Exercises on SPARQL

G. Falquet

1. Query execution

Given the following RDF graph

```
@prefix x: <http://example.org/> .
x:a x:friend_of x:b , x:c , x:d .
x:c x:friend_of x:d , x:e , x:f .
x:a x:friend_of [x:speaks "Japanese", "German", "French"] .
x:d x:friend_of x:c , x:a .
x:a x:studies [x:subject "Knowledge representation"; x:level "master"] .
x:c x:name "Jim" .
x:e x:speaks "French" ; x:speaks "German" .
x:f x:friend_of x:c ; x:speaks "German" ; x:studies [x:subject "History"] .
```

What woould be the answer to the following queries

Q1 :

```
select ?x
where {
    ?x x:friend_of ?y . ?y x:studies ?z. ?z x:subject "Knowledge representation" . }
```

Q2: (be careful)

```
select ?x
where {
    ?x x:speaks ?a .
    filter  (?a != "German" )
}
```

Solution

Q1 :

```
x:d
```

Q2 :

```
x:e
blank node (from x:a x:friend_of [x:speaks "Japanese", ...])
```

2. Querying friend-of-a-friend (foaf) graphs.

The FOAF vocabulary contains (among others) the classes

Agent	Person	Project
Organization	Group	Document
OnlineAccount	PersonalProfileDocument	Image

and the properties

name	title	img
depiction(depicts)	familyName	givenName
knows	based_near	age

made(maker)	primaryTopic(primaryTopicOf)	member
nick	mbox	homepage
weblog	openid	jabberID
mbox_sha1sum	interest	topic_interest
topic(page)	workplaceHomepage	workInfoHomepage
schoolHomepage	publications	currentProject
pastProject	account	accountName
accountServiceHomepage	tipjar	sha1
thumbnail	logo	

(The full names of these classes and properties is obtained by adding the prefix `foaf`, defined as `http://xmlns.com/foaf/0.1/`)

Use this vocabulary to express in SPARQL the following queries:

1. Find the persons who use "bob001" as nickname
2. Find the persons who know somebody based near `<http://geo.org/geneva>`
3. Find the persons who know somebody who doesn't know anybody
4. Find all the projects that have at least three (different) persons currently working on them
5. Find the persons who have a common interest but who have no common group (there is no group that has these two persons as members)

Solution

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>
select ?p
where {
  ?p a foaf:Person ; foaf:nickname "bob001"}

@prefix foaf: <http://xmlns.com/foaf/0.1/>
select ?p
where {
   ?p a foaf:Person ; foaf:knows ?somebody.
   ?somebody foaf:based_near <http://geo.org/geneva>}

@prefix foaf: <http://xmlns.com/foaf/0.1/>
select ?p
where {
   ?p a foaf:Person ; foaf:knows ?somebody.
   filter not exists{?somebody foaf:knows ?q}

@prefix foaf: <http://xmlns.com/foaf/0.1/>
select ?p
where {
   ?p a foaf:Project .
   ?p1 foaf:currentProject ?p .
   ?p2 foaf:currentProject ?p .
   ?p3 foaf:currentProject ?p .
   filter (?p1 != ?p2 && ?p1 != ?p3 && ?p2 != ?p3)
}

@prefix foaf: <http://xmlns.com/foaf/0.1/>
select ?p1 ?2
```

```
where {
   ?p1 a foaf:Person .  ?p2 a foaf:Person .
   ?p1 foaf:interest ?i. ?p2 foaf:interest ?i .
   filter not exists{?g a foaf:Group. ?g member ?p1 , ?p2 }
}
```

3. Querying networks and lists

A road network is represented by the following RDFS schema:

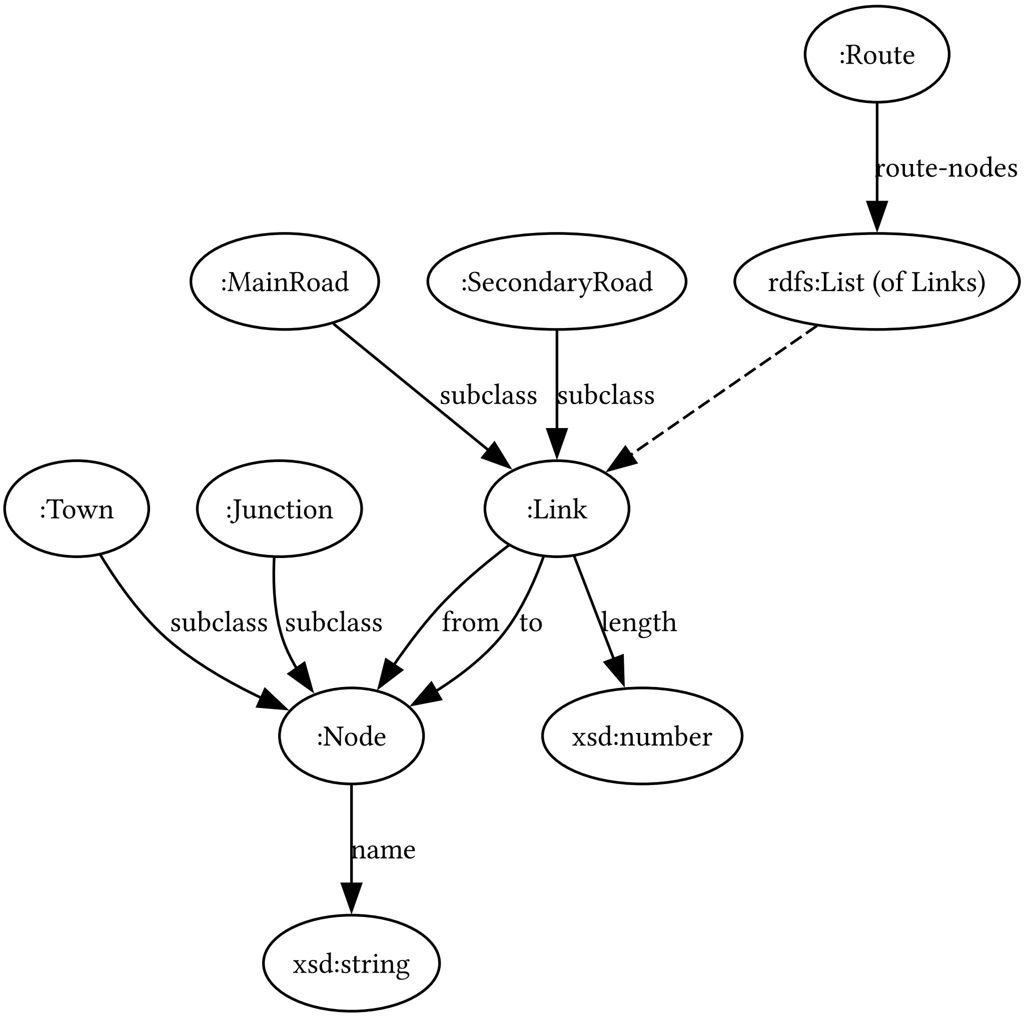
```
:Node a rdfs:Class .
:Link a rdfs:Class .
  :MainRoad rdfs:subClassOf :Link  .
  :SecondaryRoad rdfs:subClassOf :Link  .
  :Town rdfs:subClassOf :Node  .
  :Junction rdfs:subClassOf :Node  .
:Route a rdfs:Class .

:from a rdf:Property ; rdfs:domain :Link ; rdfs:range :Node .
:to a rdf:Property ; rdfs:domain :Link ; rdfs:range :Node .

:name a rdf:Property ; rdfs:domain :Node ; rdfs:range xsd:string .
:length a rdf:Property ; rdfs:domain :Link ; rdfs:range xsd:number .
:speedLimit a rdf:Property ; rdfs:domain :Link ; rdfs:range xsd:number .

:route-nodes a rdf:Property ; rdfs:domain :Route  ; rdfs:range rdf:List .
```

Schematically:



Sample data:

```
# sample data
@prefix : <http://unige.ch/rdf> .
:Geneva a :Town . :Lausanne a :Town ; :name "Lausanne"@fr.
:Sion a :Town . :Fribourg a :Town . :Zurich a :Town .
:r1 a :MainRoad ; :from :Geneva ; :to :Lausanne ; :length 58 .
:r2 a :MainRoad ; :from :Fribourg ; :to :Lausanne ; :length 78 .
:r3 a :MainRoad ; :from :Zurich ; :to :Fribourg ; :length 78 .
:r4 a :SecondaryRoad ; :to :Geneva ; :from :Lausanne ; :length 77 .
:j1 a :Junction .
:r66 a :Route ; :route-nodes (:r3 :r2 :r4) .
:r77 a :Route ; :route-nodes ([:from :Sion ; :to :j1] [:from :j1 ; :to :Zurich]) .
```

1. Write SPARQL queries to answer the following questions on this graph, when it is possible
    1. find all the towns that can be reached from the node :Zurich by following at most three links (in the from → to direction) (you can use path expressions)
    2. find all the towns that can be reached from the node :Zurich (you must use path expressions)
    3. find all the towns that cannot be reached from :Zurich
    4. find the towns that can be reached from :Zurich by following only main roads
    5. what is the total length of route “r66”?
    6. find routes whose nodes are all towns (i.e. there are no nodes that are not towns)
    7. find the towns that can be reached from :Zurcih by following a route that uses only main roads

Remark. To check if a condition holds find the entities that do not satisfy it

Solution

```
# towns reachable from :Zurich in 1, 2, or 3 steps (links)
prefix : <http://unige.ch/rdf>
select distinct ?t where {
    :Zurich ^:from/:to/(^:from/:to)?/(^:from/:to)? ?t
  }

# towns reachable from :Zurich
prefix : <http://unige.ch/rdf>
select distinct ?t where {
    :Zurich (^:from/:to)*  ?t
  }

# towns not reachable from :Zurich
prefix : <http://unige.ch/rdf>
select distinct ?t where {
    {?t a :Town} minus
    {:Zurich (^:from/:to)*  ?t}
  }

# towns that can be reached from :Zurich by following only main roads
# cannot be expressed in SPARQL
#   would require complex path expression of the form. Something like
#     {:Zurich (^:from/[a :MainRoad]/:to)*  ?t}

prefix : <http://unige.ch/rdf>
select distinct ?t where {
    :Zurich (^:from/:to)*  ?m
  }

#  length of route "r66"
prefix : <http://unige.ch/rdf>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select (sum(?lng) as ?s) where {
    :r66 :route-nodes/rdf:rest*/rdf:first/:length ?lng .
  }

# routes whose nodes are all towns (i.e. there are no nodes that are not towns)
prefix : <http://unige.ch/rdf>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?r where {
    ?r a :Route .
    filter not exists{?r :route-nodes/rdf:rest*/rdf:first/(:from|:to) ?node.
                       filter not exists {?node a :Town}} .
  }

# towns that can be reached from :Zurcih
# by following a route that uses only main roads
prefix : <http://unige.ch/rdf>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?r ?t ?ll where {
    ?r a :Route .
    ?r :route-nodes/rdf:first/:from :Zurich .
    ?r :route-nodes/rdf:rest* ?last_element .
```

```
      ?last_element rdf:rest rdf:nil; # make sure it's the last elt.
      rdf:first/:to ?t .
    filter not exists{?r :route-nodes/rdf:rest*/rdf:first ?link.
        filter not exists {?link a :MainRoad}} .
  }
```

4. Linear algebra with SPARQL

In a RDF graph a matrix celle is represented by a node with 3 properties: `r` (row number), `c` (column number), `v` (value). A matrix is represented by a node connected to the matrix cells through the a `cell` property. For example, the matrix $M = \begin{pmatrix} 1 & -1 \\ 6 & -8 \end{pmatrix}$ is represented as

```
:M1 :cell
    [:r 1; :c 1; :v 1], [:r 1; :c 2; :v -1],
    [:r 2; :c 1; :v 6], [:r 2; :c 2; :v -8],
```

Write SPARQL queries to

1. extract the second column of a matrix A
2. computes the trace of a matrix A (the sum of the elements on the diagonal (where the row and column numbers are equal))
3. compute the scalar product $A_{i\cdot} * B_{\cdot j}$ of row i of A and column j of B,

$$A_{i\cdot} * B_{\cdot j} \stackrel{\text{def}}{=} \sum_{k=1}^{n} A_{i,k} B_{k,j}$$

   (A and B must be compatible, i.e. if A has n columns then B must have n rows)
4. compute the matrix product $C = A * B$ of two compatible matrices A and B.

$$C_{i,j} \stackrel{\text{def}}{=} A_{i\cdot} * B_{\cdot j}$$

Test data:

```
@prefix : <http://unige.ch/rdf> .
:M1 :cell
[:r 1; :c 1; :v 1], [:r 1; :c 2; :v 0], [:r 1; :c 3; :v 0],
[:r 2; :c 1; :v 0], [:r 2; :c 2; :v 1], [:r 2; :c 3; :v 0],
[:r 3; :c 1; :v 0], [:r 3; :c 2; :v 0], [:r 3; :c 3; :v 1].

:M2 :cell
[:r 1; :c 1; :v 1], [:r 1; :c 2; :v 2], [:r 1; :c 3; :v 3],
[:r 2; :c 1; :v 4], [:r 2; :c 2; :v 5], [:r 2; :c 3; :v 6],
[:r 3; :c 1; :v 7], [:r 3; :c 2; :v 8], [:r 3; :c 3; :v 9].
```

Solution

```
# 1. column 2 of A

prefix : <http://unige.ch/rdf>
select  ?val
where {:A :cell [:r ?r ; :c 2 ; :v ?val] .
}
order by ?r

# 2. trace of A
```

```
prefix : <http://unige.ch/rdf>
select  (sum(?va) as ?trace)
where {:A :cell [:r ?n ; :c ?n ; :v ?va] .
}

# 3. Row iii of A scalar colum jjj of B (two integer literals)

@prefix : <http://unige.ch/rdf>
select (sum(?va * ?vb) as ?scal)
where {:A :cell [ :r iii ; :c :n ; :v ?va ].
       :B :cell [ :r ?n ; :c jjj ; :v ?vb ].
}

# 4. Matrix product of A and B

prefix : <http://unige.ch/rdf>
select  ?rnum ?cnum (sum(?va * ?vb) as ?scal)
where {:A :cell [:r ?rnum ; :c 1] .    # to get all the row numbers of A
       :B :cell [:r 1 ; :c ?cnum ].    # to get all the column numbers of B
       :A :cell [ :r ?rnum ; :c ?n ; :v ?va ].
       :B :cell [ :r ?n ; :c ?cnum ; :v ?vb ].
}
group by ?rnum ?cnum
order by ?rnum ?cnum
```