Software to Transform a Knowledge Graph into an Ontology

Hamza Abdoulhoussen
and Cressant Killian and Hadrien Rochu
Students in computer engineering
Télécom Nancy
Nancy (54000), France
Email: hamza.abdoulhoussen@telecomnancy.eu
killian.cressant@telecomnancy.eu
hadrien.rochu@telecomnancy.eu

CEST JUSTE UN PLAN AU BROuILLON POUR NOUS

- Summary of the project
- Introduction
- State of the art
  - Ontology
  - Knowledge graph
  - Existing tools to transform a knowledge graph into an ontology
- From ontology to knowledge graph
  - Chess ontology
  - Protege
  - Python script
- From knowledge graph to ontology
  - neo4j
  - python script
- Conclusion
- References

***Abstract*—The abstract goes here.**

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 are one of them. Knowledge graph are behind every research on the Internet. Ontology can also manipulate knowledge which is the abstract representation of a part of the world. Both knowledge graph and ontology have advantages and disadvantages. The project is to develop an open-source script in python to convert ontology into knowledge graph 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

The Ontology are a tool that allow us to represent precisely a domain of knowledge.To be exact, it is an explicit specification of a conceptualisation. The word ”Explicit” means that the ontology need to be clear and in detailed (leaving no room for confusion or doubt). the ”Specification” word means that ontology are a detailed description of how to make something. And finaly, the ”Conceptualization” word means that ontology is an abstract, simplified view of the world that we wish to represent for some purposes. So finaly, we can said that Ontology are a clear and detailed specification of an abstract, simplified view of the world conceptualization description of how to make something. **???is it english Hadrien ??** Ontology are finaly developed by the W3C around 2002 in a concern of standardization. As result, in 2004, the langage OWL, based on the standard RDF is born. the syntaxe is XML. OWL2 is today the most used for the ontology 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. Reasonner

A reasoner is several algorithms used to understand a special logic. To understand the ontology and particularly some restrictions, there exist few language of logic description. The most common today are DL and SHACLE (for Shape Constraint Language). In those last years, SHACLE seams more efficient to treat restrictions for getting a graph from the ontology. This is why we used that one, and a reasonner that 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 entities (object), or for keeping generality, a SPARQL property. The $\geq_n r.\phi$ must be understand as : at least n-successors of r in the graph verifying ϕ and finaly, EQ stand for equivalent. There is some others grammar possibility to describe SHACLE, but this is the theorical grammar. For example, often we also use the word "some" which is related to "$\geq_1 r.\phi$, or "max n" which describe $\leq_n r.\phi$.

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

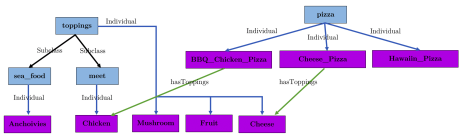


Fig. 1. Little pizza ontology

In this example, the classes are in blue and the individuals are in purple.
There is a property :
hasTopping : pizza $\rightarrow$ toppings

If we add the class meetPizza and we precise that this class is Equivalent To hasToppings some meet

With this constraint, the reasoner can infer that the individuals of meetPizza are the individuals of pizza which have meet as topping.
More formally, the individuals of meetPizza are

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

In this case, the reasoner can infer that meetPizza as 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 platform such as Wikipedia to store and compute a huge quantity of data. As all graph, a knowledge graph is a finite directed label graph that is a set of triples. In addition comparing to other graph, this one have a structure that contain "information".Triple are in 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. A knowledge graph is very deeply related to the ontology. We can imagine an ontology as the skeleton that hold the Knowledge graph. In research paper, ontology are called the Knowledge bases or KBs. in the next pages, we will describe more the link between those two things. See below an example of a knowledge graph.

In this PIDR, we had to focus on the representation in a KG of constraints designed in SHACLE language over an ontology in OWL. This topic is a current field of research, according that a formal semantics via shape assignments has barely been

Fig. 2. example of a knowledge graph

proposed those last years. The validity of a KG for a constraint ϕ_s is a non trivial problem. Behind the reasonner, there is algorithm that try to rewrite the original ontology to through what is called a "shape assignment function" to make a KG valid under this shape. For the rest 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 to only one Rank's individual. For example: the individual cell c3 is related to the individual of File c and related to the individual of Rank 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 a subclass or a individual.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 in the board. The Pieces are often the range of a property so the class was or will be described in the other description of class. "Players" contains two subclasses called "AI" and "Person". They are the entity that will played 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 movment and win_conditions. A movement is well represented by a match_move but not for the specific movements as "en_passant", the castle and the promotion. So the match_move are both related to the theoric movment and the real movment. The win_conditions is an empty class. We have met some problems to verify the conditions because the ontology is a static representation of knowledge. We currently don't know if it is possible to implement the win conditons into the ontology but if it is, we would like to implement it. The main relations 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 *Strat_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 white piece and 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 was more 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 use 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 look like this :

When the owl file is open with `protégé`, the ontology looks like this :

```
<!-- 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

Fig. 5. Protégé structure

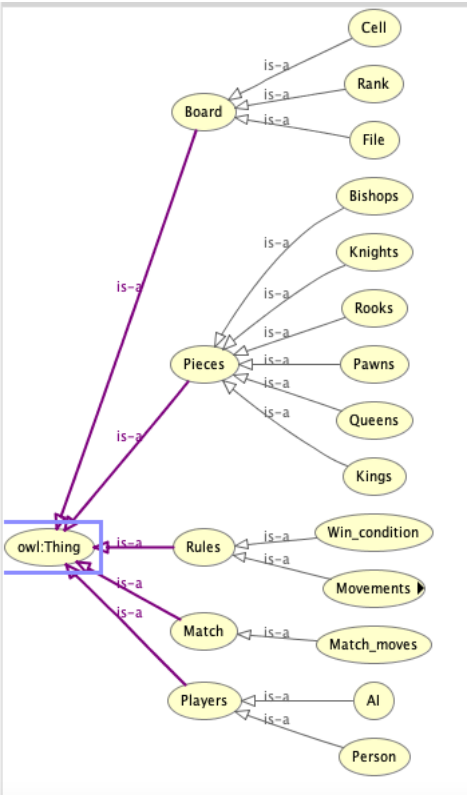


Fig. 6. OWLViz visualisation

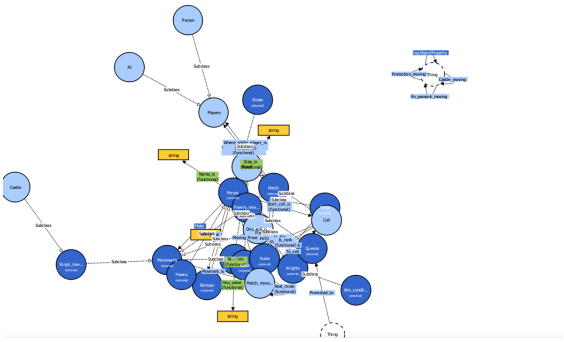


Fig. 7. Unorganised VOWL visualisation

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, the subclasses relation in the ontology. Besides, it allows to see inferred information get from the reasoner. However, we can not see the individuals and the properties.

2) VOWL: VOWL shows also the classes and the subclasses relation and the 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 purpose of this PIDR was to make python the only necessary language to manipulate an ontology and transform it in knowledge graph. As python contains many visualisation tools, graph run, and text writing, and plugins to read ontologies, it was the perfect tool for this.

Our first usage of python was to write in the

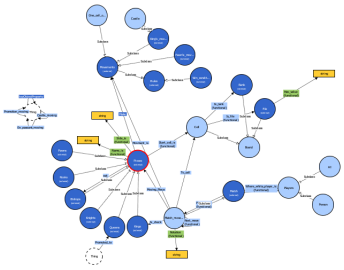


Fig. 8. VOWL visualisation

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 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 elements of the ontology : classes, subclasses, properties, etc... This library can also be used to execute SPARQL query 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 or not in the parent class individuals)
- **individual**
  contains the name, iri and parent class of the individual

The 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.

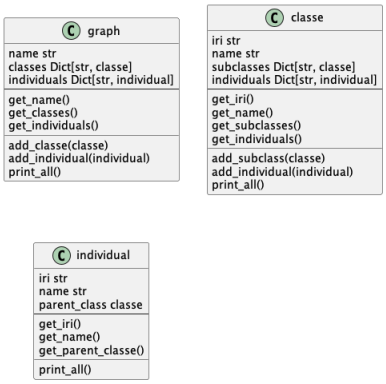


Fig. 9. Graph structure

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

This graph has a 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 analyse them with algorithms.
Also, thanks this structure, we can also compare classes and subclasses between two ontologies. We used this for example before and after launching the reasonner. 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 OWL file, the script can run the reasoner. The reasoner is an integrated engine which find 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 extract 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 create 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

```
======================================================================
================== 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

```
1  <rdf:Description rdf:about="${subject}">
2      <ex:${predicate}>
3          <rdf:Description rdf:about="${object}"/>
4      </ex:${predicate}>
5  </rdf:Description>
```

Fig. 14. Template of RDF

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 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 is no more constraints but all the logic behind remain. The second option is to keep all the constraint in a graph but find a way to express in sometimes multiples triples those restrictions. It implies maybe created new nodes. The last option is to remove all constraint and to keep only the reasonner part do as best as it can. 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 w want to use the first option,

we need to know if we can always find a solution that satisfy the shape to make a KG. After some research, we figure out that rewriting the ontology to find a valid KG is not ensure. In fact, constraint validation in presence of ontologies is a CO-NP-complete and without constraint it is NP-complete. Which means that except if Co-NP=NP, there is no general solution for the first possibility.The third solution is basically the same than the first, because all those algorithms that transform an ontology without constraints are in the reasoner. So there is 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 is no real changes of the ontology. In fact, we still need to launch the reasoner before to parse the file in triples, because we can have a more complete graph. But, we also need to find a way to get all the semantic of the constraint in a KG. We never find any document that treat this problem this way and we finally did not had the time to try it. So the current solution that we pick was the third : we do not keep all the constraints and we hope that the reasoner is doing well.

G. Flags on the script

To make the script more generic, it use 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 to create a graph, add nodes and edges using the `dot` language. . The code in `graph_viz.py` generate a `gv` file written in `dot` and it is possible to convert them to `png` using the `dot` command. The bashscript `convert_all_to_png.sh` convert 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.

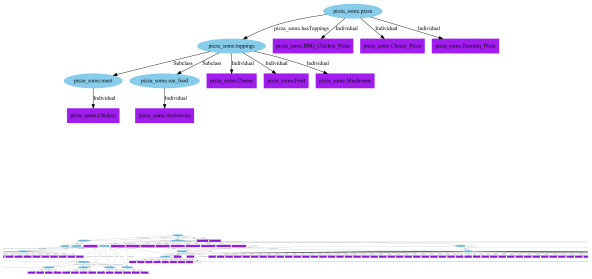


Fig. 16. Large file with graph viz (the image is cut on the edges)

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 allow 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 :

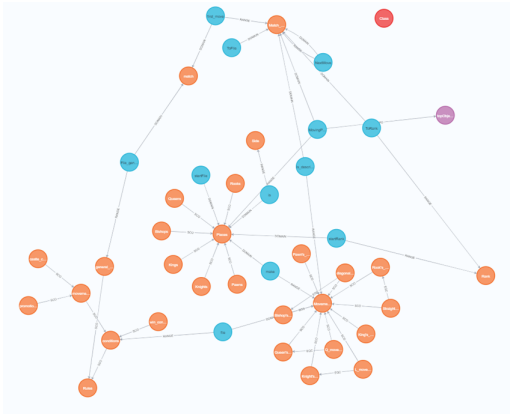


Fig. 17. Global view of our KG in Neo4J

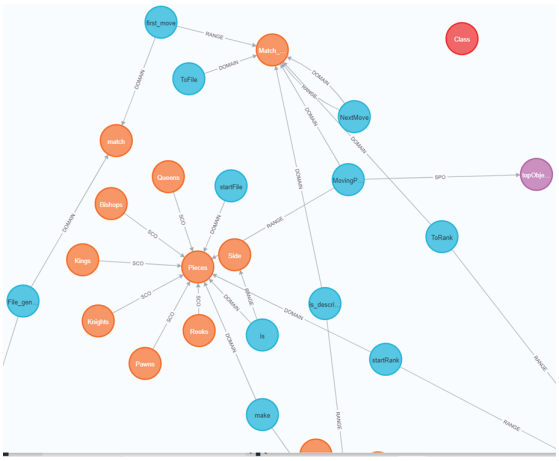


Fig. 18. Zoom on a part of that KG

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 very few to compute the other way. For this part, we decided to forget all the creation of new nodes made by the reasonner to parse constraints. This means that if the reasonner 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 build with enough constraints to be a really interested 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 test on what we needed to add in the KG to keep constraint all the time. Unfortunately, we did not had the time to treat that part.

Below our result for this part :

We started with the small handmade pizza_(with)_some ontology(figure : 19). Then we launched the reasonner. As we can see in the figure 20, meetpizza was infered as a part of pizza. In the KG, we can see figure 21 that this interpretation is kept, so that the ”some” structure : `meetpizza ->hasToppings some meet` , is understood and kept in the KG in 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 reasonner worked well and we were able to keep all those modification and so to keep all the semantique in the KG and to build a bijection function.

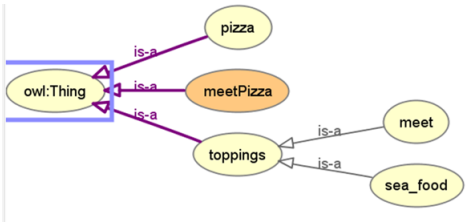


Fig. 19. Ontology before inference

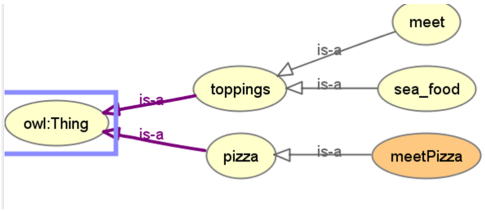
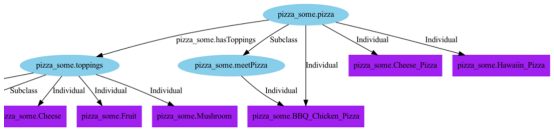


Fig. 20. Ontology after the reasonner



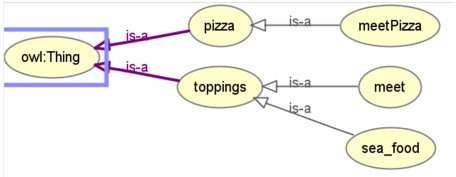


Fig. 22. Ontology from the KG

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 is was much more difficult to see differences in the middle on a large database.

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 script 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 test on big ontology and big KG. But we are able to test every part of the project trough little ontology. We had not enough time to explore one of the major part of the project : the representation of the constraint in the Knowledge graph. Instead, we spend our time also to understand what was the range of the reasonner and how we could use it to explore the semantic of an ontology. We also improve our understanding of those notion by making from nothing a chess ontology and some visualisation on python and on Neo4J.

Acknowledgment

The authors would like to thank the Phd Mariano, for his work and all his time that he gave us. For all the research that he maid for us.

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

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

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