Software to Transform a Knowledge Graph into an Ontology

Hamza ABDOULHOUSSEN
and Killian CRESSANT and Hadrien ROCHU
Students in computer science
Télécom Nancy
Nancy (54000), France
Email: hamza.abdoulhoussen@telecomnancy.eu
killian.cressant@telecomnancy.eu
hadrien.rochu@telecomnancy.eu

Mario LEZOCHE
and Mariano FERREIRONE
CRAN
Nancy (54000), France
Email: mario.lezoche@univ-lorraine.fr
mariano-julian.ferreirone@univ-lorraine.fr

May 15, 2022

***Abstract*—This document presents the study of ontologies and knowledge graphs and aims to propose a system allowing to switch from one to the other.**
Knowledge graphs are a knowledge base that uses a graph-structured data model to integrate interlinked data. Ontologies allow structuring the extracted data in a graph by describing the concepts and the types of relationships between these concepts with properties, rules, and constraints.
As a first step, we study the scope of ontologies by building an ontology by gathering our knowledge of the chess game.
Then, we used `python` to extract the basic information from the files and convert them. In parallel, some visualisations are proposed to facilitate the reading of ontologies and `python` extracted triples.
About ontology constraints, there is no specific format to translate them into knowledge graphs. We thought to propose a format allowing to include constraints, in that way, it is possible to return to the ontology without loss. It did not succeed due to a lack of time. However, an alternative solution is to use the reasoner to convert the ontology and integrate a maximum of triples, especially those inferred from the ontology's semantics.

***Index Terms*—ontology, knowledge graph, reasoner, constraints, *Protégé*, visualisation**

I. Introduction

With the development of the Internet, the virtual home assistant or simply the development of AI, tools that manipulate knowledge are precious. The knowledge graph is one of them. Knowledge graphs are behind every research on the Internet. Ontologies can also manipulate knowledge which is the abstract representation of a part of the world. Both knowledge graphs and ontologies have advantages and disadvantages. The project is to develop an open-source script in python to convert ontologies into knowledge graphs and vice versa.

This ”projet interdisciplinaire de recherche”(PIDR) is a research project conducted as part of the TELECOM Nancy curriculum.

II. State of art

A. Ontology

Ontologies are tools that allows us to represent precisely a domain of knowledge. To be exact, it is an explicit specification of a conceptualization. The word ”Explicit” means that the ontology needs to be clear and detailed (leaving no room for confusion or doubt). the ”Specification” word means that ontology are a detailed description of how to make something. And finally, the ”Conceptualization” word means that ontology is an abstract, simplified view of the world that we wish to represent for some purposes. So finally, we can say that ontologies are a clear and detailed specification of an abstract, a simplified view of the world conceptualization description of how to make something.
Ontology were developed by the W3C around 2002

in a concern of standardization. As a result, in 2004, the language OWL, based on the standard RDF has been created. the syntax is XML. OWL2 is today the most used for ontologies and this is what we used for this project. As an ontology can be seen like a NoSQL database, there is another language created to interrogate this database called SPARQL (very similar to SQL).

B. Reasoner

A reasoner is several algorithms used to understand a special logic. To understand ontologies and particularly some restrictions, there exist few languages of logical description. The most commons today are DL and SHACLE (for Shape Constraint Language). In those last years, SHACLE seems more efficient to treat restrictions for getting graphs from ontologies. This is why we used that one and a reasoner using this language. Each SHACLE constraint in a shape, is defined as a triple $< s, \tau_s, \phi_s >$ where

- s is a name
- τ_s is some entities from the ontology which verified the constraint of the shape
- ϕ_s is the constraint, an expression defined by the grammar :

$$\phi := T | s' | c | \phi_1 \wedge \phi_2 | \neg\phi | \geq_n r.\phi \mid EQ(r_1, r_2)$$

Where T is the boolean truth values, s' is a shape name (often similar), c is a constant, r can be seen as an entity (object), or for keeping generality, a SPARQL property. The $\geq_n r.\phi$ must be understood as : at least n-successors of r in the graph verifying ϕ and finally, EQ stands for equivalent. There is some others grammar possibility to describe SHACLE, but this is the theoretical grammar according to [4]. For example, often we also use the word "$some$" which is related to "$\geq_1 r.\phi$", or "max n" which describes $\leq_n r.\phi$.

Example for the reasoner :
If we take the example of this little pizza ontology (figure 1).

In this example, the classes are in blue and the individuals are in purple.

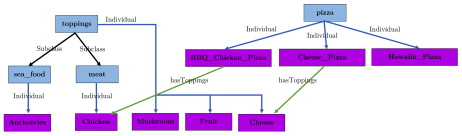

Fig. 1. Little pizza ontology

There is a property :
hasTopping : pizza → toppings

If we add the class meatPizza and we precise that this class is Equivalent To hasToppings some meat

With this constraint, the reasoner can infer that the individuals of meatPizza are the individuals of pizza has meat as a topping.
More formally, the individuals of meatPizza are

$$\begin{aligned} &\bigcup_{e \in IndividualOf(meat)} hasToppings^{-1}(\{e\}) \\ &= hasToppings^{-1}(\{Chicken\}) \\ &= \{BBQ\_Chicken\_Pizza\} \end{aligned} \tag{1}$$

In this case, the reasoner can infer that meatPizza has one individual which is BBQ_Chicken_Pizza

C. Knowledge graph

A knowledge graph (KG) is a new kind of graph born in the last decade. it was created for Google and platforms such as Wikipedia in order to store and compute a huge quantity of data. Like all graphs, a knowledge graph is a finite directed label graph that is a set of triples. In addition, compared to other graphs, this one has a structure that contains "information". Triples are in this shape : (s, p, o) over the oriented graph, where s is a constant, p is a property and o is a constant or a class. Knowledge graphs are very deeply related to ontologies. We can imagine an ontology as the skeleton that holds the knowledge graph. In research papers, ontologies are called the knowledge bases or KBs. in the next pages, we will describe more the link between those

two concepts. See below an example of a knowledge graph (figure 2).

Fig. 2. example of a knowledge graph

In this PIDR, we had to focus on the representation in KG of constraints designed in SHACLE language over an ontology in OWL. This topic is a current field of research, according to that [1], formal semantics via shape assignments have barely been proposed in those last years. The validity of a KG for a constraint ϕ_s is a non-trivial problem. Behind the reasoner, some algorithms try to rewrite an original ontology through what is called a "shape assignment function" to make a KG valid under this shape. For what is left of our work, we imagined a shape as a representation of an ontology for a given constraint.

III. From ontology to knowledge graph

A. The Chess Ontology

We have created a chess ontology in order to study how to generate a knowledge graph from an ontology. For those who don't know chess, you can learn it on Wikipedia.
The purpose of our ontology is to represent a chess match. Our chess ontology has five main classes which are: "Board", "Match", "Pieces", "Players" and "Rules". The "Board" class contains the subclasses "Cell", "File" and "Rank". The "Cell" subclass contains the sixty-four cells of the board. Those individuals are linked with two properties to only one File's individual and only one Rank's individual. For example, the Cell individual c3 is related to the File individual c and related to the Rank individual 3. They are also related to a string value. So the Cell c3 is related to the string value "c3".
Why do we add string value? The engine does not understand the name of subclasses or individuals. So if we want to link the individual to value with python, for instance, we have to create a data value for each individual.
The "Pieces" class is one of the major classes. The chess game can be seen as the evolution of the pieces on the board. Pieces is often the range of properties so the class was or will be described in the other description of classes. "Players" contains two subclasses called "AI" and "Person". They are the entity that will play the match. A match is related to two players by properties, one for each side: the black side and the white side.
"Rules" class contains the subclasses "Movements" and "Win_conditions". A movement is well represented by a "Match_moves" linked to the moving piece and the cell it goes except for the specific movements "En_passant", "Castle" and "Promotion". So Match_moves is both related to the theoric movement and the real movement. win_conditions is an empty class. We have met some problems to verify the conditions because ontologies are a static representation of knowledge. We currently don't know if it is possible to implement the winning conditions into the ontology but if it is, we would like to implement it.

The main relations of the ontology are the following.

- Match *Where_black_player_is* Player
- Match *Where_white_player_is* Player
- Match_moves *To_cell* cell
- Match_moves *Moving_Piece* Pieces
- Match_moves *Next_move* Match_moves
- Pieces *Start_cell_is* Cell

A match is played by two players. One plays the white pieces and the other plays the black pieces. A match is a set of movements alternatively of a white piece and a black piece from a starting situation to the end of the match. A movement, in our chess ontology, is basically a piece which is landing on a specific cell of the board.
In order to be inspired for our ontology, we compared it with another chess ontology found

on the Internet. The other chess ontology [2] was mostly the representation of an already played match and chess competition.
Our ontology was created with the free and open source software *Protégé* which is widely used to create ontology.
Ontologies are stored in `OWL` format. This file format can be open as a text file or with specific software like *Protégé*. Opened as a text file, the format looks like in figure 3 and 4.

```
    <!-- http://www.semanticweb.org/killi/ontologies/2022/0/Chess_Ontology#Bishops -->

    <owl:Class rdf:about="http://www.semanticweb.org/killi/ontologies/2022/0/Chess_Ontology#Bishops">
        <rdfs:subClassOf rdf:resource="http://www.semanticweb.org/killi/ontologies/2022/0/Chess_Ontology#Pieces"/>
        <owl:disjointWith rdf:resource="http://www.semanticweb.org/killi/ontologies/2022/0/Chess_Ontology#Kings"/>
        <owl:disjointWith rdf:resource="http://www.semanticweb.org/killi/ontologies/2022/0/Chess_Ontology#Knights"/>
        <owl:disjointWith rdf:resource="http://www.semanticweb.org/killi/ontologies/2022/0/Chess_Ontology#Pawns"/>
        <owl:disjointWith rdf:resource="http://www.semanticweb.org/killi/ontologies/2022/0/Chess_Ontology#Queens"/>
        <owl:disjointWith rdf:resource="http://www.semanticweb.org/killi/ontologies/2022/0/Chess_Ontology#Rooks"/>
        <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string">This is a piece. There are two bishops of
each side (black and white). See also Bishop&apos;s movement.</rdfs:comment>
    </owl:Class>
```

Fig. 3. Bishop class in `OWL` file

```
    <!-- http://www.semanticweb.org/killi/ontologies/2022/0/chess-ontology-2#Moving_Piece -->

    <owl:ObjectProperty rdf:about="http://www.semanticweb.org/killi/ontologies/2022/0/chess-ontology-2#Moving_Piece">
        <rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topObjectProperty"/>
        <rdfs:domain rdf:resource="http://www.semanticweb.org/killi/ontologies/2022/0/chess-ontology-2#Match_moves"/>
        <rdfs:range rdf:resource="http://www.semanticweb.org/killi/ontologies/2022/0/Chess_Ontology#Pieces"/>
        <rdfs:comment>Hadrien : It is not a functional property because the caste movement &quot;switch and move&quot;
both the King and the Rook</rdfs:comment>
    </owl:ObjectProperty>
```

Fig. 4. Moving_Piece property in `OWL` file

When the `OWL` file is open with *Protégé*, the ontology looks like in figure 5.

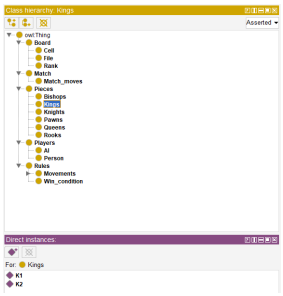

Fig. 5. *Protégé* structure

B. *Visualisation in* Protégé

In order to have visualisation in *Protégé*, it is possible to install plugins. We used the plugins `OWLViz` and `VOWL`.

1) `OWLViz`: `OWLViz` shows the taxonomy of the ontology and subclass relations in the ontology. Besides, it allows seeing inferred information get from the reasoner. However, we can not see the individuals and the properties (figure 6).

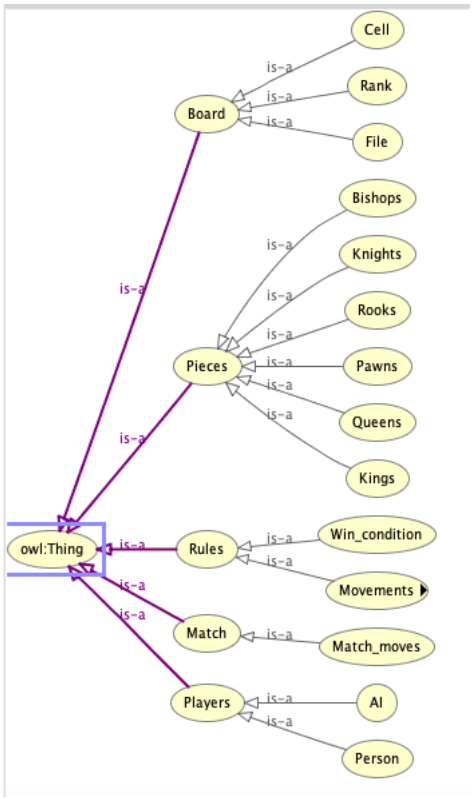

Fig. 6. `OWLViz` visualisation

2) `VOWL`: `VOWL` also shows classes and subclass relations and properties. We do not have the individuals.
For this visualisation, the classes are unorganised (figure 7) at the beginning but it is possible to move them and organise the element as wanted (figure 8).

C. *Ontology in python*

One of the purposes of this PIDR was to make `python` the only necessary language to manipulate ontologies and transform them into knowledge graphs. As `python` contains many visualisation tools, graph runner, text writer, and plugins to read

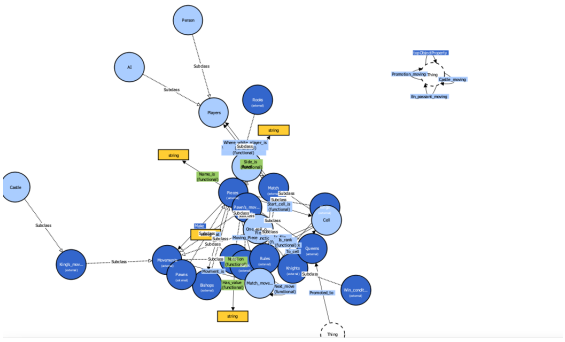

Fig. 7. Unorganised VOWL visualisation

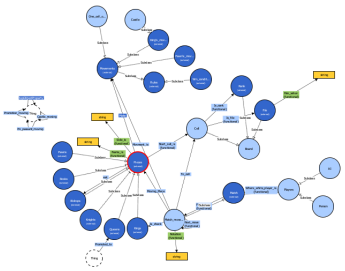

Fig. 8. VOWL visualisation

ontologies, it was the perfect language for this.
Our first usage of python was to write in the RDF file of the ontology to add new matches. For this, we just need to add in the right place some individuals and classes in the chess ontology. Thanks to python, we can now add more easily a match than on *Protégé*.
The OWL file can also be read in python with a library called *owlready2*. It allows you to obtain every element of the ontology: classes, subclasses, properties, etc... This library can also be used to execute SPARQL queries on the ontology. We use *owlready2* to extract information from the OWL file in order to create some triples of a knowledge graph.

D. Conversion into python structure

From ontologies, in order to understand and use knowledge in python, we have made a small graph structure (figure 9) :

- **graph**
  contains the name of the graph, all the classes and all the individuals
- **classe**
  contains for a class, his name and iri, all the subclasses and the individuals of the class. (the individuals of the subclasses are not in the parent class individuals)
- **individual**
  contains the name, iri and parent class of the individual

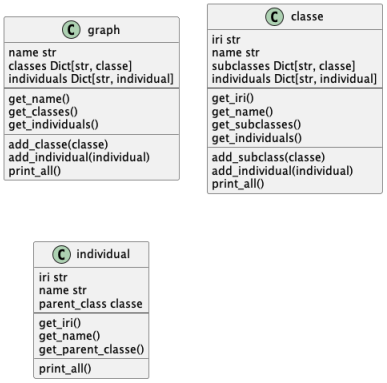

Fig. 9. Graph structure

Figure 10 is an example of the Chess Ontology loaded in the structure, it shows the same number of individuals than the ontology in *Protégé*.

This graph has practical value if you want to use the knowledge in ontologies to make a python algorithm. For example, use the chess ontology to get the structure of some matches and analyze them

Fig. 10. comparison between the number of individuals in the structure and in *Protégé*

with algorithms.
Also, thanks to this structure, we can compare classes and subclasses between two ontologies. We used this for example before and after launching the reasoner. For the rest, we did not used the complete structure. The list of triples was enough.

E. Conversion into RDF

The next step was to make a script to convert an ontology from OWL to an exclusive RDF triple file. Before extracting any information from the OWL file, the script can run the reasoner. The reasoner is an integrated engine that finds new knowledge from the current ontology, here we are using the HermiT reasoner. The ontology might be more complete and the script allows to save the new ontology in another file.
Then, the script extracts all the classes, subclasses, individuals and object properties from the OWL files. We used SPARQL query for each kind of relation (class to subclass, individual of class, from domain linked by object properties to range).
The results get on the chess ontology are in figure 11, 12 and 13.
To finish, we used a data model of RDF triples [3] to make a template (figure 14). The script creates an output with xml extension, each triple is converted into RDF triple format based on the template and written in the output.

```
================================================================
================== PROPERTIES TRIPLES ==================
================================================================
('Chess_Ontology.Match', 'Chess_Ontology.First_move', 'Chess_Ontology.Match_moves')
('Chess_Ontology.Match_moves', 'Chess_Ontology.Is_check', 'Chess_Ontology.Kings')
('Chess_Ontology.Cell', 'Chess_Ontology.Is_file', 'Chess_Ontology.File')
('Chess_Ontology.Cell', 'Chess_Ontology.Is_rank', 'Chess_Ontology.Rank')
('Chess_Ontology.Match_moves', 'Chess_Ontology.Moving_Piece',
'Chess_Ontology.Pieces')
('Chess_Ontology.Match_moves', 'Chess_Ontology.Next_move',
'Chess_Ontology.Match_moves')
('Chess_Ontology.Promotion', 'Chess_Ontology.Promoted_to', 'Chess_Ontology.Bishops')
('Chess_Ontology.Promotion', 'Chess_Ontology.Promoted_to', 'Chess_Ontology.Knights')
('Chess_Ontology.Promotion', 'Chess_Ontology.Promoted_to', 'Chess_Ontology.Queens')
('Chess_Ontology.Promotion', 'Chess_Ontology.Promoted_to', 'Chess_Ontology.Rooks')
('Chess_Ontology.Pieces', 'Chess_Ontology.Start_cell_is', 'Chess_Ontology.Cell')
('Chess_Ontology.Match_moves', 'Chess_Ontology.To_cell', 'Chess_Ontology.Cell')
('Chess_Ontology.Match', 'Chess_Ontology.Where_black_player_is',
'Chess_Ontology.Players')
('Chess_Ontology.Match', 'Chess_Ontology.Where_white_player_is',
'Chess_Ontology.Players')
('Chess_Ontology.Pieces', 'Chess_Ontology.eat', 'Chess_Ontology.Bishops')
('Chess_Ontology.Pieces', 'Chess_Ontology.eat', 'Chess_Ontology.Knights')
('Chess_Ontology.Pieces', 'Chess_Ontology.eat', 'Chess_Ontology.Pawns')
('Chess_Ontology.Pieces', 'Chess_Ontology.eat', 'Chess_Ontology.Queens')
('Chess_Ontology.Pieces', 'Chess_Ontology.eat', 'Chess_Ontology.Rooks')
('Chess_Ontology.Pieces', 'Chess_Ontology.Make', 'Chess_Ontology.Movements')
('Chess_Ontology.File', 'Chess_Ontology.Has_value', "<class 'str'>")
('Chess_Ontology.Rank', 'Chess_Ontology.Has_value', "<class 'str'>")
('Chess_Ontology.Cell', 'Chess_Ontology.Name_is', "<class 'str'>")
('Chess_Ontology.Pieces', 'Chess_Ontology.Name_is', "<class 'str'>")
('Chess_Ontology.Match_moves', 'Chess_Ontology.Notation', "<class 'str'>")
('Chess_Ontology.Pieces', 'Chess_Ontology.Side_is', "<class 'str'>")
```

Fig. 11. Triples of properties

```
================================================================
================== INDIVIDUALS TRIPLES ==================
================================================================
('Chess_Ontology.B1', 'individualOf', 'Chess_Ontology.Bishops')
('Chess_Ontology.c1', 'individualOf', 'Chess_Ontology.Cell')
('Chess_Ontology.B2', 'individualOf', 'Chess_Ontology.Bishops')
('Chess_Ontology.f1', 'individualOf', 'Chess_Ontology.Cell')
('Chess_Ontology.B3', 'individualOf', 'Chess_Ontology.Bishops')
('Chess_Ontology.c8', 'individualOf', 'Chess_Ontology.Cell')
('Chess_Ontology.B4', 'individualOf', 'Chess_Ontology.Bishops')
('Chess_Ontology.f8', 'individualOf', 'Chess_Ontology.Cell')
('Chess_Ontology.K1', 'individualOf', 'Chess_Ontology.Kings')
('Chess_Ontology.e1', 'individualOf', 'Chess_Ontology.Cell')
('Chess_Ontology.K2', 'individualOf', 'Chess_Ontology.Kings')
('Chess_Ontology.e8', 'individualOf', 'Chess_Ontology.Cell')
('Chess_Ontology.N1', 'individualOf', 'Chess_Ontology.Knights')
('Chess_Ontology.b1', 'individualOf', 'Chess_Ontology.Cell')
('Chess_Ontology.N2', 'individualOf', 'Chess_Ontology.Knights')
```

Fig. 12. Triples of individuals

F. Constraints issues

We have a problem when we try to express the constraints of the ontology into RDF triple. The ontology can contain constraints such as "some", "not", "only", etc. For example, *pawn promoted_to "Pieces & not(Kings) & not(Pawns)"*. So, a range or a subclass, which is only one element for the OWL file, cannot be transformed into one element in the RDF structure. We have in fact 3 options to treat them. The first one is to change the ontology: we can imagine another one where there are no more constraints but all the logic remains. The second option is to keep all the constraints in a graph but find a way to express those restrictions sometimes with multiple triples. It may imply creating new nodes. The last option is to remove all constraints and keep the best the reasoner finds after the conversion. For moving on, we needed to know if the existence of a rewrite that validate a KG is ensure. If we need to build a bijective function that can transform an ontology to a KG and the opposite, and if we want to use the first option, we need to know if we can always find a solution that satisfies the shape to make a KG. After some research, we figure out that rewriting the ontology to find a valid KG is not ensured. In fact, constraint validation in presence of ontologies is a CO-NP-complete and without constraint it is NP-complete. This means that except if Co-NP=NP, there is no general solution for the first possibility

```
======================================================================
================== SUBCLASS TRIPLES ==================
======================================================================
('Chess_Ontology.AI', 'subClassOf', 'Chess_Ontology.Players')
('Chess_Ontology.Castle', 'subClassOf', "Chess_Ontology.King's_movement")
('Chess_Ontology.Cell', 'subClassOf', 'Chess_Ontology.Board')
('Chess_Ontology.Double_forward', 'subClassOf', "Chess_Ontology.Pawn's_movement")
('Chess_Ontology.Eat', 'subClassOf', "Chess_Ontology.Pawn's_movement")
('Chess_Ontology.En_passant', 'subClassOf', "Chess_Ontology.Pawn's_movement")
('Chess_Ontology.Match_moves', 'subClassOf', 'Chess_Ontology.Match')
('Chess_Ontology.One_cell_one_piece', 'subClassOf', 'Chess_Ontology.Movements')
('Chess_Ontology.One_forward', 'subClassOf', "Chess_Ontology.Pawn's_movement")
('Chess_Ontology.One_square_move', 'subClassOf', "Chess_Ontology.King's_movement")
('Chess_Ontology.Person', 'subClassOf', 'Chess_Ontology.Players')
('Chess_Ontology.Promotion', 'subClassOf', "Chess_Ontology.Pawn's_movement")
("Chess_Ontology.Queen's_movement", 'subClassOf', 'Chess_Ontology.Movements')
('Chess_Ontology.Bishops', 'subClassOf', 'Chess_Ontology.Pieces')
('Chess_Ontology.Kings', 'subClassOf', 'Chess_Ontology.Pieces')
('Chess_Ontology.Knights', 'subClassOf', 'Chess_Ontology.Pieces')
('Chess_Ontology.Movements', 'subClassOf', 'Chess_Ontology.Rules')
('Chess_Ontology.Pawns', 'subClassOf', 'Chess_Ontology.Pieces')
('Chess_Ontology.Queens', 'subClassOf', 'Chess_Ontology.Pieces')
('Chess_Ontology.Rooks', 'subClassOf', 'Chess_Ontology.Pieces')
('Chess_Ontology.Win_condition', 'subClassOf', 'Chess_Ontology.Rules')
("Chess_Ontology.Bishop's_movement", 'subClassOf', 'Chess_Ontology.Movements')
("Chess_Ontology.King's_movement", 'subClassOf', 'Chess_Ontology.Movements')
("Chess_Ontology.Knight's_movement", 'subClassOf', 'Chess_Ontology.Movements')
("Chess_Ontology.Pawn's_movement", 'subClassOf', 'Chess_Ontology.Movements')
("Chess_Ontology.Rook's_movement", 'subClassOf', 'Chess_Ontology.Movements')
```

Fig. 13. Triples of subclasses

```
<rdf:Description rdf:about="${subject}">
    <ex:${predicate}>
        <rdf:Description rdf:about="${object}"/>
    </ex:${predicate}>
</rdf:Description>
```

Fig. 14. Template of RDF

according to [1]. The third solution is basically the same as the first because all those algorithms that transform an ontology without constraints are in the reasoner. So there are not always solutions, the third option is to give up the bijective aspect. We cannot go back to a similar ontology through a KG. The second solution was to incorporate directly the constraints in the KG. In that case, there are no real changes in the ontology. In fact, we still need to launch the reasoner before parsing the file in triples, because we can have a more complete graph. But, we also need to find a way to get all the semantics of the constraint in a KG. We never find any document that treats this problem this way and we finally did not have the time to try it. So the current solution we picked was the third: we do not keep all the constraints and we hope the reasoner is doing well.

G. Flags on the script

To make the script more generic, it uses flags and arguments. We can use flags to input the owl file and insert the output, we can also print the triples, create the standard version or save the reasoner inferences. There is a complete description in the documentation of the repository.

H. Triples representation

Since we are extracting triples from ontologies, it is interesting to have a graph representation of the triples.
`dot` is a graph description language. In `python`, the package *graphviz* allows creating a graph, add nodes and edges using the `dot` language. The code in `graph_viz.py` generates a `gv` file, written in `dot` and it is possible to convert them to `png` using the `dot` command (figure 15). The `bash` script `convert_all_to_png.sh` converts all the files into `png` with the command `./convert_all_to_png.sh`.
However, when the file is too large, the picture is hard to read, and the graph is often stretched. For the largest files, the image is cut (figure 16).

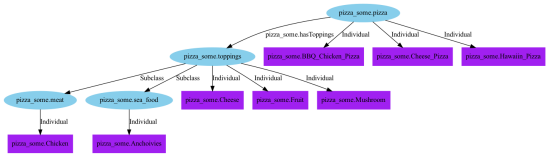

Fig. 15. `Pizza_some` with graph viz

Fig. 16. Large file with graph viz (the image is cut on the edges)

To complete our triples and our visualisations, we also learned to use another tool called *Neo4J*. This is an application (not completely open-source) of a graph database that allows us to parse in triples an `OWL` file of an ontology and then print the correspondent graph.
We used *Neo4J* to compare our triples from python and our representation with a more advanced tool. This is the kind of thing that we can obtain when we import a plugin (open-source) called *neosemantics* (figure 17 and 18).

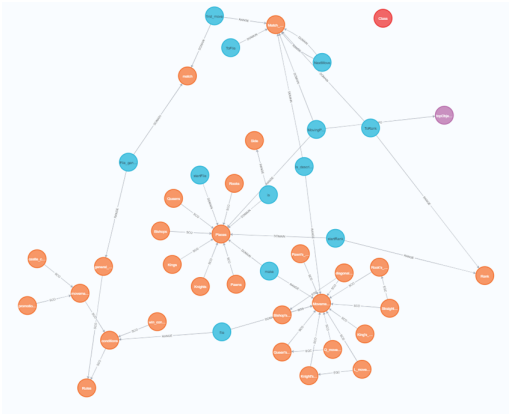

Fig. 17. Global view of our KG in *Neo4J*

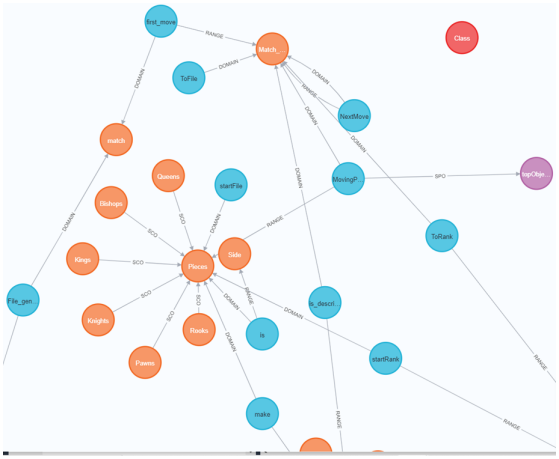

Fig. 18. Zoom on a part of that KG

IV. From knowledge graph to ontology

This is a more experimental part than the previous one. There is a lot of data about how to transform an ontology in a knowledge graph, but there is not much to compute the other way. For this part, we decided to forget all the creation of new nodes made by the reasoner to parse constraints. This means that if the reasoner did not work well, we do not have the possibility to ensure that the ontology is the same after transformation.
Then, we decided to start with smaller ontologies than our chess ontology. In fact, our chess ontology has not been built with enough constraints to be a really interesting test for that.
The idea was to rebuild an `OWL` file from the `RDF` triples file that contains the same information, without constraint. That is the first step necessary to do some tests on what we needed to add in the KG to keep restrictions. Unfortunately, we did not have the time to treat that part.
Below are our results for this part :
We started with the small handmade `pizza_(with)_some` ontology (figure 19). Then we launched the reasoner. As we can see in figure 20, `meatpizza` was inferred as a part of pizza. In the KG, we can see figure 21 that this interpretation is kept, so that the "`some`" structure: `meatpizza` → `hasToppings some meat` is understood and kept in the KG on a semantic level. And finally, we succeeded to go back from the KG, and print the shape figure 22. Here is a good example: the reasoner worked well and we were able to keep all those modifications and so to keep all the semantics in the KG and build a bijection function.

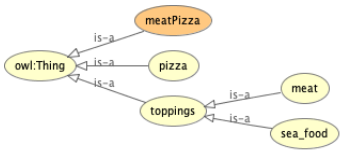

Fig. 19. Ontology before inference

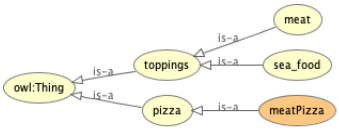

Fig. 20. Ontology after the reasonner

Sometimes, it won't work as well, but it is more complicated to see what is lost. We made the same thing with the complete "`pizza`" ontology, but our representation was too limited and it was much more difficult to see differences in the middle on a large database.

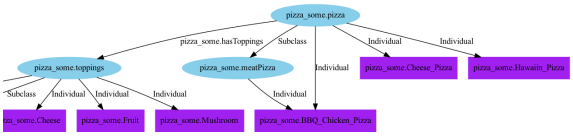

Fig. 21. KG of pizza_some inferred (cut)

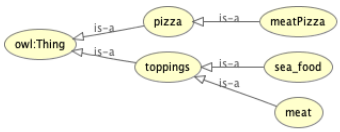

Fig. 22. Ontology from the KG

V. Conclusion

To conclude, we understood what is an ontology, what is a knowledge graph and how we can bound those two. We were able to make some python scripts to pass from one to another, but we still need some improvements. First, we were not able to judge on a ”real” (big) ontology if our work is efficient and good. We have a lack of tests on big ontologies and big KG. But we are able to test every part of the project through little ontologies. We had not enough time to explore one of the major parts of the project: the representation of constraints in knowledge graphs. Instead, we also spend our time understanding what was the range of the reasoner and how we could use it to explore the semantics of an ontology. We also improve our understanding of those notions by making from nothing a chess ontology and some visualisation on `python` and *Neo4J*.

Acknowledgment

The authors would like to thank the PhD Mariano and M. LEZOCHE, for their work and all the time they gave us. For all the research they maid for us. We also would like to thank TELECOM Nancy (Université de Lorraine) for giving us the opportunity to do research work such as this one

References

[1] Ognjen Savkovic, Evgeny Kharlamov, Steffen Lamparter, SHACLE constraint Validation over Ontology-enhanced KGs via Rewriting, December 2018.

[2] Adila Krisnadhi, Pascal Hitzler, Modeling With Ontology Design Patterns: Chess Games As a Worked Example, https://people.cs.ksu.edu/~hitzler/pub2/01-chess-example.pdf

[3] Computer Science and Operations Research Montreal University, Triples in RDF/XML, https://www.iro.umontreal.ca/~lapalme/ForestInsteadOfTheTrees/HTML/ch07s01.html, April 2019

[4] Bart Bogaerts, Maxime Jakubowski, Jan Van den Bussche SHACL: A Description Logic in Disguise, 2021

Annexe

You can find here a git repository containing all the code, documentation, and meeting minutes. **https://github.com/Hamza-ABDOULHOUSSEN/PIDR_knowledge_graph**