Universidad Zaragoza

Trabajo Fin de Grado

Consultas flexibles sobre grafos de conocimiento basadas en lógica difusa

Autor

José Félix Yagüe Royo

Director

Fernando Bobillo

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2022

Resumen

El creciente interés en los grafos de conocimiento para representar el conocimiento del mundo real y la necesidad habitual de gestionar el conocimiento impreciso en muchas aplicaciones del mundo real exigen el estudio de enfoques para resolver consultas flexibles sobre grafos de conocimiento. En este Trabajo Fin de Grado, proponemos un enfoque novedoso para resolver ese problema basado en la combinación de grafos de conocimiento y lógica difusa.

En primer lugar, se propone un algoritmo para responder tales consultas que reutiliza los estándares de la Web Semántica (RDF y SPARQL) y construye una capa difusa sobre ellos. De esta manera se podrá acceder a distintos grafos de conocimiento, como por ejemplo la DBpedia, y mediante una serie de parámetros basados en la lógica difusa devolver los individuos más acordes a la consulta y las necesidades del usuario. Además, se utilizarán ontologías para almacenar de forma ágil y estandarizada los tipos de datos difusos que se utilizan en las consultas.

En segundo lugar, se implementa la aplicación *Graph Consultant*, cuya interfaz gráfica permite a los usuarios realizar rápida, cómoda e intuitivamente consultas flexibles, abstrayéndose de las tecnologías que la aplicación utiliza por debajo, y proporcionando al usuario los resultados en segundos.

Índice

1. Introducción y objetivos **1**

1.1. Introducción 1

1.2. Objetivos 2

1.3. Estructura del documento 2

2. Contexto **5**

2.1. Grafos de conocimiento 5

2.2. Ontologías 6

2.3. Lógica difusa 7

3. Consultas flexibles **11**

3.1. Representación de las consultas 11

3.2. Resolución de las consultas 12

4. Implementación **17**

4.1. Inicialización 17

4.2. Algoritmo 20

4.3. Interfaz gráfica 23

5. Caso de uso **31**

6. Trabajo relacionado **35**

7. Conclusiones y trabajo futuro **37**

7.1. Herramientas utilizadas 37

7.2. Conclusiones 37

7.3. Trabajo futuro 38

A. Diagrama de Gannt **41**

B. Diagrama de secuencia **43**

C. Publicación en congreso KGSWC 2022 **45**

Capítulo 1

Introducción y objetivos

1.1. Introducción

La Web Semántica está recibiendo mucha atención en los últimos años para representar conocimiento en muchos dominios y aplicaciones. La Web Semántica hace uso de diferentes tecnologías para representar este conocimiento, entre ellas están las ontologías, especificaciones formales y compartidas del vocabulario de un dominio de interés [1], los grafos de conocimiento (o *"Knowledge Graphs"*) [2], un modelo de datos estructurado basado en grafos para almacenar datos a gran escala, y los datos enlazados (*"Linked Data"*), un conjunto de buenas prácticas para publicar e interconectar datos en la web [3]. Las tecnologías mencionadas anteriormente suelen utilizar el lenguaje RDF para su representación.

En muchas aplicaciones reales, hay situaciones en las que el conocimiento no es preciso, por lo que existen algunas soluciones para gestionar estos problemas. En estos casos, la teoría de conjuntos difusos (*"Fuzzy Set Theory"*) y la lógica difusa (*"Fuzzy Logic"*) han demostrado ser útiles por más de 50 años [4, 5]. Los conjuntos difusos permiten representar parcialmente la pertenencia de un elemento o instancia a un conjunto, utilizando valores entre 0 y 1, y la lógica difusa permite manejar proposiciones que son parcialmente verdaderas y hacer deducciones a través de un proceso de razonamiento aproximado.

Por esta razón, durante los últimos años se han desarrollado las extensiones difusas de las tecnologías de la web semántica. La mayor parte de las investigaciones o desarrollos se han centrado en las ontologías difusas [6, 7] o en las lógicas descriptivas difusas [8]. En cambio, la combinación de la lógica difusa con los grafos de conocimientos no ha recibido mucha atención, por lo que se decidió realizar una aproximación con estas tecnologías para este trabajo.

En este Trabajo Fin de Grado, se propondrá un enfoque para responder a consultas flexibles sobre grafos de conocimientos clásicos. Mientras que en trabajos anteriores se ha considerado el caso de axiomas difusos, donde se añade un grado de verdad a los triples RDF, y por tanto no se usa (al menos directamente) el estándar RDF, aquí nos vamos a centrar en utilizar tipos de dato difusos, como por ejemplo BajoPrecio, representado como un

conjunto difuso. Esto permite hacer consultar flexibles donde el valor de la propiedad precio es BajoPrecio, es decir, donde el valor de la propiedad se restringe de manera imprecisa.

1.2. Objetivos

El objetivo principal de este Trabajo Fin de Grado consiste en el acceso y la consulta a grafos de conocimiento, como por ejemplo *"DBpedia"*, permitiendo consultas flexibles que involucren términos con una definición imprecisa. Para ello, se combinarán los métodos de gestión de grafos de conocimiento con la capacidad de la lógica difusa para el manejo de información imprecisa. Además se añadirán distintos parámetros para que el usuario pueda adaptar las consultas a sus necesidades, como distintos operadores de fusión y modificadores difusos del grado de pertenencia.

El segundo objetivo es el diseño e implementación de una aplicación de manera que los usuarios puedan realizar las consultas flexibles de manera cómoda e intuitiva sin tener que conocer el funcionamiento del algoritmo, es decir, que haga transparentes al usuario el uso de las tecnologías necesarias para obtener los resultados.

En concreto, se ha desarrollado un algoritmo capaz de soportar consultas flexibles a grafos de conocimiento, obteniendo las instancias que pertenecen a ciertas clases y que cumplen ciertas restricciones flexibles sobre las propiedades, definidas mediante conjuntos difusos. La principal característica es que somos capaces de ceñirnos a los estándares de la web semántica, utilizando el estándar RDF y consultas SPARQL, y hemos construido un capa difusa por encima con una serie de pasos adicionales para manejar las operaciones de la lógica difusa. Adicionalmente, se ha implementado una interfaz gráfica que permite a los usuarios realizar las consultas de manera sencilla.

1.3. Estructura del documento

El resto de la memoria sobre el trabajo desarrollado está estructurada de la siguiente manera:

- El Capítulo 2 da al lector una breve explicación sobre los términos que se van a utilizar en el resto del documento, como los grafos de conocimiento, las ontologías y la lógica difusa.

- El Capítulo 3 explica y da una aproximación teórica a cómo se han enfocado las consultas flexibles sobre los grafos de conocimiento.

- El Capítulo 4 entra más en detalles prácticos sobre el trabajo desarrollado, explicando paso a paso cómo funciona el programa, los problemas encontrados y cómo se han solucionado, y cómo se ha desarrollado la interfaz gráfica.

– El Capítulo 5 explica lo que podría ser un caso de uso real.

– El Capítulo 6 contextualiza este trabajo al explicar lo que otros autores han desarrollado en proyectos similares.

– El Capítulo 7 establece algunas conclusiones y brinda algunas ideas sobre lo que se puede desarrollar en trabajos futuros.

Además, se incluyen varios apéndices:

– El Apéndice A incluye un diagrama de Gannt.

– El Apéndice B incluye un diagrama de secuencia.

– El Apéndice C incluye una publicación científica obtenida como resultado del TFG.

Capítulo 2

Contexto

En este apartado se van a explicar brevemente algunas de las tecnologías o términos que se han utilizado en el proyecto, para así poder mejorar la comprensión del trabajo desarrollado al lector.

2.1. Grafos de conocimiento

Una de las tecnologías más importantes utilizadas son los grafos de conocimiento ya que son la principal estructura en la que se almacenan los datos. Los grafos de conocimiento no tienen una definición consensuada, pero podemos verlos como un modelo estructurado de datos con el que representar entidades y sus relaciones. Influenciados por el modelo de datos RDF, los grafos de conocimiento han derivado en un conjunto de triples sujeto-predicado-objeto (SPO), a pesar de esto, como se explica en [2], el concepto detrás de los grafos puede ser mucho más que esto.

Sin vincular la definición a ningún modelo de datos en particular, Hogan et al [2] proporciona la siguiente definición: *"un grafo de conocimiento es un grafo de datos destinado a acumular y transmitir el conocimiento del mundo real, cuyos nodos representan entidades de interés y cuyas aristas representan relaciones entre estas entidades"*. Pueden encontrarse más definiciones adicionales en [2].

Dentro de la definición anterior también se encuentran los grafos RDF[1], estándar W3C para representar información en la web, que son descritos como grafos directos. Los grafos RDF están basados en triples RDF de la forma $\langle s,p,o \rangle$, donde s es el sujeto, p es la propiedad y o es el objeto. De la misma manera, $\langle s,p,o \rangle$ quiere decir que s esta relacionado a o mediante p. En general, las relaciones pueden ser no simétricas.

Los elementos del triples RDF pueden ser URIs, literales XML o nodos en blanco. Para abreviar las URIs y hacerlas más entendibles, se suele utilizar la forma `prefijo:fragmento`.

[1] https://www.w3.org/TR/rdf11-primer/

Ejemplo 1. *La URI* `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` *puede ser escrita, definiendo previamente el prefijo* `prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>`*, simplemente como* `rdf:type`*.*

Para realizar consultas a grafos de conocimientos se ha utilizado el lenguaje SPARQL[2]. Este lenguaje soporta cuatro tipos distintos de queries, `SELECT`, `ASK`, `CONSTRUCT` y `DESCRIBE`. `SELECT`se utiliza para seleccionar datos tras haber especificado un patrón o condición en el apartado `WHERE`. `ASK` permite conocer si un patrón existe o no en el grafo de conocimiento. `CONSTRUCT` se ejecuta como un `SELECT` pero permite construir nuevos triples con los datos obtenidos. `DESCRIBE`, en vez de devolver los datos, construye un grafo RDF que describe los recursos obtenidos.

Como grafo de conocimiento principal para este proyecto se ha utilizado DBpedia[3], un proyecto que extrae datos de Wikipedia para realizar una versión web semántica. Este grafo de conocimiento es extremadamente grande y se va actualizando a medida que Wikipedia crece. Actualmente cuenta con más de 1.000 tipos diferentes de clases y con mas de 4.000.000 de instancias en el grafo de conocimiento. Este grafo de conocimiento cuenta con un endpoint que admite consultas SPARQL con la que acceder a toda la información almacenada.

2.2. Ontologías

Es habitual que un grafo de conocimiento tenga como referencia una ontología [1] a modo de esquema. El término ontología hace referencia a un sistema que quiere formular un esquema conceptual en profundidad sobre un dominio dado, para así, conseguir una representación formal del mismo al definir un conjunto de conceptos y sus relaciones. Los componentes de las ontologías son:

- Clases: son los conjuntos, colecciones o conceptos existentes en la ontología. Por ejemplo: Atleta o Avión.

- Individuos o instancias: son los objetos que pueblan la ontología y pueden pertenecer a una o varias clases. Por ejemplo: Mohamed Katir o Airbus A230.

- Propiedades: son los rasgos o características que los objetos pueden tener. Existen dos tipos de propiedades los propiedades de datos (*"data properties"*), donde el valor es un tipo de dato, y las propiedades de objeto (*"object properties"*), donde el valor es otro individuo de la ontología. Por ejemplo: victorias es una propiedad de datos y prototipo una propiedad de objeto.

[2] http://www.w3.org/TR/sparql11-query

[3] http://www.dbpedia.org

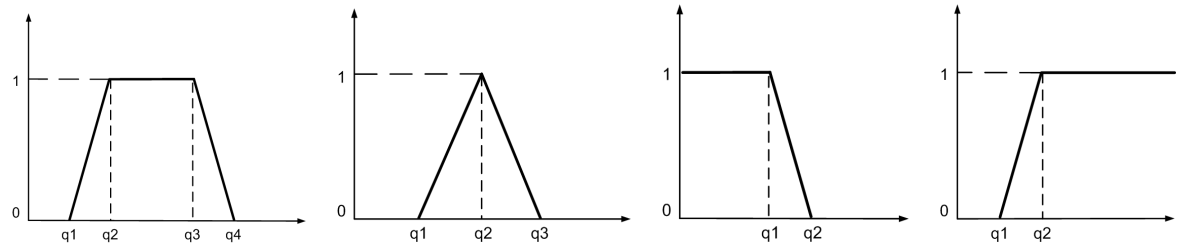
– tipos de dato: son los tipos de valores que pueden tener una propiedad de datos[4]. Por ejemplo: entero (xsd:integer), real (xsd:double) o cadena de caracteres (xsd:string).

– Axiomas: son las reglas o condiciones formales sobre los elementos que deben verificarse para preservar la estructura y semántica de la ontología. Los tipos de axiomas incluyen axiomas de subclase (por ejemplo decir que Atleta es subclase de Persona), hechos sobre conceptos (como decir que Airbus A230 pertenece a la clase Aircraft) y hechos sobre propiedades (como decir que Airbus A230 y 10441 están relacionados usando la propiedad numberBuilt).

2.3. Lógica difusa

Otro término importante en este trabajo es la lógica difusa, una técnica que nos permite trabajar con información vaga e imprecisa. Es una generalización de la lógica clásica propuesta por Zadeh donde los predicados no son necesariamente verdadero o falso, sino que se cumplen con un grado de verdad.

El pilar de la lógica difusa son los conjuntos difusos. Un conjunto difuso es una generalización del conjunto clásico donde los elementos pueden tener una pertenencia parcial al conjunto. Un conjunto difuso A se caracteriza mediante una función de pertenencia μ_A que asocia a cada elemento x un número real $\mu_A(x)$ en el rango [0,1], el cual representa el grado de pertenencia de x a A, por lo que un valor entre 0 y 1 denota una pertenencia parcial. A veces, $\mu_A(x)$ se denota simplemente $A(x)$.



(a) (b) (c) (d)

Figura 2.1: Funciones de pertenencia difusas (a) Trapezoidal, (b) Triangular, (c) Left-shoulder y (d) Right-shoulder.

Para construir las funciones de pertenencias, las opciones más comunes son las funciones trapezoidal (Figure 2.1 (a)), triangular (Figure 2.1 (b)), izquierda (*left-shoulder*) (Figure 2.1 (c)) y derecha (*right-shoulder*) (Figure 2.1 (d)).

Las funciones left-shoulder y right-shoulder funcionan de manera similar, pero de manera simétrica. Estas funciones se caracterizan por dos parámetros que indican el rango de valores

[4] https://www.w3.org/TR/xmlschema11-2/

en los que el valor proporcionado como entrada pertenece parcialmente al conjunto difuso. Por ejemplo, para la función right-shoulder, cuando el valor de la entrada no supera el menor parámetro (q_1), el grado de pertenencia es de 0, y el grado de pertenencia va creciendo constantemente hasta llegar a valer 1 cuando alcance o supere el parámetro q_2. En cambio, la función left-shoulder empieza valiendo 1 hasta el parámetro q_1, y desciende hasta tener un grado de pertenencia de 0 para entradas mayores o iguales al segundo parámetro q_2.

Para la función triangular existen tres parámetros, de manera que la entrada tiene un grado de pertenencia de 0 hasta el parámetro q_1, va aumentando hasta llegar a 1 cuando es exactamente igual que el parámetro q_2 y después decrece hasta llegar al parámetro q_3, a partir del cual el grado de pertenencia es 0.

Para la función trapezoidal hay cuatro parámetros. Funciona de la misma manera que la función triangular salvo porque entre el rango del segundo y el tercer parámetro, el grado de pertenencia es igual a 1.

Ejemplo 2. *El conjunto difuso de cervezas con* LowAlcohol *puede definirse usando una función triangular* **triangular**$(0{,}5, 3{,}1, 6{,}55)$. *Si el alcohol de la cerveza Ambar_Especial es de* $5{,}2°$*, su grado de pertenencia a* LowAlcohol *puede ser evaluado como:* $\mu_{LowAlcohol}(Ambar\_Especial) = (\mathbf{triangular}(0{,}5{,}3{,}1{,}6{,}55))(5{,}2) = 0{,}39$.

Para poder agrupar y combinar distintos valores de pertenencia (de un individuo a varios conjuntos difusos) en un único valor en [0,1] hay distintos métodos. En primer lugar, están los operadores lógicos: las t-normas, t-conormas, funciones de negación y funciones de implicación generalizan al caso difuso la intersección, unión, complemento e implicación de la lógica clásica. Ejemplos de t-normas son el mínimo y el producto. Ejemplos de t-conormas son el máximo y la suma acotada a [0,1].

También existen otras maneras de combinar distintos valores, a veces conocidas como operadores de agregación. Dos ejemplos notables son la media ponderada o el operador OWA (*"Ordered Weighted Averaging"*). La media ponderada es una función que requiere un número de pesos (valores entre 0 y 1 que suman 1) igual al número de conjuntos difusos que se quieren agrupar. El primer peso se multiplica por el grado de pertenencia al primer conjunto difuso, el segundo peso por el grado de pertenencia al segundo conjunto difuso, y así para todos los pesos y conjuntos. Finalmente, se suman todos los valores calculados y se obtiene el valor final. Dado un vector de valores a ordenar $[x_1,...,x_n]$ y un vector de pesos $[w_1,...,w_n]$, la media ponderada se define como:

$$\sum_{i=1}^{n} w_i \cdot x_i$$

El operador OWA [9] es parecido a la media ponderada, pero en vez de asignar los pesos a los conjuntos difusos, se asignan según su valor, de mayor a menor. Es decir, el primer peso se aplica al mayor valor, el segundo peso al segundo mayor valor y así hasta utilizar

todos los pesos. Formalmente, dados el vector de valores a ordenar $[x_1,...,x_n]$ y un vector de pesos $[w_1,...,w_n]$, OWA se define como:

$$\sum_{i=1}^{n} w_i \cdot x_{\sigma(i)}$$

siendo $\sigma(i)$ una permutación tal que $x_{\sigma(1)} \geq x_{\sigma(2)} \geq \cdots \geq x_{\sigma(n)}$.

Para obtener los pesos, existen varias técnicas. Una de ellas es la agregación guiada por cuantificadores (*"quantifier-guided aggregation"*) [10]. Se requiere un cuantificador Q representado por un conjunto difuso tal que $Q(0)=0$, $Q(1)=1$, y Q es no decreciente. Para calcular n pesos, cada peso w_i se calcula como:

$$w_i = Q(\frac{i}{K}) - Q(\frac{i-1}{K}) \tag{2.1}$$

Por último, también existen los modificadores difusos (o *"hedges"*) del grado de pertenencia, que permiten ampliar o reducir el grado de pertenencia. Los más comunes son el modificador very, caracterizado mediante la función $\mathsf{very}(x)=x^2$, que reduce el grado elevando al cuadrado el valor de pertenencia (ya que el valor está en [0,1]), y el modificador few, definido como $\mathsf{few}(x)=\sqrt{x}$, que aumenta el grado realizando la raíz cuadrada sobre el valor de pertenencia.

Ejemplo 3. *Si se aplica el modificador difuso* very *al conjunto difuso* LowAlcohol*, para cada cerveza x, el grado de ser una cerveza con poco alcohol puede calcularse como:* $\mu_{\mathsf{VeryLowAlcohol}}(x)=\mathsf{very}(\mu_{\mathsf{LowAlcohol}}(x))=(\mu_{\mathsf{LowAlcohol}}(x))^2$.

Capítulo 3

Consultas flexibles

En este apartado se va a dar una aproximación teórica a lo que se ha implementado en el trabajo.

3.1. Representación de las consultas

Dado un grafo de conocimiento $\mathcal{K}$, la consulta flexible se caracteriza por los siguientes elementos o parámetros:

- Un conjunto de clases C_1,C_2,...,C_N
- Un conjunto de propiedades de datos (*"data properties"*) funcionales[1] numéricas P_1,P_2,...,P_n
- Una lista de tipos de dato difusos D_1,D_2,...,D_n
- Un operador de fusión @: $[0,1]^n \rightarrow [0,1]$
- Un modificador del grado de pertenencia (opcional) h: $[0,1] \rightarrow [0,1]$

Algunos posibles operadores de fusión son t-normas, t-conormas, la media ponderada u OWA.

Ejemplo 4. *Dado un determinado grafo de conocimiento sobre cervezas, una posible consulta flexible para conseguir "cervezas Lager españolas con poco nivel de alcohol y poco amargor" puede describirse como:*

- *Clases:* Lager, SpanishBeer
- *Propiedades:* alcohol, bitterness

[1] Una propiedad es funcional si ningún individuo puede tener más de un valor para ella.

– *Tipos de datos difusos:* LowAlcohol, *definido como* **triangular**(0,5, 3,1, 6,55), *y* LowBitterness, *definido como* **triangular**(15,27,41)

– *Operador de fusión: producto (t-norma)*

– *Modificador difuso:* very$(x)=x^2$

Para representar los tipos de dato difusos, proponemos usar el lenguaje Fuzzy OWL 2 [11]. Fuzzy OWL 2 se basa en el estándar OWL 2[2] para representar los tipos de dato explicados anteriormente (trapezoidal, triangular, left-shoulder y right-shoulder). En Fuzzy OWL 2, una declaración de un tipo de dato difuso se puede asociar con una anotación OWL 2 que codifica la función de pertenencia que define un conjunto difuso, utilizando una sintaxis basada en XML, como se propuso en [11]. Adicionalmente, esas anotaciones pueden almacenarse como triples en un grafo de conocimiento o pueden formar parte de una ontología.

Ejemplo 5. *Para expresar que un tipo de dato difuso LowAlcohol corresponde a una función triangular* **triangular**(0,5,3,1,6,55)*, se puede utilizar el siguiente conjunto de triples RDF:*

```
@prefix ex: <http://www.example.org/beer/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
ex:fuzzyLabel rdf:type owl:AnnotationProperty .
ex:LowAlcohol rdf:type rdfs:Datatype .
ex:LowAlcohol ex:fuzzyLabel """<fuzzyOwl2 fuzzyType=\"datatype\">
 <Datatype type=\"triangular\" a=\"0.5\" b=\"3.1\" c=\"6.55\" />
 </fuzzyOwl2>""" .
```

3.2. Resolución de las consultas

Para resolver una consulta, proponemos seguir los siguientes pasos:

1. El primer paso es acceder al grafo de conocimiento y obtener, para cada individuo o instancia de cada clase, los valores de las propiedades asociadas a esas clases. Esto puede realizarse mediante la siguiente consulta en lenguaje SPARQL:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
SELECT ?x ?Y1 ... ?Yn
WHERE {
```

[2] https://www.w3.org/TR/owl2-overview/

```
    ?x rdf:type/rdfs:subClassOf* C1 .
    ...
    ?x rdf:type/rdfs:subClassOf* CN .
    ?x P1 ?Y1 .
    ...
    ?x Pn ?Yn .
}
```

El resultado de esta consulta es una lista de triple $\langle x,y_1^x,...,y_n^x\rangle$. Esta consulta no solo devuelve las instancias directas de las clases $C_1,...,C_n$, sino también las instancias indirectas, incluyendo instancias de sus subclases. También podría extenderse para obtener los valores $\texttt{Y}_i$ conectados a `?x` a través de alguna subpropiedad de P_i o para inferir las clases de `?x` a través de restricciones de dominio o rango.

2. El siguiente paso consiste en conseguir la función F_i que define el tipo de dato difuso D_i para cada $i \in \{1,...,n\}$. Si las definiciones están almacenadas en el grafo de conocimiento, las recuperamos con la siguiente consulta SPARQL:

```
SELECT ?Fi
WHERE {
    Di rdf:type rdfs:Datatype .
    Di fuzzyLabel Fi .
}
```

Los valores de $\texttt{?F}_i$ serían cadenas de texto en XML, con la sintaxis de Fuzzy OWL 2, que habría que parsear para extraer el tipo de función y los valores de sus parámetros. Todas las definiciones se almacenan en una lista para su uso posterior.

También puede suceder que no tengamos la posibilidad de añadir las definiciones de los tipos de dato difusos al grafo de conocimiento. Esto sucedería, por ejemplo, en la DBpedia actual. En este caso, proponemos usar una ontología externa donde se proporcionan las definiciones F_i para cada D_i. Igualmente, se tienen que parsear las anotaciones en XML.

3. Seguidamente se necesita conocer, para cada individuo, el grado de pertenencia a los conjuntos difusos asociados a las propiedades. Para cada individuo `?x` y para cada propiedad P_i, se calcula el grado de pertenencia del valor Y_i obtenido en la consulta SPARQL anterior al tipo de dato difuso F_i:

$$z_i^x = F_i(y_i^x),\ \forall i \in \{1,...,n\} \tag{3.1}$$

4. Después, se agregan todos los valores z_i^x que corresponden a cada individuo `?x` en un solo resultado r_x usando el operador de fusión

$$r_x = @_{i \in \{1,\ldots,n\}}(z_i^x) \tag{3.2}$$

5. Tras esto, para cada individuo, se tiene un grado de pertenencia para cada propiedad, por lo que es necesario conseguir un solo valor por cada individuo. Con ese objetivo, se agregan todos los valores z_i^x que corresponden a un mismo individuo `?x` en un solo resultado r_x utilizando un operador de fusión.

$$r_x = @_{i \in \{1,\ldots,n\}}(z_i^x) \tag{3.3}$$

6. A continuación, de manera opcional, se puede modificar el grado de pertenencia, aumentándolo o disminuyéndolo, para cada individuo `?x`, usando un modificador difuso h:

$$\alpha_x = h(r_x) \tag{3.4}$$

7. El objetivo es conseguir los individuos con mayor grado de satisfacción de la consulta, por lo que el último paso es ordenar la lista de individuos y valores $\langle ?x, \alpha_x \rangle$ obtenida en los pasos anteriores, y ordenarla de manera decreciente de acuerdo al valor α_x.

Ejemplo 6. *Veamos cómo resolver la consulta del Ejemplo 4 dado el siguiente conjunto de triples que hablan sobre cervezas:*

```
@prefix ex: <http://www.example.org/beer/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
ex:ImperialPilsStrongPaleLager rdfs:subclassOf ex:Lager .
ex:PaleLager rdfs:subclassOf ex:Lager .
ex:Alhambra_Reserva_1925 rdf:type ex:SpanishBeer .
ex:Alhambra_Reserva_1925 rdf:type ex:ImperialPilsStrongPaleLager .
ex:Alhambra_Reserva_1925 ex:alcohol "6.4"^^xsd:decimal.
ex:Alhambra_Reserva_1925 ex:bitterness "25"^^xsd:decimal .
ex:Ambar_Especial rdf:type ex:SpanishBeer .
ex:Ambar_Especial rdf:type ex:PaleLager .
ex:Ambar_Especial ex:alcohol "5.2"^^xsd:decimal .
ex:Ambar_Especial ex:bitterness "25"^^xsd:decimal .
ex:BrewDog_Punk_IPA ex:rdf:type ex:IndiaPaleAleIPA .
ex:BrewDog_Punk_IPA ex:alcohol "6"^^xsd:decimal .
ex:BrewDog_Punk_IPA ex:bitterness "60"^^xsd:decimal .
```

En primer lugar, se generaría la siguiente consulta SPARQL:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ex: <http://www.example.org/beer/> .
SELECT ?x ?Y1 ?Y2
WHERE {
    ?x rdf:type/rdfs:subClassOf* ex:Lager .
    ?x rdf:type/rdfs:subClassOf* ex:SpanishBeer .
    ?x ex:alcohol ?Y1 .
    ?x ex:bitterness ?Y2 .
}
```

Como resultado, se obtienen las cervezas Alhambra_Reserva_1925 y Ambar_Especial, ya que las dos son cervezas españolas y pertenecen a una subclase del estilo Lager, junto a sus valores de alcohol y acidez. Tras esto, el algoritmo calcula z_1 *y* z_2*, y después se calcula* $\alpha_x = (z_1 \cdot z_2)^2$*. Los valores obtenidos serían:*

?x	*?Y1*	*?Y2*	*z1*	*z2*	α_x
Alhambra_Reserva_1925	6,4	25	0,04	0,83	0,001
Ambar_Especial	5,2	25	0,39	0,83	0,11

Finalmente, el resultado es la siguiente lista de pares ordenados:

$\langle$*Ambar_Especial,* 0,11$\rangle$
$\langle$*Alhambra_Reserva_1925,* 0,001$\rangle$

La ventaja principal de este algoritmo es que es posible reutilizar el lenguaje estandar RDF y los endpoints (puntos de consulta) SPARQL, de manera similar a lo que los autores en [12, 13] hacen para ontologías difusas.

Observemos que es trivial extender la caracterización de la consulta para incluir un parámetro adicional k de forma que solo se devuelven los top-k resultados (los individuos con los k mayores valores de α_x).

También cabe tener en cuenta que algunos operadores de fusión y modificadores difusos pueden expresarse usando las funciones de SPARQL (o expresiones internas), permitiendo realizar más cálculos sin depender de procesos externos. Sin embargo, la consulta SPARQL necesaria podría ser bastante compleja y hay algunos operadores que no pueden ser expresados en SPARQL estándar, como por ejemplo la media geométrica (que involucra raíces n-ésimas).

Capítulo 4

Implementación

Tras explicar la teoría en el apartado anterior, ahora se va a hablar de cómo se ha implementado el algoritmo, entrando más en profundidad en los problemas que han podido surgir y qué soluciones se han aplicado para resolverlos.

La implementación, llamada *Graph Consultant*, se ha desarrollado en Java usando algunas librerías externas:

- Las consultas SPARQL se han resuelto utilizando Apache Jena y la API de Jena Java[1] a través de un endpoint SPARQL, local o remoto.

- Las definiciones de los tipos de dato difusos se han parseado, para almacenar el tipo de función y sus parámetros, utilizando la API de Fuzzy OWL 2[2].

- Para usar autocompletado en la interfaz de usuario, se ha usado la biblioteca AutoCompleter[3].

4.1. Inicialización

El primer paso para que las consultas funcionen es verificar que existe un grafo de conocimiento al que acceder para poder continuar con el proceso. Para ello, dada la URL del endpoint del grafo de conocimiento, se ejecuta una consulta SPARQL de tipo ASK pero sin ninguna condición, es decir, se busca cualquier triple que devuelva la consulta.

```
ASK {?a ?b ?c}
```

Si la respuesta a la consulta es true, se continúa con el algoritmo.

[1] http://jena.apache.org

[2] http://webdiis.unizar.es/~fbobillo/fuzzyOWL2

[3] http://serprogramador.es/autocompletar-en-java-swing/

El siguiente paso es obtener las clases sobre las que se realizarán las posteriores consultas. Para ello se hace uso del término http://www.w3.org/2002/07/owl#Class que identifica las clases siguiendo el estándar OWL 2 mediante un triple RDF. Por eso el triple que buscamos es:

```
?class rdf:type owl:Class
```

De esta manera, se obtiene mediante la variable ?class la lista con todas las clases definidas en el grafo de conocimiento.

Ejemplo 7. *Por ejemplo, realizando esta consulta al grafo que existe en la DBpedia, se obtiene una lista de* 1363 *clases distintas:*

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix owl: <https://www.w3.org/2002/07/owl#>
SELECT DISTINCT ?class
WHERE {
  ?class rdf:type owl:Class .
}
```

Una vez obtenidas las clases, se da la opción al usuario para que elija sobre cuáles va a querer realizar sus consultas. En vez de dejar al usuario seleccionar un concepto del grafo de conocimiento, lo que requiere mostrar al usuario todas las opciones posibles y podría ser abrumador en un grafo de conocimiento de gran escala donde podría haber miles de opciones para cada tipo, nos parece más apropiado usar un mecanismo de auto completado. Además de las clases, esto aplica también al caso de propiedades y tipos de dato.

Tras realizarse la elección de las clases, se procede a obtener las propiedades que tienen esas clases. Hay que tener en cuenta también que no todas las propiedades poseen las necesidades básicas para ser procesadas y obtener un valor que represente el grado de pertenencia de las instancias a esa propiedad. La necesidad básica que necesitamos es que las propiedades sean funcionales y sus valores sean propiedad sean numéricos.

Para no hacer muy pesada la consulta, ya que para cada propiedad puede haber muchos valores distintos, se hace una agrupación de la propiedad y uno de sus valores con un GROUP BY, que tiene el mismo funcionamiento que en SQL. Cabe destacar que, si se hace la consulta para más de una clase, es necesario buscar las propiedades que ambas clases tengan en común.

Ejemplo 8. *Por ejemplo, para recuperar las propiedades de las instancias de la clase* dbo:Aircraft *de la DBpedia, usamos la siguiente consulta SPARQL:*

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix dbo:<http://dbpedia.org/ontology/>
SELECT DISTINCT ?property MIN(?valor)
WHERE {
  ?instance rdf:type dbo:Aircraft .
```

```
  ?instance ?property ?valor
}
GROUP BY ?property
```

Tras obtener los resultados, se comprueba en el código de Java que los valores sean numéricos con la función `parseDouble` de la clase `Double`. Los valores que no sean numéricos producirán la excepción `NumberFormatException` que se capturará en el código.

Tras obtener la lista con todas las propiedades, también se le da la opción al usuario para que elija las propiedades que le interesan consultar sobre la clase o las clases que haya elegido anteriormente. Para cada una de estas propiedades es necesario que se escoja también un tipo de dato difuso. Para simplificar la interfaz, para cada propiedad P, se le da a elegir entre cinco opciones distintas: *"Very High", "High", "Neutral", "Low"* y *"Very Low"*. La idea es buscar el tipo de dato difuso que resulta de concatenar la opción seleccionada con el fragmento de la propiedad y el programa comprobará que el tipo de dato difuso efectivamente existe.

Ejemplo 9. *Si el usuario selecciona la opción* "High" *para la propiedad* dbp:emptyWeighKg, *se buscará un tipo de dato difuso* HighemptyWeighKg.

Además de escoger las clases, propiedades y tipos de dato, también es necesario elegir el operador de fusión, es decir, la manera que se combinaran los grados de pertenencia de los valores de las propiedades a los tipos de dato difuso (si hay más de una propiedad en la consulta). Para este caso también se puede elegir entre cinco distintas opciones: *"Producto", "Mínimo", "Máximo", "Media ponderada"* y *"OWA"*.

El producto, el mínimo y el máximo, funcionan de manera que todas las propiedades tienen el mismo peso en la evaluación del grado de pertenencia total. En el caso del producto, se multiplican entre sí todos los valores de pertenencia de las distintas propiedades, obteniendo así un número en el rango [0,1]. Para los casos de mínimo y máximo, como los nombres indican, se toma para cada individuo el valor de pertenencia al conjunto difuso más bajo para el mínimo y el más alto para el máximo.

Por último, las opciones de media ponderada y OWA hacen uso de pesos para las propiedades, por lo que el usuario debe especificarlos de alguna manera. Para simplificar la interfaz de usuario, se ha optado por diseñar métodos que impidan que el usuario deba especificar un peso para cada propiedad.

En el caso de la media ponderada, el usuario puede elegir qué propiedades son sus preferidas. Estas propiedades tendrán un peso que duplicará el valor del peso de las propiedades que no estén catalogadas como preferidas. Por ejemplo, si se han elegido tres propiedades y una de ellas es preferida esta tendrá un valor de 0,5 y la otras dos de 0,25. Si, en cambio, dos son preferidas y una no, las preferidas tendrán un peso de 0,4 y la restante un peso de 0,2. En general, si hay n propiedades de las cuales hay m favoritas, las propiedades favoritas tendrán un peso $\frac{2}{n+m}$, mientras que las demás tendrán un peso $\frac{1}{n+m}$.

En el caso de OWA, se permite seleccionar un tipo de dato difuso, se comprueba que cumple las condiciones necesarias para ser un cuantificador difuso y se calculan los pesos usando la Ecuación 2.1.

Además de estas opciones, el usuario también puede elegir un modificador difuso. Por defecto, el valor será *"None"* y no se hará nada, pero se dan dos opciones adicionales, *"Higher"* o *"Lower"* que ampliarán, mediante una elevación al cuadrado, o reducirán, mediante la raíz cuadrada, el grado de satisfacción de la consulta, tal y como se explicó en la Sección 2.3.

También se permite al usuario elegir el máximo número de instancias que devolverá el algoritmo. Estas instancias serán las que tengan un mayor grado de satisfacción de la consulta.

4.2. Algoritmo

Tras todos estos pasos, el usuario ha elegido los parámetros necesarios para que el algoritmo funcione correctamente. Una recapitulación de estos parámetros es:

- Un conjunto de clases
- Un conjunto de propiedades numéricas
- Una lista de tipos de dato difusos
- Un operador de fusión
- Un modificador difuso

Todos estos parámetros han sido mencionados en el apartado anterior, salvo la lista con los tipos de dato difusos ya que esta lista es procesada en el segundo paso del algoritmo y no depende del usuario.

Para poder consultar grafos de conocimiento que no incorporen tipos de dato difuso, se decidió crear una ontología para este propósito, que se almacena como un fichero local. Concretamente, el fichero local se almacena en los ficheros del programa con ruta `‘‘src/tfg/ontology.owl’’`. Esta ontología es muy sencilla, solo cuenta con los `DataProperty` y con sus respectivos `Datatype`. Esos `Datatype` son los que el usuario puede elegir a través de la interfaz.

Se quería buscar una manera sencilla de guardar la información para poder acceder rápidamente a las funciones de pertenencia, por lo que se decidió procesar la ontología y crear una lista de funciones de pertenencia. La ontología se maneja usando la OWL API, que nos permite leer el fichero y crear un objeto de tipo `OWLOntology` con el cual podemos trabajar.

Con la ontología cargada, podemos acceder a los axiomas de tipo `Datatype definition`. Una vez recuperados los valores de sus anotaciones fuzzyLabel, se parsean los valores XML y se almacena en un vector la información extraída.

Ejemplo 10. *Consideremos el datatype* `LowAlcohol`*, definido en el Ejemplo 5, con la anotación:*

```
<Datatype type=\"triangular\" a=\"0.5\" b=\"3.1\" c=\"6.55\" />
```

La información que se almacenaría en nuestro vector sería la siguiente:

```
LowAlcohol, tri, 0.5, 3.1, 6.55
```

Es decir, primero se almacena el tipo de dato difuso, luego el tipo de función de pertenencia, y luego los parámetros de la función de permanencia. En este caso hay tres parámetros, por ser una función triangular, pero el número de estos argumentos puede cambiar según el tipo de función.

Para almacenar la información sobre las funciones de pertenencia en el programa, se ha utilizado una clase padre llamada `FuzzyConcreteConcept` que declara un método abstracto `Double getMembershipDegree (double valor)`. A partir de esta clase, se extiende una subclase para cada tipo de función, trapezoidal, triangular, left-shoulder y right-shoulder, las cuales implementarán el código para calcular el grado de pertenencia a la función de un parámetro de entrada (un número real en [0,1]) de acuerdo con su definición. Este código dependerá de los parámetros, obtenidos de la ontología.

Tras estos pasos, ya se tiene toda la información necesaria para hacer las consultas al grafo de conocimiento con el objetivo de extraer a los individuos que pertenecen a las clases.

Ejemplo 11. *La siguiente consulta permite recuperar las instancias de las clases* `Aircraft` *y* `Automobile` *y los valores para las propiedades* `emptyWeightKg` *y* `maxSpeedKmh`*:*

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix dbpedia-owl: <http://dbpedia.org/ontology/>
prefix dbr:<http://dbpedia.org/resource/>
prefix dbo:<http://dbpedia.org/ontology/>
prefix dbp:<http://dbpedia.org/property/>
prefix dbt:<http://dbpedia.org/resource/Template:>
prefix dbc:<http://dbpedia.org/resource/Category:>
prefix dct:<http://purl.org/dc/terms/>
SELECT DISTINCT ?busqueda ?X ?Y
WHERE {
 { ?busqueda rdf:type dbo:Aircraft;
             dbp:emptyWeightKg ?X;
             dbp:maxSpeedKmh ?Y }
UNION
 { ?busqueda rdf:type dbo:Automobile;
```

```
                dbp:emptyWeightKg ?X;
                dbp:maxSpeedKmh ?Y }
}
```

Hay que tener en cuenta que puede ser que el grafo de conocimiento tenga algunos valores erróneos para algún individuo en alguna propiedad, por lo que hay que comprobar que todos los valores obtenidos sean numéricos usando el mismo método descrito anteriormente. Un ejemplo de lo que puede pasar en alguna propiedad sucede en la propiedad `aspectRatio`, la cual además del valor numérico correcto tiene otro valor vacío, como se puede apreciar en la Figura 4.1.

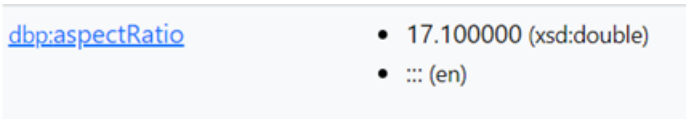


Figura 4.1: Doble valor en propiedad aspectRatio.

Tras esto, la información se almacena en un diccionario de la siguiente manera. Como clave del diccionario se utiliza el nombre de los individuos y como valor se utiliza otro diccionario. Este segundo diccionario tiene como clave el nombre de la propiedad y, como valor, el valor numérico de la propiedad para ese individuo.

Ejemplo 12. *Para el individuo* `Nieuport-Macchi_Parasol` *y las propiedades* `HighmaxSpeedKmh` *y* `NeutralemptyWeightKg` *tendríamos:*

```
{ Nieuport-Macchi_Parasol={HighmaxSpeedKmh=125,
NeutralemptyWeightKg=400}, ... }
```

Una vez almacenados todos los individuos en el diccionario de diccionarios, se procede a utilizar los valores correspondientes a cada propiedad con las funciones obtenidas anteriormente llamando al método `getMembershipDegree(double valor)` y utilizando como parámetro el valor de la propiedad. Tras procesar todos los individuos, como resultado se obtiene otro diccionario de diccionarios con la misma estructura, pero con el grado de pertenencia sustituyendo al valor en crudo.

Ejemplo 13. *Continuando el Ejemplo 12, tendríamos:*

```
{ Nieuport-Macchi_Parasol={HighmaxSpeedKmh=0.0,
NeutralemptyWeightKg=0.5736842105263158}, ... }.
```

En este punto necesitamos transformar el diccionario de diccionarios en un solo diccionario donde la clave será el individuo y el valor el grado de pertenencia al conjunto.

Para ello necesitamos los operadores de fusión los cuales están implementados cada uno en un método distinto, de manera que se ejecutará el correspondiente en función de la elección del usuario. Como se ha mencionado anteriormente hay cinco tipos diferentes de funciones implementadas el máximo, el mínimo, el producto, la media ponderada y OWA.

El funcionamiento general de los métodos es el mismo, se recorre el diccionario principal y por cada valor se recorre el diccionario interno aplicando el algoritmo correspondiente en cada individuo, insertando la información obtenida en un nuevo diccionario. Hay que tener en cuenta que, antes de recorrer los diccionarios, se calculan los pesos necesarios para las funciones de la media ponderada y OWA.

Ejemplo 14. *Tras aplicar la t-conorma del producto se obtendría un diccionario de la forma:*

```
{ Nieuport-Macchi_Parasol=0.5736842105263158, ... }.
```

Añadir nuevos operadores de fusión (por ejemplo, la suma acotada) es relativamente sencillo, solo habría que implementar una nueva función que se encargue de recorrer el diccionario y aplicar el cálculo específico (por ejemplo, la suma) necesario para fusionar los distintos valores de cada propiedad.

Tras esto, si el usuario lo he escogido, se aplica un modificador del grado de pertenencia. Esto tiene un funcionamiento más sencillo que los operadores de fusión, ya que ahora solo tenemos que recorrer un diccionario y aplicar la operación correspondiente.

Ejemplo 15. *Continuando el Ejemplo 14 con el modificador very:*

```
{ Nieuport-Macchi_Parasol=0.3291135734072022, ... }.
```

De la misma forma que con los operadores de fusión, añadir más modificadores del grado de pertenencia se haría implementando alguna función adicional que se encargue de realizar el cálculo correspondiente para cada valor del diccionario.

Por último, necesitamos ordenar el diccionario obtenido en el paso anterior de mayor a menor en función del valor que tiene cada individuo y devolver los n valores que tengan mayor grado, siendo n el valor elegido por el usuario.

4.3. Interfaz gráfica

Para facilitar al usuario el uso del algoritmo, se ha creado una interfaz gráfica con el objetivo de que sea intuitiva y sencilla de usar. Se ha escogido utilizar Java Swing para el desarrollo, una biblioteca que implementa una serie de componentes gráficos como botones, listas, desplegables, etc., que permiten construir interfaces gráficas y añadir funcionalidad e interactividad con el usuario para aplicaciones Java.

El programa comienza con una pequeña ventana donde el usuario puede escribir la dirección del endpoint donde se van a realizar las consultas SPARQL. Por defecto, se muestra la URL del endpoint de la DBpedia, "https://dbpedia.openlinksw.com/sparql".

Cuando el usuario presiona el botón *"Aceptar"*, se realiza la consulta `ASK` que comprueba si hay triples en el grafo de conocimiento. Si no los encuentra, da un mensaje de error y no pasa a la siguiente pantalla, como se muestra en la Figura 4.2.

Figura 4.2: Mensaje de error cuando se introduce un endpoint incorrecto

Antes de cargar la siguiente pantalla, se realiza la consulta SPARQL explicada anteriormente que devuelve una lista con todas las clases disponibles en el grafo de conocimiento. Al principio se planteó utilizar un desplegable para que el usuario seleccionase las clases sobre las que quiere trabajar pero, si la consulta devuelve un número demasiado grande de clases, este elemento sería bastante tedioso y frustrante. Por ejemplo, al consultar sobre la DBpedia sería insostenible tener un desplegable con más de 1000 elementos sobre los que elegir cómodamente, por lo que se decidió utilizar un elemento `JTextField` incluido dentro de la biblioteca de Java Swing.

`JTextField` es un elemento básico en el cual el usuario puede escribir lo que quiera. Aun así, no es suficientemente cómodo para manejar un gran número de opciones, porque eso implicaría que el usuario tiene que conocer previamente las URLs de las clases existentes para poder escribirlas. Por este motivo, la decisión final ha sido añadirle a este `JTextField` un elemento de autocompletado, de esta manera es como si se combinase el campo en blanco para escribir con el menú desplegable.

Para implementar el autocompletado se ha utilizado la biblioteca *AutoCompleter* para Java Swing, que se puede instalar fácilmente incluyendo el jar `AutoCompleter.jar` al proyecto en el que se va a utilizar. Su uso también es sencillo: nos da la opción de crear una clase llamada `TextAutoCompleter` que toma dos argumentos, el `JTextField` que va a utilizar para situarse en la pantalla y para poder escribir en él, y una lista de `Strings` que serán los elementos que puede autocompletar (en nuestro caso, las clases o las propiedades del grafo).

Ejemplo 16. *La Figura 4.3 muestra las clases de DBpedia que aparecen escribiendo la letra "a", es decir, las que comienzan por "A" o "a".*

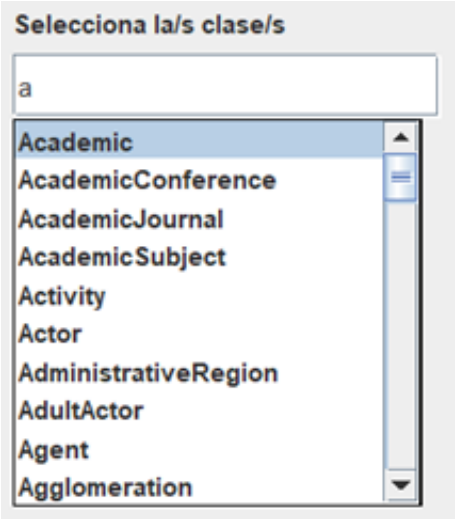


Figura 4.3: Ejemplo de autocompletado

Cuando el usuario entra a esta ventana, solo dispone de un elemento de autocompletado, explicado anteriormente, y dos botones, como se ve en la Figura 4.4. Uno tiene la utilidad de pasar a la siguiente fase del algoritmo, y el otro, con un icono "+", sirve para poder añadir más clases. La Figura 4.5 muestra cómo se usa autocompletado para seleccionar una segunda clase tras haber añadido previamente la primera.

Teóricamente, no hay un límite máximo de clases que el usuario puede escoger para realizar las consultas, pero desde el punto de vista de la interfaz no es intuitivo ni ordenado. Por este motivo, se ha decidido fijar un máximo de tres clases que el usuario puede elegir, como se ve en la Figura 4.6. Para limitar el uso del botón "+", se desactiva y ya no se puede pulsar para añadir más clases después de dos usos,

`JButton` es la clase que nos proporciona Java Swing para poder implementar botones. Esta clase puede tomar un parámetro de distintos tipos como un `String` o un `ImageIcon/Icon`. Hay que tener en cuenta que, si se quiere pasar un icono como parámetro, habrá que reescalarlo o de lo contrario no se ajustará a las medidas del botón y saldrá cortado. Conviertiendo el tipo `ImageIcon` o `Icon` a tipo `Image`, se puede aplicar el método `getScaledInstance`, que permite escalarlo al tamaño deseado. Para poder acceder a la imagen del icono guardada en el proyecto, se ha hecho uso de la clase `Toolkit`, que permite acceder a un fichero con una ruta relativa y convertirlo a una imagen.

Para que este elemento reaccione ante un click, hay que añadirle un escuchador (o *listener*). Esto se hace pasando el listener como parámetro mediante el método de la clase JButton llamado `addActionListener(listener)`. Para crear el listener, es necesario crear una clase que extienda `JFrame` e implemente `ActionListener`. De esta manera, se puede sobreescribir el método `actionPerformed`, el cual es invocado cuando se realiza un click en el botón. Este método también obtiene un parámetro de tipo `ActionEvent` para que se puedan controlar distintos tipos de acciones sobre el botón.

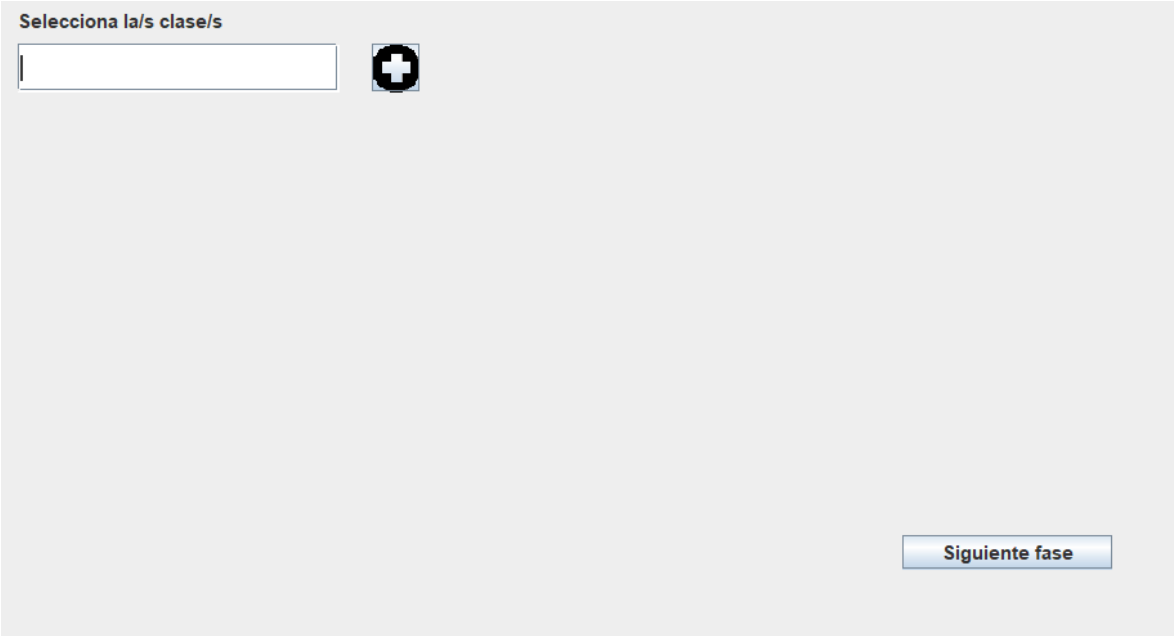


Figura 4.4: Pantalla principal vacía con el autocompletado y los dos botones

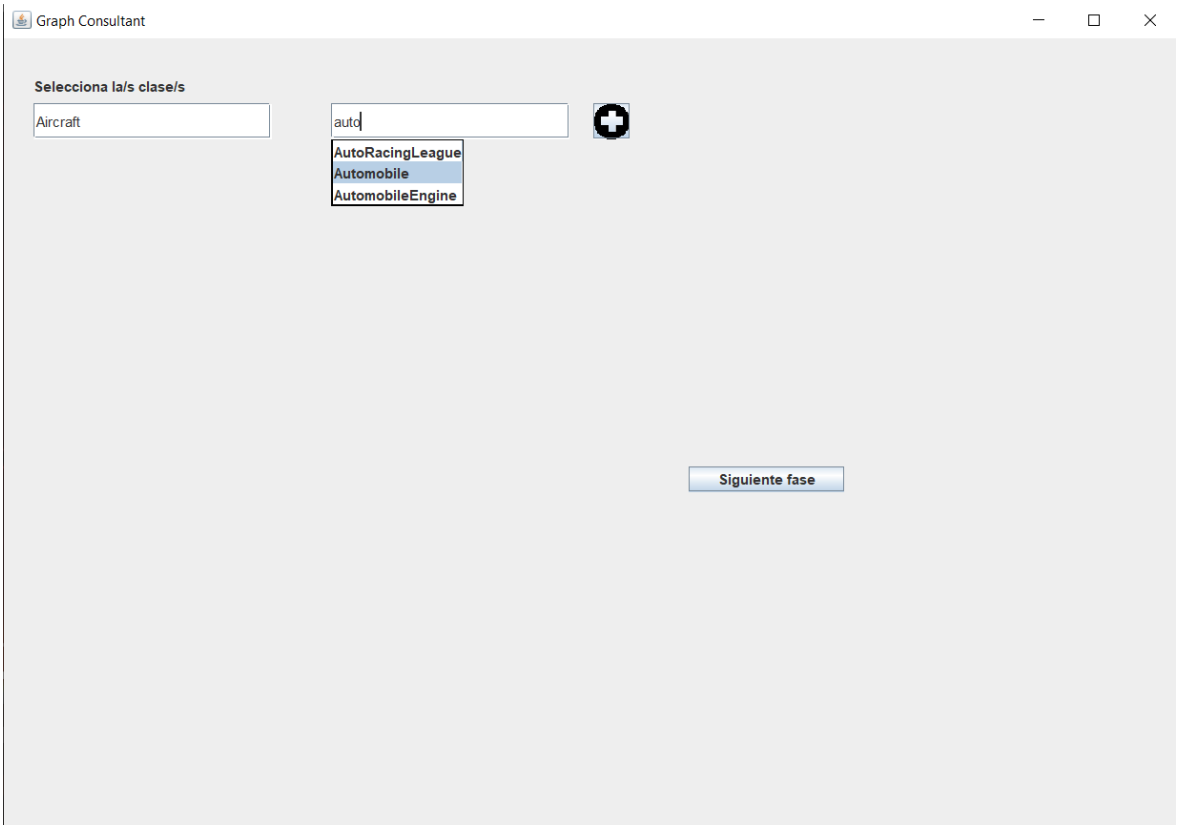


Figura 4.5: Creación de una segunda clase usando autocompletado

Una vez el usuario haya elegido las clases que crea oportunas, puede clickar en el botón

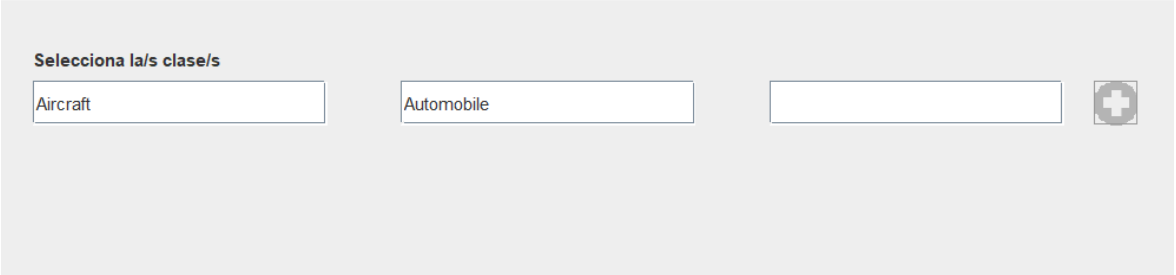


Figura 4.6: Máximo de clases

"Siguiente fase". Esto hará que ya no pueda realizar modificaciones sobre las clases escogidas y se ejecutará la consulta SPARQL que devuelve las propiedades de esas clases. Al mismo tiempo este botón desaparece para dar lugar a otro botón que da la opción para añadir más propiedades.

De las misma forma que se ha limitado el número de clases que el usuario puede escoger, también se ha hecho con las propiedades que puede escoger pero, en vez de tres, se pueden escoger cuatro propiedades. Se ha puesto un número mayor al anterior porque tiene más sentido que el usuario quiera realizar consultas con más profundidad (más propiedades) sobre un conjunto más pequeño de clases que consultas con muchas clases pero con poco sentido al no haber suficientes propiedades comunes.

Cuando el usuario pulsa el botón para añadir propiedad por primera vez, se crean distintos elementos. Para elegir la propiedad se utiliza el mismo método de autocompletado explicado anteriormente, adicionalmente el usuario necesita poder elegir la restricción flexible asociada a esa propiedad. Como solo se da a elegir entre cinco opciones, aquí sí que es apropiado utilizar un desplegable para que el usuario no tenga que escribir y así minimizar errores. Este desplegable se crea fácilmente con una instancia de la clase `JComboBox` de Java Swing, como se ve en la Figura 4.7.

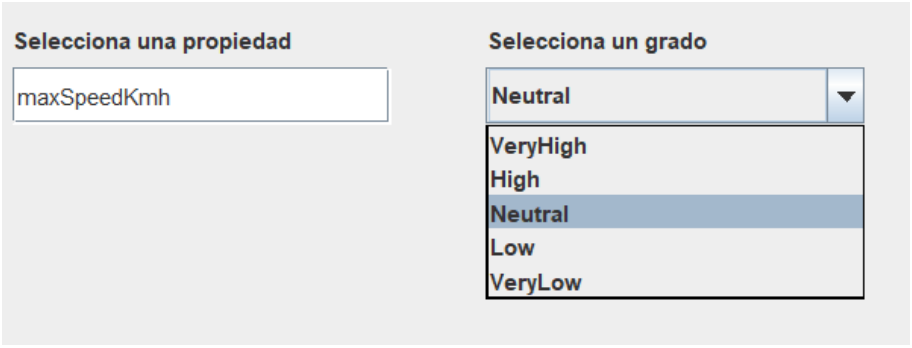


Figura 4.7: Ejemplo de selección de restricción flexible sobre una propiedad

Como se ha explicado en los apartados anteriores, hay más opciones que el usuario debe rellenar (ver Figura 4.8): el operador de fusión y el modificador difuso. Las opciones

entre las que tiene que decidir el usuario están acotadas, por lo que también se ha utilizado la clase `JComboBox` para estos casos. Además, también se ha creado un campo de tipo `JFormattedTextField` en el que el usuario puede rellenar con hasta 4 dígitos para seleccionar el máximo número de respuestas que quiere recibir. Esto se consigue gracias a la clase `numberFormatter` que podemos pasar como parámetro al crear el campo de texto. Gracias a la clase `numberFormatter` y su método `setValueClass()` podemos indicar el tipo de dato que se puede escribir, en este caso `Integer.class`, y, gracias al método `setAllowsInvalid()`, no permitimos que se inserte información errónea. Para limitar el número de caracteres a introducir es necesario introducir un `KeyAdapter` al campo de texto e implementar el método `keyTyped`. En este método se mide la longitud del valor almacenado en el campo de texto y, en caso de ser mayor o igual a cuatro, se consume la entrada producida haciendo que no haya ningún efecto.

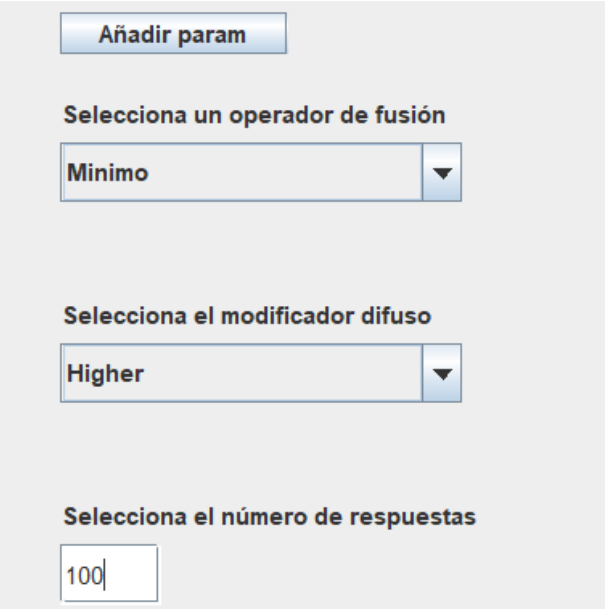


Figura 4.8: Opciones adicionales

Los operadores de fusión *"Media ponderada"* y *"OWA"* necesitan asignar unos pesos a sus propiedades, por lo que también hay que implementar algún método para que el usuario pueda elegirlos. Pensamos que elegir directamente el peso de cada propiedad puede ser difícil para el usuario medio y es poco escalable si queremos aumentar el número de propiedades, por lo que se ha buscado una solución alternativa.

- Para elegir los pesos de la media ponderada, se ha implementado un icono de una estrella para cada propiedad, que aparece cuando el usuario elige el método de *"Media ponderada"* en el desplegable. El icono tiene dos estados: falso que se representa con el contorno de la estrella y verdadero, que se representa con el interior de la estrella de color amarillo (ver Figura 4.9).

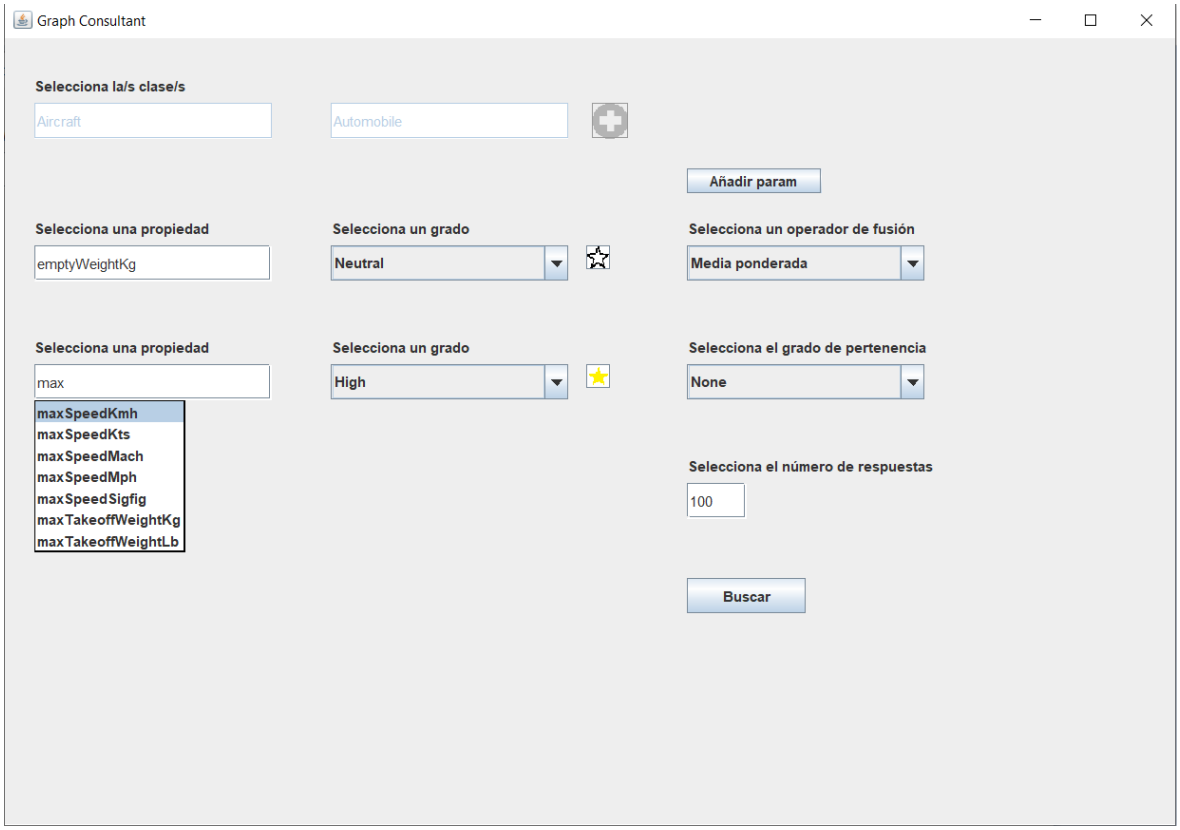


Figura 4.9: Ejemplo completo de la pantalla tras seleccionar todos los parámetros necesarios

– Para el método *"OWA"*, se ha implementado otro desplegable que aparece al lateral cuando el usuario elige esta opción. En el desplegable aparecen los tipos de dato que en la ontología tienen el prefijo "Fuzzy" y se asignan los pesos mediante agregación guiada por cuantificadores, como se explicó en aparados anteriores. Para que el texto que aparece en el desplegable tenga un nombre más legible por el usuario, se ha creado un método abstracto en la clase `FuzzyConcreteConcept` llamado `getName()` el cual está implementado en las subclases. Este método devuelve un `String` con el tipo de la función y los parámetros que tiene. Por ejemplo: "right-shoulder(0.25, 0.75)" donde right-shoulder es el tipo de la función y 0,25 y 0,75 sus dos parámetros.

Ejemplo 17. *La Figura 4.9 muestra cómo la interfaz permite resolver la siguiente consulta:*

– *Clases:* Aircraft, Automobile

– *Propiedades:* emptyWeightKg, maxSpeedKmh

– *Tipos de datos difusos:* NeutralemptyWeightKg *y* HighmaxSpeedKmh

– *Operador de fusión: media ponderada, con vector de pesos* [0,33,0,67]

– *Modificador difuso: ninguno*

Por último, en la parte inferior de la pantalla se encuentra el botón *"Buscar"*. Cuando el usuario presiona este botón, se ejecuta el algoritmo y se despliega una nueva ventana donde aparecerán los individuos devueltos por el algoritmo (ver Figura 4.10). Los resultados se filtran para mostrar únicamente los individuos que tienen un grado de cumplimiento de la consulta no nulo. Para poder mostrar los resultados en esta ventana con comodidad si el número de individuos devueltos era demasiado grande, se ha utilizado la clase de Java Swing llamada `JScrollPane`. Esta clase es utilizada para que, si el número de individuos rebasa el límite inferior de la pantalla el usuario, se pueda realizar un scroll vertical. Además, como el algoritmo devolvía unos pesos con demasiados decimales, también se han redondeado los grados de cumplimiento de la consulta a dos decimales para mejorar la claridad de los resultados. En el caso de valores inferiores a una centésima, se ha preferido no redondear.

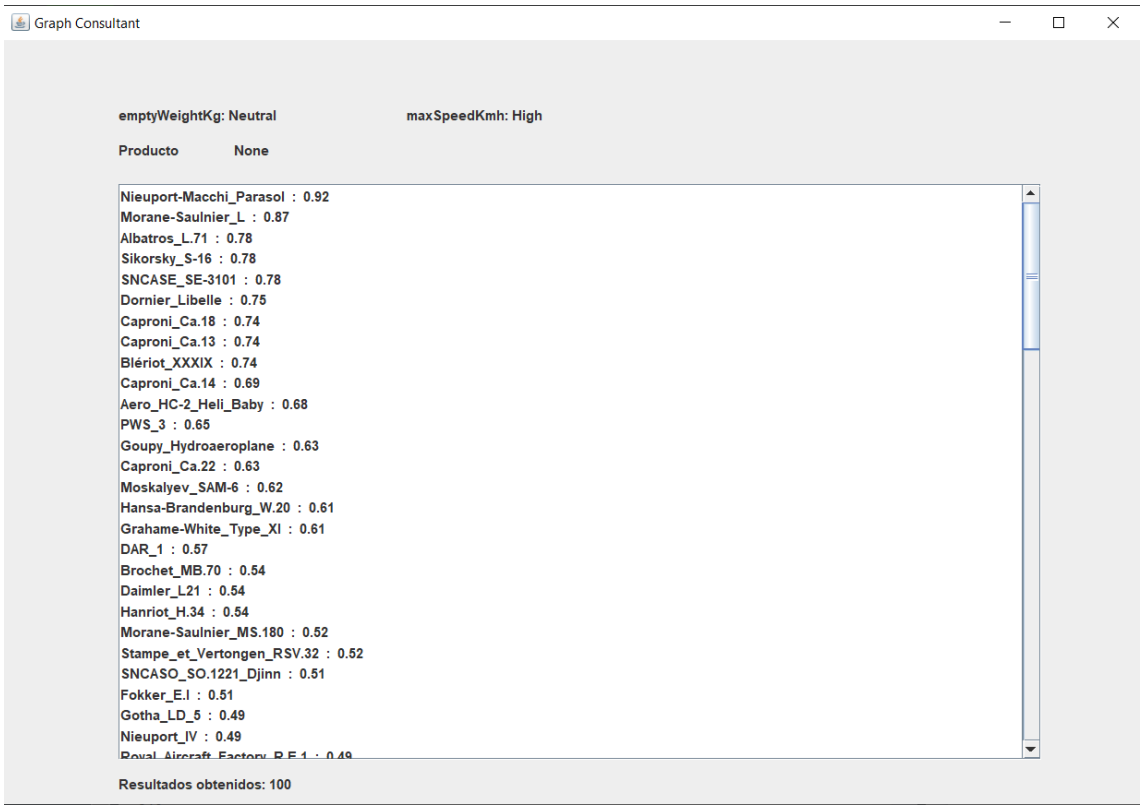


Figura 4.10: Ejemplo de la pantalla de resultados

Capítulo 5

Caso de uso

En este apartado de la memoria se va a hablar de un caso de uso para la aplicación, para ello se va a usar la DBpedia como grafo de conocimiento y se van a buscar medios de transporte rápidos (en cuanto a velocidad máxima y de crucero) y de peso medio.

Se ha seleccionado MeanOfTransportation como la clase sobre las que realizar las consultas, en la que se encuentran miles de individuos que entran dentro de distintas categorías de transportes. Además, se usarán las propiedades maxSpeedKmh, emptyWeightKg y cruiseSpeedKmh como propiedades. A continuación se pueden ver la definición de algunos tipos de dato difusos que restringirán los posibles valores de las propiedades:

```
HighmaxSpeedKmh:
 <Datatype type=\"triangular\" a=\"750\" b=\"1500\" c=\"2000\" />
NeutralemptyWeightKg:
 <Datatype type=\"triangular\" a=\"1000\" b=\"2000\" c=\"4100\" />
HighcruiseSpeedKmh:
 <Datatype type=\"triangular\" a=\"500\" b=\"1000\" c=\"6000\" />
```

Se van a utilizar distintos operadores de fusión para mostrar cómo esto puede afectar al resultado obtenido por las consultas.

En la primera consulta se usa el operador de fusión del producto. Como se realiza una multiplicación entre los grados de cumplimiento de las restricciones de las distintas propiedades, es necesario que todos sean mayores que cero para obtener un resultado no nulo. Este es el motivo por el que los valores finales son tan bajos y solo se obtienen 11 individuos a pesar de haber seleccionado un máximo de 100 se ha querido obtener 100, como se puede apreciar en la Figura 5.1.

Para la segunda consulta se ha cambiado el operador de fusión del producto por el de la media ponderada. No se ha elegida ninguna propiedad favorita, así que se obtiene un peso de $\frac{1}{3}$ para cada una. Esto hará que los valores obtenidos sean mayores que en el ejemplo anterior y que se devuelvan más individuos, ya que ahora no es necesario que todos los valores calculados para las propiedades sean mayores que 0, ni se obtienen valores tan bajos por multiplicar los

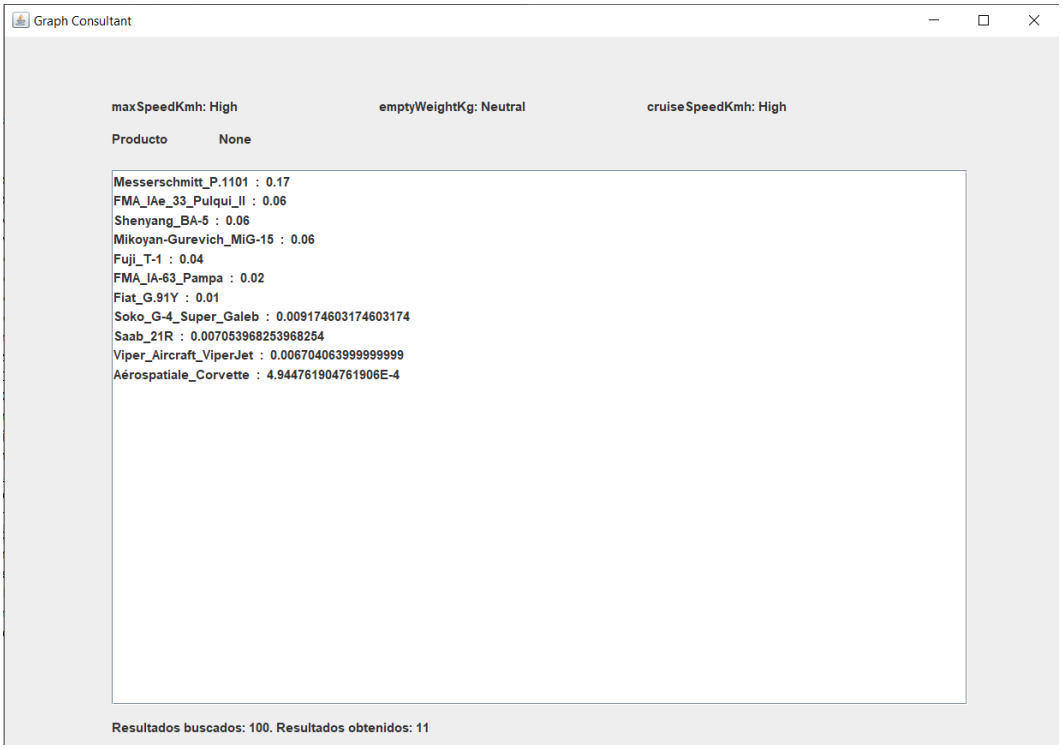


Figura 5.1: Resultados para el producto

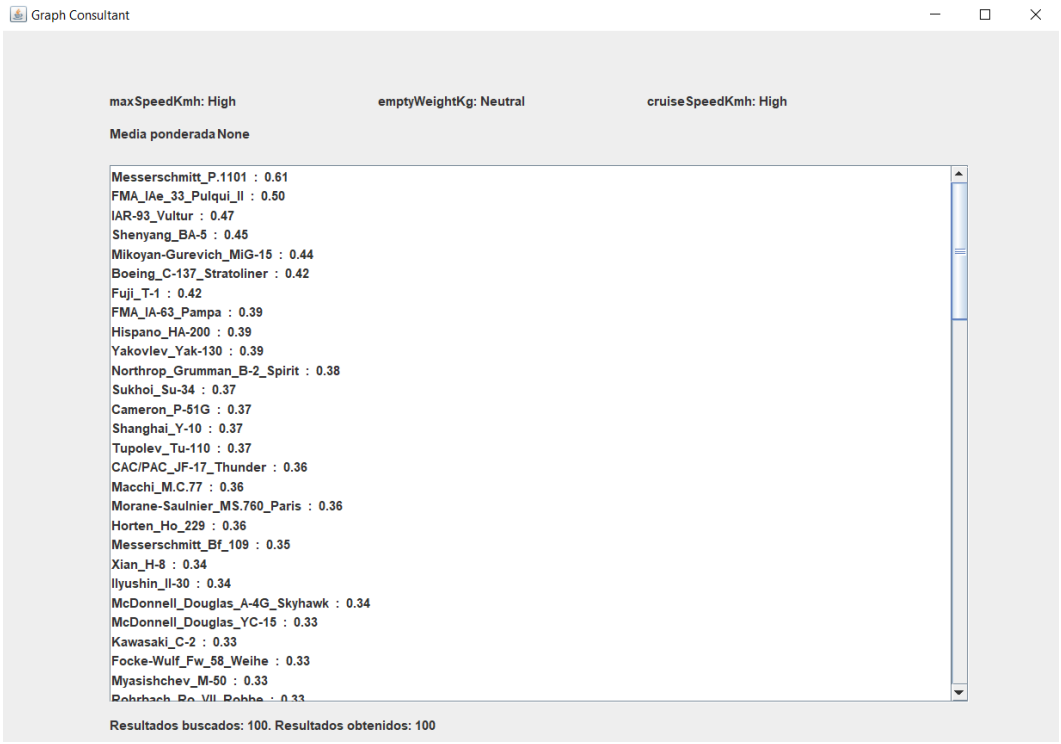


Figura 5.2: Resultados para la media ponderada

valores entre sí. Como se puede ver en la Figura 5.2, los dos primeros individuos obtenidos coinciden con el ejemplo anterior ya que, como todas las propiedades tienen valor superior a 0, es normal que la media de estas sea de las mayores. El tercer individuo ya no aparece en la consulta anterior, por lo que se puede intuir que una propiedad tiene valor 0 y las otras dos propiedades tienen un valor suficientemente alto como para ser el tercero con más puntuación.

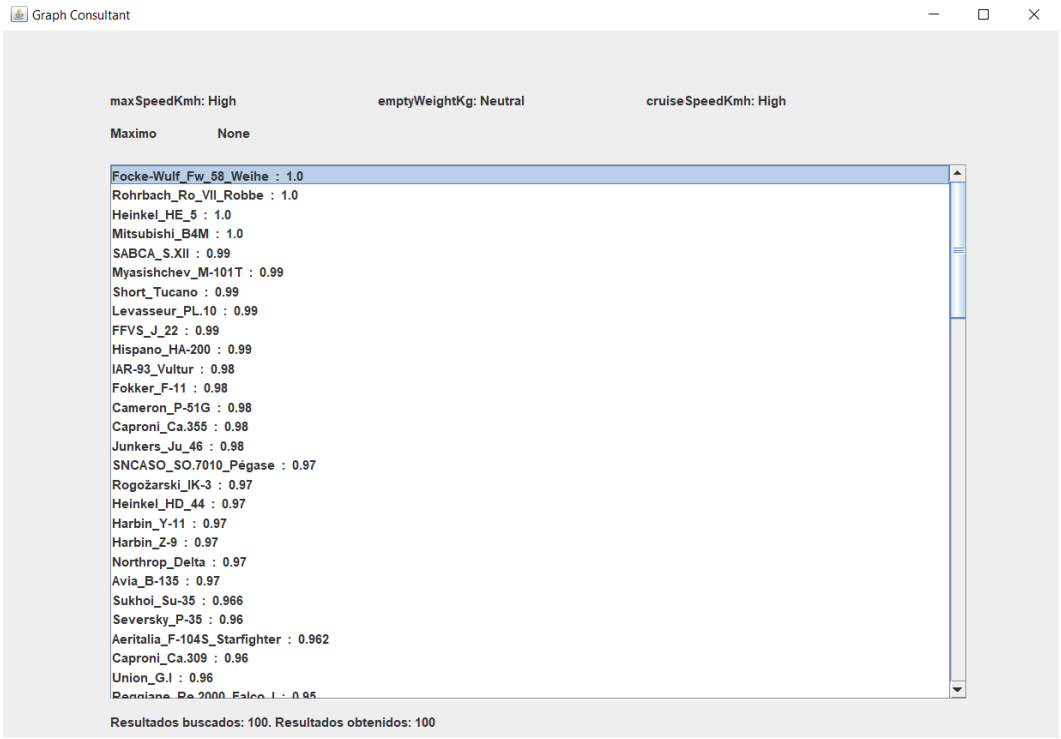


Figura 5.3: Resultados para el máximo

En el tercer ejemplo se ha utilizado el operador máximo. Cuantas más propiedades haya en la consulta, más probabilidad hay de que para cualquier individuo alguna de estas propiedades sea 1,0 o cercana a 1,0, por lo que en esta consulta los valores son mayores y los individuos difieren mucho de los ejemplos anteiores, como se puede ver en la Figura 5.3.

Para el último ejemplo, se ha cambiado el operador de fusión por uno de tipo *"OWA"*, con una función $right-shoulder(0{,}33333, 0{,}66666)$, por lo que los pesos asignados serán [0,1,0], dando toda la importancia al parámetro con el valor intermedio de entre los tres. En este caso, se obtienen como resultado los individuos que tienen unos valores más balanceados, ya que el menor y el mayor no se tienen en cuenta, esto hace que los resultados sean similares al caso de la media ponderada. Los resultados pueden verse en la Figura 5.4.

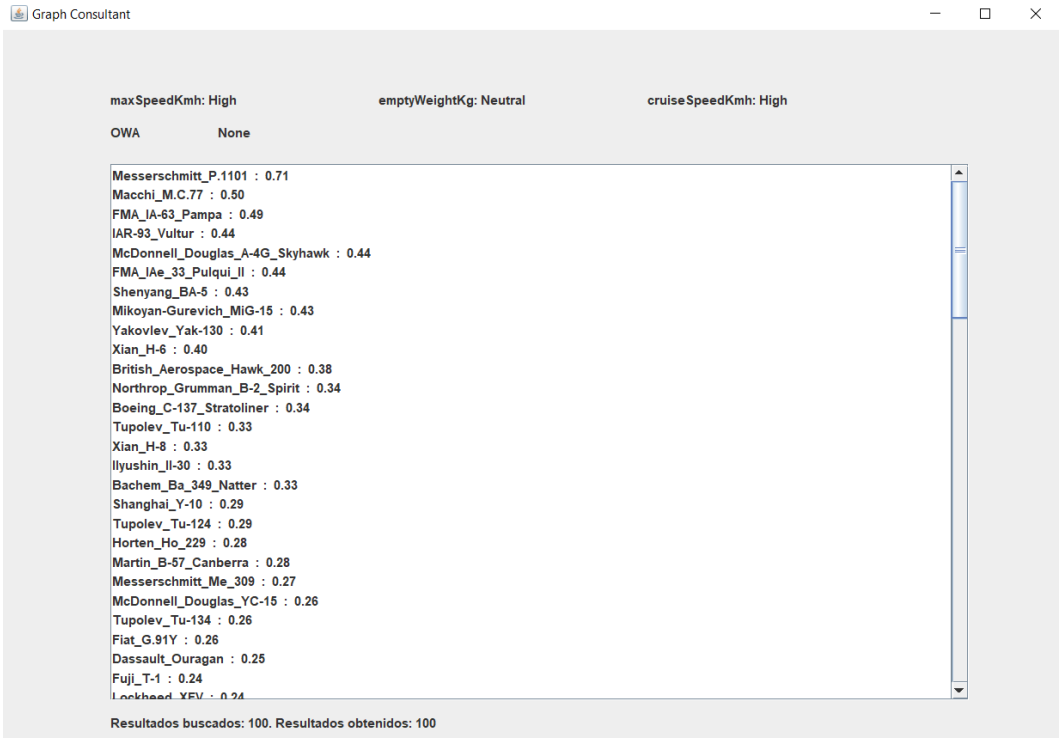


Figura 5.4: Resultados para OWA

Capítulo 6

Trabajo relacionado

Existen trabajos anteriores que han considerado lógica difusa en el lenguaje RDF. La propuesta más antigua se debe a Vaneková [14] et al., que propusieron representar un axioma difuso de la forma "un recurso s pertenece a un conjunto difuso f con un grado α", siendo $\alpha \in (0,1]$, usando un triple RDF de la forma $\langle s,f,\alpha \rangle$. Si bien es cierto que esta aproximación hace posible utilizar un único triple reutilizando el estándar RDF, hay ciertos tipos de relaciones que no se representan. Por ejemplo, los autores no representan el hecho de que un item002 es de tipo (usando rdf:type) barato, o el hecho de que el precio de item002 es barato.

M. Mazzieri y A. F. Dragoni consideran sentencias de la forma $\langle s,p,o,\alpha \rangle$ [15]. Esta idea está pensada para representar, por ejemplo, que un triple $\langle$Zaragoza,estaCercaDe,Huesca$\rangle$ tiene un grado de pertenencia de 0,7. Dependiendo de la propiedad p, se pueden tener diferentes tipos de axiomas; por ejemplo hechos sobre clases (rdf:type), subclases (rdfs:subclassOf), subpropiedades (rdfs:subPropertyOf) o, en general, hechos sobre propiedades.

A. E. A. Djebri recopila las diferentes aproximaciones para escribir sentencias de la forma sentencias de la forma $\langle s,p,o,\alpha \rangle$: reificación, propiedades n-arias, grados únicos con nombre, propiedades singleton o RDF-star [16]. Sin embargo, estas aproximaciones no tienen en cuenta tipos de dato difusos, como se está haciendo en este trabajo.

U. Straccia propuso un algoritmo de razonamiento para una extensión difusa del lenguaje ρDF (un fragmento de RDF-Schema[1]) con sentencias de la forma $\langle s,p,o,\alpha \rangle$ pero no tipos de dato difusos [17]. Además, propuso consultas conjuntivas difusas sobre un grafo difuso, que puede incluir triples difusos y asignaciones que involucren funciones de pertenencia difusas y operadores de fusión. Por ejemplo: $q(x,s) \leftarrow \langle x,rdf:type,SportsCar \rangle \wedge \langle x,hasPrice,y \rangle \wedge \{s := s1 \cdot cheap(y)\}$ [17]. Esta propuesta no detalla cómo representar la sintaxis de los tipos de dato difusos (en cambio, en este trabajo usamos tipos de dato Fuzzy OWL 2 que podrían están representados en el grafo RDF) y tampoco considera modificadores difusos. Nuestro desarrollo puede ser más amigable para el usuario, ya que no tiene la necesidad de tener que escribir consultas tan complejas como la mostrada anteriormente.

[1] https://www.w3.org/TR/rdf-schema/

J. Z. Pan et al. propusieron el lenguaje Fuzzy SPARQL, una extensión de SPARQL diseñada para realizar consultas difusas a ontologías (pero no a grafos de conocimiento difusos) [18] con sentencias de la forma $\langle s,p,o,\alpha\rangle$, pero que no considera tipos de dato difusos o modificadores difusos.

O. Pivert et al. también propusieron otra extensión de SPARQL para consultas difusas llamada "FURQL" (Fuzzy RDF Query Language) [19], permitiendo propiedades difusas (que pueden verificarse parcialmente) y tipos de dato difusos en consultas sobre un grado RDF difuso con sentencias de la forma $\langle s,p,o,\alpha\rangle$. Sin embargo, a pesar de que usan tipos de dato difusos en los ejemplos, no se detalla cómo se pueden representar en el grafo, no es posible especificar operadores de fusión y no soporta modificadores difusos.

Capítulo 7

Conclusiones y trabajo futuro

En este capítulo se describen las herramientas utilizadas, se resumen las principales conclusiones y se presentan algunas ideas de trabajo futuro.

7.1. Herramientas utilizadas

El lenguaje escogido para el desarrollo ha sido Java, ya que este lenguaje contiene todas las bibliotecas externas que necesitamos para poder realizar consultas SPARQL y para leer ontologías con facilidad.

Para desarrollar el código se ha utilizado principalmente Eclipse IDE for Java Developers en la versión 2020-09 (4.17.0). Eclipse es una herramienta fácil de usar y que facilita enormemente la gestión del código.

Adicionalmente, se ha utilizado Sublime Text 3, ya que es un editor muy ligero y es más sencillo de utilizar para trabajar con pequeñas partes del código.

Para la gestión de ontologías se ha utilizado el editor Protégé, el cual con su interfaz gráfica facilita la creación y manipulación de estas.

Por último, para crear la memoria se ha hecho uso de Microsoft Word para escribir una primera versión del texto, gracias a que permite contar las palabras y corregir fácilmente errores gramaticales, y de Overleaf para generar un documento en LaTeX.

7.2. Conclusiones

Este trabajo ha propuesto una aproximación basada en la lógica difusa para responder a consultas flexibles sobre grafos de conocimiento. Esta aproximación hace posible reutilizar los grafos RDF ya existentes y sus respectivos endpoints para responder consultas SPARQL, construyendo una capa encima de ellos que permita manejar la lógica difusa. La propuesta es general y soporta diferentes operadores de la lógica difusa. Además, soporta tipos de dato

Fuzzy OWL 2 para describir las funciones de pertenencia que definen los términos flexibles de las consultas, así como para calcular los pesos en el caso del operador de fusión OWA.

De la misma manera, se ha implementado una aplicación llamada *Graph Consultant*, con una interfaz gráfica para que los usuarios puedan realizar cómodamente estas consultas sin tener conocimiento de las tecnologías detrás de ella. La interfaz tiene un diseño modular que permite añadir nuevos operadores de fusión y modificadores lingüísticas de manera sencilla. Además, se evita la necesidad de manejar directamente pesos numéricos para los operadores de fusión, utilizando distintos mecanismos para calcularlos, y se usa autocompletado para facilitar el acceso a grafos de conocimiento de gran tamaño.

Para ilustrar la utilidad de la aplicación, se ha considerado un caso de uso: sobre la clase MeanOfTransportation y tres distintas propiedades, analizando el resultado obtenido en cada uno de ellas al modificar los operadores de fusión. La aplicación es eficiente y permite mostrar el resultado al usuario en segundos.

Algunos de los resultados de este TFG se han presentado en el congreso internacional 4th Iberoamerican Conference and 3rd Indo-American Conference on Knowledge Graphs and Semantic Web (KGSWC 2022) [20].

Desde un punto de vista personal, este TFG me ha aportado una perspectiva distinta a los trabajos realizados durante la carrera, ya que es un trabajo más ambicioso, tanto por su mayor tamaño como por el uso de tecnologías menos extendidas que requieren de más investigación para poder desarrollar un producto. Otro aspecto que lo diferencia de otros trabajos realizados anteriormente es que se realiza de forma individual, por lo que la forma de afrontarlo es diferente. Además, también ha servido para aplicar conocimientos adquiridos en otras asignaturas, sobre todo de "Sistemas de información distribuidos", ya que algunas de las tecnologías utilizadas se enseñan allí, pero también de otras como de "Ingeniería del software" o "Interacción persona ordenador".

7.3. Trabajo futuro

La principal tarea para el trabajo futuro es realización una evaluación del rendimiento del prototipo en algunos escenarios del mundo real.

Un punto que se podría mejorar de la aplicación y sería interesante es extender la interfaz para permitir más elementos de la lógica difusa: funciones de pertenencia, operadores de fusión, modificadores difusos, etc. Es posible que usuarios experimentados puedan agradecer tener una elección más amplia sobre estos elementos.

Bibliografía

[1] Steffen Staab and Rudi Studer. *Handbook on Ontologies*. International Handbooks on Information Systems. Springer, 2004.

[2] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutiérrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan F. Sequeda, Steffen Staab, and Antoine Zimmermann. *Knowledge Graphs*. Number 22 in Synthesis Lectures on Data, Semantics, and Knowledge. Morgan & Claypool, 2021.

[3] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.

[4] George J. Klir and Bo Yuan. *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Prentice-Hall, 1995.

[5] Lotfi A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

[6] Thomas Lukasiewicz and Umberto Straccia. Managing uncertainty and vagueness in Description Logics for the Semantic Web. *Journal of Web Semantics*, 6(4):291–308, 2008.

[7] Fu Zhang, Jingwei Cheng, and Zongmin Ma. A survey on fuzzy ontologies for the semantic web. *Knowledge Engineering Review*, 31(3):278–321, 2016.

[8] Fernando Bobillo, Marco Cerami, Francesc Esteva, Àngel García-Cerdaña, Rafael Peñaloza, and Umberto Straccia. Fuzzy description logics. In Petr Cintula, Christian Fermüller, and Carles Noguera, editors, *Handbook of Mathematical Fuzzy Logic Volume III*, volume 58 of *Studies in Logic, Mathematical Logic and Foundations*, chapter XVI, pages 1105–1181. College Publications, 2015.

[9] Ronald R. Yager. On ordered weighted averaging aggregation operators in multicriteria decision making. *IEEE Transactions on Systems, Man and Cybernetics*, 18(1):183–190, 1988.

[10] Ronald R. Yager. Quantifier guided aggregation using OWA operators. *International Journal of Intelligent Systems*, 11(1):49–73, 1996.

[11] Fernando Bobillo and Umberto Straccia. Fuzzy ontology representation using OWL 2. *International Journal of Approximate Reasoning*, 52(7):1073–1094, 2011.

[12] Ignacio Huitzil, Fernando Alegre, and Fernando Bobillo. GimmeHop: A recommender system for mobile devices using ontology reasoners and fuzzy logic. *Fuzzy Sets and Systems*, 401:55–77, 2020.

[13] Ignacio Huitzil, Miguel Molina-Solana, Juan Gómez-Romero, and Fernando Bobillo. Minimalistic fuzzy ontology reasoning: An application to Building Information Modeling. *Applied Soft Computing*, 103:107158, 2021.

[14] V. Vaneková, J.P. Bella, P. Gurský, and T. Horváth. Fuzzy RDF in the semantic web: deduction and induction. In *Proceedings of the 6th Workshop on Data Analysis (WDA 2005)*, pages 16–29, 2005.

[15] Mauro Mazzieri and Aldo Franco Dragoni. A fuzzy semantics for the resource description framework. In *Uncertainty Reasoning for the Semantic Web I*, volume 5327 of *Lecture Notes in Computer Science*, pages 244–261. Springer, 2008.

[16] Ahmed El Amine Djebri. *Uncertainty Management for Linked Data Reliability on the Semantic Web*. PhD thesis, Université Côte D'Azur, 2022.

[17] Umberto Straccia. A minimal deductive system for general fuzzy RDF. In *Proceedings of the 3rd International Conference on Web Reasoning and Rule Systems (RR 2009)*, volume 5837 of *Lecture Notes in Computer Science*, pages 166–181. Springer, 2009.

[18] Jeff Z. Pan, Giorgos Stamou, Giorgos Stoilos, Edward Thomas, and Stuart Taylor. Scalable querying service over fuzzy ontologies. In *Proceedings of the 17th International World Wide Web Conference (WWW 2008)*, pages 575–584, 2008.

[19] Olivier Pivert, Olfa Slama, and Virginie Thion. An extension of SPARQL with fuzzy navigational capabilities for querying fuzzy RDF data. In *Proceedings of the 2016 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE 2016)*, pages 2409–2416. IEEE, 2016.

[20] José Félix Yagüe, Ignacio Huitzil, Carlos Bobed, and Fernando Bobillo. Flexible queries over knowledge graphs. In *Proceedings of the 4th Iberoamerican and 3rd Indo-American Knowledge Graphs and Semantic Web Conference (KGSWC 2022)*, volume 1686 of *Communications in Computer and Information Science*, pages 192–200. Springer, 2022.

Apéndice A

Diagrama de Gannt

En este apartado se encuentra el diagrama de Gannt con los esfuerzos invertidos a lo largo del tiempo. El trabajo empezó con alguna reunión y documentación sobre las tecnologías que se iban a utilizar en el proyecto. Después se empezó a desarrollar el algoritmo, que es lo que más tiempo requirió para desarrollar. Tras esto tuve que documentarme otra vez para realizar la interfaz gráfica, ya que nunca había utilizado Java Swing. Por último, se ha redactado la memoria para terminar el proyecto. Además, he tenido reuniones periódicas con mi tutor para el seguimiento y correcto desarrollo del proyecto. La distribución de las horas se puede ver en la Figura A.1.

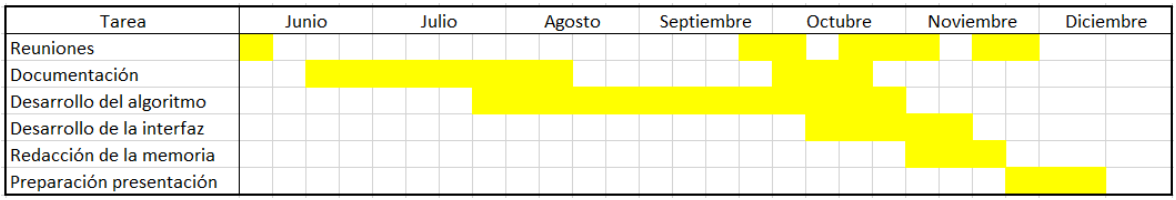


Figura A.1: Diagrama de Gannt

Apéndice B

Diagrama de secuencia

Este anexo sirve para agregar un diagrama de secuencia que facilite la comprensión del trabajo implementado.

La Figura B.1 sirve para comprender en rasgos generales el comportamiento que hay entre los distintos elementos que existen en la aplicación.

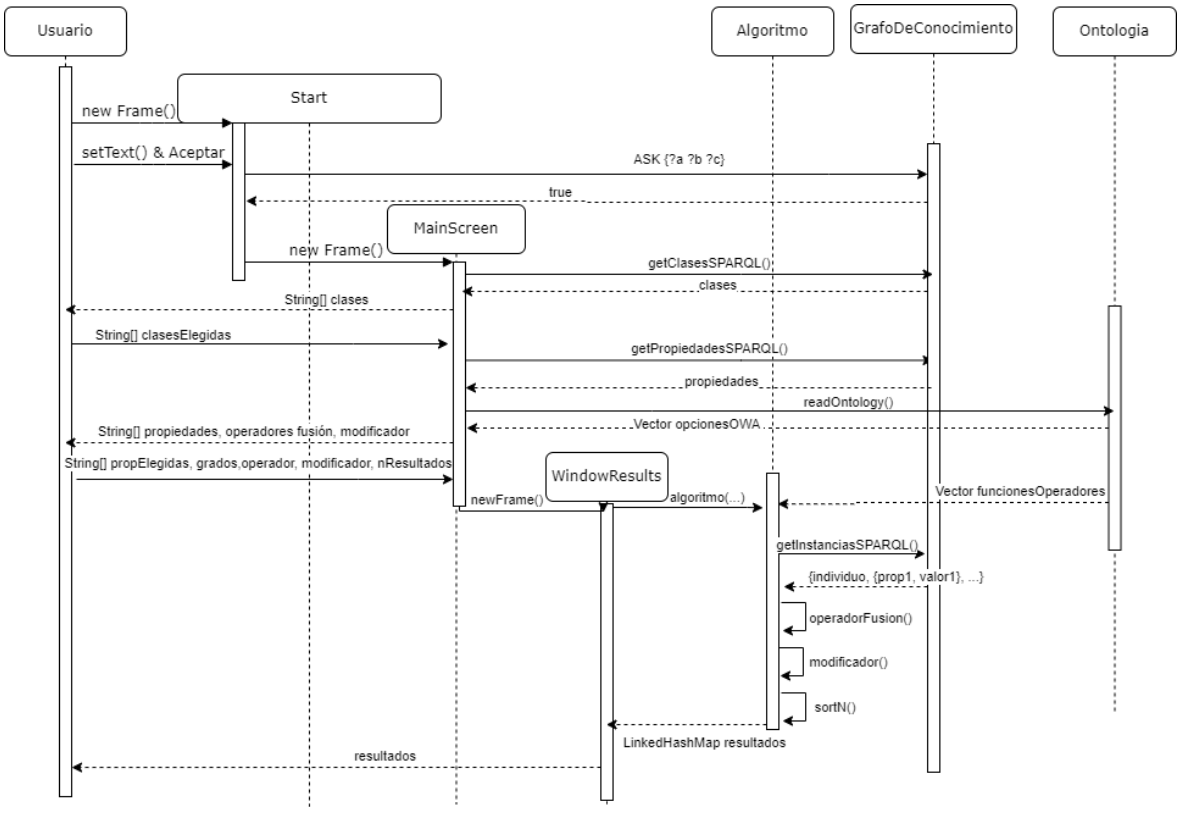


Figura B.1: Diagrama general de secuencia

Apéndice C

Publicación en congreso KGSWC 2022

Algunos de los resultados de este TFG se han presentado en el congreso internacional 4th Iberoamerican Conference and 3rd Indo-American Conference on Knowledge Graphs and Semantic Web (KGSWC 2022). Los datos completos de la publicación son los siguientes:

> *José Félix Yagüe and Ignacio Huitzil and Carlos Bobed and Fernando Bobillo. Flexible queries over knowledge graphs. Proceedings of the 4th Iberoamerican and 3rd Indo-American Knowledge Graphs and Semantic Web Conference (KGSWC 2022), Communications in Computer and Information Science 1686, pp. 192-200, Springer. Madrid (España), Noviembre 2022. DOI: 10.1007/978-3-031-21422-6_14.*

Flexible Queries over Knowledge Graphs

José Félix Yagüe[1], Ignacio Huitzil[2], Carlos Bobed[1,3], and Fernando Bobillo[1,3](✉)

[1] University of Zaragoza, Zaragoza, Spain
{cbobed,fbobillo}@unizar.es
[2] Artificial Intelligence Research Institute (IIIA), CSIC, Bellaterra, Spain
[3] Aragon Institute of Engineering Research (I3A), Zaragoza, Spain

Abstract. The increasing interest in Knowledge Graphs to represent real-world knowledge and the common need to manage imprecise knowledge in many real-world applications demand the study of approaches to solve flexible queries over Knowledge Graphs. In this paper, we propose a novel approach to solve that problem which reuses Semantic Web standards (RDF and SPARQL) and builds a fuzzy layer on top of them.

Keywords: Knowledge graphs · Fuzzy logic · Flexible querying

1 Introduction

Semantic Web technologies are receiving a lot of attention to represent the knowledge in many domains and applications. Such Semantic Web technologies include ontologies, or formal and shared specifications of the vocabulary of a domain of interest [14], Knowledge Graphs [6], a graph-based model to capture data at large scale, and Linked Data, a set of best practices for publishing and connecting data on the Web [2]. Knowledge Graphs, Linked Data, and even ontologies are usually expressed in RDF language[1].

In many real-world applications, there is a need to manage imprecise knowledge. In such cases, Fuzzy Set Theory and Fuzzy Logic have proved to be useful for more than half a century [9,17]. Fuzzy sets allow to represent the partial membership of an element to a set, and Fuzzy Logic allows to manage propositions which are partially true and make deductions through approximate reasoning.

Therefore, fuzzy extensions of Semantic Web technologies have been proposed. Most of the literature has focused on fuzzy ontologies [10,18] or fuzzy Description Logics [3], as the main formalism behind fuzzy ontologies. Unfortunately, the combination of Fuzzy Logic and Knowledge Graphs has not received a similar attention.

[1] https://www.w3.org/TR/rdf11-primer/, last accessed on 26th July 2022.

B. Villazón-Terrazas et al. (Eds.): KGSWC 2022, CCIS 1686, pp. 192–200, 2022.
https://doi.org/10.1007/978-3-031-21422-6_14

In this paper, we will propose a novel approach to answer flexible queries over classical Knowledge Graphs. While previous work focuses on the support of fuzzy axioms (using non-standard RDF), we instead restrict to fuzzy datatypes. This way, we are able to stick to the Semantic Web standards, using standard RDF and SPARQL query endpoints, and building the fuzzy layer on top of them as a series of additional steps.

The remainder of this paper is structured as follows. Section 2 provides some background on Knowledge Graphs and Fuzzy Logic. Then, Sect. 3 discusses our novel approach to answer flexible queries over Knowledge Graphs. Finally, Sect. 4 overviews some related work, and Sect. 5 sets up some conclusions and ideas for future work.

2 Background

Knowledge Graphs. Although they have been there for quite a long time under different names (e.g., Semantic Networks), Knowledge Graphs have lately received a lot of attention since the adoption of the term by Google in 2012. There is not a consensual definition of what a Knowledge Graph is, but we could consider the view on them as labeled directed graphs the most spread one. In this view, a Knowledge Graph is a labeled directed graph (E, R, L), with E being entities, R being relations between such entities, and L a labeling function that maps each element in the graph to its name/type. Influenced indeed by the RDF data model, this definition derives into regarding them as sets of triples subject-predicate-object (SPO) triples. However, as noted in [6], the notion of Knowledge Graph is broader than that. Without binding the definition to any particular data model, Hogan et al. [6] adopt the following 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* (the interested reader can find pointers to alternative definitions in [6]).

Included in the previous definition, we would find RDF graphs, which are labeled directed graphs. RDF is the W3C standard language for representing information in the Web. It is based on triples of the form $\langle s, p, o \rangle$, where s is the subject, p is the property, and o is the object. That is, $\langle s, p, o \rangle$ states the s is related with o via the property p. In general, such relationships are not symmetrical.

To define the schema that the RDF graph follows, we can use different languages with different expressivities, ranging from RDF-S (RDF-Schema[2]), which allows us to define hierarchies of concepts and properties, the domain and ranges of properties, and typing the resources asserting the concepts they belong to, to the different profiles of OWL[3], which extends RDF-S allowing to model expressive ontologies in different fragments of Description Logics [1].

[2] https://www.w3.org/TR/rdf-schema/, last accessed on 26th July 2022.
[3] https://www.w3.org/TR/owl2-overview/, last accessed on 26th July 2022.

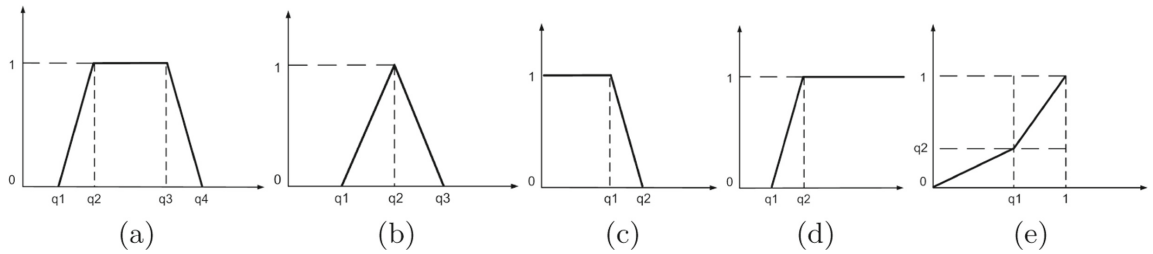


Fig. 1. (a) Trapezoidal; (b) Triangular; (c) Left-shoulder; (d) Right shoulder; (e) Linear fuzzy membership functions.

SPARQL[4] is its standard query language. It supports four different types of queries (namely, SELECT, ASK, CONSTRUCT, and DESCRIBE) whose body is described in terms of basic graph patterns (BGPs) composed using free variables. Such BGPs are to be matched crisply to the underling graph in order to provide possible mappings that fulfill the query conditions.

Fuzzy Logic. Fuzzy Logic is widely used to manage imprecise and vague knowledge. It is a generalization of classical logic proposed by Zadeh where statements are not necessarily either true or false, but hold to some degree of truth [17].

The cornerstone of Fuzzy Logic is the concept of fuzzy set, which is a generalization of a classical set where elements can have a partial membership. A fuzzy set A is characterized by a membership function $\mu_A(x)$ which associates with each object x a real number in $[0, 1]$ representing the membership degree of x in A. As in classical sets, 0 means no-membership and 1 full membership, but now an intermediate value between 0 and 1 denotes partial membership to F.

To build fuzzy membership functions, common options are the trapezoidal (Fig. 1(a)), triangular (Fig. 1(b)), left-shoulder (Fig. 1(c)), right-shoulder (Fig. 1(d)), and linear (Fig. 1(e)) membership functions.

Example 1. The fuzzy set of beers with LowAlcohol can be defined using a a triangular function **triangular**$(0.5, 3.1, 6.55)$. If the alcohol of Ambar_Especial beer is 5.2°, its membership degree to LowAlcohol can be evaluated as: $\mu_{\mathsf{LowAlcohol}}(\mathsf{Ambar\_Especial}) = (\mathbf{triangular}(0.5, 3.1, 6.55))(5.2) = 0.39$.

Fuzzy Logic enables approximate reasoning. Logical operations over classical sets are also generalized to the fuzzy case. To compute the conjunction, disjunction, complement and implication over fuzzy sets one can use different families of functions, namely a *t-norm* function $\otimes$, a *t-conorm* function $\oplus$, a *negation* function $\ominus$ and an *implication* function $\Rightarrow$ (see [9] for details). For instance, the minimum and the product are t-norms and the maximum is a t-conorm.

There are other ways to combine fuzzy sets. For example, an aggregation operator is a function that takes n values in $[0, 1]$(possibly representing the membership degrees to n fuzzy sets) and returns a single value in $[0, 1]$. Some

[4] https://www.w3.org/TR/sparql11-overview/, last accessed on 26th July 2022.

examples are the weighted mean (WMEAN) or the *Ordered Weighted Averaging* (OWA) operator.

To conclude this section, a *fuzzy modifier* (also called fuzzy hedge) modifies the shape of a fuzzy set by altering its membership function. Two common examples are the weakening modifier very, characterized by the function $\mathsf{very}(x) = x^2$, and the increasing modifier few, defined as $\mathsf{few}(x) = \sqrt{x}$.

Example 2. If we apply very to the fuzzy set LowAlcohol, for each beer x the degree of being a very low alcoholic beer can be computed as: $\mu_{\mathsf{VeryLowAlcohol}}(x) = \mathsf{very}(\mu_{\mathsf{LowAlcohol}}(x)) = (\mu_{\mathsf{LowAlcohol}}(x))^2$. Note that as the degree is in $[0, 1]$, by squaring it, we reinforce the need to belong "very" to the set.

3 Flexible Querying

Representing the Queries. Given a Knowledge Graph $\mathcal{K}$, a flexible query is characterized by the following parameters:

- A set of classes $C_1, C_2, \ldots, C_N$
- A set of functional numerical data properties $P_1, P_2, \ldots, P_n$
- A list of fuzzy datatypes $D_1, D_2, \ldots, D_n$
- An optional fusion operator $@\colon [0,1]^n \to [0,1]$
- An optional fuzzy hedge $h\colon [0,1] \to [0,1]$

Possible fusion operators are t-norms, t-conorms, weighted mean, or OWA.

Example 3. Given a Knowledge Graph about beers, a possible flexible query to retrieve "very [relevant] Spanish Lager beers with low alcohol and bitterness levels" might be described as:

- Classes: Lager, SpanishBeer
- Data properties: alcohol, bitterness
- Fuzzy datatypes: LowAlcohol, defined as **triangular**$(0.5, 3.1, 6.55)$, and LowBitterness, defined as **triangular**$(15, 27, 41)$
- Fusion operator: product t-norm
- Fuzzy hedge: $\mathsf{very}(x) = x^2$

We propose to rely on Fuzzy OWL 2 datatypes [4], which includes trapezoidal, triangular, left-shoulder, and right-shoulder datatypes. In Fuzzy OWL 2, a datatype declaration can be associated with an OWL 2 annotation encoding a fuzzy membership function, using an XML-like syntax proposed in [4]. Interestingly, such annotations can be encoded as triples in the Knowledge Graph.

Example 4. To express that fuzzy datatype *LowAlcohol* corresponds to a triangular function **triangular**$(0.5, 3.1, 6.55)$, we use the following set of RDF triples:

```
@prefix ex: <http://www.example.org/beer/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
ex:fuzzyLabel rdf:type owl:AnnotationProperty .
ex:LowAlcohol rdf:type rdfs:Datatype .
ex:LowAlcohol ex:fuzzyLabel """<fuzzyOwl2 fuzzyType=\"datatype\">
 <Datatype type=\"triangular\" a=\"0.5\" b=\"3.1\" c=\"6.55\" />
 </fuzzyOwl2>""" .
```

Solving the Queries. To solve the query, we perform the following steps:

1. The first step is to retrieve, for each member of all the classes, the values of the data properties. This can be done using the following SPARQL query:

   ```
   SELECT ?x ?Y1 ... ?Yn
   WHERE {
      ?x rdf:type/rdfs:subClassOf* C1 .
      ...
      ?x rdf:type/rdfs:subClassOf* CN .
      ?x P1 ?Y1 .
      ...
      ?x Pn ?Yn .
   }
   ```

   The result of this step is a list of tuples $\langle x, y_1^x, \ldots, y_n^x \rangle$. Note that this query not only retrieves the direct instances of the concepts $C_1, \ldots, C_n$, but also their indirect instances (i.e., including the instances of some of their subclasses). It could also be extended to retrieve values `?Y`$_i$ linked to `?x` via a subproperty of P_i, or to infer the classes of `?x` via some domain or role restrictions.
2. Next, for each $i \in \{1, \ldots, n\}$, we retrieve the definition F_i of the input fuzzy datatype D_i, using the following SPARQL query:

   ```
   SELECT ?Fi
   WHERE {
      Di rdf:type rdfs:Datatype .
      Di fuzzyLabel Fi .
   }
   ```

3. The next step is to compute, for each individual `?x` and each data property P_i, the membership degree of the value Y_i to the fuzzy datatype F_i:

$$z_i^x = F_i(y_i^x),\ \forall i \in \{1, \ldots, n\} \tag{1}$$

4. Then, we aggregate all the values z_i^x corresponding to an individual `?x` into a single result r_x using the input fusion operator

$$r_x = @_{i \in \{1,\ldots,n\}}(z_i^x) \tag{2}$$

5. Now it is time to use the fuzzy hedge to modify the membership degree for each `?x`:
$$\alpha_x = h(r_x) \tag{3}$$
6. Finally, the answer to the query is a list of values $\langle ?x, \alpha_x \rangle$, sorted in decreasing order according to the value α_x.

Example 5. Let us solve the query in Example 3 given the following set of triples on a beer domain:

```
@prefix ex: <http://www.example.org/beer/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
ex:ImperialPilsStrongPaleLager rdfs:subclassOf ex:Lager .
ex:PaleLager rdfs:subclassOf ex:Lager .
ex:Alhambra_Reserva_1925 rdf:type ex:SpanishBeer .
ex:Alhambra_Reserva_1925 rdf:type ex:ImperialPilsStrongPaleLager .
ex:Alhambra_Reserva_1925 ex:alcohol "6.4"^^xsd:decimal.
ex:Alhambra_Reserva_1925 ex:bitterness "25"^^xsd:decimal .
ex:Ambar_Especial rdf:type ex:SpanishBeer .
ex:Ambar_Especial rdf:type ex:PaleLager .
ex:Ambar_Especial ex:alcohol "5.2"^^xsd:decimal .
ex:Ambar_Especial ex:bitterness "25"^^xsd:decimal .
ex:BrewDog_Punk_IPA ex:rdf:type ex:IndiaPaleAleIPA .
ex:BrewDog_Punk_IPA ex:alcohol "6"^^xsd:decimal .
ex:BrewDog_Punk_IPA ex:bitterness "60"^^xsd:decimal .
```

The first SPARQL query retrieves both Alhambra_Reserva_1925 and Ambar_Especial, as they are both Spanish beers of a (subclass of) Lager style, together with their values of alcohol and bitterness. After that, the algorithm computes z_1 and z_2, and then $\alpha_x = (z_1 \cdot z_2)^2$. More precisely, the obtained values are:

?x	?Y1	?Y2	z1	z2	α_x
Alhambra_Reserva_1925	6.4	25	0.04	0.83	0.001
Ambar_Especial	5.2	25	0.39	0.83	0.11

Finally, the result is the following ordered list of pairs:

⟨Ambar_Especial, 0.11⟩
⟨Alhambra_Reserva_1925, 0.001⟩

The main advantage of this algorithm is that it is possible to reuse standard RDF language and SPARQL query endpoints, similarly as the authors in [7,8] do

for fuzzy ontologies. Note that it is trivial to extend the characterization of the query to include an additional parameter k so that only the top-k results (i.e., the individuals with the k highest values of α_x) are retrieved. Note as well that some fusion operators and fuzzy hedges can be expressed using SPARQL built-in functions (or inner expressions), allowing to perform more computations without relying on external processing. However, the resulting SPARQL query could be quite complex so as to be included here, and there are operators that cannot be expressed in standard SPARQL, such as the geometric mean (involving n-th roots).

Implementation. We are currently implementing a prototype tool. It is developed in Java. The SPARQL queries are solved using Apache Jena and Jena Java API[5] through a (local or remote) SPARQL endpoint. The definitions of the fuzzy datatypes are parsed (to retrieve the function type and its parameters) using Fuzzy OWL 2 API.

The idea is develop a graphical user interface so that the user can easily provide all the relevant information:

- Providing the URL of the endpoint storing the Knowledge Graph.
- Providing the names of the concepts, data properties, and fuzzy datatypes. Rather than allowing the user to select a concept/property/datatype from the Knowledge Graphs (which requires showing all possible options, overwhelming the user in large Knowledge Graphs), some auto-complete mechanism seems more appropriate.
- Selecting the fusion and hedge operators from a list of options built into the implementation.

4 Related Work

Some previous works consider Fuzzy Logic and RDF language. The oldest proposal was made by Vaneková [16] et al., who propose to represent a fuzzy fact of the form "a resource s belongs to a fuzzy set f with degree α", with $\alpha \in (0, 1]$, using an RDF triple $\langle s, f, \alpha \rangle$. While this approach makes it possible to use a single triple reusing standard RDF, there is an implicit relationship type which is not being represented. For example, the authors do not represent the fact that item002 is of type (rdf:type) cheap, or the fact item002's price is cheap. M. Mazzieri and A. F. Dragoni consider statements of the form $\langle s, p, o, \alpha \rangle$ [11]. The idea is to represent, for example, that the triple $\langle$Zaragoza, isCloseTo, Huesca$\rangle$ holds with degree 0.7. Depending on the property p, we can have concept assertions (rdf:type), subclass axioms (rdfs:subclassOf), subproperty axioms (rdfs:subPropertyOf) ore, more generally, property assertions. A. E. A. Djebri discusses different approaches to annotate such statements: reification, n-ary properties, single named graph, singleton properties, and RDF-star [5]. However, these approaches do not consider fuzzy datatypes, as we do.

[5] http://jena.apache.org.

U. Straccia proposed a reasoning algorithm for a fuzzy extension of ρDF (a fragment of RDF Schema) with statements of the form $\langle s, p, o, \alpha \rangle$ but no fuzzy datatypes [15]. He also proposed fuzzy conjunctive queries over a fuzzy graph, which can include fuzzy triples, and assignments involving fuzzy membership functions and fusion operators, e.g., $q(x, s) \leftarrow \langle x, rdf : type, SportsCar \rangle \wedge \langle x, hasPrice, y \rangle \wedge \{s := s1 \cdot cheap(y)\}$ [15]. This approach does not detail how to represent the syntax of the fuzzy datatypes (we instead use Fuzzy OWL 2 datatypes already represented in the RDF graph) and does not consider fuzzy hedges. Our approach could be friendlier to the user as there is no need to write such complex queries.

J. Z. Pan et al. proposed Fuzzy SPARQL, an extension of SPARQL designed to query fuzzy ontologies (but not fuzzy Knowledge Graphs) [12] with statements of the form $\langle s, p, o, \alpha \rangle$, but it does not consider fuzzy datatypes or fuzzy hedges.

O. Pivert et al. proposed another fuzzy extension of SPARQL called FURQL (Fuzzy RDF Query Language) [13], allowing fuzzy properties (that partially hold) and fuzzy datatypes in queries over a fuzzy RDF graph with statements of the form $\langle s, p, o, \alpha \rangle$. However, although fuzzy datatypes are used in the examples, it is not discussed how to represent them in the graph, it is not possible to specify the fusion operator, and fuzzy hedges are not supported.

5 Conclusions and Future Work

This paper has proposed a Fuzzy Logic based approach to answer flexible queries over Knowledge Graphs. Our approach makes it possible to reuse existing RDF graphs and SPARQL endpoints, building a fuzzy layer on top of them. It also supports Fuzzy OWL 2 datatypes to describe fuzzy membership functions.

The main direction for future work is the completion of the prototype implementation, with a modular design so that new fusion operators can be added, and an intuitive user graphical interface. After finishing the implementation, we would like to evaluate the performance on some real scenarios.

Acknowledgments. C. Bobed and F. Bobillo were supported by the I+D+i project PID2020-113903RB-I00, funded by MCIN/AEI/10.13039/501100011033, and by DGA/FEDER.

References

1. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, Cambridge (2003)
2. Bizer, C., Heath, T., Berners-Lee, T.: Linked Data - the story so far. Int. J. Semant. Web Inf. Syst. **5**(3), 1–22 (2009)
3. Bobillo, F., Cerami, M., Esteva, F., García-Cerdaña, À., Peñaloza, R., Straccia, U.: Fuzzy description logics. In: Cintula, P., Fermüller, C., Noguera, C. (eds.) Handbook of Mathematical Fuzzy Logic Volume III, Studies in Logic, Mathematical Logic and Foundations, Chap. XVI, vol. 58, pp. 1105–1181. College Publications (2015)

4. Bobillo, F., Straccia, U.: Fuzzy ontology representation using OWL 2. Int. J. Approx. Reason. **52**(7), 1073–1094 (2011)
5. Djebri, A.E.A.: Uncertainty management for linked data reliability on the Semantic Web. Ph.D. thesis, Université Côte D'Azur (2022). https://hal.archives-ouvertes.fr/tel-03679118
6. Hogan, A., et al.: Knowledge Graphs, vol. 22. Morgan & Claypool, San Rafael (2021)
7. Huitzil, I., Alegre, F., Bobillo, F.: GimmeHop: a recommender system for mobile devices using ontology reasoners and Fuzzy Logic. Fuzzy Sets Syst. **401**, 55–77 (2020)
8. Huitzil, I., Molina-Solana, M., Gómez-Romero, J., Bobillo, F.: Minimalistic fuzzy ontology reasoning: an application to building Information Modeling. Appl. Soft Comput. **103**, 107158 (2021)
9. Klir, G.J., Yuan, B.: Fuzzy Sets and Fuzzy Logic: Theory and Applications. Prentice-Hall, Englewood Cliffs (1995)
10. Lukasiewicz, T., Straccia, U.: Managing uncertainty and vagueness in Description Logics for the Semantic Web. J. Web Semant. **6**(4), 291–308 (2008)
11. Mazzieri, M., Dragoni, A.F.: A fuzzy semantics for the resource description framework. In: da Costa, P.C.G., d'Amato, C., Fanizzi, N., Laskey, K.B., Laskey, K.J., Lukasiewicz, T., Nickles, M., Pool, M. (eds.) URSW 2005-2007. LNCS (LNAI), vol. 5327, pp. 244–261. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89765-1_15
12. Pan, J.Z., Stamou, G., Stoilos, G., Thomas, E., Taylor, S.: Scalable querying service over fuzzy ontologies. In: Proceedings of the 17th International World Wide Web Conference (WWW 2008), pp. 575–584 (2008)
13. Pivert, O., Slama, O., Thion, V.: An extension of SPARQL with fuzzy navigational capabilities for querying fuzzy RDF data. In: Proceedings of the 2016 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE 2016), pp. 2409–2416. IEEE (2016)
14. Staab, S., Studer, R.: Handbook on Ontologies. International Handbooks on Information Systems, Springer, Heidelberg (2004)
15. Straccia, U.: A minimal deductive system for general fuzzy RDF. In: Polleres, A., Swift, T. (eds.) RR 2009. LNCS, vol. 5837, pp. 166–181. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05082-4_12
16. Vaneková, V., Bella, J., Gurský, P., Horváth, T.: Fuzzy RDF in the Semantic Web: deduction and induction. In: Proceedings of the 6th Workshop on Data Analysis (WDA 2005), pp. 16–29
17. Zadeh, L.A.: Fuzzy sets. Inf. Control **8**, 338–353 (1965)
18. Zhang, F., Cheng, J., Ma, Z.: A survey on fuzzy ontologies for the Semantic Web. Knowl. Eng. Rev. **31**(3), 278–321 (2016)