CITS3005 Knowledge Representation

Lecture 7: Knowledge Graphs

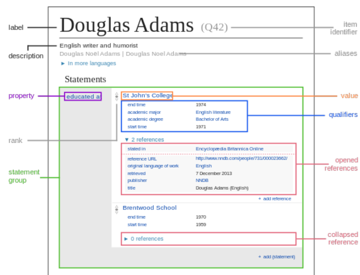
Tim French

The University of Western Australia

2023

Part II: Knowing and Reasoning at Scale

The study of knowledge, deduction and reasoning is fascinating from an introspective point: we are interested in how we are intelligent, and there are many interesting philosophical questions that arise from this.
Machines are better at simple, repeatable processes that can be done very quickly, and this is where knowledge representation is most likely to make an impact.
In this section of the course we will look at how knowledge representation technologies can be used to store, process, and search large information stores.



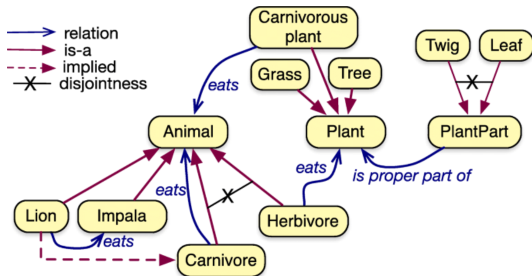
Technologies and terminologies

- A *database* is intended to store information, and allow it to be retrieved efficiently. However the process by which it is retrieved is via query language over a schema the is designed external to the database. That is, the *meaning* of the information in the database is external (with the designer), and the semantic interpretation of the data is only incorporated in the interface of the database.
- A *knowledge graph* is an attempt to build a *universal schema* for interacting with data, via triples. Arbitrary facts can be represented as triples, and we can extend this to rules. The knowledge graph can contain its own schema and the relationships between types and entities.
- An ontology is a formal representation of the meaning of data in a knowledge graph. This is achieved via a series of logical constraints over the relations in a knowledge graph, giving them meaning. An ontology can be stored in a knowledge graph.

Functionality

When considering these technologies we will consider the following functionalities:

- Representation: How is information represented and stored (property graph, graph DB, triple store).
- Querying: How is information retrieved (sparql, cypher)
- Semantics: How are is meaning represented (OWL, SWRL, SHACL)
- Proof: How is reasoning demonstrated (Pellet, Hermit).
- Learning: How is knowledge derived (Induction, Graph Neural Networks).



Overview

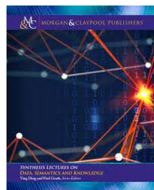
- In this lecture we will look at the basic concept of a knowledge graph, and how it relates to prolog, problog and databases. We will look at the basic methods to define and edit schema, and the basic methods to query graph databases with SPARQL or Cypher.
- Week 8 we will consider how graph schema are specified and used to inject meaning into knowledge graphs.
- In week 9 we will then consider ontologies and reasoning, and consider first order logic representations of knowledge at scale.
- In week 10, we will the look at the challenges of Ontology Design and Knowledge Engineering, and the importance of consistency and reuse.
- Week 11 will consider the problems of incomplete knowledge and knowledge graph induction.

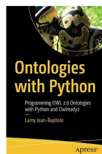
Tools

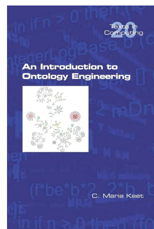
The tools we will use for this will generally come from the W3C semantic web stack, and work with python: That is we will use RDF for triple representations (and RDFLib in python, as a triple store) OWL for representing ontologies (and owlready2 as a python library), with networkx for visualisation.

The main texts we will rely on in the section of the course are:

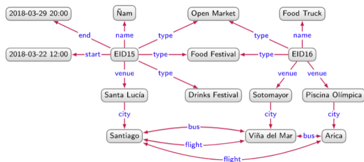
- Knowledge Graphs (Hogan et al., General structure and theory);
- Ontologies with Python (Jean-Baptiste, OWlReady2); and
- An Introduction to Ontology Engineering (Keet).

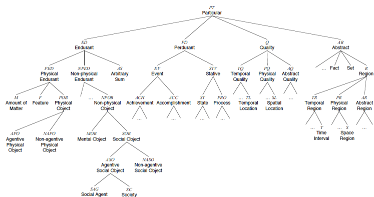


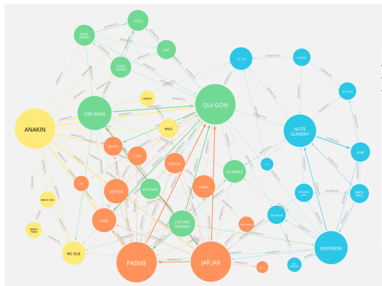


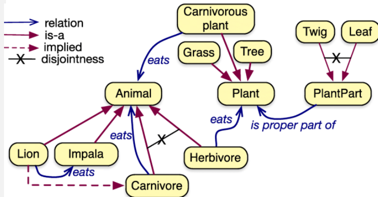


Examples







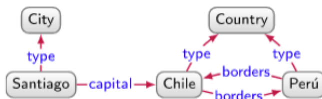


What are Knowledge Graphs?

Definition

A knowledge graph is a graph of data intended to accumulate and convey knowledge of the real world, whose nodes represent entities of interest and whose edges represent relations between these entities.

By knowledge, we refer to something that is known. Knowledge may be composed of simple statements, such as “Santiago is the capital of Chile”, or quantified statements, such as “all capitals are cities”. Deductive methods can then be used to entail and accumulate further knowledge (e.g., “Santiago is a city”).



(a) Directed edge-labelled graph

Databases vs Knowledge Graphs

Consider an Event table with five columns: `Event(name, venue, type, start, end)` where `name` and `start` together form the primary key of the table.

This is efficient from a memory and processing point of view, but does not sit well with the open world nature of data. Is this everything we want to know about *Events*.

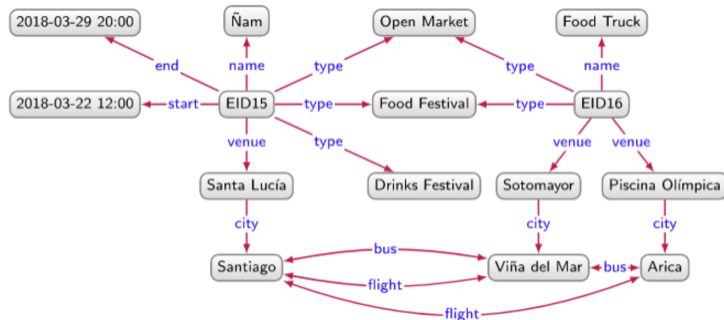
Alternatively we can give an event an id, and each property can be related to this id: `EventName(id,name)`, `EventStart(id,start)`, `EventEnd(id,end)`, `EventVenue(id,venue)`, `EventType(id,type)`

This schema essentially represent columns as edges from an entity to a value as a representation of some property.

Knowledge graph has several interpretations: we will formalise them as directed edge-labelled graphs.

Directed Edge Labelled Graphs

Directed Edge Labelled Graphs is defined as a set of nodes – like `Santiago`, `EID16` and a set of directed labelled edges between those nodes, like `Santa Lucía - city - Santiago`.



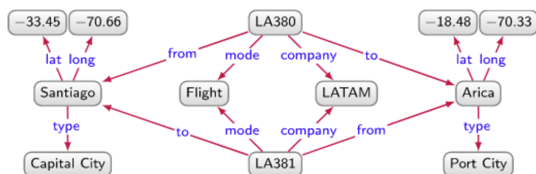
Definitions

We denote by *Con* a countably infinite set of constants.

Definition (Directed edge-labeled graph)

A *directed edge-labelled graph* is a tuple $G = (V, E, L)$, where $V \subseteq Con$ is a set of nodes, $L \subseteq Con$ is a set of edge labels, and $E \subseteq V \times L \times V$ is a set of edges.

To efficiently represent knowledge *heterogeneous graphs* and *property graphs* (like Neo4J databases) have been defined, however we will use Directed edge-labelled graphs as they are theoretically simpler and more flexible.



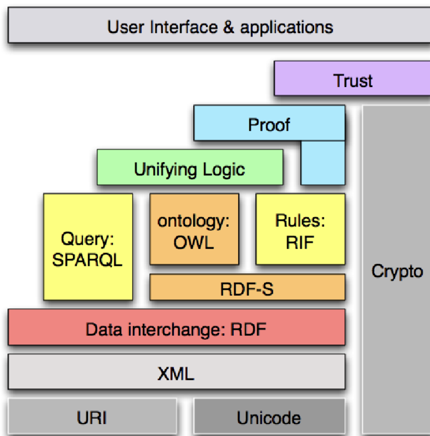
(a) Directed edge-labelled graph

Resource Description Framework (RDF)

RDF is a simple but powerful realisation of directed edge-labelled graphs.
RDF stands for:

- Resource: Everything that can have a unique identifier (URI).
- Description: attributes, features, and relations of the resources
- Framework: model, languages and syntaxes for these descriptions

RDF was published as a W3C recommendation in 1999.



Example

RDF models (KGs) consist of statements (triples) of

- `subject predicate object`, or
- `subject predicate literal`

```
<rdf:RDF
xmlns:rov=\http://www.w3.org/TR/vocab-regorg/ \
xmlns:org=\http://www.w3.org/TR/vocab-org/"
xmlns:locn=\http://www.w3.org/ns/locn" >

<rov:RegisteredOrganization rdf:about=\http://example.com/org/217279811
  <rov:legalName> \Nike"< /rov:legalName>
  <org:hasRegisteredSite rdf:resource=\http://example.com/site/1234"/>
</rov:RegisteredOrganization>

<locn:Address rdf:about=\http://example.com/site/1234"/>
  <locn:fullAddress>
    " Dahliastraat 24, 2160 Wommelgem"
  </locn:fullAddress>
</locn:Address>
</rdf:RDF>
```

Example continued

While XML is useful for machine readability, turtle format (and others) have been developed to be more human readable:

```
@prefix rov: <http://www.w3.org/TR/vocab-regorg/> .
@prefix org: <http://www.w3.org/TR/vocab-org/> .
@prefix locn: <http://www.w3.org/ns/locn####> .

<http://example.com/org/2172798119 >
  a <rov:RegisteredOrganization> ;
  rov:legalName \Niké \;
  org:hasRegisteredSite <http://example.com/site/1234> .

<http://example.com/site/1234>
  a <locn:Address> ;
  locn:fullAddress \Dahliastraat 24, 2160 Wommelgem" .
```

RDF in Python

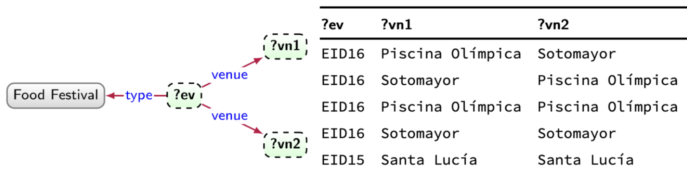
RDF is supported in Python using rdflib. This supports a triple store that allows creation, editing and querying of knowledge graphs.

```
from rdflib import Graph, Literal, RDF, URIRef
# rdflib knows about quite a few popular namespaces, like W3C ontologies, schema.org etc.
from rdflib.namespace import FOAF , XSD

# Create a Graph
g = Graph()

# Create an RDF URI node to use as the subject for multiple triples
donna = URIRef("http://example.org/donna")

# Add triples using store's add() method.
g.add((donna, RDF.type, FOAF.Person))
g.add((donna, FOAF.nick, Literal("donna", lang="en")))
g.add((donna, FOAF.name, Literal("Donna Fales")))
g.add((donna, FOAF.mbox, URIRef("mailto:donna@example.org")))

# Add another person
ed = URIRef("http://example.org/edward")

# Add triples using store's add() method.
g.add((ed, RDF.type, FOAF.Person))
g.add((ed, FOAF.nick, Literal("ed", datatype=XSD.string)))
g.add((ed, FOAF.name, Literal("Edward Scissorhands")))
g.add((ed, FOAF.mbox, Literal("e.scissorhands@example.org", datatype=XSD.anyURI)))

# Iterate over triples in store and print them out.
print("--- printing raw triples ---")
for s, p, o in g:
    print((s, p, o))

# For each foaf:Person in the store, print out their mbox property's value.
print("--- printing mboxes ---")
for person in g.subjects(RDF.type, FOAF.Person):
    for mbox in g.objects(person, FOAF.mbox):
        print(mbox)

# Bind the FOAF namespace to a prefix for more readable output
g.bind("foaf", FOAF)
```

Querying Knowledge Graphs

At the core of every structured query language for graphs lie basic graph patterns. Terms in basic graph patterns are thus divided into constants, such as `Africa` or `venue`, and variables, which we prefix with question marks, such as `?event` or `?rel`.

A basic graph pattern is evaluated against the data graph by generating mappings from terms to constants in the graph, so the pattern is contained in the graph.



?ev	?vn1	?vn2
EID16	Piscina Olímpica	Sotomayor
EID16	Sotomayor	Piscina Olímpica
EID16	Piscina Olímpica	Piscina Olímpica
EID16	Sotomayor	Sotomayor
EID15	Santa Lucía	Santa Lucía

Graph Patterns

For these definitions, we introduce a countably infinite set of variables Var ranging over (but disjoint from: $Con \cap Var = \emptyset$) the set of constants. We let $Term = Con \cup Var$.

Definition (Basic directed edge-labelled graph pattern)

We define a *basic directed edge-labelled graph pattern* as a tuple $Q = (V, E, L)$, where $V \subseteq Term$ is a set of node terms, $L \subseteq Term$ is a set of edge terms, and $E \subseteq V \times L$ is a set of edges (triple patterns).

For evaluating graph patterns, define a partial mapping $\mu : Var \to Con$ from variables to constants, whose domain is $dom(\mu)$. Given a basic graph pattern Q, let $Var(Q)$ denote the set of all variables in Q We further denote by $\mu(Q)$ the image of Q under μ, meaning that any variable $v \in Var(Q) \cap dom(\mu)$ is replaced in Q by $\mu(v)$.

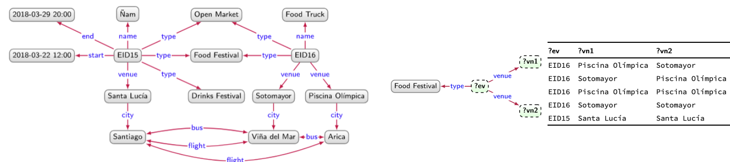
Evaluating Graph Patterns

For two directed edge-labelled graphs $G1 = (V1, E1, L1)$ and $G2 = (V2, E2, L2)$, we say that $G1$ is a *sub-graph* of $G2$, denoted $G1 \subseteq G2$, if and only if $V1 \subseteq V2$, $E1 \subseteq E2$, and $L1 \subseteq L2$.

Definition (Evaluation of a basic graph pattern)

Let Q be a basic graph pattern and let G be a data graph. The evaluation of the basic graph pattern Q over G is the set of mappings

$$Q(G) = \{\mu | \mu(Q) \subseteq G \text{ and } dom(\mu) = Var(Q)\}$$



?ev	?vn1	?vn2
EID16	Piscina Olímpica	Sotomayor
EID16	Sotomayor	Piscina Olímpica
EID16	Piscina Olímpica	Piscina Olímpica
EID16	Sotomayor	Sotomayor
EID15	Santa Lucía	Santa Lucía

Complex Graph Patterns

Complex graph patterns are built from simple graph patterns using a relational algebra, with the operators *projection* (π), *selection* (σ), *renaming* (ρ), *union* ($\cup$), *difference* (-), and *join* ($\bowtie$).

Definition (Complex graph pattern)

Complex graph patterns are defined recursively, as follows:

- If Q is a basic graph pattern, then Q is a complex graph pattern.
- If Q is a complex graph pattern, and $V \subseteq Var(Q)$, then $\pi V(Q)$ is a complex graph pattern.
- If Q is a complex graph pattern, and R is a selection condition with Boolean and equality connectives ($\wedge$, $\vee$, $\neg$, $=$), then $\sigma_R(Q)$ is a complex graph pattern.
- If both $Q1$ and $Q2$ are complex graph patterns, then $Q1 \bowtie Q2$, $Q1 \cup Q2$, and $Q1 - Q2$

Evaluating Complex Graph Patterns

Letting R denote a Boolean selection condition and μ a mapping, we denote by $\mu \models R$ that μ satisfies the Boolean condition.
Two mappings μ_1 and μ_2 are compatible, denoted $\mu_1 \sim \mu_2$, if and only if $\mu_1(v) = \mu_2(v)$ for all $v \in dom(\mu_1) \cap dom(\mu_2)$.

Definition (Complex graph pattern evaluation)

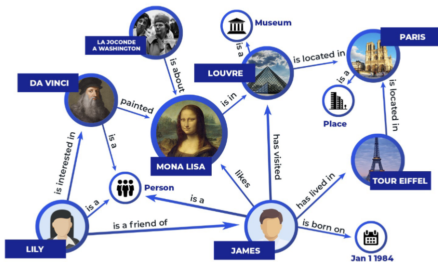
If Q is a basic graph pattern, then $Q(G)$ is given. Otherwise, $Q(G)$is defined as follows:

$$
\begin{aligned}
\pi_V(Q)(G) &= \{mu[V] \mid \mu \in Q(G)\} \\
\sigma_R(Q)(G) &= \{\mu \mid \mu \in Q(G), \mu \models R\} \\
Q_1 \bowtie Q_2 &= \{\mu_1 \cup \mu_2 \mid \mu_1 \in Q_1(G), \mu_2 \in Q_2(G), \mu_1 \sim \mu_2\} \\
Q_1 \cup Q_2 &= \{\mu \mid \mu \in Q_1 \text{ or } \mu \in Q_2\} \\
Q_1 - Q_2(G) &= \{\mu \mid \mu \in Q_1(G),\ \mu \notin Q_2(G)\}
\end{aligned}
$$

Querying RDF with SPARQL

SPARQL is the *Simple Protocol and RDF Query Language* and is a data manipulation language for RDF and triple stores like *Blazegraph*.
SPARQL queries represent Complex Graph Patterns:

```
SELECT ?person ?museum WHERE{
  ?person ex:likes ?painting .
  ?painting ex:is_in ?museum .
}
```



SELECT queries in SPARQL

- Basic form: `SELECT projection WHERE { pattern }`
  - the projection is a list of variables
  - the pattern is (essentially) a list of triples
  - returns a table with one row per result and
  - one column per projection variable
- Optional and combinable variations:
  - `SELECT * WHERE { ... }`
  - `SELECT * WHERE { ..pattern.. MINUS {..pattern..} }`
  - `SELECT * WHERE { ..pattern.. UNION ..pattern.. }`

Navigational Graph Patterns

A path expression r is a *regular expression* that allows for matching arbitrary-length paths between two nodes using a regular path query (x, r, y), where x and y can be variables or constants.

- The base path expression is where r is a constant (an edge label).
- If r is a path expression, then r^* (Kleene star: zero-or-more) is also a path expression.
- If r_1 and r_2 are path expressions, then $r_1 \mid r_2$ (disjunction) and $r_1 \cdot r_2$ (concatenation) are also path expressions.
- If r is a path expression, then r^-(inverse) is a path expression.

Navigational Patterns in SPARQL

Property paths (in SPARQL 1.1):

- Concatenation: $a \backslash b$ (means "first a, then a's b")
- Inversion: ˆa (means "a backwards")
- Grouping: (...) (nested composite properties)
- Repetition: $a*$ (0:n), $a+$ (1:n), $a?$ (0:1)
- Alternative: $a \mid b$ (either a or b)
- Negation: !a (any other property than a)

For example: `?uncle ^(:hasParent / :hasBrother) ?nephew .` and `?uncle (^:hasBrother / ^:hasParent) ?nephew .` are the same relation.

Query Interfaces

Knowledge Graphs Systems often provide support for interacting without using raw SPARQL, including

- **Faceted browsing:** Users start by specifying a simple search. They are then presented with a set of matching results, and a set of facets, which are typically attributes and values present in the current results set.
- **Query Building:** Users are provided with a form or graphical interface that can be used to specify a graph query without needing to understand the syntax of a specific query language.
- **Query-by-example:** Users provide examples of positive and sometimes negative answers to their queries, and the system seeks to reverse engineer a suitable query.
- **Question answering:** Users express their queries as questions in natural language; for example, they might ask "What food festivals will be held in Arica?"