Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumediene

Faculté Informatique

Département IA & SD

Filière : Informatique

Spécialité : SII

Rapports de travaux pratiques

RCR 1

Réalisé par :

TAOUCI Kenza

OUCHAOU Chafaa

Professeur :

F.Haned

Table des matières

Table des figures :

Tp2 : Logique des Prédicats

I. Introduction

Dans ce qui suit, nous présentons notre utilisation de la logique des prédicats avec la librairie Java Tweety pour modéliser des situations impliquant des personnes, des activités et des relations entre eux. Nous démontrons comment nous avons défini des prédicats, des constantes, des foncteurs, et des formules logiques pour représenter notre exemple, ainsi que la manière dont nous avons utilisé la librairie Tweety pour manipuler ces entités logiques.

II. Méthodologie

Choix d'un exemple concret

Nous explorerons l'utilisation de la librairie Tweety pour modéliser les relations entre différentes entités en utilisant des prédicats et des fonctions, avec comme scénario la représentation de relations entre personnes et activités. Voici une description détaillée de l'exemple :

1. *Définition des entités logiques :* définir les sortes (ou types) pour les entités, telles que "personne" et "activité".

```
FolSignature sig = new FolSignature();
Sort person = new Sort(name:"person");
sig.add(person);
Constant Chafaa = new Constant(name:"Chafaa", person); // Chafaa est une personne
sig.add(Chafaa);
Constant Kenza = new Constant(name:"Kenza", person); // Kenza est une personne
sig.add(Kenza);
Constant david = new Constant(name:"David", person); // David est une personne
sig.add(david);

Sort activity = new Sort(name:"activity");
sig.add(activity);
Constant hiking = new Constant(name:"hiking", activity); // hiking est une activité
sig.add(hiking);
Constant swimming = new Constant(name:"swimming", activity); // swimming est une activité
sig.add(swimming);
Constant painting = new Constant(name:"painting", activity); // painting est une activité
sig.add(painting);
```

Figure 1:Definition des entités logiques

2. *Définitions des prédicats et des foncteurs :* pour représenter des relations entre les entités, tels que "Participates" (participation à une activité) et "Organizes" (organisation d'une activité). Le prédicat "Organizes" est défini avec une arité de 2, indiquant qu'il s'agit d'un prédicat prenant deux termes en argument. Une fonction "knows est définie avec une arité de 1, indiquant qu'elle prend un terme de type person en argument.

```
Predicate participates = new Predicate(name:"Participates", 1); // création du prédicat Participates d'arité 1
sig.add(participates);
l = new ArrayList<Sort>();
l.add(person);
l.add(activity);
Predicate organizes = new Predicate(name:"Organizes", 1); // création du prédicat Organizes qui est d'arité 2
sig.add(organizes);
l = new ArrayList<Sort>();
l.add(person);
// isFriendWith est une fonction qui prend un terme qui est de type personne
Functor knows = new Functor(name:"knows", 1, person);
sig.add(knows);
```

Figure 2:Definition des prédicats et des foncteurs

3. *Construction de la base de connaissances :*

En utilisant la classe **FolBeliefSet**, nous avons créé une base de connaissances logique pour stocker nos formules logiques.
Pour l'ajout des formules logiques à la base de connaissances le parser **FolParser est utilisé.**

```
FolBeliefSet b = new FolBeliefSet();
b.setSignature(sig);
FolParser parser = new FolParser();
parser.setSignature(sig);
```

Figure 3:Creation de la base de connaissances

4. *Ajout des axiomes :*

On ajoute plusieurs axiomes pour exprimer des règles logiques concernant les relations entre les personnes, les activités et les connaissances.
Pour tout individu X, s'il participe à une activité, alors il existe une activité Y qu'il organise.
Si une personne participe à une activité, alors toute personne qui la connaît participe également à cette activité
Kenza organise une randonnée (hiking).
Chafaa participe à une activité.

```
b.add(parser.parseFormula(text:"forall X:(Participates(X) => (exists Y:(Organizes(X,Y))))"));
// si X participe à une activité, alors knows(X) participe également à cette activité
b.add(parser.parseFormula(text:"forall X:(Participates(X) => Participates(knows(X)))"));
b.add(parser.parseFormula(text:"Participates(Chafaa)")); // Chafaa participe à une activité
b.add(parser.parseFormula(text:"Organizes(Kenza, hiking)")); // Kenza organise une randonnée
```

Figure 4:Ajout des axiomes

Résultats

Nous avons affiché les formules logiques de notre base de connaissances pour vérifier qu'elles ont été correctement ajoutées.

```
Organizes(Kenza,hiking)
Participates(Chafaa)
forall X: ((Participates(X)=>exists Y: (Organizes(X,Y))))
forall X: ((Participates(X)=>Participates(knows(X))))
```

Figure 5:Resultats de formules logiques

Les résultats obtenus ont montré que nos formules logiques étaient cohérentes et conformes aux règles que nous avions définies.

Conclusion

Ce travail démontre l'efficacité de l'utilisation de la logique des prédicats et de la librairie Java Tweety pour modéliser des problèmes impliquant des relations logiques entre différents types d'entités.

Tp3 : Logique Modale

I. Introduction

Ce rapport présente une étude pratique de la logique modale à travers deux approches distinctes : la modélisation avec **Modal Logic Playground** et l'implémentation en Java avec **la librairie Tweety**. L'objectif principal de cette étude est de vérifier la valeur de vérité de quelques formules dans le cadre de la logique modale.

II. Méthodologie

Modal Logic Playground

1. *Fonctionnement :* Modal Logic Playground offre une interface conviviale permettant aux utilisateurs de saisir des formules de logique modale dans un langage formel et de les vérifier. Il permet de définir des mondes possibles, d'attribuer des valeurs de vérité à des propositions atomiques, et d'appliquer des opérateurs modaux tels que □ (nécessairement) et ◇ (possiblement).
2. *Utilisation pour la modélisation :* Dans cette partie de l'expérience, nous avons utilisé Modal Logic Playground pour modéliser différentes situations et vérifier la validité de formules de logique modale. Nous avons saisi les formules pertinentes dans l'outil, défini les mondes possibles et interprété les résultats pour évaluer la vérité des propositions modales.

3. *Modélisation :*

On choisit le nombre de variables propositionnelles : dans notre exemple = 4 (p, q, r, s) : initialement elle est ainsi :

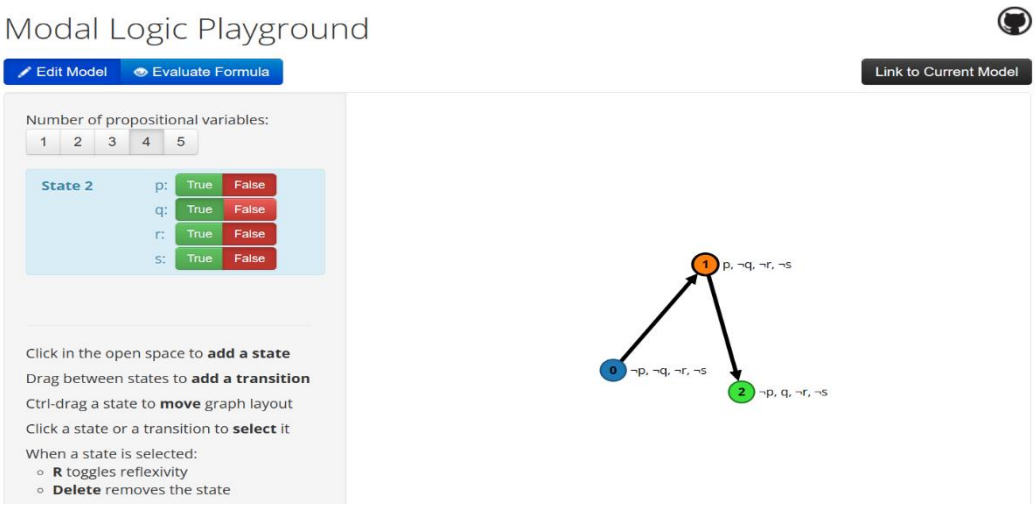

Figure 6:Modal Logic Playground

On prend un exemple de 7 mondes avec 4 variables puis on passe à l'évaluation des formules.

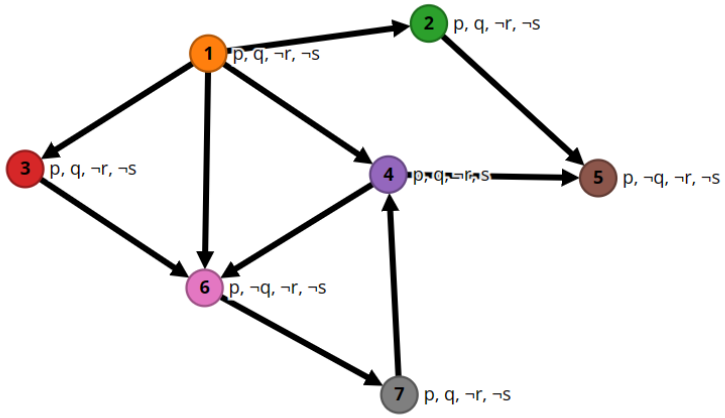

Figure 7:Exemple de modélisation d'un modèle modale

4. *Evaluation :*

[] <> (p | ~ r)

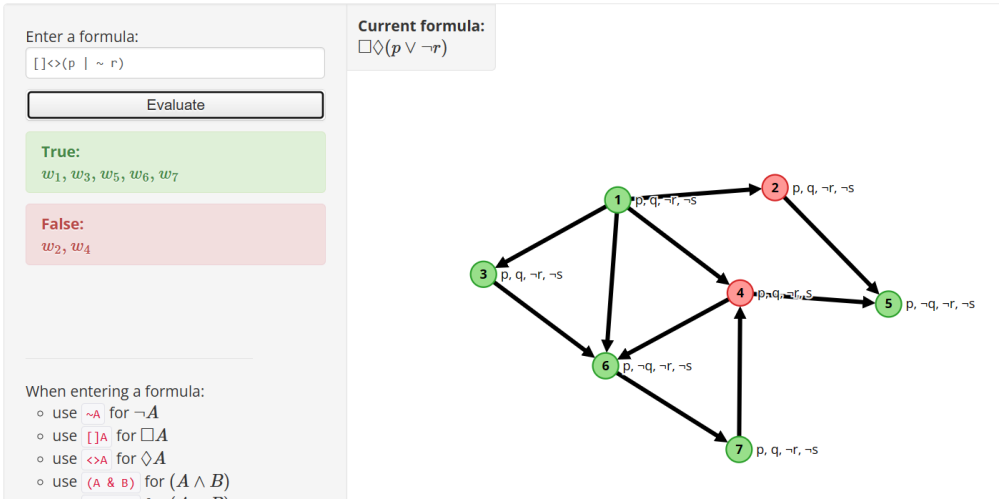

Figure 8:Evaluation de la première formule

<> (p & q)

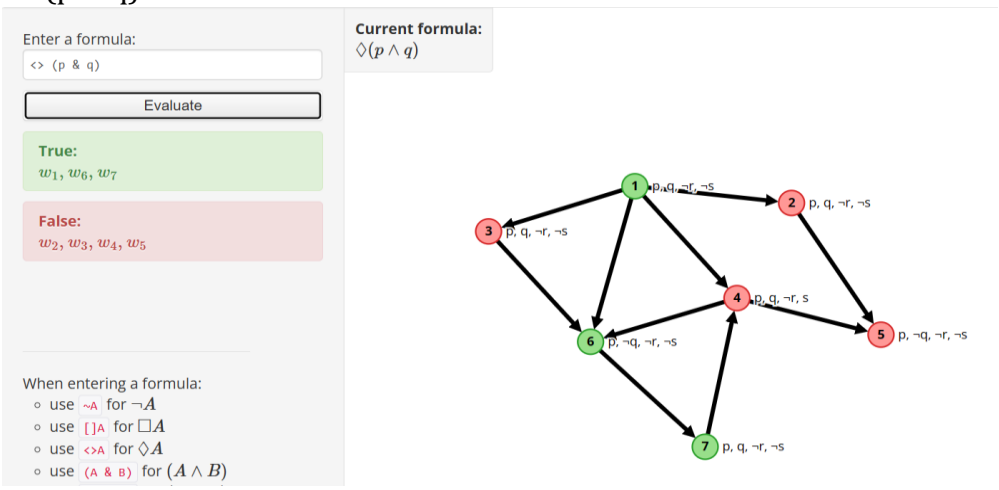

Figure 9:Evaluation de la deuxième formule

~[]<>p | ~s

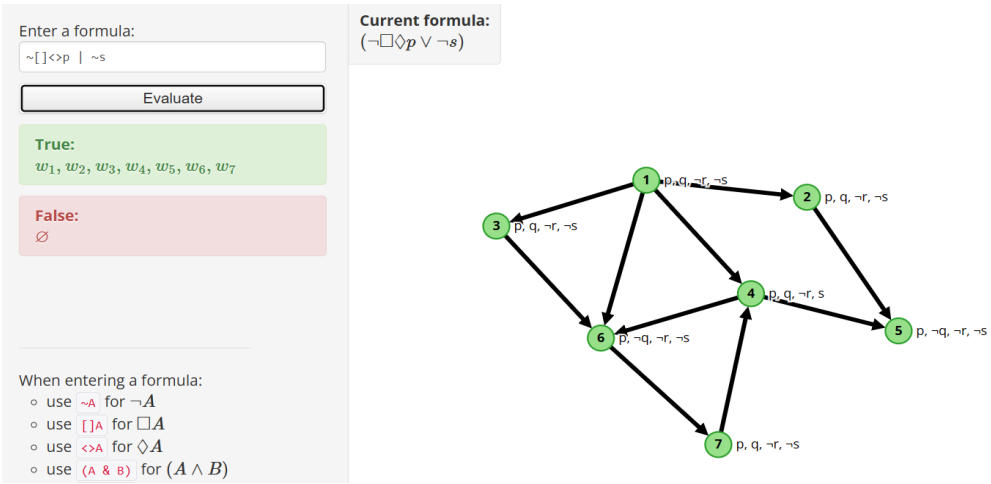

Figure 10:Evaluation de la troisième formule

~<> <> (p -> q | r)

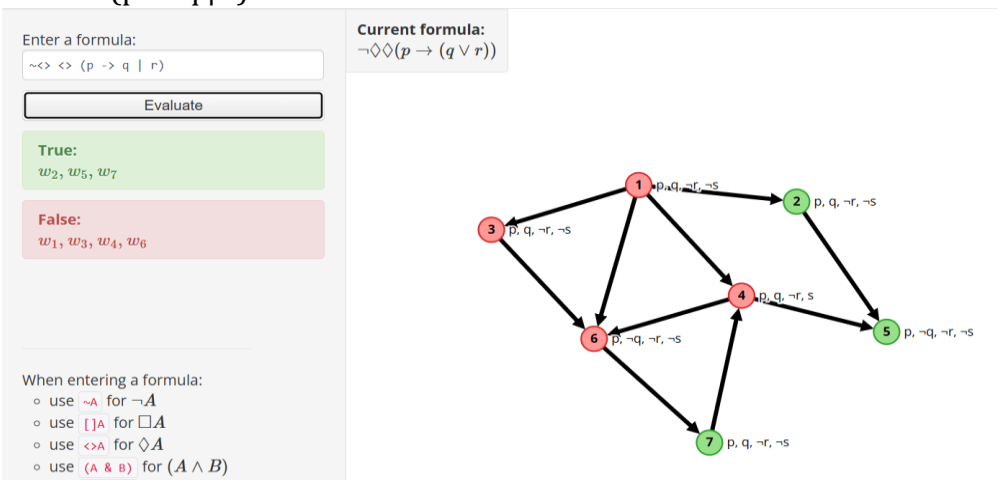

Figure 11:Evaluation de la formule 4

[] [] (p -> q)

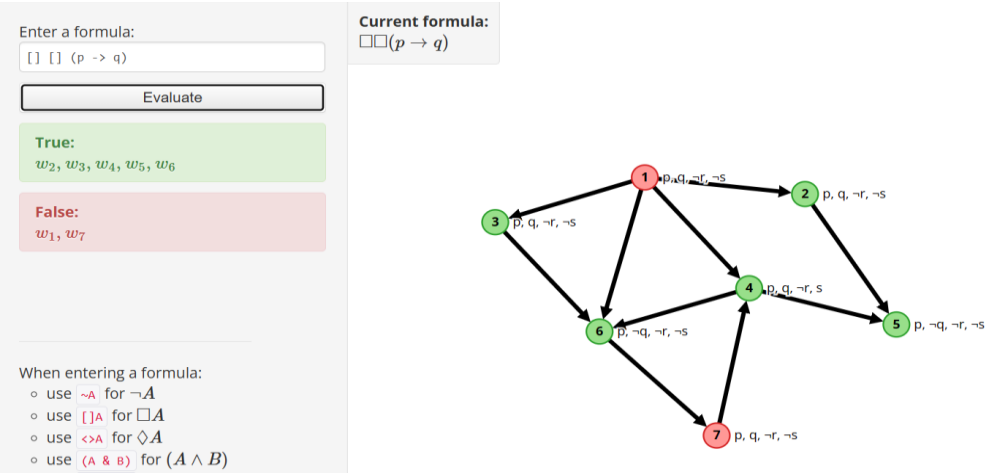

Figure 12:Evaluation formule 5

<> <> (p | s)

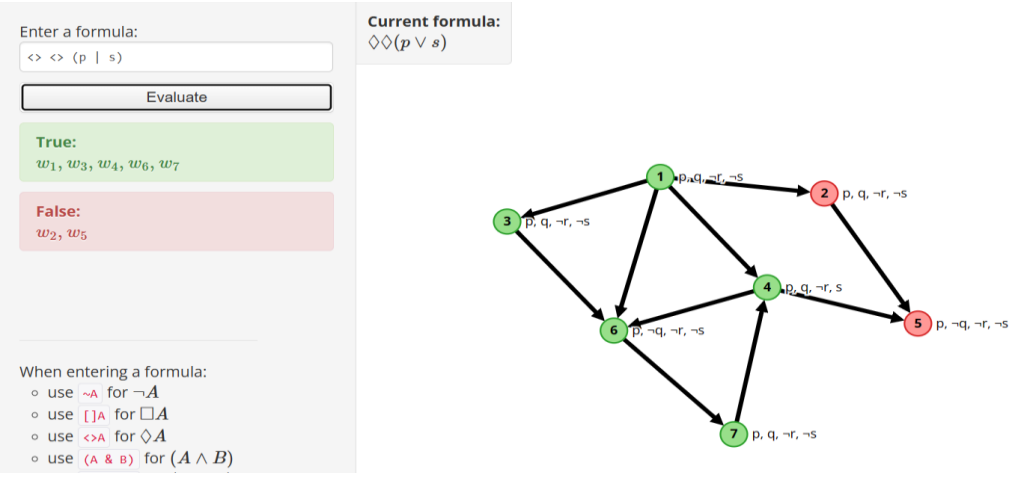

Figure 13:Evaluation formule 6

Librairie Java Tweety

1- *Fonctionnalités :* Tweety offre un ensemble de classes et de méthodes pour représenter des modèles logiques, définir des opérateurs modaux, et effectuer des opérations telles que la vérification de la satisfiabilité et la déduction.

2- *Utilisation pour l'implémentation en Java :*
Voici un exemple démontrant l'utilisation de la librairie Java Tweety pour travailler avec des formules modales et des formules de logique du premier ordre. L'exemple montre comment créer une base de connaissances modale, ajouter des formules, utiliser un analyseur syntaxique (parseur) pour interpréter les formules, et utiliser un moteur d'inférence (raisonneur) pour répondre aux requêtes sur la base de connaissances modale.

- Création de la base de connaissance modale :

```
MlBeliefSet bs = new MlBeliefSet(); // Mlbeliefset est utilise pour stocker les formules modales
```

Figure 14:Creation de la BCM

- Initialisation du parseur :

```
MlParser parser = new MlParser(); // Mlparser est utilise pour parser les formules modales
```

Figure 15:Initialisation du parseur

- Définition des symboles de la logique du premier ordre :

```
FolSignature sig = new FolSignature();  // FolSignature est utilise pour stocker les symboles de la logique du premier ordre
// On ajoute les symboles de la logique du premier ordre dans la signature de la logique modale
sig.add(new Predicate(name:"fenetres_fermees", arity:0));
sig.add(new Predicate(name:"porte_verrouillee", arity:0));
// On ajoute la signature de la logique du premier ordre dans le parser de la logique modale
parser.setSignature(sig);
```

Figure 16:Definition des symboles de la LPO

- Ajout des formules modales a la base de connaissances :

```
// On ajoute les formules modales dans la base de connaissances modale
bs.add((RelationalFormula)parser.parseFormula(text:"<>(fenetres_fermees && porte_verrouillee)"));
bs.add((RelationalFormula)parser.parseFormula(text:"[](!fenetres_fermees || porte_verrouillee)"));
bs.add((RelationalFormula)parser.parseFormula(text:"[](porte_verrouillee && <>(!porte_verrouillee))"));
```

Figure 17:Ajout des formules modales a la BC

La base de connaissances modale que nous avons créée dans l'exemple comprend trois formules modales :

La première formule signifie "Il est possible que les fenêtres soient fermées et la porte soit verrouillée".

La deuxième formule signifie "Il est nécessaire que si les fenêtres ne sont pas fermées, alors la porte est verrouillée".

La troisième formule signifie "Il est nécessaire que si la porte est verrouillée, alors il existe un monde où la porte n'est pas verrouillée".

- Utilisation du moteur d'inférence pour répondre aux requêtes :

```
//reasoner est utilise pour repondre aux requetes sur la base de connaissances modale par la methode query
SimpleMlReasoner reasoner = new SimpleMlReasoner();
System.out.println("[](!fenetres_fermees)      " + reasoner.query(bs, (FolFormula)parser.parseFormula(text:"[](!fenetres_fermees)")) + "\n");
System.out.println("<>(fenetres_fermees && porte_verrouillee)  " + reasoner.query(bs, (FolFormula)parser.parseFormula(text:"<>(fenetres_fermees && porte_verrouillee)")) + "\n");
System.out.println("!(porte_verrouillee)  " + reasoner.query(bs, (FolFormula)parser.parseFormula(text:"!(porte_verrouillee)")) + "\n");
```

Figure 18: Le moteur d'inférence pour les requêtes

3- Résultats d'exécution :

```
Modal knowledge base: { <>(fenetres_fermees&&porte_verrouillee), [](!fenetres_fermees||porte_verrouillee), [](porte_verrouillee&&<>(!porte_verrouillee)) }
[](!fenetres_fermees)      true

<>(fenetres_fermees && porte_verrouillee)  true

!(porte_verrouillee)  true
```

Figure 19:Résultats d'exécution

Tp4 : Logique des défauts

I. Introduction

La logique des défauts est un domaine de l'intelligence artificielle qui permet de raisonner de manière incertaine sur des connaissances incomplètes ou des situations ambiguës. Dans ce rapport, nous explorerons l'utilisation d'un *Default Logic Reasoner* pour modéliser et résoudre des problèmes de logique des défauts. Nous nous appuierons sur une implémentation en Java utilisant la bibliothèque Orbitale. L'outil utilisé est un raisonneur de logique par défaut basé sur la logique par défaut de Reiters, développé par Evan Morrison.

1. **Contexte :** *On* s'intéresse à l'application de la logique par défaut à un problème d'exploration spatiale. Le cas d'étude porte sur un astronaute effectuant une sortie extravéhiculaire (EVA).
2. **Objectifs :** L'objectif de ce rapport est de :
   - Décrire l'implémentation d'un raisonneur par défaut pour le cas d'étude.
   - Expliquer le fonctionnement du raisonneur et son application au problème posé.
   - Analyser les résultats obtenus et discuter de leurs implications.

II. Méthodologie

Rappels sur la logique des défauts

1- *Un monde* Représente l'état actuel des connaissances ou des faits dans un contexte donné.
Généralement exprimé par un ensemble de formules logiques décrivant les propositions vraies dans ce monde.

2- *Une Règle* Définit une relation entre des conditions nécessaires appelées "prérequis", des faits ou observations justifiant l'application de la règle, et des conséquences découlant de l'application des prérequis.
L'application d'une règle étend le monde en ajoutant ses conclusions lorsque ses prérequis sont satisfaits.

3- *Extensions* Représentent les différents états possibles du monde résultant de l'application de règles à un monde initial.
Obtenu en explorant toutes les combinaisons possibles de règles applicables et en déduisant les conséquences dans chaque cas.

Initialisation du monde et des règles Et raisonnement

Dans l'implémentation, le monde est représenté par la classe **WorldSet** et les règles sont représentées par la classe **RuleSet**. Voici comment ces éléments sont initialisés :

1. **Exemple 1 : Astronaute a bord :**
   a. **Initialisation du monde :**

```
WorldSet myWorld = new WorldSet();
// Commencer par croire que l'astronaute 1 est à bord
myWorld.addFormula(_wff:"astronaute1_a_bord");
```

Figure 20:Initialisation du monde

Dans cette ligne, un nouvel ensemble de monde est créé, puis la formule "astronaute1_a_bord" est ajoutée à cet ensemble, indiquant que l'astronaute 1 est à bord du vaisseau spatial.

b. Initialisation des règles :

```
DefaultRule rule1 = new DefaultRule();
rule1.setPrerequisite(prerequisite:"astronaute1_a_bord");
rule1.setJustificatoin(justificatoin:"vaisseau_pret");
rule1.setConsequence(consequence:"vaisseau_pret");

DefaultRule rule2 = new DefaultRule();
rule2.setPrerequisite(prerequisite:"astronaute1_a_bord");
rule2.setJustificatoin(justificatoin:"astronaute_en_mission");
rule2.setConsequence(consequence:"astronaute_en_mission");

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

Figure 21:Initialisation des règles

Ces lignes définissent deux règles par défaut : une règle indiquant que si l'astronaute est à bord, alors le vaisseau est prêt (**astronaute1_a_bord ==> vaisseau_pret**) et une autre règle indiquant que si l'astronaute est à bord, alors il est en mission (**astronaute1_a_bord ==> astronaute_en_mission**). Ces règles sont ajoutées à l'ensemble de règles **myRules**.

c. Raisonnement :

Le raisonneur de logique par défaut est utilisé pour déterminer les extensions possibles du monde initial en fonction des règles définies.

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

Figure 22:Raisonnement sur les défauts

2. Exemple 2 : EVA en cours :

Dans cet exemple, l'astronaute effectue une EVA (activité extravéhiculaire). Voici comment cela est ajouté au monde existant et comment cela affecte les extensions possibles :

a. **Mise à jour du monde :** En plus de la formule "astronaute1_a_bord", la formule "eva_en_cours" est ajoutée pour indiquer que l'EVA est en cours.

```
WorldSet myWorld = new WorldSet();
// Commencez par croire que l'astronaute 1 est à bord
myWorld.addFormula(_wff:"astronaute1_a_bord");
a.e.println(msg:"Nous recevons l'information selon laquelle l'astronaute1 effectue une EVA.");
// Apprenez un jour que l'astronaute 1 effectue une EVA
myWorld.addFormula(_wff:"eva_en_cours");
// Apprenez que si l'astronaute 1 est à bord et qu'une EVA est en cours, alors la mission de l'astronaute est terminée
myWorld.addFormula("(astronaute1_a_bord & eva_en_cours " + a.e.IMPLIES + " " + a.e.NOT + "astronaute_en_mission)");
```

Figure 23:Mise à jour du monde

b. Définition des règles :

```
DefaultRule rule1 = new DefaultRule();
rule1.setPrerequisite(prerequisite:"astronaute1_a_bord");
rule1.setJustificatoin(justificatoin:"vaisseau_pret");
rule1.setConsequence(consequence:"vaisseau_pret");

DefaultRule rule2 = new DefaultRule();
rule2.setPrerequisite(prerequisite:"astronaute1_a_bord");
rule2.setJustificatoin(justificatoin:"astronaute_en_mission");
rule2.setConsequence(consequence:"astronaute_en_mission");

DefaultRule rule3 = new DefaultRule();
rule3.setPrerequisite(prerequisite:"astronaute1_a_bord & eva_en_cours");
rule3.setJustificatoin(justificatoin:"eva_en_cours");
rule3.setConsequence(a.e.NOT + "astronaute_en_mission");

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

Figure 24:Définition des règles de l'exemple 2

Une nouvelle règle est définie pour gérer le cas où l'astronaute est à bord et qu'une EVA est en cours, ce qui signifie que la mission de l'astronaute est terminée.

c. **Raisonnement :** Les extensions possibles sont recalculées avec le monde mis à jour et la nouvelle règle en utilisant le DefaultReasoner :

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

Figure 25:raisonnement de l'exemple 2

III. Application sur les deux exemples

Résultats d'exécution d'Exemple 1

- A : astronaute_a_bord
- V : vaisseau_pret
- M : astronaute_en_mission

 =<W, D>

W= {A} ¬

D= {A: V/V, A:M/M}

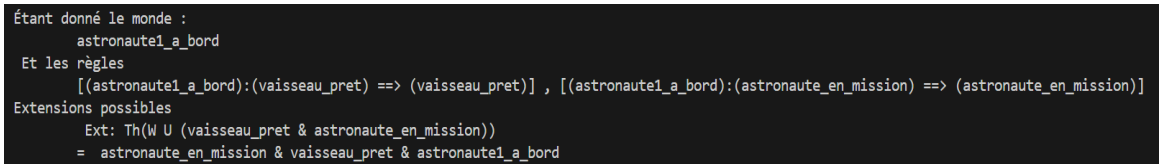

Figure 26:Resultats d'exécution

Extension: Th (W U (V&M))

Résultats d'exécution d'Exemple 2

Nous recevons l'information selon laquelle l'astronaute1 effectue une EVA.

□ =<W', D'>

W'= {A, E, A&E -> ¬ M}

D'= {A: V/V, A:M/M, A&E: E/¬ M}

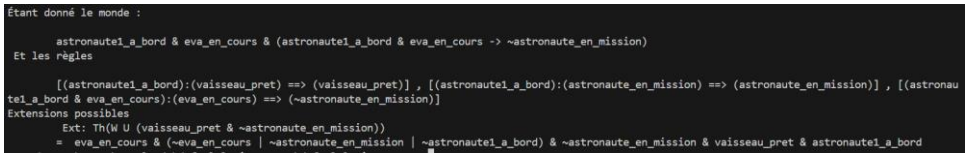

Figure 27:Résultats d'exécution d'exemple 2

Extension: Th (W U (V&¬M))

TP5 : Réseau sémantique

I. Introduction

Ce rapport présente les travaux pratiques réalisés sur les réseaux sémantiques. Un réseau sémantique est une représentation graphique des connaissances sous forme de nœuds (concepts) et d'arêtes (relations). Ces structures sont utilisées pour modéliser les relations entre différents concepts, facilitant ainsi la compréhension et l'analyse de ces relations. Ce TP a pour objectif de construire un réseau sémantique pour des concepts informatiques et de répondre à des questions complexes en se basant sur ce réseau.

Rappels sur les Réseaux Sémantiques

Un réseau sémantique est une structure de données graphique qui représente des connaissances sous forme de concepts et de relations. Les concepts sont des nœuds du graphe, tandis que les relations sont les arêtes qui relient ces nœuds. Les réseaux sémantiques sont utilisés dans divers domaines tels que la linguistique, l'intelligence artificielle, et la recherche d'informations pour modéliser les relations entre les concepts.

Exemple 1 :

Cet exemple contient l'algorithme avec des liens d'exception, et donc une différente démarche par rapport au premier.

Outils utilise

On a utilisé le langage de programmation JAVA afin de programmer le réseau

Explication

La classe "**SemanticNetwork**" encapsule la fonctionnalité liée au réseau sémantique, incluant les nœuds, les relations et l'algorithme de marquage. Elle possède deux attributs :

• "**nodes**", qui est une collection associant des clés (sous forme de chaînes de caractères) à des valeurs (nœuds), représentant les différents nœuds de notre réseau sémantique.

• "**edges**", qui est une liste d'objets de type "Edge", représentant les différentes relations entre les nœuds.

```java
public class SemanticNetwork {
    private static String IS_A = "is a";
    private HashMap<String, Node> nodes;
    private List<Edge> edges;

    public SemanticNetwork() {
        nodes = new HashMap<>();
        edges = new ArrayList<>();
    }
```

Figure 28: Réseau sémantique

De plus, la classe dispose d'un attribut de classe appelé "**IS_A**" qui est utilisé pour définir la relation "est un".

Attributs :

Elle contient 4 attributs :

- **name** : qui est l'étiquette du nœud

– **edges** : toutes les relations du nœud avec les autres.

- **exceptionLinks** : tous les liens d'exceptions.

- **visitedFrom** : un attribut qui sert pour l'algorithme de marquage.

```
public static class Node {
    private String name;
    private List<Edge> edges;
    private int visited_from;
    public Node(String name) {
        this.name = name;
        this.edges = new ArrayList<>();
        this.visited_from = -1;
    }
    public String getName() {
        return name;
    }
    public void addEdge(Edge edge) {
        edges.add(edge);
    }
    public List<Edge> getEdges() {
        return edges;
    }
    public int getVisitedFrom() {
        return visited_from;
    }
    public void setVisitedFrom(int visited_from) {
        this.visited_from = visited_from;
    }
    public void resetVisitedFrom() {
        this.visited_from = -1;
    }
}
```

Figure 29: Noeud

Méthodes : Les getters et les setters, et les méthodes suivantes :

- **addEdge()** : permet d'ajouter une relation au nœud.

- **getExceptionLinks**() : Cette méthode renvoie la liste des liens d'exception du nœud.

- **addExceptionLink(Edge edge):** Cette méthode ajoute un lien d'exception (passé en paramètre) à la liste des liens d'exception du nœud.

```
public void addEdge(String fromNode, String toNode, String edgeName) {
    Node from = nodes.get(fromNode);
    Node to = nodes.get(toNode);
    if (from != null && to != null) {
        Edge edge = new Edge(from, to, edgeName);
        edges.add(edge);
        from.addEdge(edge);
    }
}
```

Figure 30:methode ajout d'un noeud

Classe Edge :

Attributs : Elle contient 3 attributs :

- **from**: le nœud de départ de la relation.

- **to**:le nœud final de la relation.

- **name**: qui est l'étiquette de la relation.

```
public static class Edge {
    private Node from;
    private Node to;
    private String name;
    public Edge(Node from, Node to, String name) {
        this.from = from;
        this.to = to;
        this.name = name;
    }
    public Node getFrom() {
        return from;
    }
    public Node getTo() {
        return to;
    }
    public String getName() {
        return name;
    }
}
```

Figure 31:Classe Edge Relation

addNode et addEdge permettent de rajouter un nœud et une relation dans “nodes” et “edges” respectivement

public List *propagate*(Node start,int mark1,Node end ,int mark2) :

Cette méthode va prendre en paramètre le nœud de départ et le nœud d'arrivée généralement on applique l’algorithme de marquage pour répondre à une question du genre “**Quelles sont les guerres qui ont généré des pertes financières ?**” Ici le nœud de

départ c'est "**guerre**" et le nœud d'arrivé est "**pertes financières**" et le but de l'algorithme c'est de retrouver toutes les guerres qui génère des pertes financières.

Le paramètre "**mark1**" désigne la marque M1 et "**mark2**" désigne la marque M2 de l'algorithme de marquage.

La méthode **propagate** va faire appel à une autre méthode propagate deux fois qui est surchargé en commençant à propager la marque M1 depuis le noeud de départ (appel 1) puis de propager la marque M2 (appel 2) puis retourner les noeuds qui ont la marque M1 et qui sont en relation avec les noeuds de la marque M2.

private boolean isException(Node node, Edge edge):

Cette méthode vérifie si un lien donné (**edge**) est une exception pour un nœud donné (**node**). Elle parcourt les liens d'exception du nœud et vérifie si l'un d'entre eux correspond exactement au lien donné.

public List inheritance(Node node) :

Cette méthode implémente un algorithme pour obtenir les propriétés **héritées** d'un nœud. Elle parcourt les arêtes sortantes d'un nœud donné, **elle vérifie si le lien est une exception en appelant la méthode isException**. Si le lien est identifié comme une exception, la méthode ignore cette arête et passe à l'arête suivante **sans ajouter le nœud de destination correspondant à la liste des propriétés héritées**.

Ensuite, elle continue à explorer les arêtes sortantes des nœuds nouvellement ajoutés jusqu'à ce qu'il n'y ait plus de liens "**is a**" à suivre.

```java
public List<Node> inheritance(Node node) {
    List<Node> properties = new ArrayList<>();
    List<Node> parents = new ArrayList<>();
    parents.add(node);
    while (!parents.isEmpty()) {
        Node parent = parents.remove(0);
        for (Edge edge : parent.getEdges()) {
            if (edge.getName().equals(SemanticNetwork.IS_A)) {
                if (isException(node, edge)) {
                    continue;
                }
                Node child = edge.getTo();
                if (!properties.contains(child)) {
                    properties.add(child);
                    parents.add(child);
                }
            }
        }
    }
    return properties;
}
```

Figure 32: Méthode pour faire l'héritage

MAIN :

On crée un réseau sémantique en utilisant la classe **SemanticNetwork** et effectue différentes opérations sur ce réseau. Tout d'abord, le réseau est initialisé avec des nœuds représentant des concepts liés à la guerre, tels que "guerre conventionnelle", "guerre nucléaire", "guerre froide", etc. Ensuite, des relations sont établies entre ces nœuds, en

utilisant des liens "**est un**" et des liens spécifiques comme "**a généré**".

```
semanticNetwork.addNode(nodeName:"chouhada");
semanticNetwork.addNode(nodeName:"poilus");
semanticNetwork.addNode(nodeName:"pertes financieres");
semanticNetwork.addNode(nodeName:"victime militaire");
semanticNetwork.addNode(nodeName:"perte humaine");
semanticNetwork.addNode(nodeName:"fln");
semanticNetwork.addNode(nodeName:"parti politique");
semanticNetwork.addNode(nodeName:"larbi benmhidi");
semanticNetwork.addNode(nodeName:"hassiba benbouali");


// Ajout de relations
semanticNetwork.addEdge(fromNode:"guerre", toNode:"conflit", SemanticNetwork.IS_A);
semanticNetwork.addEdge(fromNode:"guerre conventionnelle", toNode:"guerre", SemanticNetwork.IS_A);
semanticNetwork.addEdge(fromNode:"guerre nucleaire", toNode:"guerre", SemanticNetwork.IS_A);
semanticNetwork.addEdge(fromNode:"guerre technologique", toNode:"guerre", SemanticNetwork.IS_A);
semanticNetwork.addEdge(fromNode:"guerre technologique", toNode:"pertes financieres", edgeName:"genere");
semanticNetwork.addEdge(fromNode:"usa russie",toNode:"guerre froide", SemanticNetwork.IS_A);
semanticNetwork.addEdge(fromNode:"guerre froide", toNode:"guerre", SemanticNetwork.IS_A);
semanticNetwork.addEdge(fromNode:"guerre algerie", toNode:"guerre conventionnelle", SemanticNetwork.IS_A);
semanticNetwork.addEdge(fromNode:"guerre algerie", toNode:"chouhada", edgeName:"a genere");
semanticNetwork.addEdge(fromNode:"1 guerre mondiale", toNode:"guerre conventionnelle", SemanticNetwork.IS_A);
semanticNetwork.addEdge(fromNode:"1 guerre mondiale", toNode:"poilus", edgeName:"a genere");
```

Figure 33: Création du réseau sémantique

Ensuite, un lien d'exception est ajouté pour spécifier qu'il y a une exception à la relation "**génère**" entre "**guerre technologique" et "pertes financières**". Cela signifie que *dans la plupart des cas, la guerre technologique génère des pertes financières, mais il existe des situations exceptionnelles où cela n'est pas le cas*.

Application de la méthodologie : Propagation -> inherance -> propagation

O effectue différentes opérations sur le réseau sémantique : **Quelle guerre a des victimes militaires ?**

```
public static void main(String[] args) {
    // Création du réseau sémantique
    SemanticNetwork semanticNetwork = new SemanticNetwork();
    // utilisation example (guerre)
    semanticNetwork = SemanticNetwork.example1();
    // Affichage du réseau sémantique

    Node n1  = semanticNetwork.getNode(nodeName:"guerre");
    Node n2  = semanticNetwork.getNode(nodeName:"victime militaire");

    // propager les marqueurs 1 et 2 (guerre et victime militaire)
    List<Node> nodes_res = semanticNetwork.propagate(n1, mark1:1, n2, mark2:2);

    // pour chaque noeud sans la liste nodes1
    for (Node node : nodes_res) {
        System.out.println(node.getName());
    }
}
```

Figure 34: Exemple de question ?

Tout d'abord, on utilise la méthode **propagate** pour propager des marqueurs à travers le réseau. Dans cet exemple, **un marqueur 1** est propagé à partir du nœud "guerre" et un marqueur 2 à partir du nœud "**victime militaire**". La méthode retourne une liste de nœuds

qui répondent aux critères de propagation, c'est-à-dire les nœuds qui sont connectés d'une manière spécifique à ces marqueurs.

Ces nœuds sont affichés en réponse à la question **"Quelle guerre a des victimes militaires ?**".

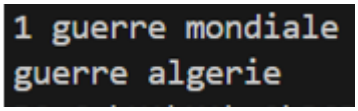

Figure 35: Réponse à la question

Ensuite, la méthode inheritance est utilisée pour récupérer les propriétés héritées d'un nœud spécifique. Dans cet exemple, les propriétés héritées de "guerre algérie" sont affichées. Cela signifie que les nœuds qui sont directement ou indirectement connectés à "guerre algérie" par une relation "est un" sont récupérés.

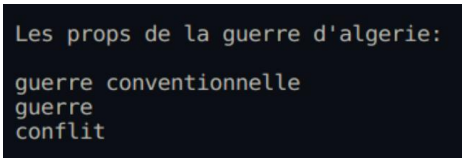

Figure 36: Réponse à l'inheritance

Enfin, la méthode ***propagate*** est utilisée à nouveau pour trouver les nœuds qui répondent à une autre question, à savoir "**Quelles guerres technologiques génèrent des pertes financières ?**". Le nœud "guerre technologique" est propagé avec un marqueur 3 et le nœud "pertes financières" avec un marqueur 4. Les nœuds qui répondent à cette propagation sont affichés.

Figure 37:Réponse a la question

Exemple 2 :

II. Méthodologie

Outil Utilisé

Pour réaliser ce TP, nous avons utilisé la bibliothèque **GraphStream** en **Java**. GraphStream est un outil puissant pour la manipulation et la visualisation de graphes dynamiques.

Choix d'un Exemple Concret

Nous avons choisi de travailler avec des concepts liés à l'informatique, tels que les langages de programmation (Python, Java, C++, Ruby), les applications, les développeurs, et les bibliothèques open-source. Ce choix est motivé par la richesse des relations possibles entre ces concepts et la pertinence de ces concepts dans le domaine informatique.

```java
// Classe représentant un concept informatique
class ConceptInformatique {
    String nom;
    List<RelationInformatique> relations;

    ConceptInformatique(String nom) {
        this.nom = nom;
        this.relations = new ArrayList<>();
    }

    void ajouterRelation(RelationInformatique relation) {
        relations.add(relation);
    }

    @Override
    public String toString() {
        return nom;
    }
}
```

Figure 38: Classe représentant un concept Informatique

```java
// Classe représentant une relation informatique entre deux concepts
class RelationInformatique {
    ConceptInformatique source;
    ConceptInformatique cible;
    String type;

    RelationInformatique(ConceptInformatique source, ConceptInformatique cible, String type) {
        this.source = source;
        this.cible = cible;
        this.type = type;
    }

    @Override
    public String toString() {
        return source.nom + " " + type + " " + cible.nom;
    }
}
```

Figure 39: Classe représentant une relation Informatique

Initialisation des Nœuds et Relations

Le réseau qu'on va modéliser est le suivant :

a. Les langages de programmation sont des outils.
b. Python, Java, C++ et Ruby sont des langages de programmation.
c. L'utilisation de Python a conduit à la popularité de bibliothèques

d. La première version de Python a été créée par Guido van Rossum.
e. Python, Java et C++ sont des langages de programmation de haut niveau.
f. Certains développeurs doutent que Ruby soit un langage de programmation de haut niveau.
g. La compétition entre Google et Apple concerne le développement d'assistants virtuels.
h. Le développement d'assistants virtuels génère des avancées dans le traitement du langage naturel.
i. La rivalité entre Google et Microsoft concerne le marché des systèmes d'exploitation.
j. Les utilisateurs de Python et de Java sont des développeurs logiciels.
k. Les développeurs logiciels sont des professionnels de l'informatique.
l. En général, les utilisateurs de Python préfèrent utiliser des bibliothèques open-source.
m. En général, les utilisateurs de Java travaillent dans des entreprises utilisant des applications d'entreprise.
n. Python est utilisé dans le domaine de l'analyse de données.

Nous avons initialisé notre réseau en définissant d'abord les concepts (nœuds) et les relations (arêtes) entre eux. Voici un extrait du code utilisé pour cette initialisation :

Création des concepts et relations

```
ConceptInformatique python = new ConceptInformatique(nom:"Python");
ConceptInformatique java = new ConceptInformatique(nom:"Java");
ConceptInformatique cpp = new ConceptInformatique(nom:"C++");
ConceptInformatique ruby = new ConceptInformatique(nom:"Ruby");
//ConceptInformatique outil = new ConceptInformatique("outil");
ConceptInformatique application = new ConceptInformatique(nom:"application");
ConceptInformatique entreprise = new ConceptInformatique(nom:"entreprise");
ConceptInformatique systemeExploitation = new ConceptInformatique(nom:"système d'exploitation");
ConceptInformatique assistantVirtuel = new ConceptInformatique(nom:"assistant virtuel");
ConceptInformatique developpeur = new ConceptInformatique(nom:"développeur logiciel");
ConceptInformatique bibliothequeOpenSource = new ConceptInformatique(nom:"bibliothèque open-source");
ConceptInformatique analyseDonnees = new ConceptInformatique(nom:"analyse de données");

// Ajout des concepts au réseau informatique
reseauInformatique.ajouterConcept(langage);
reseauInformatique.ajouterConcept(python);
reseauInformatique.ajouterConcept(java);
reseauInformatique.ajouterConcept(cpp);
reseauInformatique.ajouterConcept(ruby);
//reseauInformatique.ajouterConcept(outil);
reseauInformatique.ajouterConcept(application);
reseauInformatique.ajouterConcept(entreprise);
```

Figure 40: Création et ajout des concepts au réseau

```
RelationInformatique r1 = new RelationInformatique(langage, python, type:"est un");
RelationInformatique r2 = new RelationInformatique(langage, java, type:"est un");
RelationInformatique r3 = new RelationInformatique(langage, cpp, type:"est un");
RelationInformatique r4 = new RelationInformatique(langage, ruby, type:"est un");
RelationInformatique r5 = new RelationInformatique(python, application, type:"conduit à");
RelationInformatique r6 = new RelationInformatique(python, bibliothequeOpenSource, type:"conduit à");
RelationInformatique r7 = new RelationInformatique(developpeur, java, type:"utilise");
RelationInformatique r8 = new RelationInformatique(java, entreprise, type:"travaille dans");
RelationInformatique r9 = new RelationInformatique(java, systemeExploitation, type:"concerne");
RelationInformatique r10 = new RelationInformatique(ruby, developpeur, type:"doute");
RelationInformatique r11 = new RelationInformatique(ruby, application, type:"doute");
RelationInformatique r12 = new RelationInformatique(entreprise, application, type:"utilise");
RelationInformatique r13 = new RelationInformatique(systemeExploitation, assistantVirtuel, type:"concerne");
RelationInformatique r14 = new RelationInformatique(systemeExploitation, application, type:"concerne");
RelationInformatique r15 = new RelationInformatique(developpeur, analyseDonnees, type:"utilise");
RelationInformatique r16 = new RelationInformatique(python, analyseDonnees, type:"utilisé dans");
RelationInformatique r17 = new RelationInformatique(java, analyseDonnees, type:"utilisé dans");
RelationInformatique r18 = new RelationInformatique(cpp, analyseDonnees, type:"utilisé dans");

// Ajout des relations au réseau informatique
reseauInformatique.ajouterRelation(r1);
reseauInformatique.ajouterRelation(r2);
reseauInformatique.ajouterRelation(r3);
```

Figure 41: Création et ajout des relations au réseau sémantique

Visualisation du réseau

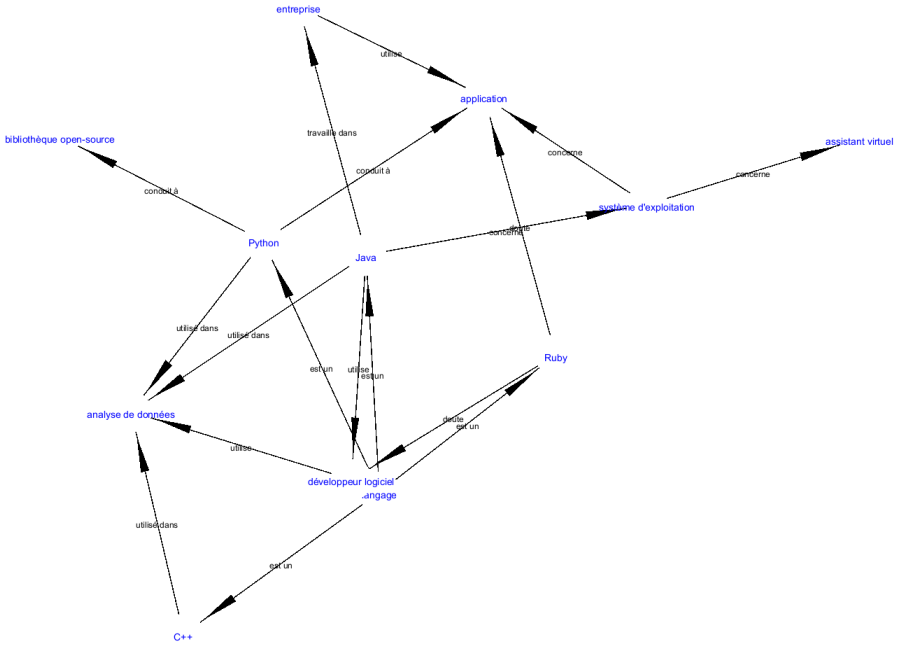

Figure 42: Visualisation du Réseau sémantique

Propagation de Marqueurs

Pour répondre aux questions posées au réseau sémantique, nous utilisons une méthode de propagation de marqueurs. Cette méthode consiste à marquer les nœuds et les relations pertinents en fonction de la question, et à propager ces marqueurs à travers le réseau pour identifier les concepts et relations pertinents.

Résultats d'Exécution des Exemples

Question 1 : **Quels sont les langages de programmation ?**

```java
// Répondre à la première question
String question1 = "Quels sont les langage de programmation ?";
List<ConceptInformatique> conceptsMarques1 = reseauInformatique.repondreQuestionInformatique(question1,verbe:"est un");
List<ConceptInformatique> reponses1 = new ArrayList<>(conceptsMarques1); // Copie des réponses

// Afficher la réponse à la première question
System.out.println("Réponse à la question informatique \"" + question1 + "\" :");
if (reponses1.isEmpty()) {
    System.out.println(x:"Aucune réponse trouvée pour la question. Manque de connaissances.");
} else {
    for (ConceptInformatique concept : reponses1) {
        System.out.println("- " + concept.nom);
    }
}
```

Figure 43: Premiere Question

Réponse :

Python, Java, C++, Ruby

```
Réponse à la question informatique "Quels sont les langage de programmation ?" :
- C++
- Python
- Java
- Ruby
```

Figure 44: Réponse a la première question

Visualisation

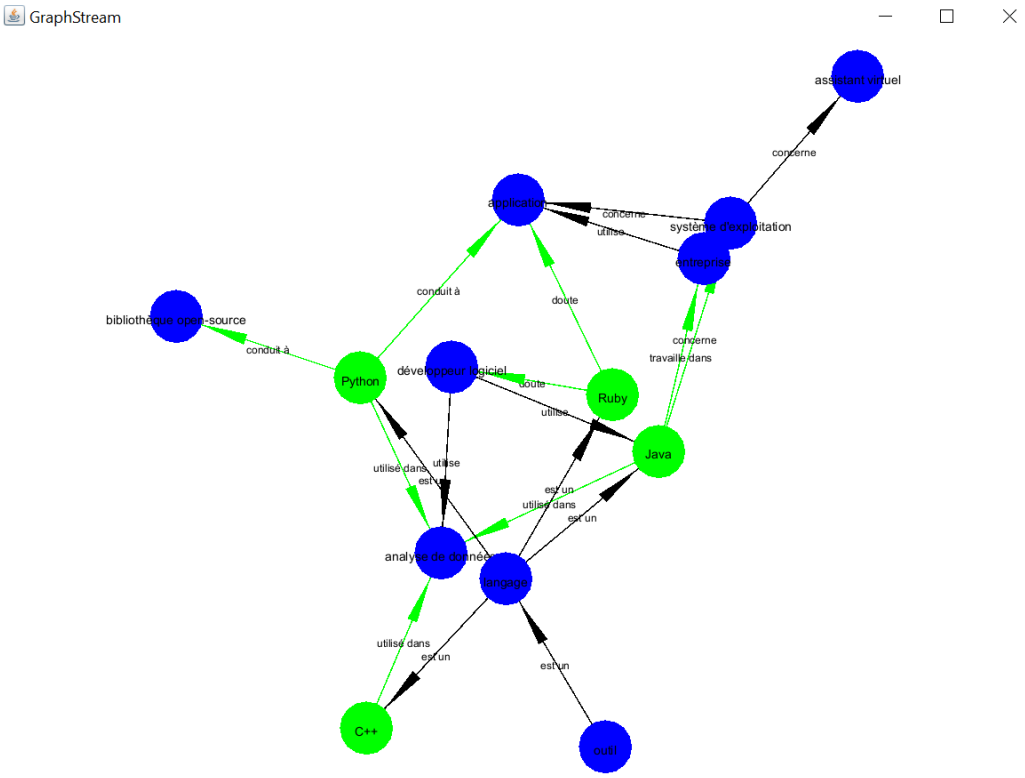

Figure 45: Réponse à la première question

Question 2 : Qu'est-ce que Python conduit à ?

Réponse

bibliothèque open-source

application

```
Réponse à la question informatique "Qu'est-ce que Python conduit à ?" :
- bibliothèque open-source
- application
```

Figure 46: Réponse a la deuxième question

Visualisation

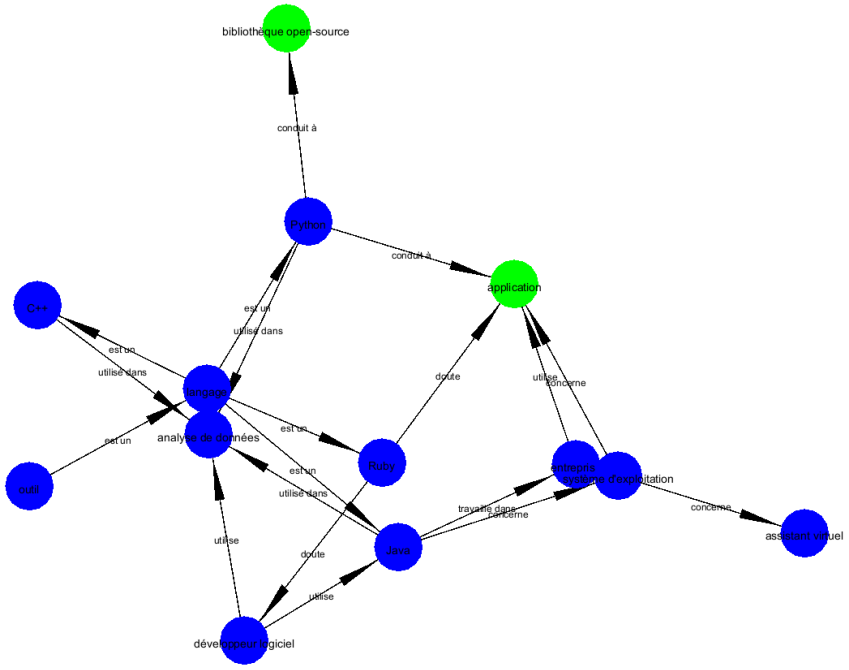

Figure 47 : Réponse a la deuxième question

Question 3 : Qu'est-ce que Python est utilisé dans ?

Réponse :

Analyse de données

```
Réponse à la question informatique "Qu'est-ce que Python est utilisé dans ?" :
- analyse de données
```

Visualisation

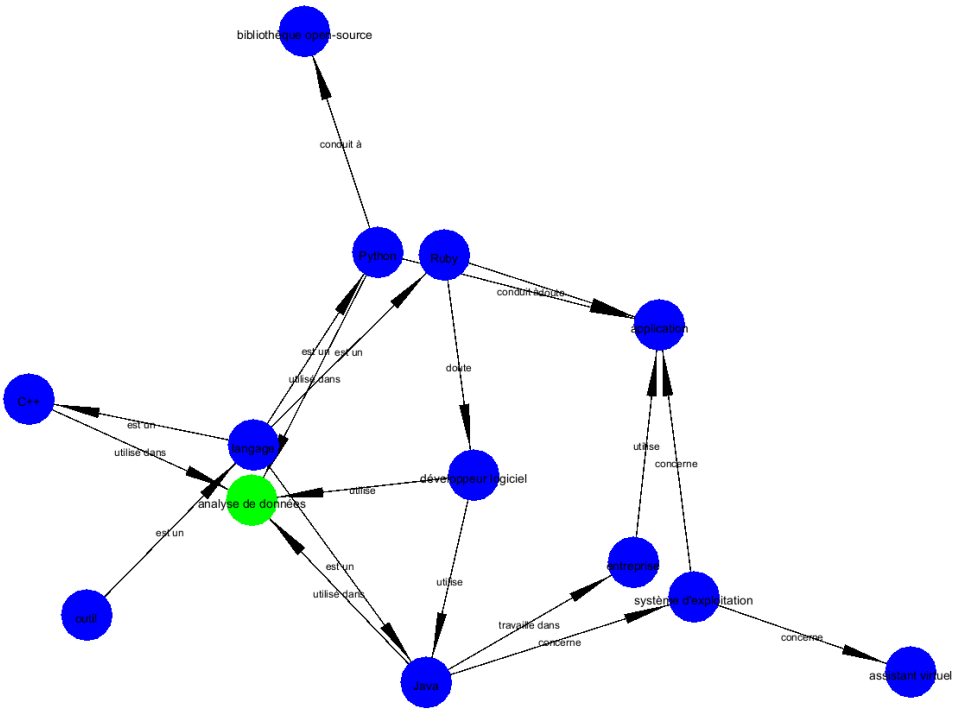

Figure 48: Réponse a la troisième question

Question 4 : Qu'est-ce que Python utilise?

Réponse :

Aucune en raison de manque de connaissances.

```
Réponse à la question informatique "Qu'est-ce que Python utilise ?" :
Aucune réponse trouvée pour la question. Manque de connaissances.
```

Figure 49 : cas de non réponse

Conclusion

Ce TP a permis de démontrer l'utilité des réseaux sémantiques pour modéliser des connaissances et répondre à des questions complexes. En utilisant GraphStream, nous avons construit un réseau sémantique pour des concepts informatiques et répondu à plusieurs questions en propagant des marqueurs à travers le réseau. Les résultats obtenus montrent que les réseaux sémantiques sont des outils puissants pour la modélisation et l'analyse des relations entre concepts.