Programming Assignment: Hierarchical Reasoning

November 18, 2021

1 Introduction

In this programming assignment, we are going to implement two reasoning approaches, i.e. forward and backward chaining, for hierarchical reasoning. Hierarchical reasoning is a very simple reasoning task that infers the hierarchy of concepts.

2 Background

2.1 Ontologies

An ontology formally describes concepts, properties, and their relations, within a certain domain. One can define special relation between concepts when there is a hierarchical dependency between concepts, i.e. a concept is a *subclass* of another concept. For example, the concept of *Student* would be a *subclass* of the concept *Person*. An ontology consists of two parts: a Terminological Box (TBox) and Assertion Box (ABox). The TBox defines the *schema* of the ontology and the ABox defines the *assertions* according to the schema. You can compare this to classes and objects in Object Oriented Programming (OOP). The TBox used to state that a *Student* is a subclass of *Person* would contain the axiom *:Student rdfs:subClassOf :Person.* Not that rdfs is a *Prefix* and rdfs:subClassOf is short for *http://www.w3.org/2000/01/rdf-schema#subClassOf*.

The ABox can describe that an instance :joe is an instance of the concept Student: *:joe rdf:type :Student.*

Note that any relations between concepts can be defined, for example *:Student :followsCourse :Course* might indicate that *Students* follow *Courses*. However, we will focus here on the subClassOf relations, as they allow to define hierarchies of concepts.

2.2 SPARQL

SPARQL Protocol and RDF Query Language (SPARQL) is the default query language for retrieving data from knowledge graphs or ontologies. We will focus

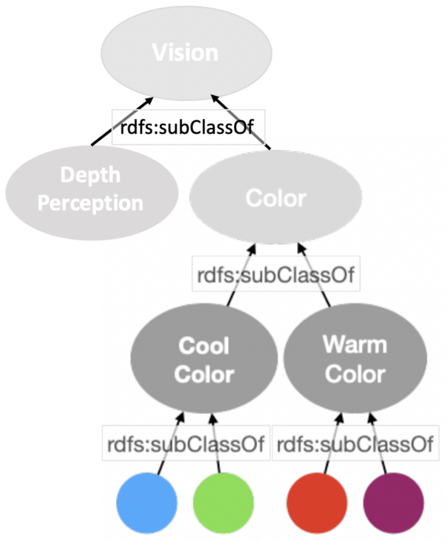

Figure 1: Graphical representation of the Color Ontology.

on very simple queries, i.e. queries consisting of a single *Triple Pattern* (TP) that retrieve instances of a certain concept. A TP to retrieve all students can be written as *?student a :Student*, where *a* is short for rdf:type. A SPARQL query consisting of this simple TP would look like this:

```
PREFIX : <http://persons.org/persons#>
SELECT ?student
WHERE {
 ?student a :Student .
 }
```

Listing 1: An example query consisting of a single Triple Pattern that looks for all Students

?student is a variable and will be bound to the data it matches when evaluating the query. In our case, *?student* will be bound to *:joe* after query evaluation. The *SELECT* statement is a *projection* and allows us to specify which bindings that need to shown in the results. The *PREFIX* statement allows to defined different prefixes, allowing to shorten the query.

2.3 Reasoning techniques

To perform hierarchical reasoning, we can use two reasoning techniques: forward chaining and backward chaining.

2.3.1 Forward Chaining

Forward chaining is the procedure of executing all reasoning steps in order to infer all possible derivations on the TBox. This technique is often used to *materialize* the ontology, such that the reasoning step is executed once and multiple queries can be evaluated without any additional reasoning steps.

Let's say we have the following rules in our ontology TBox, as can be extracted from the Color Ontology as visualized in Figure 1:

(R_1) :Blue *rdfs:subClassOf* :CoolColor

(R_2) :Green *rdfs:subClassOf* :CoolColor

(R_3) :Red *rdfs:subClassOf* :WarmColor

(R_4) :Brown *rdfs:subClassOf* :WarmColor

(R_5) :CoolColor *rdfs:subClassOf* :Color

(R_6) :WarmColor *rdfs:subClassOf* :Color

(R_7) :Color *rdfs:subClassOf* :Vision

(R_8) :DepthPerception *rdfs:subClassOf* :Vision

When we have the following triple in our ABox *(:blue*$_1$*, a, :Blue)*, forward chaining would produce the following reasoning steps. Based on a hierarchical reasoning rule that states that **if** A *rdfs:subClassOf* B **and** a *rdf:type* A, **then** a *rdf:type* B[1], we can infer the following steps:

(I_1) :blue$_1$ *rdf:type* :CoolColor (based on rule R$_1$ and input triple)

(I_2) :blue$_1$ *rdf:type* :Color (based on rule R$_5$ and result of I$_1$)

(I_3) :blue$_1$ *rdf:type* :Vision (based on rule R$_7$ and result of I$_2$)

The idea of *materialization* is that we can now execute a query, for example, to fetch all colors, without needing any intermediate reasoning steps. This is necessary, because the statement *(:blue*$_1$*, a, :Color)* is not defined. Only that is of the color *Blue*. We thus need to consult the hierarchy as defined in Figure 1 to infer that *:blue*$_1$ is indeed an instance of *:Color*.

```
PREFIX : <http://colors.org/colors#>
SELECT ?c
WHERE {
 ?c a :Color .
 }
```

Listing 2: An example query based on the Ontology hierarchy

When we have a query that looks for all Colors, as in Listing 2, the query engine can find the result $blue_1, a, Color$ as it has been added to the ontology through the forward chaining reasoning process.

[1]With A and B concepts defined in the ontology TBox.

2.3.2 Backward Chaining

Backward chaining or query rewriting is the process of exploiting the ontology TBox to find the data that is being queried. The advantage of this technique is that there is no additional memory or storage required to store the additional inferred statement as with Forward Chaining. The drawback is that the reasoning is performed at query time, thus lowering performance when multiple queries need to be evaluated.

When performing backward chaining or query rewriting for a query that looks for all Color, using the color ontology, we will end up looking for all possible subclasses of Color in the hierarchy. Backward chaining uses a top-down approach. Lets say we want to find all instances of the concept *:Color* (*?c a :Color*), then we can rewrite the query using the following steps:

(I_1) ?c a :CoolColor (based on rule R_5 and input query)

(I_2) ?c a :WarmColor (based on rule R_6 and result of I_1)

(I_3) ?c a :Red (based on rule R_3 and result of I_2)

(I_4) ?c a :Brown (based on rule R_4 and result of I_2)

(I_5) ?c a :Blue (based on rule R_1 and result of I_1)

I_5 would match the input triple *(:blue$_1$, a, :Blue)*. Note that when rewriting this process inside a query, we need to inject all possible options, resulting in many UNIONs. Each union can be evaluated independently and can be seen as an OR operator. Backward chaining would stop at I_5 as it matches the input triple, for query rewriting we need to evaluate all possible options, thus resulting in an additional step:

(I_6) ?c a :Green (based on rule R_2 and result of I_1)

It is thus important when performing query rewriting, that the whole hierarchy is present in the rewritten query. Also, note that we only started from the queried concept *:Color*, even though there is a parent class *:Vision*. As *:Vision* is not a subclass of *:Color* it can be ignored here.

Using query rewriting, this would result in the following SPARQL query:

```
PREFIX : <http://colors.org/colors#>
SELECT ?c
WHERE {
 {?c a :Color .}
UNION {?c a color:CoolColor}
UNION {?c a color:WarmColor}
UNION {?c a color:Blue}
UNION {?c a color:Green}
UNION {?c a color:Red}
UNION {?c a color:Brown}
 }
```

```
}
```

Listing 3: An example query rewriting based on the Ontology hierarchy

3 The Assignment

3.1 Assignment Setup

For this assignment, we will use Python 3.x to write a simple hierarchical reasoner. We will use the rdflib[2] package to load the data and perform the query evaluation using SPARQL. The assignment will focus specifically on performing the reasoning steps.

The starting code is available on Github: `https://github.com/pbonte/ReasoningAssignment`. Once you are done with the assignment, you need to create a *pull request*[3] to submit your solution. More information on how to use git can be found here[4].

In *main.py* you will find the start code for this assignment. There is a *QueryEngine* class present that takes an ABox and TBox (already provided) and allows to execute a certain query. The goal of the assignment is to implement the *ForwardChaining* and *BackwardChaining* classes. In order to make sure that they comply with the same interface provided by the *QueryEngine*, you need to make use of polymorphism as typically used in OOP.

3.2 The Data

In this assignment, we will use a hierarchy of *Creative Works*, consiting of ArtWork, Software, WrittenWork, etc. You can investigate the data yourself in *data/ontology.ttl*. Figure 2 show the complete hierarchy. The TBox thus consits only of the depicted hierarchy, consting of statments such as *:Artwork rdfs:subClassOf :CreativeWork*. The ABox consist of the sole statement *:lightNovel1 rdf:type :LightNovel*. We will be looking for all *CreativeWork*s through the query:

```
SELECT ?work
WHERE {
?work a <http://www.dbpedia.org/CreativeWork>.
}
```

It is the goal of the reasoning process to make sure we have matches for this query.

[2]`https://rdflib.readthedocs.io/en/stable/`

[3]`https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/creating-a-pull-request`

[4]`https://docs.github.com/en/get-started/quickstart`

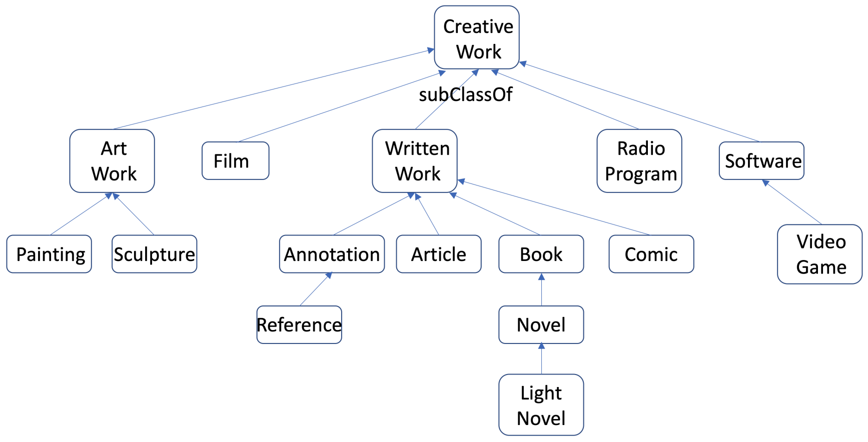

Figure 2: Graphical representation of the Creative Work Hierarchy.

3.3 Assignment 1: the forward chaining reasoner

In the first assignment, the forward reasoner needs to be implemented. This can be done in the *forwardreasoner.py* file. Make sure you do not change the interface as provided by the *QueryEngine*. The goal of this assignment is to *materialize* the ontology, i.e. infer all hierarchy rules over the available statements in the ABox, such that the queries can be answered. To simplify, the TBox only consists of *rdfs:subClassOf* statements. Once you have inferred the additional type assertions based on the defined hierarchy, e.g. (*:lightNovel1 rdf:type :WrittenWork*), you can add these to the ABox list in the *QueryEngine* class. By adding the inferred types to the ABox list, the query evaluation process will be able to find the additional types and resolve the query. Following the hierarchy, the reasoning process should infer something similar to the following steps:

1. :lightNovel1, a , :Novel
2. :lightNovel1, a, :Book
3. :lightNovel1, a, :WrittenWork
4. :lightNovel1, a, :CreativeWork

3.4 Assignment 2: the backward chaining reasoner through query rewritting

In this assignment, we will implement the backward chaining reasoner through the use of query rewriting. This can be done in the *backwardreasoner.py* file. For this case, also make sure not to change the provided interface. Instead of inferred additional ABox statements, in this assignment, you will rewrite the query such that it includes all the subclasses of the concept that is being queried. To simplify, you can assume that the query will only consist of the single TP *?var a <someConcept>.*, e.g. (*?work a <http://www.dbpedia.org/CreativeWork>.*). We will also not make use of any prefixes, thus the full name of the concepts can be used. The result of the rewritting process should result in a query similar to the one below:

```
SELECT ?work
WHERE {
 {?work a <http://www.dbpedia.org/CreativeWork>.}
UNION {?work a <http://www.dbpedia.org/Artwork>.}
UNION {?work a <http://www.dbpedia.org/WrittenWork>.}
UNION {?work a <http://www.dbpedia.org/RadioProgram>.}
UNION {?work a <http://www.dbpedia.org/Software>.}
UNION {?work a <http://www.dbpedia.org/Film>.}
UNION {?work a <http://www.dbpedia.org/VideoGame>.}
UNION {?work a <http://www.dbpedia.org/Article>.}
UNION {?work a <http://www.dbpedia.org/Book>.}
UNION {?work a <http://www.dbpedia.org/Annotation>.}
UNION {?work a <http://www.dbpedia.org/Comic>.}
UNION {?work a <http://www.dbpedia.org/Manga>.}
UNION {?work a <http://www.dbpedia.org/Manhua>.}
UNION {?work a <http://www.dbpedia.org/ComicStrip>.}
UNION {?work a <http://www.dbpedia.org/Reference>.}
UNION {?work a <http://www.dbpedia.org/Novel>.}
UNION {?work a <http://www.dbpedia.org/LightNovel>.}
UNION {?work a <http://www.dbpedia.org/Sculpture>.}
UNION {?work a <http://www.dbpedia.org/Painting>.}
 }
```

4 Submission

To submit your solution, you can create a pull request[5]. As a reminder:

- Make sure to keep the current interface as provided by the *QueryEngine* class.
- Make sure to write *clean* code.
- There are many solutions to both problems. Make sure to take efficiency and readability into account.
- The only package used is rdflib, you can install it using pip (pip -install rdflib).

[5]`https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/creating-a-pull-request`

- You should not change te main.py file, the forwardreasoner.py and backwardreasoner.py need to be implemented.

If something is unclear, or you have any questions, please contact me.