COMP 6721 Applied Artificial Intelligence (Fall 2023)

Lab Exercise #08: Knowledge Graphs, Part I

Solutions

Question 1 Your first task is to translate the knowledge graph you developed on Worksheet #7 into a real RDF graph. Write down the triples using the Turtle format[1] discussed in the lecture. Some notes:

- We have not yet covered the details how predicates like *"studies at"* are encoded in URIs. The details will come in the next lecture & lab; for now, just use a URI like `http://example.org/studiesAt`.
- Remember to use the correct form for Wikidata URIs using the `entity` path, e.g., `http://www.wikidata.org/entity/Q326342` for Concordia. The `wiki/` path is a redirect for displaying a human-readable HTML page in a browser and cannot be used for working with RDF triples in a program.
- For now, use the URIs `user:joe` and `user:jane` instead of the `user:joe#me` and `user:jane#me` we had on the worksheet.[2]

Now, validate your graph:

- Using a browser, go to `http://www.ldf.fi/service/rdf-grapher/` and paste your RDF triples into the text field.
- Make sure "Turtle" is selected as the input format.
- Click the "Visualize" button.
- Examine the results of parsing the input. Correct any mistakes that you might have made accordingly.
- If no mistakes are found in the input, you should see the output in form of a graph; e.g., the first three triples should look like Figure 1.

There are a number of other RDF-related tools available online; for example, try out the RDF converter at `https://issemantic.net/rdf-converter` and convert your Turtle file (`.ttl`) into JSON-LD and RDF/XML to get an idea how these formats look like. The validator at `https://www.w3.org/RDF/Validator/` only accepts RDF/XML, but it can also draw you a graph corresponding to your triples.

[1]Use `.ttl` for the file extension, like in `mytriples.ttl`

[2]Unlike we did it on the worksheet, the fragment identifier is typically used as part of a namespace, as we will see in the next lecture.

Note: you will probably encounter references to `rdfs:` (RDF Schema) in examples you find online; we will cover RDFS in Lecture #9 *("Knowledge Graphs, Part II")*.

Here's a possible solution using RDF triples in Turtle format:

```
@prefix user: <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

user:joe
    foaf:knows user:jane ;
    user:studiesAt <http://www.wikidata.org/entity/Q326342> ;
    foaf:mbox <mailto:joe@example.com> .

user:jane
    user:studiesAt <http://www.wikidata.org/entity/Q201492> .
```

This is how to read it:

- `@prefix` is used to define the namespaces used in the document. `user:` is the prefix for an example organization, and `foaf:` is the prefix for the *Friend of a Friend* (FOAF) vocabulary (we will cover this next time).
- `foaf:knows` is used from the FOAF vocabulary to denote that the person identified by `user:joe` knows the person identified by `user:jane#me`.
- `user:studiesAt` is a placeholder property to indicate where the person identified by `user:joe` studies. Here, it points to the entity at Wikidata with the identifier `Q326342`, which represents Concordia University.
- `foaf:mbox` is used from the FOAF vocabulary to denote the mailbox (email) of the person identified by `user:joe`. The URI for the email address follows the `mailto:` scheme.
- `user:studiesAt` is also used to indicate where the person identified by `user:jane` studies, pointing to the entity at Wikidata with the identifier `Q201492`, which would represent McGill University.

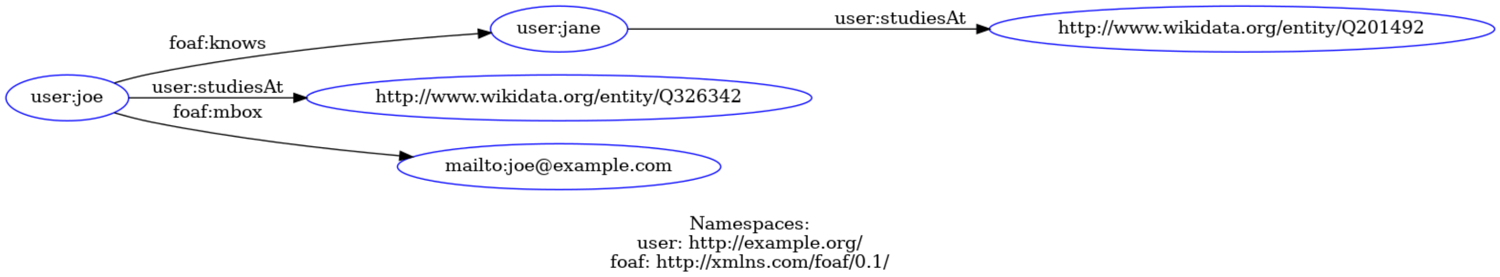
user:jane
user:studiesAt
http://www.wikidata.org/entity/Q201492
foaf:knows
user:joe
user:studiesAt
http://www.wikidata.org/entity/Q326342
foaf:mbox
mailto:joe@example.com
Namespaces:
user: http://example.org/
foaf: http://xmlns.com/foaf/0.1/

Figure 1: Visualizing the RDF triples encoding the knowledge "Joe's email is `joe@example.com`", "Joe knows Jane", "Joe studies at Concordia University", and "Jane studies at McGill University"

- The Turtle file ends with a period (`.`) which is used to terminate each set of statements about a subject.

Note that we did not add a triple about Concordia being located in Montreal, since this triple is already available under the URI `http://www.wikidata.org/entity/Q326342`: `<http://www.wikidata.org/entity/Property:P131> <http://www.wikidata.org/entity/Q340>`.

Question 2 For working with RDF and related standards, there are a multitude of libraries available. For example, a popular open source framework for Java is Apache Jena.[3] Here, we will use the RDFLib for Python.[4]

First, we need to install RDFLib. Run the command below in your Conda environment:

```
pip install rdflib
```

For the details, please refer to the RDFLib documentation section, *"Getting started with RDFLib"*.[5]

Now, we want to load the graph you prepared in Question #1. The code you need from RDFLib is `<graph>.parse`, as shown below:

```python
import rdflib

# Create a Graph
g = rdflib.Graph()

# Parse an RDF file in Turtle format
g.parse(<RDFFile_Path>, format='turtle')
```

You can now print out your whole graph `g` using the code below:

```python
# Loop through each triple in the graph (subj, pred, obj)
for s, p, o in g:
   # Print the subject, predicate and the object
   print(s, p, o)
```

Now, go through the RDFLib documentation section, *"Loading and saving RDF"*,[6] to see how to read and write RDF graphs in different formats. Add code to write the triples in a different format, e.g., RDF/XML and N-Triples.

You can use `g.parse(rdffilename)` or `g.load(rdffilename)` function to load a graph in different formats:

```python
g.load(<rdffilename>, format='turtle')
```

[3] See `http://jena.apache.org/`

[4] See `https://rdflib.dev/` and `https://github.com/RDFLib/rdflib`

[5] See `https://rdflib.readthedocs.io/en/stable/gettingstarted.html`

[6] See `https://rdflib.readthedocs.io/en/stable/intro_to_parsing.html`

or

```
g.parse(<rdffilename>, format='turtle')
```

To write triples from a graph in different formats, you use `serialize`:

```
# Write the graph in RDF/XML format
g.serialize(destination='output.rdf', format='xml')

# Write the graph in N-Triples format
g.serialize(destination='output.nt', format='nt')
```

Question 3 We can also create triples directly with RDFLib. To begin, read the documentation section, *"Creating RDF triples"*.[7]

Now, start your program by importing the following libraries:

```
from rdflib import Graph, Literal, RDF, Namespace
from rdflib.namespace import FOAF, RDFS
```

Then, create a new graph instance with:

```
g = Graph()
```

Now you can use the `g.add()` function to add triples to the graph. Write code that adds triples representing:

- `<Joe> <is a> <foaf:Person>`
- `<Joe> <rdfs:label> "Joe"`
- `<Joe> <foaf:knows> <Jane>`

For printing the generated graph, you can use the `g.serialize()` function.

Note: RDFLib has pre-defined name spaces that can be used for graph creation. In order to add another prefix to the knowledge graph, like `foaf`, use the `g.bind()` function.[8]

Here is a possible solution:

```
# Add the namespaces for known prefixes to the graph
g.bind("foaf", FOAF)
g.bind("rdfs", RDFS)
g.bind("rdf", RDF)

# Commands to create a user-defined namespace and bind it to the graph
user = Namespace("http://example.org/")
g.bind("user", user)
```

[7]See `https://rdflib.readthedocs.io/en/stable/intro_to_creating_rdf.html`
[8]See `https://rdflib.readthedocs.io/en/stable/namespaces_and_bindings.html` for details

```python
# Commands to add triples to graph for Joe using "user" namespace
g.add((user.joe, RDF.type, FOAF.Person))
g.add((user.joe, FOAF.knows, user.jane))
g.add((user.joe, RDFS.label, Literal("Joe")))

# Print the graph, serialized in a specific format
print(g.serialize(format='turtle'))
```

Note: if your output looks like this (a byte string, indicated by a leading `b'`):

```
b'@prefix foaf: <http://xmlns.com/foaf/0.1/> .\n@prefix rdfs: <http://www.
    ↪ w3.org/2000/01/rdf-schema#> .\n@prefix user: <http://example.org/> .\
    ↪ n\nuser:joe a foaf:Person ;\n rdfs:label "Joe" ;\n foaf:knows user:
    ↪ jane .\n\n'
```

you're using Python 3.7 or earlier; in this case, print the graph with:

```python
print(g.serialize(format='turtle').decode("utf-8"))
```

Question 4 The next step is to learn how to *navigate* graphs, in order to retrieve the knowledge we need, for example, to answer a question.

Remember that triples are a way to represent graphs, so you can navigate a knowledge graph by querying its triples. RDFLib supports basic *triple pattern matching* through the `triples()` function. You can match specific parts of a graph with the methods `objects()`, `subjects()`, `predicates()`, etc. Look at the documentation section *"Navigating Graphs"*[9] for the details.

Now, write a Python program that prints out all triples in your graph for the subject URI corresponding to `Joe`.

Here are a few examples for pattern matching and graph navigation:

```python
# Find the subject for a given predicate and object
name = list(g.subjects(RDFS.label, Literal('Joe')))
if name:
   print(name[0]) # prints the subject node, i.e., http://example.org/joe
else:
   print("No subject found with label 'Joe'")

# Find the list of all predicates and objects for a given subject
joe_details = g.predicate_objects(rdflib.term.URIRef('http://example.org/joe'))
# Iterate through each predicate-object pair and print them
for pred, obj in joe_details:
   print(pred, obj)
```

Note: Here, `None` is treated as a wildcard.

[9]See `https://rdflib.readthedocs.io/en/stable/intro_to_graphs.html`

Question 5 A powerful feature of knowledge graphs is that we can *merge* different graphs together, in order to connect knowledge from different sources. In this question, we will merge the graph you created in above with the knowledge about Concordia from Wikidata.

RDFLib supports various graph operations, like union (addition), intersection, difference (subtraction) and XOR. Like before, we first import RDFLib and create a graph instance, which we'll call `g1`:

```
import rdflib

# Create graph g1
g1 = rdflib.Graph()
```

We can now parse the Wikidata graph, by directly loading it from the online knowledge base URI:

```
# Parse the Wikidata graph about Concordia (in Turtle format)
g1.parse("http://www.wikidata.org/entity/Q326342.ttl")
```

To check the number of triples in our Wikidata graph about Concordia University, use the command below. The output should show a large number of triples, more than 3500:[10]

```
# Print the number of triples in the graph
print(len(g1))
```

Similarly, load the graph your create above in Question #1 using the `g.parse()` function. To merge the two graphs (union), use the addition (+) operator.

Print out the merged graph using the `g.serialize()` function and verify that they were indeed merged using Concordia's URI:

Here is a possible solution:

```
# Load the local graph create above
g2.parse("Lab8_Q1.ttl", format="turtle")

# Print the number of triples in the graph g2
print(len(g2))

# Merge (union) the two graphs
g = g1 + g2

# Print the total number of triples present in the merged graph
print(len(g))

# Write the merged graph to a file in Turtle format
with open("Merged_Lab8.ttl", "wb") as f:
```

[10]The exact number of triples changes over time as knowledge about Concordia is updated on Wikidata.

```
   f.write(g.serialize(format='turtle'))
```

Note that you can now answer questions from this merged graph that were not possible to answer from each of them alone, for example, *"In which city is the university located that Joe is studying at?"*. Try running your code from Question 4 on this merged graph to see all the information you now have available. Then, write some triple patterns to print out the city where Joe is studying.

You can add code like this to find the city

```
# Which city is Joe studing in? First define the namespaces:
user = rdflib.Namespace("http://example.org/")
wdt = rdflib.Namespace("http://www.wikidata.org/prop/direct/")

# Find the university where Joe studies
for uni in g.objects(subject=user.joe, predicate=user.studiesAt):
   # Find the city where this university is located, using Wikidata Property:P276
   for city in g.objects(subject=uni, predicate=wdt.P276):
      print(f"Joe studies in the city: {city}")
```

This exercise serves as an introduction to the expansive capabilities of knowledge graphs in AI systems. By further exploring linked data, such as loading additional details about Montreal (Q340) from Wikidata, an autonomous agent can enhance its understanding of related concepts like the country in which the city is located. This approach exemplifies the power of knowledge graphs in providing comprehensive answers to complex queries. It underscores the importance of data provenance, ensuring that all responses generated by the AI are backed by traceable and verifiable sources. The integration of diverse knowledge sources like this can be applied in numerous real-world applications, ranging from intelligent personal assistants to advanced data analysis in research, offering a glimpse into the potential of knowledge graphs in building informed, reliable AI solutions.