanismes pour gérer les erreurs et les situations imprévues de manière appropriée, les exceptions permettent aux programmes de maintenir un comportement cohérent même en cas de problème. Dans le contexte spécifique des réseaux sémantiques, les exceptions peuvent être utilisées pour gérer des scénarios tels que des erreurs de lecture de fichiers JSON, des nœuds inexistants ou des arêtes incorrectes. En les traitant correctement, les développeurs peuvent améliorer la qualité et la stabilité des systèmes basés sur les réseaux sémantiques.

Conclusion

Au cours de ce travail, nous avons exploré plusieurs concepts fondamentaux en informatique, notamment les réseaux sémantiques et la gestion des exceptions. Les réseaux sémantiques, utilisés pour représenter les connaissances et les relations entre les entités, sont des outils puissants pour modéliser des domaines complexes. Nous avons examiné comment lire et manipuler des réseaux sémantiques à l'aide de langages de programmation tels que Java, en mettant l'accent sur des opérations telles que la propagation de marqueurs et l'héritage.

Parallèlement, nous avons abordé l'importance de la gestion des exceptions pour assurer la fiabilité des applications. En fournissant des mécanismes pour détecter, signaler et gérer les erreurs, les exceptions jouent un rôle crucial dans la création de logiciels robustes et résilients.

Enfin, bien que non explicitement abordées dans ce contexte, les logiques de premier ordre et la logique modale sont également des sujets importants en informatique. Les logiques de premier ordre sont utilisées pour formaliser les raisonnements sur les relations entre objets, tandis que la logique modale permet de raisonner sur les modalités telles que la possibilité, la nécessité et la croyance.

En combinant ces concepts, nous avons une base solide pour concevoir, développer et analyser une variété de systèmes informatiques, allant des applications de traitement des connaissances aux systèmes d'intelligence artificielle en passant par les systèmes de gestion de base de données et bien plus encore. La compréhension de ces principes est essentielle pour les professionnels de l'informatique cherchant à construire des solutions logicielles efficaces et fiables dans un large éventail de domaines d'application.