Integrációs és ellenőrzési technikák (VIMIAC04)

I. házi feladat: Szemantikus modellezés és információkeresés

Készítette: Strausz György
2021. március

Tartalom

1. Bevezetés

A feladat célja bemutatni egy egyszerűsített feladat keretében egy tárgyterület modellezését és információkeresést gráfadatbázisok alkalmazásával.
Az Integrációs és ellenőrzési technikák tárgy gyakorlatainak keretében RDF adatbázisok és OWL ontológiák modellezését és lekérdezését vizsgáltuk. A mostani feladat az RDF adatbázisok elérését mutatja és gyakoroltatja be programozási környezetből.
A házi feladat keretében a már ismert múzeumi adatbázist kell bővíteni és lekérdezni dinamikusan, API-n keresztül egy egyszerű szolgáltatás keretében.

2. Követelmények

- Végezze el az alábbiakban leírt feladatokat. Használhat saját környezetet vagy virtuális gépet a niif.cloud.bme.hu (Visual Studio) vagy a fured.cloud.bme.hu (Java Eclipse) felhőben található „SZVlabMIT2” nevű sablon alapján. A virtuális gépen a feladat megoldásához szükséges környezetek és adatfájlok közvetlenül is elérhetőek. (A gyakorlaton alkalmazott virtuális gépekhez képest ez a környezet tartalmaz egy telepített Java (Eclipse) fejlesztő környezetet is.
- Készítsen dokumentációt a feladatoknál leírt követelmények szerint tetszőleges formátumban.

- Az elkészített dokumentáció PDF nyomtatását és az elkészített projektet töltse fel a tárgy oktatási felületére (edu.vik.bme.hu).
- A feladatot Java, C++ vagy Phyton eszközök felhasználásával is megoldhatja, a segédprogramok Java Eclipse környezetben készültek.

3. Feladatok

3.1. RDF adatbázis bővítése

A feladat: Tervezzen és valósítsa meg a Szépművészeti Múzeum képeit tartalmazó RDF adatbázis bővítését. Az adatbázis (szepmuveszeti-dbpedia.2017-02-24.trig vagy szepmuveszeti.20151028.n3) a melléklet CIDOC Conceptual Reference Model szabvány szerint lett elkészítve. A feladat három tetszőlegesen kiválasztott művész adatainak bővítése legalább két-két új tulajdonsággal.
Használja az RDF4J böngészőt a művészek adatainak elérésére, a már tárolt tulajdonságok lekérdezésére, majd készítsen programot, amely a betöltött adatokat bővíti az új tulajdonságokkal, értékekkel.

Felhasználható adatok: A virtuális gépen az RDF4J workbench indítása után közvetlenül is elérhető „szép" repository (saját környezetben a mellékelt szepmuveszeti-dbpedia.2017-02-24.trig vagy szepmuveszeti.20151028.n3 adatfájl betöltése után érhető el. .
Leadandó: Az adatbázis bővítés leírása, elkészített program.
Eszközök: Az adatbázis kezelésében a Segédletekben megadott RD4J REST API specifikáció és mintaprogram linkek segítenek.
Tanácsok: Javasoljuk, hogy az új tulajdonságokat a CIDOC Conceptual Reference Model szabvány szerint vegyék fel, például a művészek Wikipedia oldalai alapján. Az előkészített virtuális gépen az RDFJ workbench automatikusan elindul, saját gépen történő fejlesztéshez a telepítési útmutató elérhető itt a dokumentum alján.

3.2. RDF adatbázis lekérdezés

A feladat: készítsen egy egyszerű festmény kereső alkalmazást a Szépművészeti Múzeum képeit tartalmazó RDF adatbázis lekérdezésével. A lekérdő szolgáltatásnak képesnek kell lennie egy művész neve vagy legalább egy tulajdonsága alapján (pl. nemzetisége, korszaka, születési évszázada) alapján lekérdezni a szűrésnek megfelelő művész(ek) festményeinek címét. Nem szükséges, de ajánlott (és a maximális pont eléréséhez feltétel) egy egyszerű grafikus felület elkészítése, pl. webes szolgáltatás fejlesztése.
Leadandó: a lekérdező megoldás rövid leírása, futási példa, elkészített példa.
Felhasználható adatok: az 3.1. feladatban is alkalmazott adatbázis.
Tanácsok: Az RDF4J workbench segítségével futasson minta lekérdezéseket, majd a sikeres lekérdezéseket építse meg a programkódból is a felhasználótól bekért információk alapján.

4. IMSC Feladat

4.1. Ontológia bővítése

A feladat: Tervezze meg és valósítsa meg a 3.1 feladatban leírt bővítést a gyakorlatokon alkalmazott ontológián is, tehát bővítse az ontológiát új elemekkel és készítsen egy egyszerű lekérdező felületet festmények kereséséhez. A gyakorlat során használt ontológiát a virtuális gépen is megtalálja: szepmuveszeti-dbpedia.2017-03-10.owl.
Az ontológia és a következtető gép programból történő kezeléséhez a Segédletben (6.6 fejezet) talál útmutatót.

5. Értékelési szempontok

A házi feladat elégséges szinten történő elfogadásához (40%) legalább az egyik feladat elkészítése szükséges, mindkét programozási feladat dokumentálva leadva jó értékelést (75%) kap, a maximális pont eléréséhez egy egyszerű GUI (nem feltétlenül webes, de javasolt) megvalósítását várjuk.
Az IMSC pontok megszerzésére hasonló feltételek vonatkoznak.

6. Segédletek

6.1. Virtuális gép indítása a kari felhőben

1. Böngészőben belépni a kari felhőbe az egyetemi azonosítóval: https://niif.cloud.bme.hu vagy https://fured.cloud.bme.hu

2. A virtuális gépek ablak alatti zöld "Új" gombra kattintva egy új virtuális gép indítása:
A megjelenő sablonok közül kiválasztani a " SZVlabMIT2" virtuális gépet.

3. Ha a virtuális gép fut, akkor a lokális gépen elindítani egy távoli asztal kapcsolatot a vm.smallville.cloud.bme.hu:XXXX távoli géppel

ahol XXXX a futó virtuális gép port címe, ami megtalálható a szükséges jelszóval együtt az induló virtuális gép weblapján.

6.2. RDF4J API specifikációk

RDF4J REST API specifikáció: https://rdf4j.org/documentation/reference/rest-api/
Mintapélda az API alkalmazására:
https://graphdb.ontotext.com/documentation/free/using-graphdb-with-the-rdf4j-api.html

6.3. RDF4J telepítése saját gépre

Telepítési útmutató:
https://share.mit.bme.hu/index.php/s/nMZPQ3yPQxrcjmbf

6.4. RDF4J workbench alkalmazása

A felhőben található virtuális gépeken a Szep adatbázis (Repository) közvetlenül elérhető.

Saját környezetben nyissa meg böngészőben az OpenRDF Workbench felületét:
http://localhost:8080/rdf4j-workbench/
A bal oldali menüben válassza a “Repositories / New repository” menüpontot, majd az alábbi beállításokkal hozhatja létre az adatbázist:

- `Type: Native Java Store`
  (“In Memory Store” esetén előfordulhat adatvesztés az adatbázis leállásakor.)
- `ID: szepmuveszeti`

- `Next`
- `Triple indexes: spoc,posc,cspo,csop,cops`
  Az alapértelmezett (spoc,posc) helyett további indexek megadásával a lekérdezések futtatása nagy adatbázisban gyorsabb lehet.
- `Create`

A fejlécben az "Repository:" után látja mindig az aktuálisan kiválasztott adatbázis azonosítóját. Ha több adatbázisa is van, váltani a "Repositories" menüpontban tud.
Az adatok betöltéséhez a "Modify / Add" menüpontban:

- Kapcsolja ki a "`use base URI as context identifier`" checkbox-ot.
- `RDF Data File: Choose File`
  Válassza ki az asztalon a "`szepmuveszeti.20151028.nt`" file-t.
- `Data format: N3`
  Az "`(autodetect)`" erre a file-ra nem működik. A file tallózása után állítsa be, különben visszaállítódik.
- `Upload`

Ha később újra akarja kezdeni a munkát, a "Repositories / Delete repository" menüpontban törölheti az adatbázist, és újra létrehozhatja.

6.5. Ajánlott dokumentumok, adatfájlok listája

- CIDOC Conceptual Reference Model: http://www.cidoc-crm.org/Version/version-7.0.1
- CIDOC szabvány további információk: http://www.cidoc-crm.org/cidoc-crm-tutorial
- Szépművészeti múzeum adatbázis:
  https://share.mit.bme.hu/index.php/s/t6eNL58eTPXN9JC

6.6. OWL következtető használata

Az OWL következtető használtalához töltse le a HermiT következtetőt a webről (legfrisebb) vagy innen: https://share.mit.bme.hu/index.php/s/dzDDwQRzXHS9Qcd
(A virtuélis gépen található mintapélda Eclipse projekt tartalmazza.)

A letöltött állomány kicsomagolása után a HermiT.jar-t fel kell venni a Java classpath-ba.

A következtetőt az OWL API-n (http://owlapi.sourceforge.net/)keresztül érjük el. Az OWL API dokumentáció releváns részei az alábbiak:

- OWL API reference (Javadoc) http://owlapi.sourceforge.net/documentation.html
- remek (angol nyelvű) mintapéldák és tutorial az OWL API dokumentáció honlapján (a feladathoz nem feltétlenül szükséges)

Valójában bármely más OWL API-kompatibilis következtetőt (pl. Pellet, Fact++) is használhatnánk azonos programkóddal, csak a következtető inicializálását végző sort kellene kicserélni. A HermiT következtetőt az alábbi sorral lehet inicializálni:

```
OWLReasonerFactory reasonerFactory = new org.semanticweb.HermiT.Reasoner.ReasonerFactory();
```

Az alábbiakban közlünk egy kiindulásként felhasználható mintapéldát az OWL API és a HermiT használatához. A példaprogram beolvassa a PC-Shop ontológiát, majd az osztályhierarchia alapján

keres releváns kulcsszavakat az "alkatrész" termhez. További magyarázatot találnak a kommentekben.

A PelletReasoningExample példaprogram megtalálható az informaciokereses.zip projektben.

```
001 package hu.bme.mit.iir;
002
003 import java.io.File;
004 import java.util.Collections;
005 import java.util.HashSet;
006 import java.util.Set;
007
008 import org.semanticweb.owlapi.apibinding.OWLManager;
009 import org.semanticweb.owlapi.model.*;
010 import org.semanticweb.owlapi.reasoner.*;
011 import org.semanticweb.owlapi.vocab.OWLRDFVocabulary;
012
013 import com.clarkparsia.pellet.owlapiv3.PelletReasonerFactory;
014
015 public class PelletReasoningExample {
016
017   public static final String PCSHOP_ONTOLOGY_FNAME = "pc_shop.owl.xml";
018   public static final String PCSHOP_BASE_URI =
019     "http://mit.bme.hu/ontologia/.....";
020   public static final IRI ANNOTATION_TYPE_IRI =
021       OWLRDFVocabulary.RDFS_LABEL.getIRI();
022
023   OWLOntologyManager manager;
024   OWLOntology ontology;
025   OWLReasoner reasoner;
026   OWLDataFactory factory;
027
028   public PelletReasoningExample(String ontologyFilename) {
029     // Hozzunk létre egy OntologyManager példányt. Többek között ez tartja
030     // nyilván, hogy mely ontológiákat nyitottuk meg és azok hogyan
031     // hivatkoznak egymásra.
032     manager = OWLManager.createOWLOntologyManager();
033
034     // Töltsük be az ontológiát egy fizikai címről.
035     // [Az ontológiáknak külön van logikai és fizikai címe, a kettő közötti
036     // leképezést az URIMapper-ek határozzák meg: manager.addURIMapper(...).
037     // Erre ontológiák közötti függőségek (import) kialakításánál van
038     // szükség.]
039     ontology = null;
040     try {
041       ontology = manager.loadOntologyFromOntologyDocument(
042           new File(ontologyFilename));
043     } catch (Exception e) {
044       System.err.println("Hiba az ontológia betöltése közben:\n\t"
045           + e.getMessage());
046       System.exit(-1);
047     }
048     System.out.println("Ontológia betöltve: " +
049         manager.getOntologyDocumentIRI(ontology));
050
051     // Létrehozzuk a következtetőgép egy példányát. Mi most a Pellet-et
052     // használjuk, de az OWL API univerzális, másik következtető
```

```
053     // használatához csak a Factory osztály nevét kellene kicserélni.
054     //
055     // A classpath-ban szerepelnie kell a Pellet jar file-jainak:
056     //   pellet-core.jar pellet-datatypes.jar pellet-explanation.jar, ...
057     OWLReasonerFactory reasonerFactory = new PelletReasonerFactory();
058     //
059     // Hozzunk létre egy következtetőgépet, betöltve az ontológiát.
060     // [Ha vannak az ontológiának függőségei, azok automatikusan
061     // betöltődnek a manager segítségével.]
062     reasoner = reasonerFactory.createReasoner(ontology);
063
064     try {
065        // Ha az ontológia nem konzisztens, lépjünk ki.
066        if (!reasoner.isConsistent()) {
067           System.err.println("Az ontológia nem konzisztens!");
068
069           Node<OWLClass> incClss = reasoner.getUnsatisfiableClasses();
070           System.err.println("A következő osztályok nem konzisztensek: "
071              + Util.join(incClss.getEntities(), ", ") + ".");
072           System.exit(-1);
073        }
074     } catch (OWLReasonerRuntimeException e) {
075        System.err.println("Hiba a következtetőben: " + e.getMessage());
076        System.exit(-1);
077     }
078
079     // Példányosítsuk az OWLDataFactory-t, aminek a segítségével létre tudunk
080     // hozni URI alapján OWL entitásokat (osztályt, tulajdonságot, stb).
081     factory = manager.getOWLDataFactory();
082  }
083
084  // Lekérdezi egy osztály leszármazottait a PC-Shop ontológiában.
085  //   className: a keresett osztály neve
086  // Az OWL következtető Set<Set<..>> struktúrákban, azaz halmazok
087  // halmazaként adja vissza a lekérdezés eredményét. A belső halmazok
088  // egymással ekvivalens, jelentés szempontjából megegyező dolgokat
089  // tartalmaznak. Például egy lekérdezés eredménye:
090  // { { Kutya, Eb }, { Macska, Cica }, { Ló } }
091  // Ez a függvény a fenti halmazt "kilapítva" adja vissza:
092  // { Kutya, Eb, Macska, Cica, Ló }
093  public Set<OWLClass> getSubClasses(String className, boolean direct) {
094     // Létrehozzuk az osztály URI-ját a base URI alapján.
095     IRI clsIRI = IRI.create(PCSHOP_BASE_URI + className);
096     // Ellenőrizzük, hogy az osztály szerepel-e az ontológiában.
097     // (Ha olyan dologra kérdezünk rá a következtetővel, ami nem szerepel az
098     // ontológiában, az nem vezet hibához az OWL nyílt világ feltételezése
099     // miatt.)
100     if (!ontology.containsClassInSignature(clsIRI)) {
101        System.out.println("Nincs ilyen osztály az ontológiában: \"" +
102              className + "\"");
103        return Collections.emptySet();
104     }
105     // Létrehozzuk az osztály egy példányát és lekérdezzük a leszármazottait.
106     OWLClass cls = factory.getOWLClass(clsIRI);
107     NodeSet<OWLClass> subClss;
108     try {
109        subClss = reasoner.getSubClasses(cls, direct);
```

```
110     } catch (OWLReasonerRuntimeException e) {
111         System.err.println("Hiba az alosztályok következtetése közben: "
112                 + e.getMessage());
113         return Collections.emptySet();
114     }
115     // Kiírjuk az eredményt, az ekvivalens osztályokat
116     // egyenlőségjellel elválasztva.
117     System.out.println("Az \"" + className + "\" osztály leszármazottai:");
118     for (Node<OWLClass> subCls : subClss.getNodes()) {
119         System.out.println("  - " + Util.join(subCls.getEntities(), " = "));
120     }
121     return subClss.getFlattened();
122   }
123
124   // Lekérdezi egy OWL entitás (osztály, tulajdonság vagy egyed) annotációit
125   // az ontológiából. (Ehhez nincs szükség a következtetőre.)
126   // Az annotációknak sok fajtája van, a lekérdezett típust az
127   // ANNOTATION_TYPE_IRI konstans tartalmazza, jelen esetben rdfs:label.
128   public Set<String> getClassAnnotations(OWLEntity entity) {
129     OWLAnnotationProperty label =
130         factory.getOWLAnnotationProperty(ANNOTATION_TYPE_IRI);
131     Set<String> result = new HashSet<String>();
132     for (OWLAnnotation a : entity.getAnnotations(ontology, label)) {
133       if (a.getValue() instanceof OWLLiteral) {
134         OWLLiteral value = (OWLLiteral)a.getValue();
135         result.add(value.getLiteral());
136       }
137     }
138     return Collections.unmodifiableSet(result);
139   }
140
141   // Példaprogram: beolvassa a pc-shop OWL ontológiát, majd listázza
142   // az "alkatrész" osztály valamennyi leszármazottját és azok annotációit.
143   public static void main(String[] args) {
144     // Példányosítsuk a fenti egyszerű Pellet következtető osztályt.
145     PelletReasoningExample p = new PelletReasoningExample(
146         PCSHOP_ONTOLOGY_FNAME);
147
148     // Végezzük keresőszó-kiegészítést az "alkatrész" kulcsszóra
149     // az osztály leszármazottai szerint!
150     final String term = "alkatrész";
151     Set<OWLClass> descendants = p.getSubClasses(term, false);
152     System.out.println("Query expansion a leszármazottak szerint: ");
153     for (OWLClass cls : descendants) {
154       // Az eredmények közül a beépített OWL entitásokat ki kell szűrnünk.
155       // Ezek itt az osztályhierarchia tetejét és alját jelölő
156       // "owl:Thing" és "owl:Nothing" lehetnek.
157       if (!cls.isBuiltIn()) {
158         // Kérdezzük le az osztály címkéit (annotation rdfs:label).
159         Set<String> labels = p.getClassAnnotations(cls);
160         System.out.println("\t- "
161             + term + " -> " + cls.getIRI().getFragment()
162             + " [" + Util.join(labels, ", ") + "]");
163       }
164     }
165   }
166 }
```