January 12, 2023

SD-213

Cognitive Approach to Natural Language Processing

Micro-study

teaching.dessalles.fr/CANLP

Name: Rémy Tang

Using YAGO to find coincidences and contradictions and formulating them

Abstract

Notre objectif initial était d'extraire des connaissances de la base de connaissances YAGO et d'identifier les coïncidences et les contradictions dans les données, pour ensuite les présenter en utilisant le langage naturel.

Le jeu de données que nous avons choisi d'extraire de Yago est un jeu de données lié aux acteurs et à leur entourage. Cela a abouti sur la création de + de 430 000 prédicats.

Ensuite nous nous sommes répartis 2 idées :

- Eliott utilise du pattern matching afin de créer différentes anecdotes sur les acteurs
- Rémy utilise CAN pour combler des trous relationnels de la base de données

Problem

Le but de ma partie est d'utiliser la procédure CAN (Conflit-Abduction-Négation) sur une portion de la base de connaissance de YAGO pour identifier des informations manquantes ou d'éventuelles incohérences entre les données.

Une approche avec CAN n'est pas strictement nécessaire pour résoudre ces deux problèmes si on les prend de manière isolée. Cependant, il suffit d'une mise en situation simple, entre un administrateur de base de données et une machine par exemple, pour se convaincre de sa pertinence. Cet exemple sera donné plus bas sous la forme d'un dialogue typique illustrant CAN (analogue à ceux que nous avons vus en cours).

Method

Dans ce projet, la procédure CAN repose sur une partie de la base de données d'acteurs extraite et convertie en prédicats Prolog binaires (predicat(X,Y)).

Les relations extraites sont :

- children(X,Y) : Y est l'enfant de X
- parent(X,Y) : Y est le parent de X
- spouse(X,Y) : Y est l'époux.se de X.

On peut noter que la relation spouse(•,•) est symétrique, et qu'il s'agira par conséquent de faire attention lors de l'implémentation en prolog de bien vérifier les deux sens. Dans le code, on utilise le "ou inclusif" (;) pour vérifier les deux cas.

On remarque aussi que children(X,Y) est équivalent à parent(Y,X).

À partir de ces trois relations, nous pouvons identifier

des données manquantes :

children(X,Y) présent mais pas parent(Y,X)

parent(X,Y) présent mais pas children(Y,X)

une incohérence (artificielle dans la mesure où l'on n'a pas de dates associées aux spouses, mais à valeur illustrative pour le projet) :

spouse(X,Y) et spouse(X,Z) avec Y\=Z (deux spouse associés à une même personne)

children(parent1, enfant) et children(parent2, enfant) mais pas spouse(parent1, parent2) (enfant né d'un couple qui n'est pas défini en tant que spouse)

parent(enfant, parent1) et parent(enfant, parent2) mais pas spouse(parent1, parent2) (deux parents qui ont un enfant mais qui ne sont pas dans un prédicat spouse)

le cas où un seul parent est connu (parent(X,Y) et children(Y,X)) est laissé de côté

Le code joint à ce rapport est fortement inspiré du code fourni dans le TP4. Il a dû être compris puis adapté à la situation (gestion d'une base de données) et aux relations entre les entités (ici des relations familiales entre des personnes).

La partie la plus chronophage a été de bien comprendre le programme pour pouvoir bien poser le problème, puis de déboguer les problèmes d'implémentation (plutôt difficiles à comprendre en Prolog).

Dans la suite du rapport, nous détaillons la manière dont nous avons compris le programme puis commentons le fonctionnement haut niveau du programme, ceci afin de faire paraître les motivations qui justifient l'implémentation réalisée.

La première contrainte du projet est que nous travaillons sur des relations, relations qui nous permettent de transmettre des arguments entre prédicats (nécessaire pour exploiter la base de donnée Prolog). Cependant, le TP que nous avons réalisé en cours portant sur la peinture d'une porte ne faisait pas figurer de relations dans la procédure CAN. Fort heureusement, le code fourni contenait rel_projector.pl, qui contient les clauses définissant la situation de CAN avec des relations.

Le principal challenge alors était de traduire les lignes de la base de connaissances en utilisant les prédicats mis à disposition dans l'implémentation du CAN telle qu'elle était proposée dans le TP.

De la BDD au CAN

Pour effectuer ce parallèle, les fonctions assert_init/1, assert_spouse/1, assert_parent/1 et assert_children/1 ont été définies pour pouvoir déclarer les initial_situation/1 convenablement à partir des clauses.

Dans le projet, nous avons choisi de travailler à partir de toutes les lignes contenant 'trump' pour simplifier une partie du débogage.

Par ailleurs, ces clauses sont mises par défaut pour tester le programme sur les conflits de petite taille :

```
parent(donald_trump,mary_anne_macleod_trump).
parent(donald_trump,fred_trump).
parent(barron_trump,melania_trump).
parent(barron_trump,donald_trump).
children(fred_trump,donald_trump).
spouse(marla_maples,donald_trump).
spouse(melania_trump,donald_trump).
```

Ces clauses suffisent à obtenir les situations de conflit (1.a), (2.a) et (2.c), le tout simplement à partir de quelques données centrées sur Donald Trump.

Étant donné que dans le programme par défaut, les clauses étaient écrites directement là où elles avaient lieu d'être, il a fallu paramétriser l'assertion des conditions initiales. Un point

crucial a été d'ajouter :- dynamic(initial_situation /1). à rel_knowl.pl pour pouvoir réaliser les assertions après compilation, ce qui n'était pas le cas avant.

CAN

Pour résoudre les situations de conflit susmentionnées, il est nécessaire de définir des règles sous forme d'implication ainsi que les actions qui permettent de les résoudre.

Les règles (lois de la base de données) découlent de la définition des conflits et leur assertion est également paramétrée pour pouvoir être générée à partir de la base de données (ou, de manière équivalente, à partir des situations initiales).

En plus de ces règles, on génère également des conflits en associant des préférences négatives à chacune des règles induites par une certaine combinaison de situations. De même que précédemment, il a été nécessaire de définir :- dynamic(preference /2). pour réaliser les assertions à partir des données.

Les actions agissent directement sur un des éléments générateurs d'un conflit, de sorte à ce que la réalisation d'une action supprime le conflit. Leur conséquence est définie avec < - - - tandis que les prérequis (triviaux) sont définis avec < = = =.

Enfin, pour que toutes les règles et conditions initiales soient déclarées en tapant go., il a été nécessaire d'ajouter une étape init/0 à w_init_world/0 dans rel_world.pl.

Situation

Dans l'idée, la situation illustrée par le CAN est analogue au dialogue suivant :

P1 : Donald Trump a un enfant avec Marla alors que Marla n'est pas son épouse.

R1 : Compris, j'ajoute Marla Trump comme épouse de Donald Trump.

P2 : Donald Trump a maintenant deux épouses (Marla Trump et Melania Trump) dans ma base de données, cela ne devrait pas être le cas, parce que que la polygamie est tout au plus contentieuse aux États-Unis.

R2 : Il doit s'être remarié alors. Je mets à jour la base pour indiquer qu'il s'est remarrié à Melania Trump.

P3 : Au fait, un enfant est manquant dans la base alors qu'il existe dans la relation parent.

R3 : J'ajoute la relation enfant manquante, …

Results

Le programme implémenté identifie les sources de conflits et prend les décisions qui amènent à leur résolution, cependant, le programme semble afficher les mauvais conflits car ils sont définis deux fois (symétrie des relations).

Discussion

L'implémentation proposée pose quelques limites plus ou moins contraignantes. Tout d'abord, il est nécessaire d'appuyer à de nombreuses reprises sur ';' pour obtenir le prochain conflit, la raison étant inconnue (parfois plus de 20 ; sont nécessaires).

De plus, je n'ai pas su utiliser les clauses default/0 pour appuyer le CAN, ce qui aurait pu être utile (une ébauche avec l'idée de polygamie est glissée dans le code).

Aussi, l'implémentation génère des doubles conflits qui semblent fausser l'affichage du conflit à résoudre.

De manière générale, l'implémentation du CAN s'est révélée beaucoup plus ardue que prévue, en raison d'abord du fait que le code fourni est déjà assez complexe, en plus du fait que la création de l'interface entre la base de données et le CAN nécessite de bien poser les choses.

Bibliography

SD213, SD206.