Department of Computer Science
Technical University of Cluj-Napoca

Knowledge-Based Systems

Laboratory activity
Project title: 20 Questions
Team name: Team14
Students: Linca Paul, Muresan Daniel
Email: linca_paul@yahoo.com, m_dani_26@yahoo.com

Assoc. Prof.dr. eng. Adrian Groza
Adrian.Groza@cs.utcluj.ro

Contents

Chapter 1

Project Overview

1.1 Competency questions

Use cases:

- Guess the character/person you are thinking of by asking a series of questions.
- Maximum 20 questions:
  - Generate yes/no questions whose answers narrow the search domain.
  - Generate follow-up questions based on previous answers.
  - Generate competent guesses based on previous answers.

Competency questions the application should answer:

1. Is X real or fictional?
2. What is the gender of X?
3. What is X's time period?
4. What is the occupation of X?
5. How is X related to Y?
6. What category does X belong to?
7. What physical traits does X have?
8. What personality traits does X have?

X = character to guess, Y = related character.

1.2 Related ontologies

Having narrowed our domain to cartoon and comic book characters, we have identified a couple of existing ontologies that could be used in our project:

- DBpedia film ontology - https://dbpedia.org/ontology/Film
- DBpedia tv series ontology - https://dbpedia.org/ontology/televisionSeries

In addition to these ontologies, there are several movie APIs that can be used:

- Superheroes API - https://superheroes.docs.apiary.io/
- Marvel API - https://developer.marvel.com/

1.3 DBPedia

DBPedia is a project aiming to extract structured content from the information created in the Wikipedia project. It allows users to semantically query relationships and properties of Wikipedia resources, including links to other related datasets. The DBpedia data set describes 6.0 million entities, out of which 5.2 million are classified in a consistent ontology.

1.4 SPARQL

Data from DBPedia is accessed using an SQL-like query language called SPARQL. It allows users to write queries in order to retrieve and manipulate data stored in RDF (Resource Description Framework) format. Thus, the entire database is a set of "subject-predicate-object" triples.

Most forms of SPARQL query contain a set of triple patterns called a basic graph pattern. Triple patterns are like RDF triples except that each of the subject, predicate and object may be a variable.

Example:

```
PREFIX ex: <http://example.com/exampleOntology#>
SELECT ?capital
       ?country
WHERE
{
    ?x  ex:cityname       ?capital   ;
        ex:isCapitalOf    ?y         .
    ?y  ex:countryname    ?country   ;
        ex:isInContinent  ex:Africa  .
}
```

This is the query one would write in order to answer the question: "What are all the country capitals in Africa?"

- **ex** stands for *http://example.com/exampleOntology* which defines the source ontology for our data.
- Variables are indicated with **?** or **$**
- The variables **?capital** and **?country** represent the returned data.
- When a triple ends with a semicolon, the subject from this triple will implicitly complete the following pair to an entire triple. So for example `ex:isCapitalOf ?y` is short for `?x ex:isCapitalOf ?y`.

We will use SPARQL in order to query DBPedia so as to find the most information possible about the character the application is trying to guess.
Thus, each question help us build a base query into a more and more complex query that will narrow down the possibilities.

Chapter 2

Approach

2.1 Java

The application will be developed using Java. We chose Java because of our familiarity with the language and because of its support for building and executing SPARQL queries as well as retrieving and manipulating the results.

The main package we will be using is `org.apache.jena.query`. It has support for the SPARQL RDF Query language.
Main classes:

- `Query` - class that represents the application query.
- `QueryFactory` - methods which provide access to the various parsers.
- `QueryExecution` - represents one execution of a query.
- `QueryExecutionFactory` - a place to get QueryExecution instances.
- `ResultSet` - All the QuerySolutions. An iterator.
- `ResultSetFormatter` - turn a ResultSet into various forms; into json, text, or as plain XML.

The UI of the application will be developed using JavaFX - a software platform for creating and delivering desktop applications.

2.2 SPARQL queries from Java

Steps for executing a SPARQL query in Java:

- Define the SPARQL query string format;
- Build the query string with the required parameters.
- Execute the query using `QueryExecution`
- Interpret the query results from `ResultSet`

The following example retrieves 10 male comic book characters and prints them out on the console:

```
//Build the SPARQL query
String queryString =
    "PREFIX dbo: <http://dbpedia.org/ontology/>\n" +
    "PREFIX foaf: <http://xmlns.com/foaf/0.1/>\n" +
    "PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>\n" +
    "PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>\n" +
    "SELECT DISTINCT (STR(?characterLabel) as ?characterName)\n" +
    "WHERE {\n" +
    "      ?character rdf:type dbo:ComicsCharacter.\n" +
    "      ?character rdfs:label ?characterLabel.\n" +
    "      ?character foaf:gender ?gender.\n" +
    "      FILTER (LANG(?characterLabel) = \"en\")\n" +
    "      FILTER(STR(?gender) = \"%s\")\n" +
    "      LIMIT %d";
// Set the values for the variable fields
String gender = "male";
int resultsLimit = 10;
//Add the variable fields in the query
queryString = String.format(queryString, gender, resultsLimit);
//Preview the query
System.out.println(queryString);
//Execute the query and display results
QueryExecution queryExecution = QueryExecutionFactory
    .sparqlService("http://dbpedia.org/sparql", queryString);
ResultSet resultSet = queryExecution.execSelect();
ResultSetFormatter.out(resultSet);
```

2.3 Building the Query

In order to build the query that will guess the player's character, we will think of it as a String consisting of three parts:

- A beginning that is constant, containing all the used ontologies and needed triples for filtering the results as comic book characters:

```
private final String queryBeginning =
    "PREFIX dbo: <http://dbpedia.org/ontology/>\n" +
    "PREFIX dbr: <http://dbpedia.org/resource/>\n" +
    "PREFIX dbp: <http://dbpedia.org/property/>\n" +
    "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>\n"+
    "PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>\n" +
    "PREFIX foaf: <http://xmlns.com/foaf/0.1/>\n" +
    "SELECT DISTINCT (STR(?characterLabel) as ?characterName) ?
       character\n" +
    "WHERE {\n" +
    "      ?character rdfs:label ?characterLabel.\n" +
    "      ?character rdf:type dbo:ComicsCharacter.\n";
```

- An end, also constant, that closes out the query braces and filters the characters so that only their english names are retrieved:

```
private final String queryEnd =
    "FILTER (LANG(?characterLabel) = \"en\")\n" +
    "}\n";
```

- And lastly, a list of triples (or, as we named them in our project, Filters), that is initially empty, but gets populated as the questions are getting answered:

```
private List<String> filters = new ArrayList<>();
```

2.4 Choosing what questions to ask

The questions are generated by choosing from a large pool of triples (filters) and displaying them as a natural language question. That being said, questions have to be asked so that they make sense in regard to the previous questions.
There are some pre-eliminary questions that establish standard attributes that all characters have: race, gender, etc. This will build a simple query that returns a pool of characters in which the final guess resides. Thus, the next question to be asked, must apply to at least one of those characters. (it makes no sense to ask if the character has blond hair if in a previous question it was established that the character is bald).
To determine if the next question makes sense, we wrote a method that builds an additional query that checks whether a triple makes sense in for the existing pool of guesses:

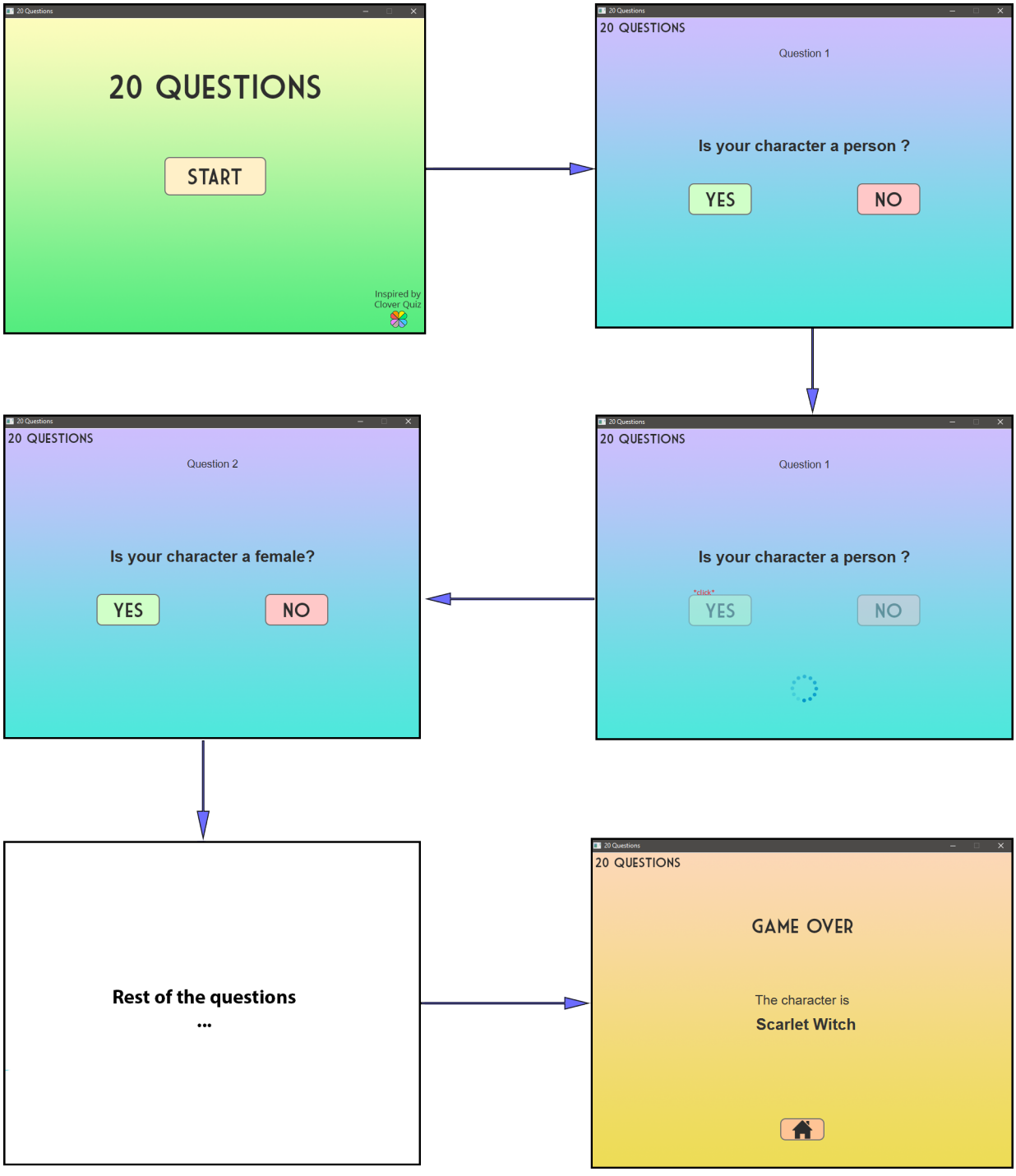
```
public static boolean executeBooleanQuery(String pageName, String
   attribute, String value)
{
    String booleanQuery =
            "PREFIX dbo: <http://dbpedia.org/ontology/>\n" +
            "PREFIX dbr: <http://dbpedia.org/resource/>\n" +
            "PREFIX dct: <http://purl.org/dc/terms/>\n" +
            "PREFIX dbc: <http://dbpedia.org/resource/Category:>\n" +
            "PREFIX dbp: <http://dbpedia.org/property/>\n" +
            "PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns
               #>\n" +
            "PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>\n" +
            "PREFIX foaf: <http://xmlns.com/foaf/0.1/>\n" +
            "PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>\n" +
                "ask where {<%s> %s %s}\n" ;

    booleanQuery = String.format(booleanQuery, pageName, attribute,
       value);
    QueryExecution queryExecution = QueryExecutionFactory.
       sparqlService("http://dbpedia.org/sparql", booleanQuery);

    return queryExecution.execAsk();
}
```

Chapter 3

Final Product



Appendix A

Original code

All of the original code is available at `https://github.com/PaulLinca/KBS-Project`.