UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

SISTEMA VISUAL PARA EXPLORAR SUBGRAFOS TEMÁTICOS EN WIKIDATA

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

CRISTÓBAL PATRICIO TORRES GUTIÉRREZ

PROFESOR GUÍA:
AIDAN HOGAN

MIEMBROS DE LA COMISIÓN:
GONZALO NAVARRO BADINO
RODRIGO FREZ PULGAR

SANTIAGO DE CHILE
2023

SISTEMA VISUAL PARA EXPLORAR SUBGRAFOS TEMÁTICOS EN WIKIDATA

Wikidata posee una gran cantidad de información, ya sea de personas, animales, grupos de música, países, etc. Algunos de sus usuarios siguen agregando información nueva, actualizándola e incluyendo fuentes. Los datos que dispone están estructurados, lo que permite que puedan ser leídos automáticamente por ordenadores.

Se pueden realizar consultas sobre los datos de Wikidata. Sin embargo, elaborar estas consultas requiere el conocimiento del lenguaje SPARQL. Esto es una barrera de entrada en el uso para personas que no poseen conocimientos acerca de lenguajes de consulta, lo que les aleja de aprovechar completamente Wikidata.

Con lo anterior en consideración, el objetivo principal de esta memoria es el desarrollo de un sistema visual que permita explorar subgrafos temáticos en Wikidata y obtener eficientemente relaciones. Esto podría acercar el uso de Wikidata a personas que no conocen esta fuente de información o bien no saben acerca del lenguaje de consulta SPARQL.

La creación de los subgrafos consta de desplegar los caminos (nodos y aristas intermedias) entre entidades de Wikidata. Para esto se implementa una estructura de almacenamiento de los datos de Wikidata en memoria principal. A continuación, se desarrolla un algoritmo de búsqueda de caminos entre los nodos (entidades de Wikidata). Finalmente se desarrolla un sistema visual para explorar subgrafos temáticos usando los datos de Wikidata ya almacenados en memoria y el algoritmo propuesto.

La solución implementada se evalúa a nivel de estructura de datos, eficiencia algorítmica y usabilidad. En primera instancia, se evalúan las estructuras de almacenamiento mediante la comparación de su uso de memoria y rapidez de obtención de vecinos de los nodos. Se observan mejores resultados en la estructura que usa arreglos nativos y posee una estructura similar a una lista de adyacencia. Se evalúa el algoritmo de búsqueda de caminos midiendo la cantidad de aristas pertenecientes a caminos que se obtienen en un minuto. El resultado es un algoritmo que posee una mediana que supera las 100 aristas obtenidas en menos de un minuto para caminos de largo 3 (a lo más 3 aristas entre los nodos que se buscan caminos). Por último, se evalúa la aplicación web desarrollada mediante una encuesta de usabilidad. A partir de 80 respuestas voluntarias se obtiene el puntaje promedio de 81,19 en la escala de usabilidad del sistema.

Se plantea como trabajo futuro, evaluar otras estructuras de almacenamiento del grafo, mejorar el aspecto visual para hacerla más amigable con los usuarios y cambiar la arquitectura del servidor, de modo que se pueda tener una máquina que almacene el grafo de Wikidata y otras lleven a cabo las búsquedas de caminos.

A mi hermano, obviamente.

Agradecimientos

Agradezco a quienes me apoyaron en este camino y me siguen apoyando en mis metas. Agradezco por sobretodo a quienes han confiado en mí sus metas y desafíos.

Agradezco a mi familia por brindarme una vida tranquila, por hacerme sentir querido y apoyarme en mis metas. A mi padre, por su apoyo incondicional y por preocuparse de que nunca me faltara nada.

Agradezco a mi hermano quien me ayudó a encontrar la motivación de entrar a esta universidad y estudiar esta carrera. Por siempre confiar en mis capacidades aún más que yo. Quien me ha acompañado toda mi vida, alguna vez me enseñó, me aconsejó y también fue mi compañero de U.

A mi gato Mateo, quien me acompañó en el desarrollo de mi memoria y que incluso el solo hecho de verlo dormir me alegraba.

A mis familiares de Santiago y Quirihue, quienes se han alegrado de mis metas y me han brindado su apoyo.

Al profesor Aidan Hogan por haberme aceptado como memorista. Por confiar en mi un tema desafiante y haberme apoyado en este proyecto del que hoy me siento orgulloso y feliz.

A mis amigos del colegio, sobretodo a Francisco y Lukas, porque aún en la distancia encontramos momentos para juntarnos y reír.

A mis compañeros y amigos con quienes compartí en mis primeros años de universidad. Gracias a ellos mi estadía se hizo más amena. En especial a mis primeros amigos: Ignacio, Francisco, Javier y Pablo.

A mis compañeros y amigos del departamento de computación. Gente de la que siempre pude aprender y encontrar más motivación por la carrera. En especial a Rodrigo con quien nos apañamos harto en el transcurso de la especialidad.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

A lo largo del tiempo siempre ha sido un pilar fundamental el acceso a información. Sin importar su uso, se hace necesario recurrir a periódicos, enciclopedias, videos, libros, noticias, etc., para acceder a ella. En la actualidad la forma más rápida y efectiva de acceder a información pública consiste en la búsqueda por internet mediante un computador o celular. Esto ha sido un paso fundamental para masificar el acceso, además de permitir acceso a una mayor cantidad de fuentes de información.

Las fuentes de información en la internet destacan por su gran cantidad de datos y referencias. Esto las hace la alternativa preferida, sin embargo, las más usadas son las que poseen un uso fácil y acceso rápido (sin necesidad de más pasos). Lo anterior ha limitado el acceso del usuario que no posee grandes conocimientos en computación a fuentes más complejas, como, por ejemplo, Wikidata [1].

Wikidata es una base de conocimientos editada en colaboración, muy similar en este sentido a Wikipedia. En ella, la información no se distribuye como párrafos, sino como relaciones entre elementos. Al acceder a algún elemento se obtiene cómo se relaciona con otros elementos. Esta estructura de relaciones tiene el formato RDF [2]. Este formato es un estándar que posee el lenguaje de consulta SPARQL [3]. Lo anterior influye en la gran ventaja de Wikidata: la posibilidad de consultar sobre datos integrados automáticamente de varias entidades de forma simultánea.

Con las ventajas que posee Wikidata, se vuelve una fuente de información valiosa. El problema que surge es que los usuarios, para beneficiarse de esta, deben tener conocimientos en el uso y manejo de su estructura. Esto requiere estudiar y entender los conceptos del formato RDF y el lenguaje de consulta SPARQL. Estos son conocimientos avanzados en el ámbito de la computación, lo que aleja a un usuario que no conozca de esta área.

Dado lo anterior, es necesario abordar un software que facilite y apoye el acceso de los usuarios no expertos a la información de Wikidata. Con esto, se abordará un acercamiento entre personas (sin importar si posee conocimientos en computación) y Wikidata, priorizando que la obtención de información sea rápida e incluyendo un enfoque en el aspecto visual y la usabilidad.

En este trabajo se diseñará y desarrollará una aplicación web que permita al usuario crear y visualizar subgrafos temáticos de Wikidata. En particular, el usuario provee un conjunto

de dos o más entidades y se van a desplegar los caminos de hasta largo 3 entre ellas. Esto incluirá entidades y relaciones intermedias.

El uso de esta aplicación permitirá enriquecer el conocimiento e información que busca el usuario acerca de un elemento específico. Así podrá entender relaciones con otros elementos y motivar la búsqueda de esos conceptos. Se podrá usar este sistema para iniciar una búsqueda exploratoria acerca de un tema, entendiendo las relaciones de las entidades en un tópico y también como un punto de inicio para explorar dentro de este.

Un uso posible es formar redes de colaboración. Por ejemplo, al buscar dos o más personas, junto (opcionalmente) con algunas organizaciones, artículos o temas de interés, se pueden identificar relaciones entre estas entidades que proveen estrategias para colaborar.

Otro uso que se plantea es poder detectar relaciones entre entidades que puedan ser controversiales. Esto podría fomentar y plantear un mayor estudio dentro de estas entidades y sus relaciones. Un ejemplo es detectar fraudes o corrupción al ver relaciones entre personajes públicos o de gran relevancia, dueños de empresas u organizaciones en cuestión y otras entidades relacionadas.

La implementación de este sistema visual incluye desafíos a nivel de programación, siendo uno de los principales la obtención de los caminos entre las entidades que quiera el usuario. Esto implica el diseño de un algoritmo que obtenga los caminos en poco tiempo. Para definir y diseñar este algoritmo se hace necesario entender el tipo de información con la que se va a trabajar y los algoritmos que permiten recorrerla; esto requiere la investigación de estudios al respecto.

Dentro de los desafíos a nivel de programación se incluyen: orientar la solución a una arquitectura de cliente servidor; escribir código eficiente para una ejecución en menos tiempo; lograr un uso eficiente de los recursos, esto ya que los datos se trabajarán en memoria principal.

Otro desafío es lograr una aplicación usable y útil para todo tipo de usuario (con y sin conocimientos de computación). Esto exige un enfoque en los aspectos visuales de la solución y que hagan fácil su uso y obtención de resultados.

1.1. Objetivos

1.1.1. Objetivo General

Diseñar e implementar una herramienta Web que sea un sistema visual para explorar subgrafos temáticos en Wikidata, incluyendo los caminos entre las entidades que desee el usuario, manteniendo un enfoque en la usabilidad y accesibilidad para los usuarios no expertos en el uso de Wikidata y eficiente en la obtención de relaciones.

1.1.2. Objetivos Específicos

- Estudiar algoritmos para recorrido de grafos.

- Diseñar e implementar un algoritmo eficiente en tiempo de ejecución de obtención de caminos para un largo cualquiera entre entidades en un grafo RDF.
- Evaluar algoritmo de obtención de caminos para un largo específico entre entidades con un conjunto de prueba.
- Diseñar e implementar un sistema visual que, a partir de un conjunto de entidades que entregue un usuario, despliegue los caminos entre estas incluyendo relaciones y otras entidades intermedias.
- Validar con usuarios la usabilidad del sistema visual que despliega las relaciones y entidades.

1.2. Estructura de la memoria

Primero se describirá el estado del arte. Aquí se comentan investigaciones y literatura relacionada al tema de la memoria, siendo principalmente conceptos de la Web Semántica y algoritmos de recorrido sobre grafos. También se buscan otras soluciones existentes a la problemática general y estudios que sirvan para abordar la solución en ámbitos del algoritmo y el sistema visual.

A continuación, se explicita la solución propuesta. Esta aborda dos grandes áreas: la búsqueda de caminos y la aplicación web. Por el lado de la búsqueda de caminos se estudian estructuras de almacenamiento de los datos de Wikidata en memoria principal y se plantea un algoritmo que permita obtener los caminos entre las entidades almacenadas. Por el lado de la aplicación web se debe elaborar un sistema visual que permita a cualquier usuario consultar por los caminos entre entidades de Wikidata y que use el algoritmo planteado.

Luego se detalla la evaluación de la solución desarrollada. En ella se evalúan las estructuras de almacenamiento y el algoritmo de búsqueda de caminos, midiendo tiempos de búsqueda, cantidad de aristas de caminos obtenidas y memoria usada. También se detallan los resultados de la evaluación de la aplicación, mediante pruebas de usabilidad con usuarios.

Por último, se presentan las conclusiones a partir del trabajo desarrollado. Esto incluye comentar acerca de los objetivos logrados, un análisis acerca de la solución y posibles trabajos futuros.

Capítulo 2

Marco Teórico

En este capítulo se exponen los principales conceptos involucrados y tecnologías consideradas para el desarrollo de la memoria. También se describen soluciones existentes y su influencia en el desarrollo de la memoria.

2.1. Web Semántica

La meta de la web semántica es hacer legible y distinguible la información de la web para las máquinas. Se basa en la idea de añadir metadatos semánticos y ontológicos a la *World Wide Web* (web o red informática mundial). Esa nueva información describe el contenido, significado y relaciones entre datos y se debe proporcionar de manera formal para que sea fácil evaluarla por máquinas. El objetivo de la web semántica es mejorar la Internet ampliando la interoperabilidad entre los sistemas informáticos.

2.1.1. RDF

Resource Description Framework (RDF o Marco de Descripción de Recursos) es un modelo de datos que representa información sobre recursos en la web. También es una base para procesar metadatos.

RDF representa la información en forma de expresiones sujeto-predicado-objeto. Su formato proporciona interoperabilidad entre aplicaciones al contener información legible por máquinas en la web.

Existen múltiples formatos de serialización para RDF, algunos son:

- Turtle, formato de fácil interpretación para personas y compacto, que agrupa sentencias para un mismo sujeto. A continuación, se muestra en el Código 2.1 un ejemplo del formato, donde primero se define un prefijo y a continuación el grafo. Cada entidad y relación usa también el prefijo.

```
@prefix ex: <http://ex.org/>.

ex:Robert ex:padreDe ex:Jim .
ex:Alice ex:madreDe ex:Jim .
ex:Bob ex:padreDe ex:Jill .
ex:Helena ex:madreDe ex:Jill .
ex:Jim ex:padreDe ex:James .
ex:Jill ex:madreDe ex:James .
ex:James ex:nacioEn ex:Chile ; ex:estudiaEn ex:UniversidadDeChile .
```

Código 2.1: Grafo de ejemplo en formato Turtle.

- N-Triples, formato delimitado por líneas donde cada una es una sentencia sujeto-predicado-objeto. A continuación, se muestra en el Código 2.2 un ejemplo del formato, en este caso no es necesario definir el prefijo para usarlo.

```
<http://ex.org/Robert> <http://ex.org/padreDe> <http://ex.org/Jim> .
<http://ex.org/Alice> <http://ex.org/madreDe> <http://ex.org/Jim> .
<http://ex.org/Bob> <http://ex.org/padreDe> <http://ex.org/Jill> .
<http://ex.org/Helena> <http://ex.org/madreDe> <http://ex.org/Jill> .
<http://ex.org/Jim> <http://ex.org/padreDe> <http://ex.org/James> .
<http://ex.org/Jill> <http://ex.org/madreDe> <http://ex.org/James> .
<http://ex.org/James> <http://ex.org/nacioEn> <http://ex.org/Chile> .
<http://ex.org/James> <http://ex.org/estudiaEn> <http://ex.org/UniversidadDeChile> .
```

Código 2.2: Grafo de ejemplo en formato NTriples.

Los ejemplos anteriores describen el mismo grafo que corresponde al que se presenta a continuación en la Figura 2.1.

2.1.2. SPARQL

SPARQL Protocol and RDF Query Language (SPARQL) [3] es un lenguaje estandarizado de consultas sobre grafos que poseen una estructura RDF. Existen múltiples implementaciones para este lenguaje.

En SPARQL las consultas se construyen en un formato similar a RDF, incluyendo distintos operadores y permitiendo el uso de variables. A continuación, se muestra una consulta de ejemplo sobre el grafo anterior. Esta consulta busca obtener el país en el que nació James.

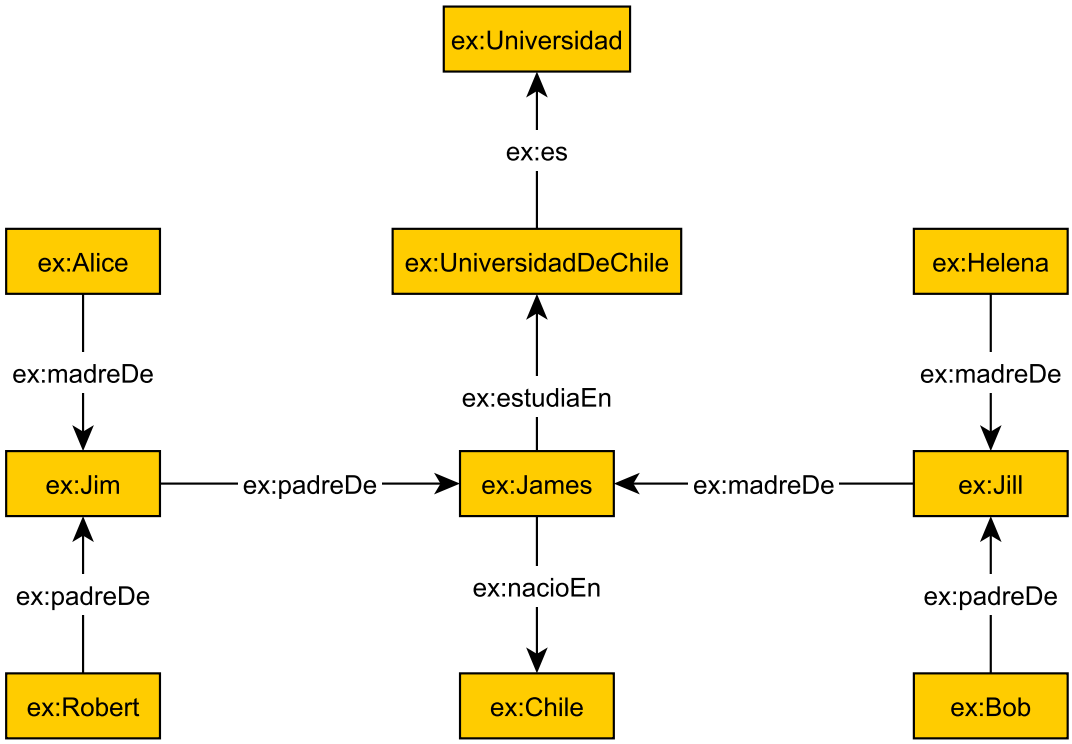


Figura 2.1: Grafo de ejemplo resultante de ambos formatos.

```
PREFIX ex: <http://ex.org/>

SELECT ?pais
WHERE
{
  ex:James ex:nacioEn ?pais .
}
```

Código 2.3: Ejemplo consulta en SPARQL.

La consulta del Código 2.3 posee los operadores de SPARQL; *SELECT* y *WHERE*. Tiene cómo variable *?pais* e incluye la estructura RDF (sujeto-predicado-objeto) donde `ex:James` es el sujeto, `ex:nacioEn` es el predicado y `?pais` es el objeto. Esta consulta obtendrá los países en los que nació James, sin embargo, al ser solo uno (Chile), entregará ese resultado.

2.1.3. Wikidata

Wikidata [1] es una fuente con mucha información relevante de todo tipo; actualmente posee 104.947.641[1] elementos, entre los que se incluyen países, artistas, animales, etc. También posee propiedades como: lugar de nacimiento, género, país, creador, etc. Esta gran cantidad

[1] Obtenido de datos oficiales https://www.wikidata.org/wiki/Wikidata:Statistics/es

de información se ha logrado mediante las 1.929.053.484[2] ediciones desde el lanzamiento del proyecto. Al ser de dominio público y colaborativa puede seguir expandiéndose y actualizando su información.

A continuación, en la Figura 2.2 se muestra una captura de una entidad vista en la página web de Wikidata. Además, se señalan el título e ID de Wikidata de la entidad y la ubicación de sus tripletas.

Figura 2.2: Captura de una entidad de la página web de Wikidata.

Sus características más importantes se enfocan en su estructura, teniendo elementos con un identificador único. La información en Wikidata se estructura como un grafo con el formato RDF, mediante declaraciones de relaciones entre dos entidades (de modo que posee aristas como relaciones y vértices como entidades).

Para consultar estos datos, la página web de Wikidata incluye un servicio de consultas en el lenguaje SPARQL [4]. También permite la descarga de los datos como archivos y acceso a APIs públicas para consultarlos.

[2] Obtenido de datos oficiales https://www.wikidata.org/wiki/Wikidata:Statistics/es

2.2. Recorrido sobre Grafos

Los datos de Wikidata poseen la estructura de un grafo, de modo que la búsqueda de caminos entre entidades implica recorrer en el grafo. Se requiere conocer de algoritmos en grafos para obtener de forma eficiente caminos entre entidades.

2.2.1. Algoritmos clásicos

La principal forma de buscar nodos en un grafo es recorrerlo junto a sus aristas. Para esto se usan principalmente dos algoritmos; búsqueda en anchura, en inglés *Breadth First Search* (*BFS*) y búsqueda en profundidad, en inglés *Depth First Search* (*DFS*). Estos dos permiten seleccionar solo algunos caminos entre nodos.

2.2.1.1. Depth First Search

En búsqueda en profundidad (*DFS*), se debe seleccionar un nodo inicial y desde este recorrer todos y cada uno de los nodos que va localizando, de forma recursiva, siguiendo solo un camino. Cuando no quedan más nodos que visitar en este camino, regresa y repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.

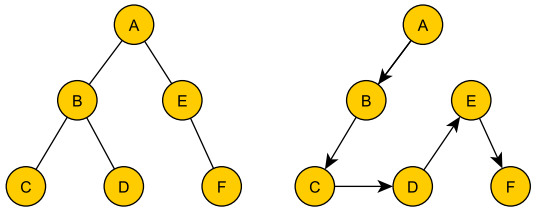


Figura 2.3: Ejemplo de llevar a cabo DFS sobre un grafo. A la izquierda el grafo original y la derecha el grafo resultante.

En la Figura 2.3 se lleva a cabo un DFS sobre el grafo de la izquierda iniciando en el nodo **A**. El resultado es el grafo de la derecha en el cual los arcos (x, y) entre los nodos indica que se visitó y inmediatamente después del nodo x en la DFS.

2.2.1.2. Breadth First Search

En búsqueda en anchura (*BFS*), se selecciona un nodo inicial y se exploran sus vecinos. Después para cada nodo vecino se exploran sus vecinos adyacentes: esto continúa hasta recorrer todo el grafo.

En la Figura 2.4, se lleva a cabo un BFS sobre el grafo de la izquierda iniciando en el nodo **A**. El resultado es el grafo de la derecha en el cual los arcos (x, y) entre los nodos indica que se visitó y inmediatamente después del nodo x en la BFS.

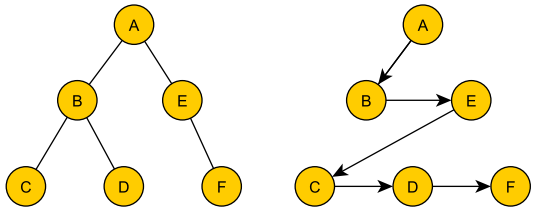


Figura 2.4: Ejemplo de llevar a cabo BFS sobre un grafo. A la izquierda el grafo original y la derecha el grafo resultante.

2.2.1.3. Implementación

Everitt et al. [5] estudian las propiedades y características de ambos algoritmos fundamentales de búsqueda (*BFS* y *DFS*). También analizan su tiempo y uso de recursos para recomendar cuál usar para recorrer según el tipo de grafo que se presente.

Además de estos algoritmos, existen otros para la obtención de caminos de menor costo entre nodos como *A**. Este algoritmo trabaja con una heurística que permite comparar caminos y, en la medida que descubra uno nuevo, compararlo con otro y seguir trabajando con el mejor.

Lie-Quan et al. [6] explican y desarrollan la implementación de una biblioteca de grafos. Plantean un modelo que incluye las clases aristas, vértices (o nodos) y al grafo completo, abstrayendo el concepto de grafo al lenguaje de programación C++. Sobre este diseño se implementan los algoritmos de búsqueda y recorrido de grafos *DFS* y *BFS*; además se incluyen evaluaciones de rendimiento para distintos casos.

En la cátedra del curso *Parallel and Sequential Data Structures and Algorithms* de la escuela de ciencias de la computación de la Universidad Carnegie Mellon [7] se muestra una representación de grafos usando solamente arreglos. La estructura consta de un arreglo cuyos índices coinciden con la información de los nodos, que incluye los valores de los vecinos de cada nodo. Esta estructura logra ser más liviana al usar arreglos nativos en lugar de listas o conjuntos de objetos.

2.2.2. Algoritmos multi-nodo

Los algoritmos clásicos inician desde un nodo y recorren de manera secuencial las aristas. Esto puede consumir mucho tiempo para algunos conjuntos de datos grandes. Además, involucra mucha computación redundante cuando se ejecuta múltiples veces desde diferentes vértices de inicio.

Then et al. [8] proponen el concepto de *Multi-Source BFS* (*MS-BFS*): un algoritmo diseñado para correr múltiples *BFSs* de manera concurrente y simultánea sobre el mismo grafo. Este algoritmo aprovecha las propiedades de grafos del tipo *Small World* (grafos donde la mayoría de los nodos no se conectan entre ellos, pero se puede llegar a la mayoría de los nodos desde cualquiera recorriendo pocas conexiones). Permite un recorrido eficiente ya que comparte información común entre cada *BFS* realizado y reduce el acceso aleatorio a la memoria.

2.3. Búsqueda de Caminos en la Web Semántica

Algunos trabajos ya han abarcado la búsqueda de caminos en la web semántica. Entre estos está la búsqueda de todos los caminos entre ciertas entidades, con el uso de SPARQL y la búsqueda de caminos más cortos entre entidades.

2.3.1. Property Paths en SPARQL

SPARQL incluye una opción para trabajar con caminos: *property paths*. Estas son expresiones regulares sobre caminos que permiten describir rutas entre nodos sobre grafos RDF. A continuación, se da un ejemplo de su uso:

```
1 PREFIX ex: <http://ex.org/>
2
3 SELECT ?abue
4 WHERE {
5    ?abue (ex:madreDe|ex:padreDe)/(ex:madreDe|ex:padreDe) ex:James .
6 }
```

Código 2.4: Código SPARQL usando property paths.

En el Código 2.4 se muestra una consulta de SPARQL que busca obtener los abuelos y abuelas de `ex:James`. Para esto se tiene una declaración que a la izquierda tiene la variable `?abue` que corresponderá a las entidades abuelos o abuelas de *James*, luego el camino usando *property paths* y a la derecha la entidad `ex:James`.

El *property path* usado es `(ex:madreDe|ex:padreDe)/(ex:madreDe|ex:padreDe)`, este caso usa dos símbolos: `|`, que corresponde a que la secuencia use cualquiera de los dos, y `/`, que corresponde a que tiene que incluir ambos (uno seguido del otro). Así, este *property path* permite incluir los cuatro caminos para obtener abuelos o abuelas, es decir, considerando la Figura 2.1: el padre del padre (`ex:Robert`), el padre de la madre (`ex:Bob`), la madre del padre (`ex:Alice`) y la madre de la madre (`ex:Helena`).

Los *property paths* son una herramienta avanzada para obtener nodos describiendo caminos con una expresión regular. Poseen otros símbolos para añadir más complejidad a los caminos. Si bien, permite obtener nodos conectados por caminos cuyas etiquetas coinciden con la expresión regular, no permite obtener las relaciones dentro de los caminos.

2.3.2. Sistemas para Visualizar Caminos

Se han desarrollado sistemas que permiten visualizar caminos entre nodos de grafos de la Web Semántica, teniendo enfoque en mostrar subgrafos, buscar caminos relevantes o, incluso, como herramienta didáctica.

2.3.2.1. RelFinder

Heim et al. [9, 10] implementan un algoritmo de obtención de caminos que unen entidades específicas en un grafo y lo usan en la aplicación RelFinder para desplegar relaciones entre elementos de forma interactiva. El algoritmo implementado se basa en consultas mediante SPARQL a los datos de DBpedia, como se ve en el Código 2.5.

```
SELECT * WHERE {
   db:Kurt_Cobain ?pf1 ?of1 .
   ?of1 ?pf2 ?c .
   db:Chile ?ps1 ?os1 .
   ?os1 ?ps2 ?c .
   FILTER ...
} LIMIT 13
```

Código 2.5: Ejemplo de consulta que usa RelFinder.

En el Código 2.5, se muestra un código similar al del documento, donde se obtienen las relaciones que unen a *db:Kurt_Cobain* con una entidad *?of1*, luego esta misma con la entidad *?c*, después une a la entidad *db:Chile* con la entidad *?os1* y por último la entidad *?os1* con la entidad *?c*. De esta forma, se obtiene un patrón de consulta que captura el camino que se ve en la Figura 2.5.

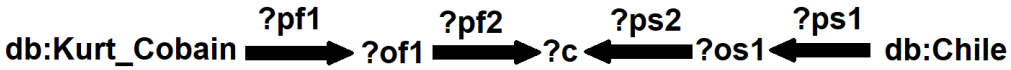


Figura 2.5: Camino implementado por la consulta del Código 2.5.

El camino de la Figura 2.5 posee el largo de 4 (*?pf1*, *?pf2*, *?ps2* y *?ps1*), sin embargo existen 16 caminos posibles de largo 4 (todas las combinaciones al cambiar el sentido de cada relación). La consulta del Código 2.5 incluye filtros para evitar los ciclos y un límite para acotar el número de resultados.

Este procedimiento crece de manera exponencial en consultas según el largo del camino que se requiera, necesitando 2^l sub-consultas, donde l es el largo del camino. Se pueden combinar estas consultas en una sola usando unión, pero puede resultar en una consulta muy larga y costosa.

RelFinder usaba los datos de DBpedia. La implementación se llevó a cabo con Adobe Flex y se compiló con Flash, así la aplicación podía ser usada al instalarse Flash Player; esto implica que ya no posee soporte ni se puede usar.

En la Figura 2.6 se muestra cómo era la aplicación RelFinder. Un aspecto muy importante de este sistema es cómo plantea la visualización de las relaciones y la interfaz. Se asigna un espacio para que el usuario pueda agregar entidades y otro espacio para el despliegue del grafo que se formaba, donde continuamente aparecía cada nueva relación encontrada.

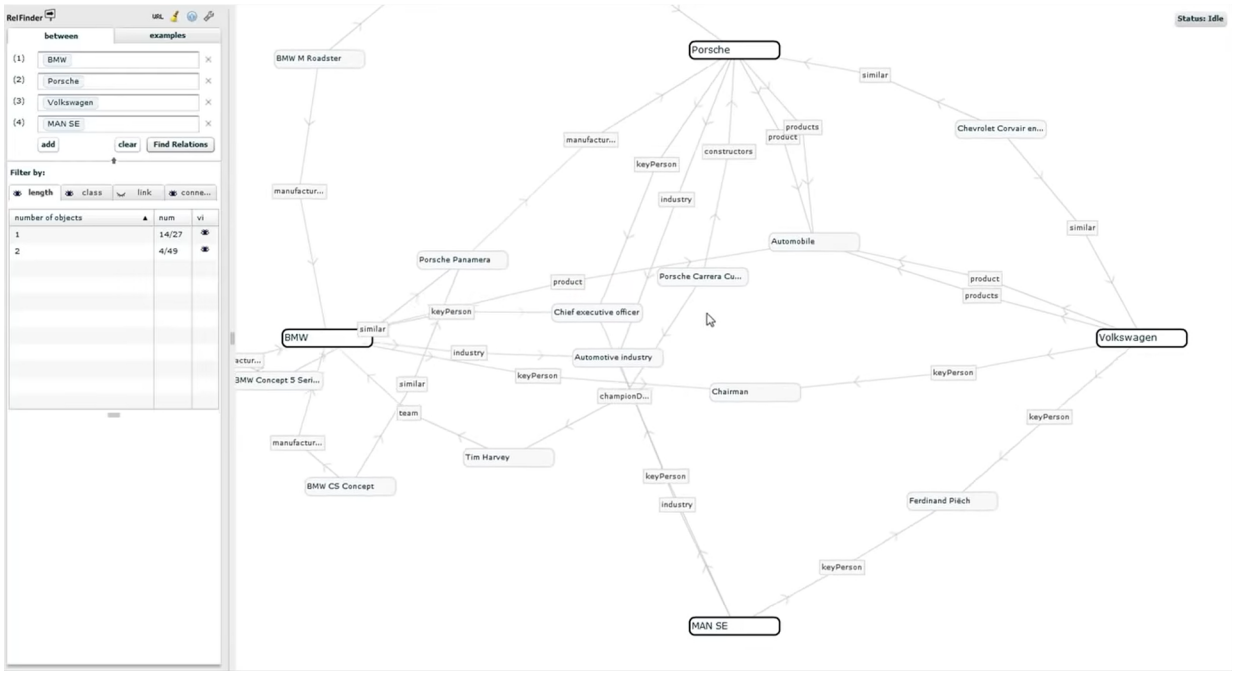


Figura 2.6: Captura de la aplicación RelFinder[3].

2.3.2.2. Búsqueda de Caminos Relevantes en Grafos RDF

Con enfoque en la eficiencia, Tartari et al. [11] estudiaron la búsqueda de caminos más cortos en grafos RDF entre dos entidades de Wikidata. Este trabajo plantea un estudio de la eficiencia en la búsqueda de estas relaciones. Usa la relevancia de los caminos basada en agregar un peso a las aristas y un valor según la cantidad de conexiones que enlazan a una entidad con las entidades que proveen estos enlaces. Además, desarrollaron una aplicación web, pero que permite solamente desplegar un camino entre dos entidades.

2.3.2.3. RDF Playground

Inostroza et al. [12] trabajaron en una aplicación web que apoye el curso "La web de datos" de la Universidad de Chile que se llama RDFPlayground. Esta aplicación permite visualizar, validar, consultar y actualizar grafos RDF. Un aspecto clave es la arquitectura usada en la solución, considerando velocidad de cómputo, bibliotecas para la visualización de grafos y un fuerte enfoque en aspectos visuales de la aplicación. Incluye la evaluación de la aplicación con usuarios. El sistema no permite buscar caminos entre entidades particulares.

La Figura 2.7 muestra la aplicación RDFPlayground, además despliega un grafo cargado anteriormente.

[3] Captura obtenida del video de la página oficial http://www.visualdataweb.org/relfinder.php.

Figura 2.7: Captura de la aplicación RDFPlayground[4].

2.4. Novedad

Se plantea diseñar e implementar un sistema interactivo que permita a usuarios explorar subgrafos y buscar múltiples caminos entre las entidades que elijan. Los usuarios podrán acceder a información trabajada de Wikidata sin requerir conocimientos de SPARQL.

Un aspecto fundamental es la necesidad de un nuevo algoritmo para la obtención de varios caminos entre múltiples nodos específicos. Esto ya que una solución mediante consultas con SPARQL es muy lenta al trabajar con datos de Wikidata y los resultados se obtienen al terminar la consulta.

[4] Captura obtenida de la página oficial http://rdfplayground.dcc.uchile.cl/.

Capítulo 3

Solución

A continuación, se describe la solución desarrollada, la cual está disponible en Github[5]. Esta se separa principalmente en dos partes; la búsqueda de caminos y la aplicación.

La búsqueda de caminos abarca la carga de los datos y el trabajo algorítmico. La carga de los datos incluye la creación de cachés; luego se describen estructuras de almacenamiento que usan estos cachés para ser pobladas. Finalmente se explica el algoritmo usado para la búsqueda de caminos entre entidades.

En la segunda parte se detalla el desarrollo de la aplicación. Esto incluye una descripción de la arquitectura general de la aplicación y herramientas usadas para el desarrollo y diseño de esta. También se explican las características que permiten llevar a cabo las búsquedas, entre las principales están: el autocompletado, la comunicación entre el usuario y la aplicación, el despliegue del grafo y los filtros de los caminos.

3.1. Búsqueda de caminos

Se desarrollan estructuras de almacenamiento de grafos a las que se cargan los datos de Wikidata. Luego se desarrolla un algoritmo de búsqueda de caminos entre los nodos del grafo.

3.1.1. Lenguaje de Programación

El lenguaje de programación que se decide usar es Java, orientándose a una máquina que posee Java 11.

Java, al ser un lenguaje de programación compilado, es más rápido en tiempo de ejecución que lenguajes interpretados. Esto se requiere para obtener más rápidamente los caminos entre nodos.

Java posee bibliotecas que permiten facilitar tareas dentro de la solución, las principales son: análisis gramático de los datos de Wikidata, para generar el grafo y la existencia de *frameworks* de desarrollo web que facilitarán la comunicación entre los resultados de la

[5] https://github.com/TinSlim/WD-PathFinder

búsqueda de caminos y el despliegue de estos al usuario.

Lo último que favorece la decisión es JVM (*Java Virtual Machine*) o bien la Máquina Virtual de Java. JVM es el software que ejecuta el programa de Java, permite la ejecución de un programa escrito en Java en cualquier dispositivo que posee JVM. Esto facilita la integración del programa en los servidores.

La decisión de usar Java 11 se debe a que es una versión LTS (Long Term Support) que aún posee soporte. Además, es la versión más común de ver en paquetes de Debian; esto permite incluir lo trabajado en otras máquinas sin requerir una mayor configuración.

3.1.2. Lectura de Datos de Wikidata

Wikidata permite la descarga de copias de sus datos. El archivo que se usa es *latest-truthy.nt.gz*; posee declaraciones que no son obsoletas según Wikidata, por ejemplo; incluye la población más reciente para las ciuidades, omite hechos disputados u obsoletos, como que la Tierra sea plana, etc. Este archivo posee el formato N-Triples y es un archivo comprimido usando la tecnología GNU zip (formato gz). Este archivo fue obtenido el día 30 de marzo del año 2023 correspondiendo a la versión del 24 de marzo del año 2023. Posee el tamaño de 59.456.991.776 bytes, que es aproximadamente 59 gigabytes en formato comprimido. Son 7.425.200.149 triples (líneas) los que posee este archivo. En la Figura 3.1 se muestra un ejemplo de cómo son las líneas del archivo *latest-truthy.nt.gz* al descomprimirlas.

```
<http://www.wikidata.org/entity/Q31> <http://www.wikidata.org/prop/direct/P1344> <http://www.wikidata.org/entity/Q108836> .
<http://www.wikidata.org/entity/Q31> <http://www.wikidata.org/prop/direct/P1151> <http://www.wikidata.org/entity/Q3247091> .
<http://www.wikidata.org/entity/Q31> <http://www.wikidata.org/prop/direct/P1546> <http://www.wikidata.org/entity/Q1308013> .
```

Figura 3.1: Ejemplo líneas descomprimidas del archivo *latest-truthy.nt.gz*.

3.1.2.1. Descompresión del archivo

El archivo está comprimido; para llevar a cabo su lectura y posterior creación de nodos y aristas se debe descomprimir. Dado el gran peso del archivo la descompresión completa de este tendría un tamaño muy grande y podría superar el almacenamiento del disco.

Se decide entonces hacer una lectura línea a línea, descomprimiendo solo la línea que se lee y con ella se crean las aristas y nodos según corresponda. Se usa la clase `FileInputStream` que permite la lectura del archivo por partes, sin sobrecargar la memoria de la máquina. Ya que el archivo además está comprimido, se usa la clase `GZIPInputStream`; esta recibe un *stream* y permite la lectura descomprimida de este. Ambas clases usadas están incluidas de forma nativa en Java.

3.1.3. Caché de los datos

Los datos que incluye el archivo *latest-truthy.nt.gz* no son solo conexiones entre entidades. También se incluyen valores como el nombre de las entidades, identificadores únicos, entre

otros. Estos datos no aportan en la búsqueda de caminos entre entidades. Eliminarlos permite acelerar la carga del grafo, disminuyendo la cantidad de líneas que se deben leer del archivo.

También, el archivo *latest-truthy.nt.gz* posee un formato que no permite asignar espacios de memoria antes o en la medida que se lee. Se requiere un trabajo sobre los datos para generar un caché que facilite la asignación de memoria y así poder lograr el almacenamiento del archivo en el grafo, usando arreglos nativos de Java.

3.1.3.1. Borrado de atributos

Se crea un caché del archivo *latest-truthy.nt.gz*. Corresponde a una copia del archivo, en el mismo formato pero que solo posee valores válidos, es decir, el primer y tercer valor de la tripleta son entidades e incluyen 'http://www.wikidata.org/entity/Q', el segundo valor corresponde a una propiedad e incluye 'http://www.wikidata.org/prop/direct/P'. Este nuevo archivo se llama *latest-truthy_small.nt.gz*.

En la Figura 3.2 se muestra un rectángulo superior que posee líneas de un archivo de ejemplo. Al aplicarle el filtro se reconoce que las líneas 2, 4 y 5 deben ser eliminadas ya que poseen valores de una propiedad y no unen entidades. Se obtiene el resultado de la conversión que corresponde al rectángulo de abajo en la figura.

Figura 3.2: Ejemplo de borrado de atributos. El rectángulo superior posee las líneas antes de ser filtradas y el rectángulo inferior posee las líneas resultantes.

Al borrar los atributos el archivo resultante posee 715.906.922 triples (líneas). Esto es equivalente a un 9.6 % del archivo original.

3.1.3.2. Caché Triple

Se requiere elaborar un caché que relacione a un nodo (mediante su ID de Wikidata) con las aristas que posee. El formato de este caché contendrá en cada fila el ID de Wikidata de un nodo y a continuación los índices de sus aristas. Además, estará ordenado por los IDs de Wikidata de los nodos.

Para esto, se leen las líneas de *latest-truthy_small.nt.gz*. A cada línea (arista) le asigna un índice entero a partir de 0. En un diccionario se almacenará como llave el ID de Wikidata que corresponde al nodo y como valor del diccionario, una lista con los índices de las aristas que le corresponden.

Al recorrer cada línea del archivo se escribirá la información en el diccionario para el nodo origen y nodo destino. Después de leer todas las líneas, el diccionario contendrá como llave los IDs de Wikidata de cada nodo y como valores los índices de las aristas que le corresponden a cada uno.

El diccionario resultante se ordena según sus llaves (IDs de Wikidata) y se exporta en el archivo comprimido *latest-truthy_small_triple.gz*.

En la Figura 3.3 se muestra un ejemplo de creación de este caché. El rectángulo superior de la figura posee líneas de un archivo de ejemplo en formato NTriples. Al llevar a cabo la conversión cada línea es indexada, en este caso desde 0 a 4. El resultado final es un archivo en que cada fila posee el ID de Wikidata de un nodo y a continuación los índices de las líneas en las que participa, como se muestra en el rectángulo inferior de la figura.

Figura 3.3: Ejemplo de creación de caché triple. El rectángulo superior posee las líneas antes de la conversión y el rectángulo inferior posee las líneas resultantes.

3.1.3.3. Caché Adyacencia

Se requiere elaborar un caché que relacione a un nodo (usando su ID de Wikidata) con los nodos vecinos que posee, mediante una arista. El formato de este caché contendrá en cada fila el ID de Wikidata de un nodo y a continuación grupos donde el primer valor es la etiqueta de una arista y seguidamente IDs de Wikidata de los nodos vecinos que conectan usando esa arista.

Para la construcción de este caché se leen las líneas de *latest-truthy_small.nt.gz*. Con cada línea se obtiene información para dos nodos; el origen y destino. Con esa información se rellena un diccionario que almacena como llave el ID de Wikidata de un nodo y los valores almacenados en el diccionario serán listas que incluyen la etiqueta de Wikidata de la arista junto a los IDs de Wikidata de los nodos que se conectan mediante esta. Además, el sentido de la arista será definido usando su etiqueta como negativo si los nodos junto a esta son origen. Si son destino el valor de la arista será positivo.

El diccionario resultante se ordena según sus llaves (IDs de Wikidata) y se exporta en el archivo comprimido *latest-truthy_small_adyacencia.gz*.

En la Figura 3.4 se muestra un ejemplo de creación de este caché. El rectángulo superior de la figura posee líneas de un archivo de ejemplo en formato NTriples. Al llevar a cabo la

conversión de cada línea, genera información de los nodos origen y destino; en el caso de la primera y segunda fila se obtiene que el nodo 31 conecta con los nodos 32 y 33 mediante la arista de etiqueta 32 y, los nodos 32 y 33 conectan mediante la etiqueta -32 (ya que es sentido contrario se usa el valor negativo) con el nodo 31. El resultado final es un archivo en que cada fila posee el ID de Wikidata de un nodo y a continuación la etiqueta de una arista y los IDs de Wikidata de los nodos que conectan mediante esta, como se muestra en el rectángulo inferior de la figura.

Figura 3.4: Ejemplo de creación de caché adyacencia. El rectángulo superior posee las líneas antes de la conversión y el rectángulo inferior posee las líneas resultantes.

3.1.4. Estructura de almacenamiento de datos

La solución propuesta requiere almacenar los datos de *latest-truthy_small.nt.gz* en memoria principal y permitir consultar los vecinos de cada nodo rápidamente. Se elaboraron distintas propuestas de estructura que se abordarán a continuación.

3.1.4.1. Objetos de Java

La primera intuición fue usar clases y objetos de Java. La ventaja que posee es ser una abstracción de fácil interpretación y programación, además extensible.

Este grafo se pobla con los datos del caché *latest-truthy_small.nt.gz*. Se elaboran tres clases principales. Las estructuras que usa se describen a continuación:

- Grafo

  Se usa la clase `Graph` como abstracción del grafo: posee un `HashMap` que almacena los nodos (`Vertex`) indexando con el ID de Wikidata que le corresponde.

- Nodos

  Se usa la clase `Vertex` como abstracción del nodo: almacena el ID que le corresponde en Wikidata y una lista enlazada (`LinkedList` de Java) para almacenar sus aristas.

- Aristas

  Se usa la clase `Edge` como abstracción de una arista: posee su etiqueta según Wikidata y referencias al nodo origen y al nodo destino.

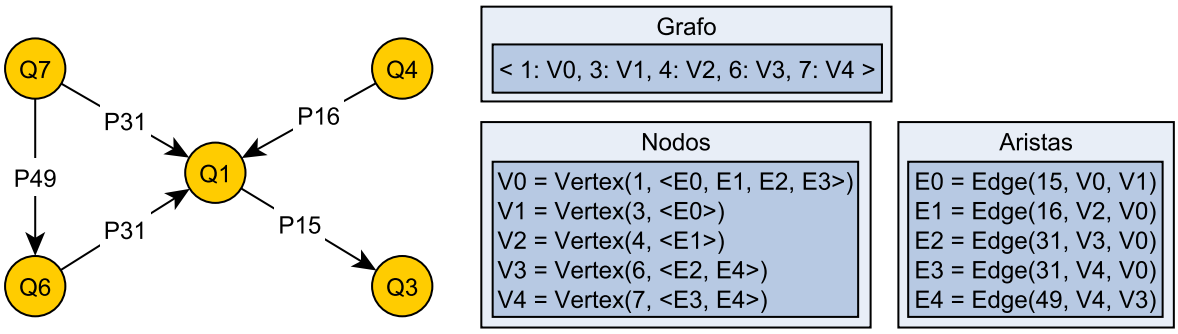


Figura 3.5: Ejemplo estructura con objetos de Java.

En la Figura 3.5 se muestra un grafo y su representación usando clases de Java. **Grafo** posee el `HashMap` donde almacena los nodos usando como llave sus ID de Wikidata. **Nodos** son los objetos que incluye el `HashMap` y poseen una lista con las aristas que le corresponden al nodo. **Aristas** son los objetos que representan a las conexiones entre nodos, incluyendo referencias al nodo origen y nodo destino.

Esta estructura usa clases, algunas nativas de Java como `HashMap` y `LinkedList`. Otras fueron implementadas: `Graph`, `Vertex` y `Edge`.

3.1.4.2. Triples Densos

La segunda intuición es almacenar las aristas (su origen, destino y etiqueta) separadas de la información de un nodo (vecinos que posee). Para esto se usa un arreglo nativo (`int[ ][ ]`) que almacena las aristas y otro arreglo nativo (`int[ ][ ]`) que almacena el ID de un nodo con los índices de las aristas que le corresponden.

En este caso se busca disminuir la cantidad de memoria usada recurriendo al almacenamiento en arreglos nativos. Se almacenan las aristas como triples y los nodos en un arreglo denso, ya que no posee filas vacías. A continuación, se describen las estructuras que usa:

- Nodos (`int [cantidad de nodos][ ]`)

  Es un arreglo de dos dimensiones; cada fila corresponde a la información de un nodo. Esto incluye el ID de Wikidata que le corresponde al nodo y los índices de las aristas que lo incluyen. Para la creación del arreglo completo se usa el archivo *latest-truthy_small_triple.gz*.

- Aristas (`int[cantidad de aristas][3]`)

  Es un arreglo de dos dimensiones; para cada arista almacena tres valores: el ID de Wikidata del nodo origen, la etiqueta de la arista y el ID de Wikidata del nodo destino. Para la creación de este arreglo se usa el archivo *latest-truthy_small.gz*.

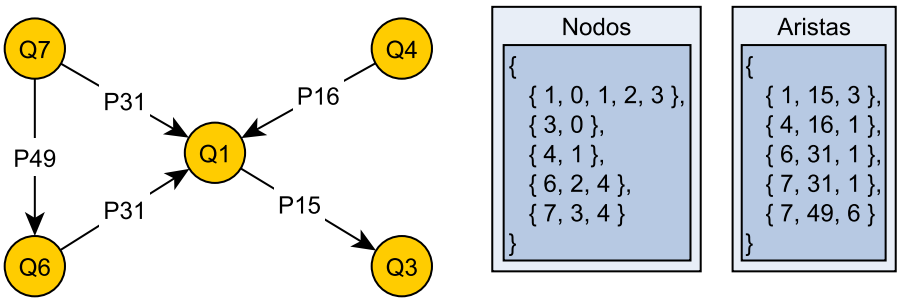


Figura 3.6: Ejemplo estructura de Triples Densos.

En la Figura 3.6 se muestra un grafo y su representación usando arreglos nativos. **Nodos** posee el arreglo donde se almacena el ID de Wikidata y los índices de las aristas que le corresponden para cada nodo; por ejemplo, el arreglo `{ 6, 2, 4 }` indica que el nodo con ID 6 está en las aristas con índice 2 y 4 (los índices empiezan desde el 0). **Aristas** es el arreglo que incluye ID del nodo origen, etiqueta de la arista e ID del nodo destino para cada arista; por ejemplo, la arista `{ 6, 31, 1 }` indica que el nodo con ID 6 apunta al nodo de ID 1 con la arista de etiqueta 31.

En esta estructura, no existen filas vacías y para buscar las aristas de un nodo se debe hacer una búsqueda binaria con el primer elemento de la fila. Si bien logra la carga completa del grafo, se puede mejorar el tiempo de búsqueda de un nodo.

3.1.4.3. Triples no Densos

Esta intuición sigue la idea de almacenar las aristas (su origen, destino y etiqueta) separadas de la información de un nodo (vecinos que posee), usando arreglos nativos (`int[ ][ ]`). Pero, en el arreglo de los nodos no se almacena el ID de Wikidata sino que cada nodo corresponde con su índice en el arreglo.

Esta abstracción posee una idea similar a la anterior, almacenando las aristas en triples. Sin embargo, aprovecha que los IDs de las entidades de Wikidata son incrementales y densos, de modo que el mayor ID de las entidades de Wikidata será mayor o igual a la cantidad de nodos. Esta estructura posee filas vacías. Se usan las siguientes abstracciones:

- Nodos (`int[máximo ID de Wikidata de los nodos + 1] [ ]`)

  Es un arreglo de dos dimensiones, y está indexado usando el ID de Wikidata que le corresponde a un nodo. Almacena un arreglo con los índices de las aristas que le corresponden. Para la creación del arreglo completo se usa el archivo *latest-truthy_small_triple.gz*.

- Aristas (`int[cantidad de aristas][3]`)

  Es un arreglo nativo de dos dimensiones; para cada arista almacena tres valores: el ID de Wikidata del nodo origen, la etiqueta de la arista y el ID de Wikidata de nodo destino. Para la creación de este arreglo se usa el archivo *latest-truthy_small.gz*.

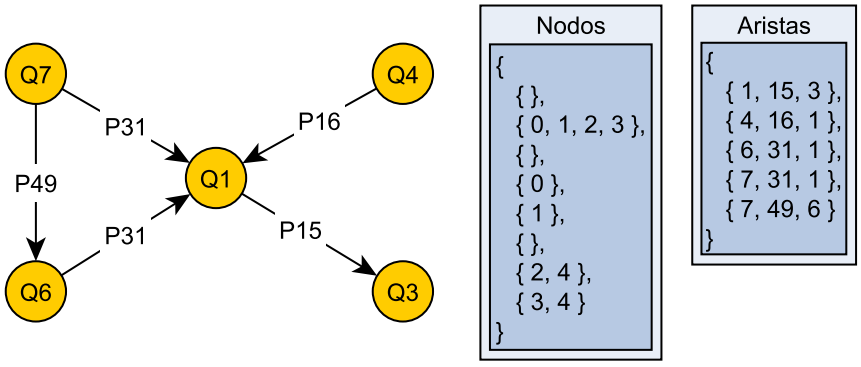


Figura 3.7: Ejemplo estructura Triples no Densos.

En la Figura 3.7 se muestra un grafo y su representación usando arreglos nativos e índices. **Nodos** posee el arreglo donde se almacenan los índices de las aristas que le corresponden a cada nodo según su índice; por ejemplo, la línea con índice 6; `{ 2, 4 }` indica que el nodo de ID 6 tiene dos aristas, la de índice 2 e índice 4 en el arreglo de aristas. **Aristas** es el arreglo que incluye ID nodo origen, etiqueta e ID nodo destino para cada arista, tal como en la estructura Triples Densos.

Esta idea sí logra la carga completa de los datos, incluyendo información de las aristas y los nodos.

3.1.4.4. Adyacencia Densa

Esta idea intenta mejorar el uso de memoria principal. Almacena los datos usando menos valores enteros para describir al grafo. Para esto almacena aristas y nodos en un mismo arreglo nativo (`int [ ][ ][ ]`) juntando las dos abstracciones principales de un grafo en un solo arreglo tridimensional.

Se caracteriza por el concepto de adyacencia en el arreglo y lo mantiene denso al no contener filas vacías. A continuación, se describen las estructuras que posee:

- Grafo (`int [cantidad de nodos][ ][ ]`)

  Es un arreglo que almacena en cada fila información de un nodo. Esta información incluye primero un arreglo de un solo valor con el ID de Wikidata que le corresponde al nodo y a continuación arreglos que incluyen la etiqueta de Wikidata de la arista y los IDs de Wikidata de los nodos con los que se conecta usándola. Usa el archivo *latest-truthy_small_adyacencia.gz* para cargar los datos.

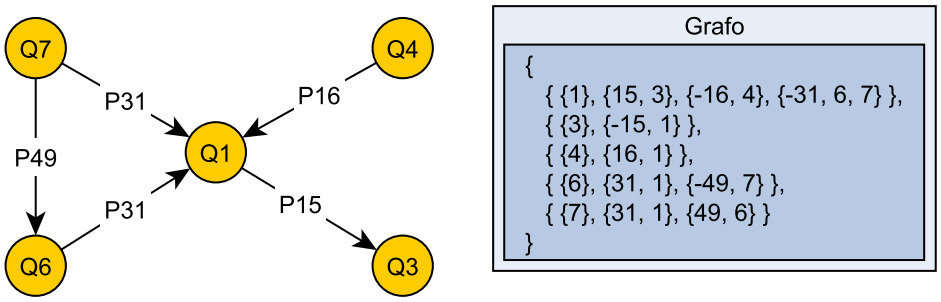


Figura 3.8: Ejemplo estructura Adyacencia Densa.

En la Figura 3.8 se muestra un grafo y su representación usando la estructura Adyacencia Densa. **Grafo** posee el arreglo donde almacena el ID de Wikidata y las conexiones que posee, para cada nodo, por ejemplo: `{ {6}, {31, 1}, {-49, 7} }` significa que el nodo de ID 6 tiene dos aristas, la primera con la etiqueta 31 que va hacia el nodo de ID 1 y la segunda de etiqueta 49 que es desde el nodo de ID 7.

Esta estructura requiere usar búsqueda binaria para encontrar la información de un nodo.

3.1.4.5. Adyacencia no Densa

Esta intuición sigue la idea de almacenar las aristas y nodos en un mismo arreglo (`int [ ][ ][ ]`). Pero, en las filas no se almacena el ID de Wikidata sino que cada nodo corresponde con su índice en el arreglo.

Esta abstracción posee una idea similar a la anterior, manteniendo el concepto de adyacencia. Aprovecha que los ID de las entidades de Wikidata son incrementales y densos, de modo que puede tener filas vacías que implican que ese nodo no existe en los datos. A continuación se describen las estructuras que posee:

- Grafo (`int [máximo ID de Wikidata de los nodos + 1][ ][ ]`)

  Es un arreglo que almacena en cada fila información de un nodo. Esta información posee arreglos que incluyen la etiqueta de Wikidata de la arista y los ID de Wikidata de los nodos con los que se conecta usándola. La información de una fila corresponde su índice con el ID de Wikidata del nodo. Usa el archivo *latest-truthy_small_compressed.gz* para cargar los datos.

En la Figura 3.9 se muestra un grafo y su representación usando la estructura Adyacencia no Densa. **Grafo** posee el arreglo donde almacena las conexiones para cada nodo, por ejemplo: en la línea con índice 6; `{ {31, 1}, {-49, 7} }` significa que el nodo de ID 6 (de acuerdo al índice) tiene dos aristas, la primera con la etiqueta 31 que va hacia el nodo de ID 1 y la segunda de etiqueta 49 es desde el nodo de ID 7.

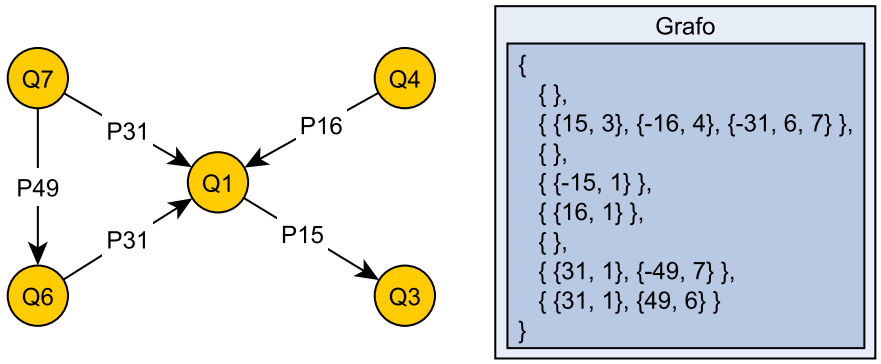


Figura 3.9: Ejemplo estructura Adyacencia no Densa.

Esta estructura mantiene el poco uso de memoria principal (asumiendo un intervalo denso de IDs en Wikidata) y la búsqueda de nodos es rápida ya que corresponden con su índice en el arreglo.

3.1.5. Algoritmo de búsqueda de caminos

El algoritmo debe permitir la búsqueda de caminos entre dos o más entidades, además de la concurrencia ya que se implementará su uso en una aplicación web. La última consideración es que sea soportado por la memoria disponible de la máquina.

Se plantea desarrollar un algoritmo que recorra el grafo ya cargado usando el algoritmo BFS desde nodos elegidos anteriormente.

Se usa un algoritmo de BFS multinodo, es decir, desde cada nodo inicial se llega a sus vecinos, luego a los vecinos de estos y se sigue repitiendo. Existirán BFSs para cada nodo inicial. Ya que los caminos buscados poseen un largo máximo, se hace crecer cada BFS en largo uno. Esto se repite hasta llegar a la mitad de largo máximo; en caso de que el largo máximo sea impar, se llega a la mitad más uno.

Los nodos que recorre se almacenan, formando un subgrafo del grafo ya cargado. En la medida que se recorre, cada nodo almacena los nodos con los que se llegó a este, el nodo que inició su BFS, la distancia a este y la distancia al inicio de un BFS distinto. La distancia a un BFS distinto inicia como “-1” y cuando ese nodo pertenece a un camino válido, se actualiza.

Cada BFS tiene un distinto inicio; cuando dos BFSs se intersectan, es decir uno llega a un nodo al que ya se llegó mediante otro BFS, significa que se encontró un camino. En ese caso se revisa que el camino sea válido de acuerdo con el largo máximo definido. Si lo es, se reportan los dos nodos y las aristas que los unen y, se lleva a cabo un *back tracking* para ambos nodos, lo que reporta nodos y aristas que llegan al inicio de sus BFSs.

El concepto de *back tracking* corresponde a recorrer en reversa desde un nodo, siguiendo de manera sucesiva los nodos que permiten llegar al actual. Esto despliega muchos candidatos de caminos; para encontrar caminos válidos revisa que no se repitan nodos (para que no ocurran ciclos) y que lleguen a un nodo de los que se elige en el inicio.

Otro caso posible es cuando un BFS alcanza un nodo al que ya se llegó mediante el mismo BFS pero avanzando en otra dirección. Si el nodo al que se llega no pertenece a un camino válido, se agrega el otro nodo a la lista de nodos que llegan a él. Si, por otro lado, pertenece a un camino válido, significa que se encontró otro camino válido y se lleva a cabo un *back tracking* desde ese nodo. También se reportan las aristas entre ambos nodos y el nuevo nodo.

A continuación, se presentará el pseudocódigo del algoritmo. Se requiere primero el concepto de `VertexWrapper` (VW) que será una copia del nodo que almacena la distancia al inicio de su BFS, la distancia al inicio de un BFS distinto, el ID de su BFS (se usa el nodo que lo inicia), un `Set` de nodos que llegan al nodo actual y la cantidad de veces que ha sido incluido en la cola. En el Código 3.1 se muestran sus variables.

```
VW // Vertex Wrapper
 id       // ID de Wikidata del nodo
 SCD      // Same Color Distance: Cantidad de aristas hacia el inicio de su BFS
 OCD      // Other Color Distance: Cantidad de aristas hacia el inicio de otro BFS
 from     // Set<VW> de nodos que llegan al nodo actual
 BFSid    // Nodo inicial de su BFS
 qTimes   // Veces que se ha unido el nodo a la cola (q)
```

Código 3.1: Definición de nodos.

A continuación, en el Código 3.2 se muestra el pseudocódigo del algoritmo. Incluye también comentarios para ayudar a la interpretación de algunas líneas. En este pseudocódigo se usan funciones planteadas en el Código 3.3.

Código 3.2: Pseudocódigo del algoritmo.

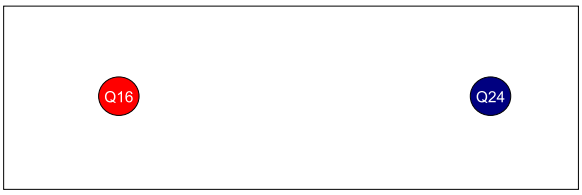
```
// Variables externas
g = Grafo;                    // Grafo con datos cargados
Map<Integer, VW> n;           // Map que almacena los nodos visitados indexando con el id
Set<Integer> p;               // Set de nodos pertenecientes a caminos válidos

// Función de búsqueda de caminos de un largo máximo entre nodos iniciales.
// iNodes: ID entero de los nodos entre los que se buscan caminos
// s       : Largo máximo de los caminos
// RETURN Imprime nodos y aristas de caminos entre nodos iniciales que no superan el
    ↪ largo máximo y no se han impreso
function search (Set<Integer> iNodes, int s):
   List<VW> q = new List<VW>() // Instancia Queue
   addInitialNodes(iNodes,n,q,p)   // Añade nodos iniciales a estructuras de almacenamiento
   WHILE (q.size() > 0):
      VW aVW = q.dequeue()         // Obtiene un nodo de la Queue
      IF (aVW.SCD > (s/2) + s%2):  // Si el nodo no pertenece a caminos del largo máximo
         CONTINUE
      FORALL (adjV IN g.getAdjacentVertex(aVW.id)):
         IF (aVW.id == adjV):     // Si nodo adyacente es si mismo
            CONTINUE
         IF (n.get(adjV) == null): // Si no se ha instanciado una copia del nodo
            VW newVW = new VW (aVW, adjV);
```

```
            n.put(adjV, newVW);
            q.add(newVW);
         ELSE:
            VW bVW = n.get(adjV)
            IF (aVW.onlyParent(bVW)): // Si solo ha sido visitado una vez
               CONTINUE               // y fue por el nodo adyacente
            IF (aVW.BFSid == bVW.BFSid):  // Si nodo actual y adyacente pertenecen a
    ↪ mismo BFS
               IF (bVW.OCD > -1 && bVW.OCD + aVW.SCD + 1 <= s):  // Si adyacente
    ↪ pertenece a un camino válido
                  IF (aVW.OCD == -1):  // Si actual no pertence a camino válido
                     aVW.OCD = bVW.OCD + 1
                     backTracking(aVW, size, iNodes)
                  makeEdges(bVW,aVW)
               IF (bVW.qTimes == 1): // si el nodo adyacente se unió a la cola una sola vez
                  q.enqueue(bVW)      // se agrega a la cola
                  bVW.qTimes += 1
               bVW.from.add(aVW)
            ELSE: // Si nodo actual y adyacente son de distinto BFS
               IF (bVW.SCD + aVW.SCD + 1 <= s): // Si el camino que forma es menor a
    ↪ largo máximo
                  makeEdges(bVW,aVW)
                  IF (bVW.OCD != -1):
                     bVW.OCD = min(bVW.OCD,aVW.SCD + 1)
                  ELSE:
                     bVW.OCD = aVW.SCD + 1
                     backTracking(bVW, s, iNodes)
                  IF (aVW.OCD != -1):
                     aVW.OCD = min(aVW.OCD,bVW.SCD + 1)
                  ELSE:
                     aVW.OCD = bVW.SCD + 1
                     backTracking(aVW, s, iNodes)
```

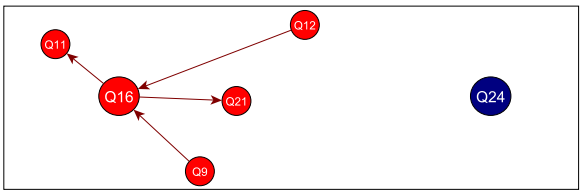
```
// Recorre hacia el inicio de un BFS encontrando caminos válidos, imprime nodos y aristas
// que pertenecen a caminos válidos
// nVW  : Nodo desde el que se hace backtracking
// s      : Largo máximo de los caminos
// iNodes: Nodos iniciales
// RETURN Imprime nodos y aristas de caminos que no superan el largo máximo desde un
    ↪ nodo hasta su nodo inicial
function backTracking (nVW,s,iNodes)

// Imprime nodos y aristas que corresponden a un camino válido
// aVW: Nodo 1
// bVW: Nodo 2
// RETURN Imprime dos nodos y aristas entre ellos si no han sido impresas
function makeEdges (aVW,bVW)

// Instancia VertexWrapper para cada nodo inicial y los añade a las estructuras de
//almacenamiento
// iNodes: Nodos iniciales
// n      : Map (diccionario) de nodos.
// q      : Queue (cola) de nodos que extraen y revisan
// p      : Set (conjunto) de nodos que pertenecen a caminos
// RETURN No devuelve nada
function addInitialNodes(iNodes,n,q,p)
```

Código 3.3: Funciones auxiliares.

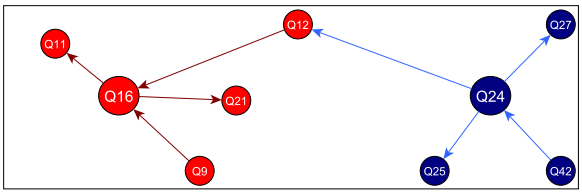
En la Figura 3.10 se muestra un ejemplo de la ejecución del algoritmo. Al inicio en la Figura 3.10.a se presentan dos nodos iniciales entre los que se ejecutará una búsqueda de caminos de largo máximo 3. A continuación en la Figura 3.10.b se obtienen los nodos adyacentes del nodo inicial rojo, avanzando el BFS rojo en un paso. En la Figura 3.10.c se lleva a cabo con el nodo de color azul, resultando en avanzar un paso el BFS de color azul. En este caso se encuentra un camino de largo 2, lo que se marca en verde en la Figura 3.10.d. Nuevamente avanza el BFS de color rojo resultando en la Figura 3.10.e, donde aparece un nodo que se une con un camino existente. Esto genera nuevos caminos entre los nodos iniciales marcándose en verde en la Figura 3.10.f. El BFS azul avanza una vez en la Figura 3.10.g encontrando nuevos nodos y un camino, sin embargo, este camino supera el largo máximo de 3. Además ambos BFS alcanzaron el largo máximo posible para seguir encontrando caminos de largo 3. El algoritmo finaliza y resulta en el subgrafo de la Figura 3.10.h.



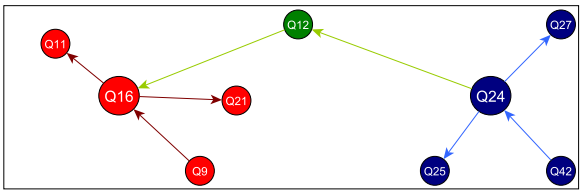
(a) Nodos iniciales de búsqueda.



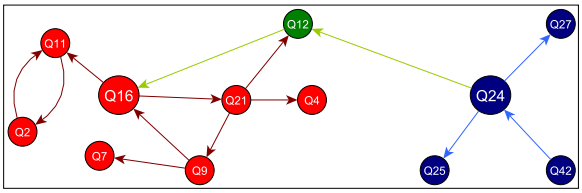
(b) Se obtienen vecinos del nodo inicial rojo.



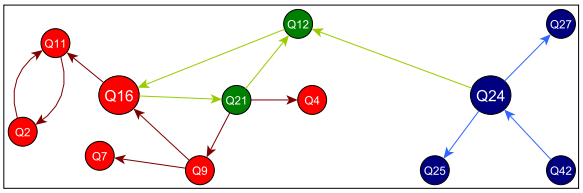
(c) Se obtienen vecinos del nodo inicial azul.



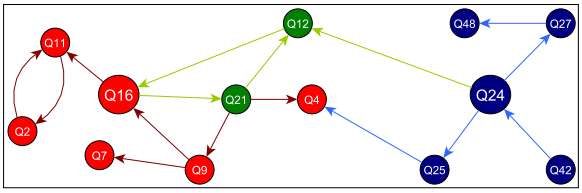
(d) La intersección de dos BFSs significa que hay un camino.



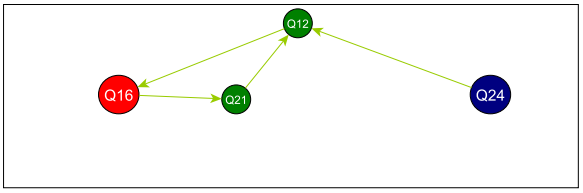
(e) Se obtienen vecinos de los nodos rojos más lejanos.



(f) Un BFS llega a un camino existente, significa que hay otro camino.



(g) Se obtienen vecinos de los nodos azules más lejanos. Se encuentra un camino de largo que supera el largo máximo.



(h) Subgrafo resultante.

Figura 3.10: Ejemplo de uso del algoritmo en la búsqueda de caminos entre dos nodos.

3.1.6. Optimización del Algoritmo

Este algoritmo de búsqueda de caminos obtiene muchos nodos y caminos que pueden no ser relevantes y se consideran como ruido. Esto ocurre cuando hay entidades intermedias de muchas aristas, por ejemplo, *humano*. En caso de que una entidad sea un país y otra se conecte con *humano* consideraría como caminos a todos los humanos de ese país.

Para evitar el ruido, cuando un nodo posee más de una cantidad definida de vecinos, estos vecinos no se incluyen en la búsqueda de caminos. Esta restricción no se hace si son nodos iniciales de la búsqueda. Implementar esto limita los caminos entre nodos intermedios de

muchas aristas, sin embargo, se permite que usen estos nodos si vienen de uno que posea menos aristas que las de límite.

La opción anterior mejora el uso de memoria de la aplicación, ya que no crea siempre todos los nodos vecinos (se harán experimentos en el Capítulo 4 para ver el efecto de este filtro). Además, disminuye la cantidad de ruido y permite obtener caminos que puedan ser más interesantes al contener nodos con menos vecinos.

3.2. Aplicación

Se desarrolla una aplicación que permita a los usuarios realizar las búsquedas de caminos y desplegar un subgrafo con los resultados. Esta aplicación debe usar el algoritmo planteado en la sección anterior. Los caminos buscados se definen con un largo máximo de **3 aristas**. Para que el algoritmo se incluya de forma nativa a la aplicación se plantea su desarrollo también usando Java.

La aplicación web se llama **WoolNet**. Su nombre se basa en la imagen de una lana enrollada en la que no se logra distinguir un inicio o un final. Pero si se ven los hilos. Esto es una analogía con los grafos resultantes en la aplicación.

3.2.1. Arquitectura

La arquitectura de la aplicación es de carácter monolítica. Al momento de iniciar la aplicación se realiza la carga de los datos de Wikidata desde el caché construido.

La aplicación posee tres controladores descritos a continuación:

1. Índice: Es un controlador que despliega la plantilla de la aplicación, es decir, el contenido estático.
2. Autocompletado: Es una API que recibe texto y retorna un JSON con propuestas de autocompletado. Estas propuestas incluyen ID, etiqueta y descripción de las entidades.
3. Obtención de Caminos: Es un WebSocket. Recibe los IDs de entidades junto al idioma de búsqueda e inicia la búsqueda de caminos usando el algoritmo implementado. Envía nodos y aristas que pertenecen a los caminos.

Además, la aplicación realiza consultas a la API de Wikidata. Esto se lleva a cabo en el autocompletado y la obtención de caminos. Los *endpoints* de Wikidata que se consultan son:

1. `wbsearchentities`: Este *endpoint* se usa para obtener los autocompletados. Entregando el ID de la entidad, etiqueta y su descripción.
2. `wbgetentities`: Este *endpoint* se usa para obtener las imágenes de las entidades. Entrega los valores de las propiedades para una entidad.

A continuación, la Figura 3.11 muestra un diagrama con los componentes descritos anteriormente y cómo se conectan entre ellos. Se incluye también al usuario que se comunica con WoolNet.

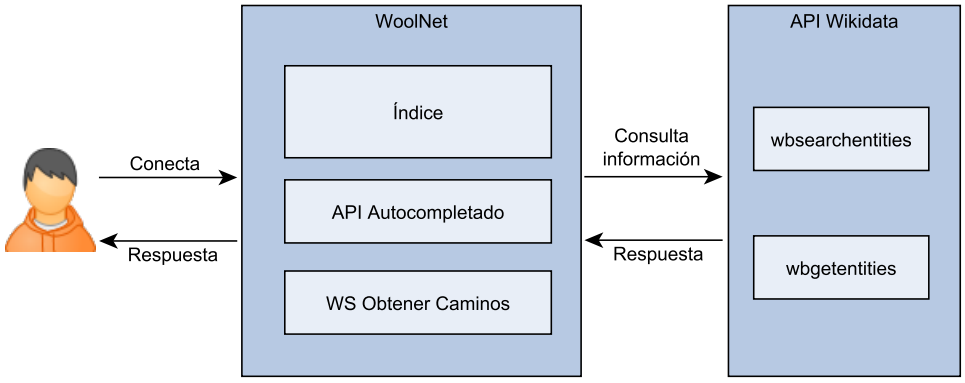


Figura 3.11: Diagrama de los componentes.

3.2.2. Framework

Se usa el *framework* Spring. La razón principal es que al ser un *framework* de Java, permite incluir el desarrollo del algoritmo de manera nativa.

También se usa la herramienta Spring Boot que facilita el inicio de un proyecto de Spring además permite incluir las dependencias y características propias de Spring para el manejo de la aplicación.

3.2.3. Diseño general

El diseño de la aplicación web está incluido en el servidor de Spring. Se desarrolla usando la biblioteca de JavaScript, Reactjs[6]. Esto facilita el desarrollo y mejora la extensibilidad del código.

Se usan las bibliotecas MUI[7] y Bulma[8] para agregar estilo y dinamismo al diseño. Además permiten el uso de iconos, fuentes, textos y colores que incluyen ambas bibliotecas. En la Figura 3.12 se muestra cómo se ve la aplicación al entrar desde un computador.

3.2.4. Autocompletado

El usuario posee una entrada para texto. Al escribir en esta se despliegan entidades con su descripción, nombre e ID de Wikidata para que pueda seleccionarlas. Esto corresponde a un autocompletado que facilita al usuario encontrar las entidades entre las que quiere buscar caminos.

[6] React: The library for web and native user interfaces
[7] MUI: The React component library you always wanted.
[8] Bulma: the modern CSS framework that just works.

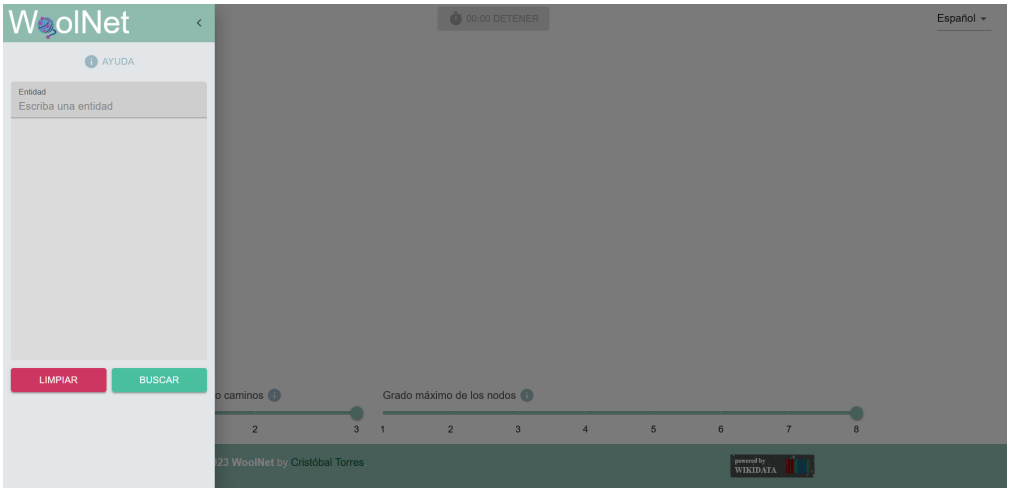
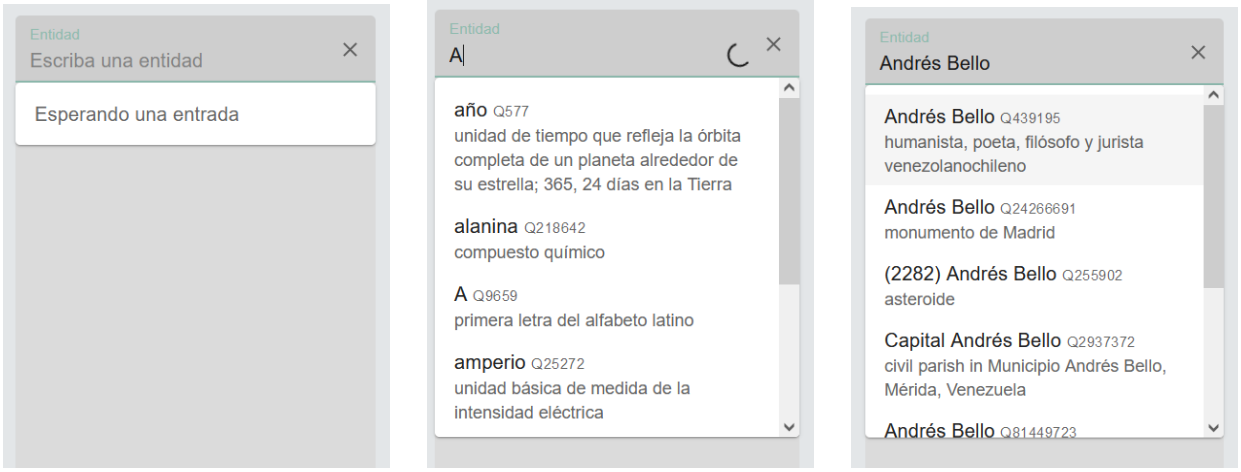


Figura 3.12: Captura de la aplicación WoolNet.

Se desarrolla un *endpoint* en la aplicación que consulta a la API de acceso público que incluye la web de Wikidata. Con esto, cuando el usuario ingrese caracteres en la entrada de texto, se consultará al *endpoint* y se desplegará al usuario un grupo de posibles entidades que cumplan con los caracteres ingresados.

El *endpoint* de la API de Wikidata es: https://www.wikidata.org/w/api.php?action=wbsearchentities&format=json. A este *endpoint* se le puede agregar el parámetro *search*, con el texto que se desea autocompletar. La respuesta propone entidades entregando su información; esto incluye la etiqueta de la entidad (o nombre) y su ID de Wikidata. Además, se pueden incluir los parámetros `uselang` y `lang` para obtener un autocompletado en otro idioma.

En la Figura 3.13 se ven las tres etapas del autocompletado; en la Figura 3.13.a se selecciona sin escribir texto; a continuación, en la Figura 3.13.b se comienza a escribir, esto genera las primeras propuestas; en la Figura 3.13.c se escribe completamente la entidad donde la primera propuesta corresponde con la entidad escrita.



(a) Autocompletado sin entrada.

(b) Autocompletado proponiendo entidades.

(c) Autocompletado con la entrada ya escrita.

Figura 3.13: Autocompletado de la aplicación.

3.2.5. Envío de entradas del usuario y recepción de respuestas

A partir del autocompletado, cuando el usuario clickea una entidad de las que se despliegan, se almacena su ID. Puede repetir esto seleccionando cuantas entidades desee. Luego, cuando el usuario quiera buscar los caminos entre las entidades ya seleccionadas, clickea el botón BUSCAR. A continuación, se envían los IDs al servidor Spring mediante un WebSocket incluyendo el idioma actual.

El WebSocket es una vía de comunicación bidireccional. Cuando el servidor recibe los IDs de Wikidata de las entidades, lleva a cabo el algoritmo de búsqueda de caminos. Mientras encuentra caminos válidos, envía aristas y nodos al usuario mediante el WebSocket en un formato de diccionario.

En la Figura 3.14 se ven tres entidades seleccionadas; cada una posee un botón para quitarla. También se tiene el botón *LIMPIAR* que permite quitar todas las entidades seleccionadas. Finalmente el botón *BUSCAR* inicia la búsqueda de caminos.

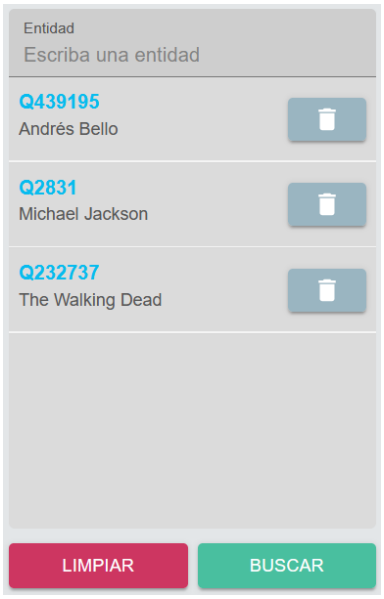


Figura 3.14: Entidades seleccionadas.

3.2.6. Despliegue del Grafo

Cuando el WebSocket recibe información en el lado del usuario, reconoce si es un *Nodo* o una *Arista* y los dibuja en la pantalla según corresponda. También existe información del tipo *Editar*; esta corresponde a información nueva de un nodo para su actualización. La información que se puede actualizar es: la posición de los nodos iniciales o el grado del camino al que pertenece un nodo.

Para desplegar el grafo se usa la biblioteca de JavaScript VISjs. Esta biblioteca de visualización permite el manejo de grandes cantidades de datos, junto a su manipulación e interacción. Se usan las componentes `DataSet` para almacenar nodos y aristas y `Network` para el despliegue de los caminos entre entidades que se van formando.

En la Figura 3.15 se muestra un grafo de los caminos que unen tres entidades.

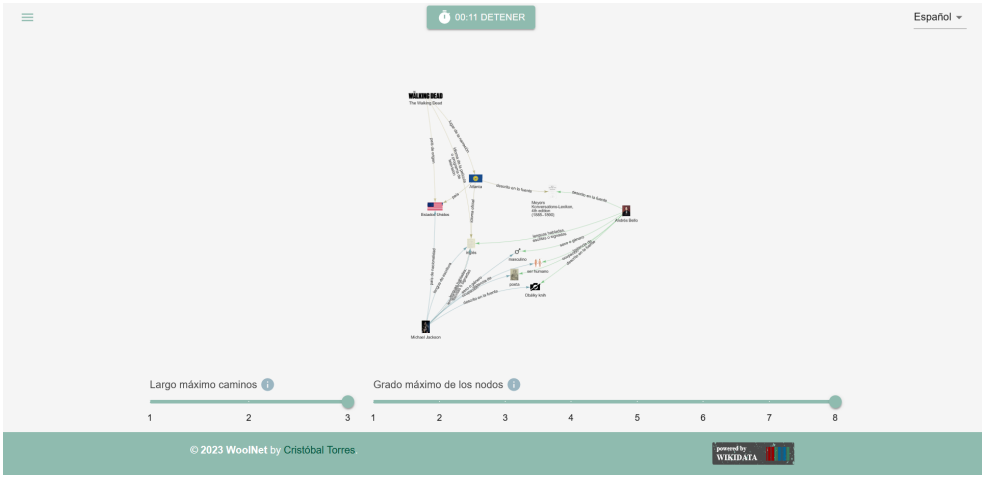


Figura 3.15: Grafo desplegado.

3.2.7. Detención de la Búsqueda de Caminos

Cuando el usuario desee detener la búsqueda debe apretar el botón *DETENER*. Este botón incluye un cronómetro para medir el tiempo que lleva la búsqueda y no se puede apretar si no hay una búsqueda en proceso.

El botón cierra el WebSocket y detiene la búsqueda del usuario. Para que el servidor pueda saber que se cerró, cada segundo envía un mensaje para revisar la conexión. Al reconocer que se cerró el WebSocket se detiene la búsqueda en el servidor.

Si la búsqueda de caminos en el servidor termina, cierra la conexión. Esto detiene la búsqueda de caminos para el usuario.

3.2.8. Mejora en visualización del Grafo

La cantidad de aristas de un camino corresponde al largo de este. Se añade un slider que permite mostrar caminos que poseen un largo menor o igual al que decide el usuario.

En la Figura 3.16 se usa el slider para filtrar caminos de largo mayor a dos.

Cada nodo tiene un grado que corresponde a la cantidad de aristas que posee según los datos del servidor. Los usuarios poseen un slider con valores enteros que permite desplegar solo los caminos que poseen nodos tal que $\log_{10}(gradoDelNodo)$ sea menor o igual al valor elegido por el usuario.

En la Figura 3.17 se posiciona el slider en el valor de 5; esto filtra caminos que poseen nodos con grado mayor a 10^5.

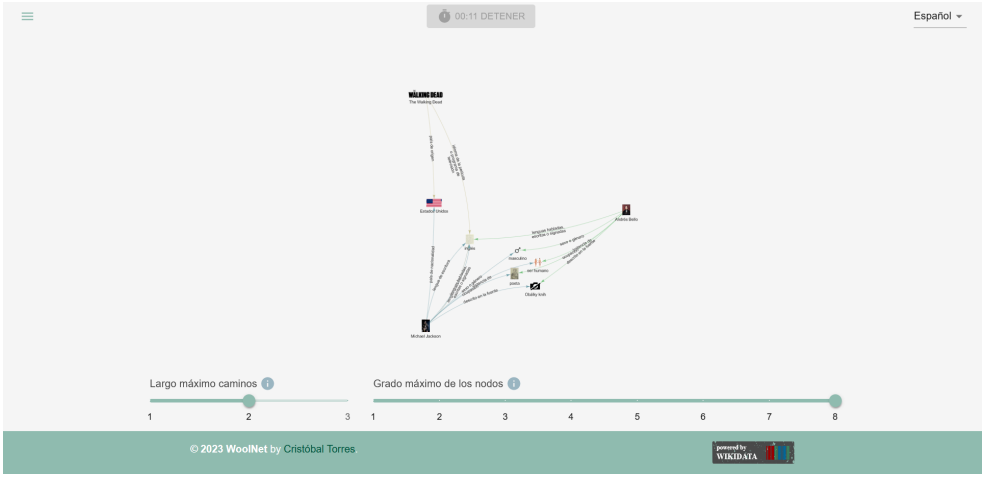


Figura 3.16: Uso del slider para filtrar caminos según su largo.

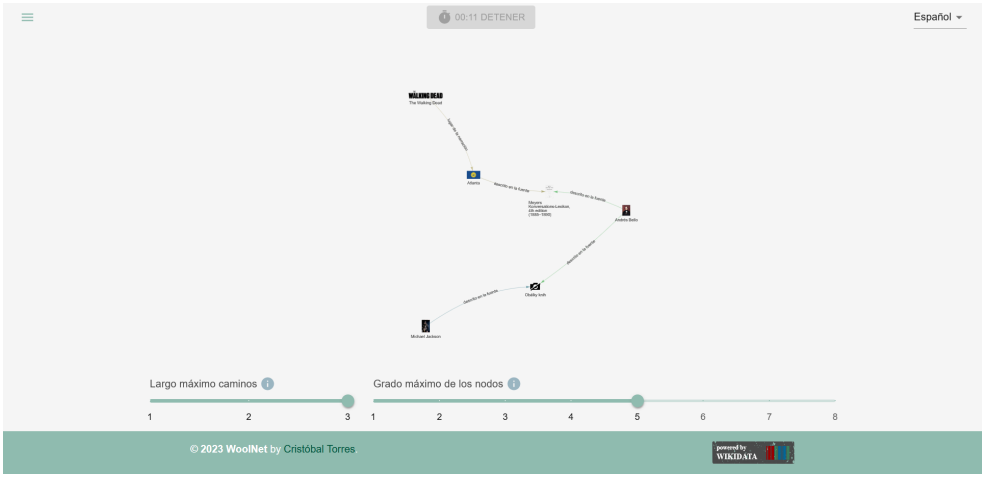


Figura 3.17: Uso del slider para filtrar caminos según el grado de sus nodos.

3.2.9. Servidor y Web

Esta aplicación se monta en el servidor del Instituto Milenio Fundamentos de los Datos. La dirección y puertos que usa la aplicación están expuestos, de modo que el acceso a esta se puede hacer desde cualquier navegador web mediante la url de la aplicación web en https://woolnet.dcc.uchile.cl. Las características de la máquina que aloja la aplicación son:

- 59.6 GB de RAM
- Arquitectura: x86_64
- CPU(s): 12
- Sistema operativo: Devuan GNU/Linux 3 (beowulf)
- Procesador: Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz
- Versión de Java: openjdk version "11.0.18"

Capítulo 4

Evaluación

En esta sección se abordarán los experimentos y la evaluación tanto del algoritmo usado en la búsqueda de caminos como de la aplicación para los usuarios.

En el contexto del algoritmo, se evaluará midiendo tiempos de búsqueda, caminos que se obtienen y memoria usada, comparando las estructuras y usando distintos conjuntos que varían en cantidad de entidades entre las que se buscan caminos.

Lo primero a abordar es la estructura de almacenamiento del grafo. Para esto se responde a la pregunta: **¿cuál de las estructuras carga más rápido los datos?** y **¿qué estructura usa menos memoria para almacenar los datos?**. Además se evalúa según las consultas que se hacen a la estructura, respondiendo a **¿qué estructura entrega los vecinos de un nodo en el menor tiempo?**.

Después se compara el algoritmo de búsqueda de caminos y su uso de memoria para las distintas estructuras; respondiendo a las siguientes preguntas: **¿qué estructura obtiene más aristas pertenecientes a caminos en una búsqueda?** y **¿qué estructura usa menos memoria en una búsqueda de caminos?**.

También se realiza la optimización que consta de no continuar la búsqueda de caminos en vecinos de nodos que poseen grado mayor o igual a un valor. Esto permite resolver **¿qué grado máximo de los nodos en los caminos obtiene búsquedas más eficientes?**, esto considera la cantidad de aristas pertenecientes a caminos y el uso de memoria de las búsquedas. Se realizan experimentos variando ese valor, pero usando la estructura que demuestre ser la más eficiente en los otros experimentos.

Finalmente con los resultados se decide la estructura y sus parámetros. Esta se incluye en la aplicación web que será evaluada. Se recurre a usuarios para que prueben la aplicación y respondan una encuesta. Esta encuesta busca resolver las siguientes preguntas acerca de la aplicación: **¿es usable?** y **¿es útil?**. Además, los usuarios podrán entregar retroalimentación acerca de la aplicación.

4.1. Ambiente de Prueba

Los experimentos de a continuación se realizan en una máquina virtual compartida por el Instituto Milenio Fundamentos de los Datos (IMFD); esta máquina también será el servidor de la aplicación web. Estas son sus especificaciones:

- 59.6 GB de RAM
- Arquitectura: x86_64
- CPU(s): 12
- Sistema operativo: Devuan GNU/Linux 3 (beowulf)
- Procesador: Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz
- Versión de Java: openjdk versión "11.0.18"

4.2. Preprocesamiento

A continuación se describe el preprocesamiento de los datos. Esto incluye la creación de subconjuntos de los datos de Wikidata y la creación de cachés para la carga de estos a las estructuras. El preprocesamiento se programa usando Python.

4.2.1. Subconjuntos

Primero se requiere desarrollar subconjuntos del archivo que posee todos los datos (*latest-truthy.nt.gz*). Estos subconjuntos solo poseen entidades con IDs de Wikidata menores o iguales al descrito en el subconjunto, por ejemplo, los archivos que poseen en su nombre *subset100000* solo incluyen entidades con ID de Wikidata menor o igual a 100000.

Tabla 4.1: Cantidad de aristas, cantidad de nodos, ID máximo y uso de memoria para cada subconjunto.

Subconjunto	**Cantidad de aristas**	**Cantidad de nodos**	**ID máximo**	**Uso de memoria [megabytes]**
latest-truthy_small	715.906.922	99.609.308	117.288.116	4.188,634
subset100000000	652.319.619	87.110.322	99.999.996	3.839,527
subset10000000	42.682.387	8.159.611	10.000.000	260,608
subset1000000	5.977.585	829794	1.000.000	36,372
subset100000	600.493	92.654	100.000	3,051

La Tabla 4.1 describe información de los subconjuntos; se incluye además el subconjunto *latest-truthy_small*, que es la versión de *latest-truthy* que solo incluye entidades (es decir, el conjunto completo de los datos). La información descrita posee la cantidad de aristas, cantidad de nodos, el ID máximo de los nodos y el uso de memoria de cada subconjunto. Estos subconjuntos son archivos comprimidos en formato `gzip`. Esta información permite

diferenciar las escalas de los datos y se usan para la creación de las estructuras según se requieran.

4.2.2. Caché de los datos

Algunas estructuras de almacenamiento del grafo requieren de un caché para construirse. Se presenta a continuación información del caché para cada subconjunto. Estos se almacenan en archivos comprimidos en formato `gzip`.

Tabla 4.2: Tiempo de creación y uso de memoria del caché Triple para cada subconjunto.

Subconjunto	**Tiempo de creación [s]**	**Uso de memoria [megabytes]**
latest-truthy_small	11.122,45	5.013,562
subset100000000	9.940,23	4.606,408
subset10000000	561,07	280,247
subset1000000	65,77	37,963
subset100000	6,12	3,439

En la Tabla 4.2 se muestran los tiempos de creación del caché que permite la carga de datos a estructuras Triples Densos y Triples no Densos. También se incluye su uso de memoria en el disco.

Tabla 4.3: Tiempo de creación y uso de memoria del caché Adyacencia para cada subconjunto.

Subconjunto	**Tiempo de creación [s]**	**Uso de memoria [megabytes]**
latest-truthy_small	16.425,99	5.064,993
subset100000000	14.730,41	4.692,552
subset10000000	946,91	283,313
subset1000000	120,21	38,734
subset100000	11,26	3,016

En la Tabla 4.3 se muestran los tiempos de creación del caché para las estructuras Adyacencia Densa y Adyacencia no Densa. También se incluye su uso de memoria en el disco.

Los resultados de tiempo para cada creación de caché varían, siendo menores para el caché Triple que para el caché Adyacencia. Este tiempo se debe principalmente a que el caché Adyacencia lleva a cabo más trabajo en su creación que el caché Triple.

Un detalle importante es que las estructuras Triples Densos y Triples no Densos usan dos archivos para su construcción: el archivo original que posee información de las aristas y el caché que tiene la información de los nodos y las aristas que le corresponde a cada uno.

4.3. Estructura de Almacenamiento

Se requiere una estructura que logre almacenar completamente el grafo en memoria principal y que esta carga sea en el menor tiempo posible. Para esto se mide el uso de memoria y tiempo de carga para las distintas estructuras planteadas en el Capítulo 3. Además para estas estructuras se obtienen tiempos de búsqueda de vecinos de los nodos.

4.3.1. Almacenamiento de datos en las Estructuras

Se compara uso de memoria y tiempos de carga de las estructuras. Para esto se cargan en cada una de ellas los subconjuntos. Cada estructura usa los datos en el formato que le corresponde.

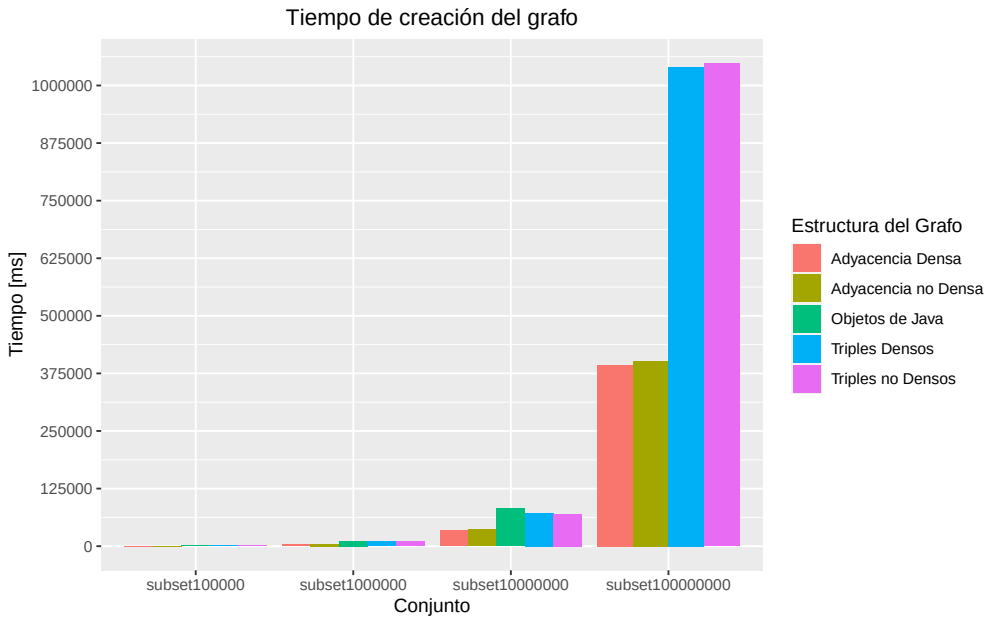


Figura 4.1: Tiempo de creación del grafo para cada estructura.

En la Figura 4.1 se muestran los tiempos de creación y carga de los datos para cada estructura usando cada subconjunto. Este tiempo se mide en milisegundos. En el caso de la estructura usando objetos de Java, no se logra la carga de los datos usando el subconjunto *subset100000000*. Se ve que las estructuras que demoran menos en cargar los datos son Adyacencia Densa y Adyacencia no Densa.

En la Figura 4.2 se muestra el uso de memoria en bytes para cada estructura usando cada subconjunto. En el caso de la estructura que usa objetos de Java, no se logra la carga de los datos usando el subconjunto *subset100000000*. Se ve que la estructura que usa la menor cantidad de memoria es Adyacencia no Densa.

Los resultados anteriores plantean que la estructura Adyacencia no Densa posee menor uso de memoria permitiendo almacenar grafos más grandes. Además, junto a la estructura Adyacencia Densa, poseen los menores tiempos de creación del grafo.

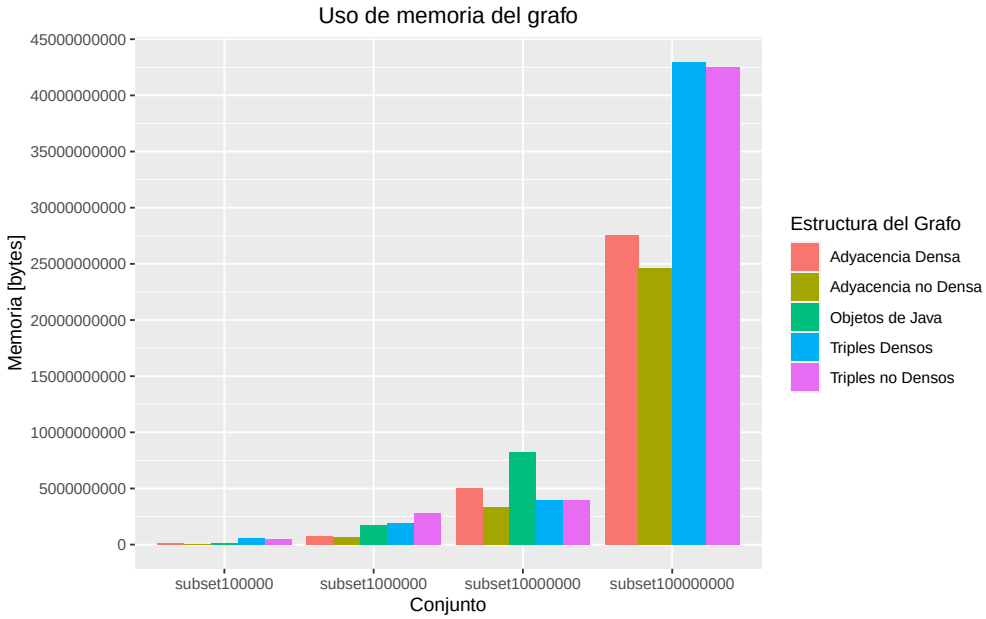


Figura 4.2: Memoria usada por el grafo para cada estructura.

4.3.2. Búsqueda de Vecinos en las Estructuras

Primero se obtienen 10000 IDs de nodos aleatorios para generar búsquedas, usando como probabilidad de obtención su cantidad de aristas para no sesgar la muestra hacia los muchos nodos de grado menor (en general, se asume que el usuario va a elegir nodos de alto grado con más frecuencia siendo nodos más notables). Estos 10000 IDs varían según el subconjunto pero no según la estructura de datos usada.

A continuación se realiza la búsqueda de vecinos para cada uno de los 10000 valores aleatorios en el subconjunto que le corresponde y se mide el su tiempo de obtención.

En la Figura 4.3 se muestran los resultados de las búsquedas de vecinos para 10000 datos usando un gráfico de cajas con un eje en escala logarítmica. En particular, se presenta la distribución de los tiempos (con eje y en escala logarítmica) para cada una de las 10000 búsquedas. La estructura que concentra los menores tiempos es Adyacencia no Densa.

4.4. Búsqueda de Caminos

Se evalúa el rendimiento del algoritmo de búsqueda de caminos para cada estructura de almacenamiento; llevando a cabo búsquedas de caminos de largo máximo 3 y, se compara la cantidad de aristas pertenecientes a los caminos que se obtienen y el uso de memoria de la búsqueda.

Luego, se evalúa el rendimiento del algoritmo de búsqueda de caminos en la estructura que obtuvo mejores resultados. Se comparan distintos límites de búsqueda, de modo que, si el nodo actual posee un grado mayor al límite no se continúa buscando caminos con sus vecinos.

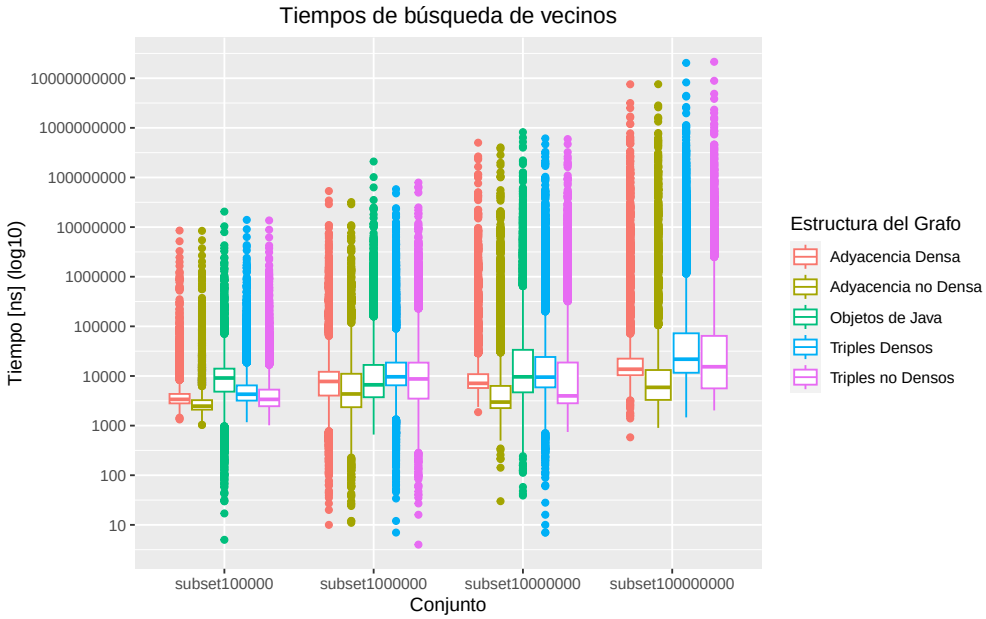


Figura 4.3: Tiempo en obtener vecinos para 10000 nodos.

4.4.1. Conjunto de datos

Para evaluar el algoritmo de búsqueda de caminos, se plantea como experimento contar la cantidad de aristas de caminos que se encuentran al ejecutarlo durante 60 segundos. Se supone este tiempo de uso aproximando por usuario.

Lo anterior se realiza para 100 grupos de 2, 3, 4 y 5 entidades. Estos grupos se generan a partir de los nodos aleatorios usados en la evaluación anterior. Esto permite encontrar una tendencia en la cantidad de aristas y uso de memoria a partir de la búsqueda. La evaluación se lleva a cabo sobre los datos de *latest-truthy_small* para las estructuras planteadas sin incluir la que usa objetos de Java ya que no soporta todos los datos.

4.4.2. Búsqueda de Caminos en Estructuras

En la Figura 4.4 se muestran los resultados de la búsqueda de caminos durante 60 segundos; se usa una escala logarítmica de base 10 para contar aristas encontradas pertenecientes a caminos entre las entidades de cada grupo.

Según los resultados, las estructuras Adyacencia Densa y Adyacencia no Densa obtienen mayor cantidad de caminos y aristas en la misma cantidad de tiempo. Esto ocurre ya que los algoritmos de obtención de vecinos son más rápidos en estas estructuras y se usa mucho en la obtención de caminos.

A continuación en la Figura 4.5 se compara el uso de memoria en las búsquedas de caminos durante 60 segundos. El uso principal corresponde al subgrafo que se construye y las variables auxiliares para guardar los caminos.

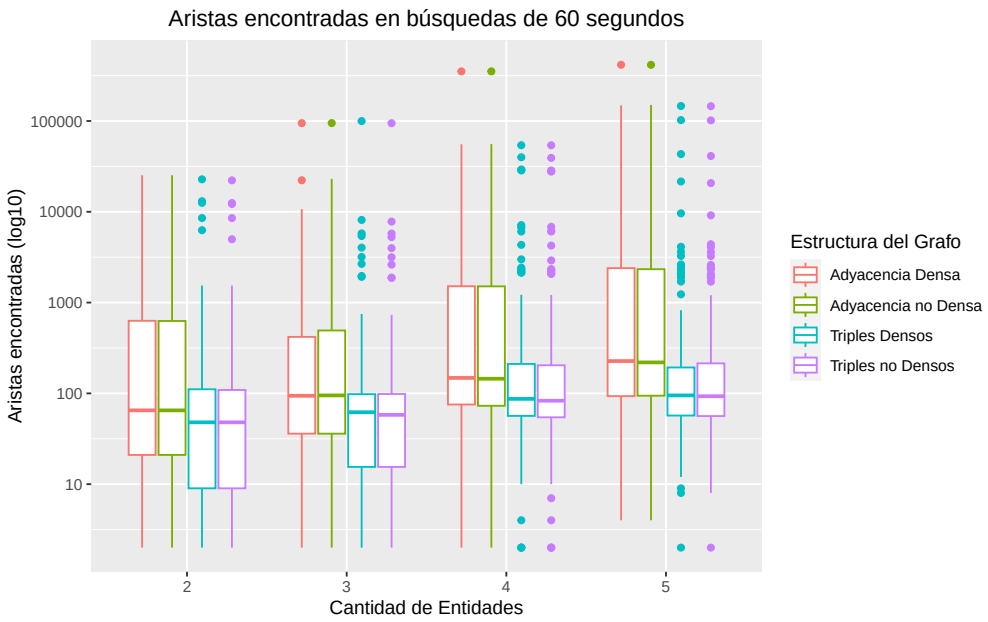


Figura 4.4: Aristas encontradas en escala logarítmica en búsquedas de caminos durante 60 segundos para cada grupo de entidades.

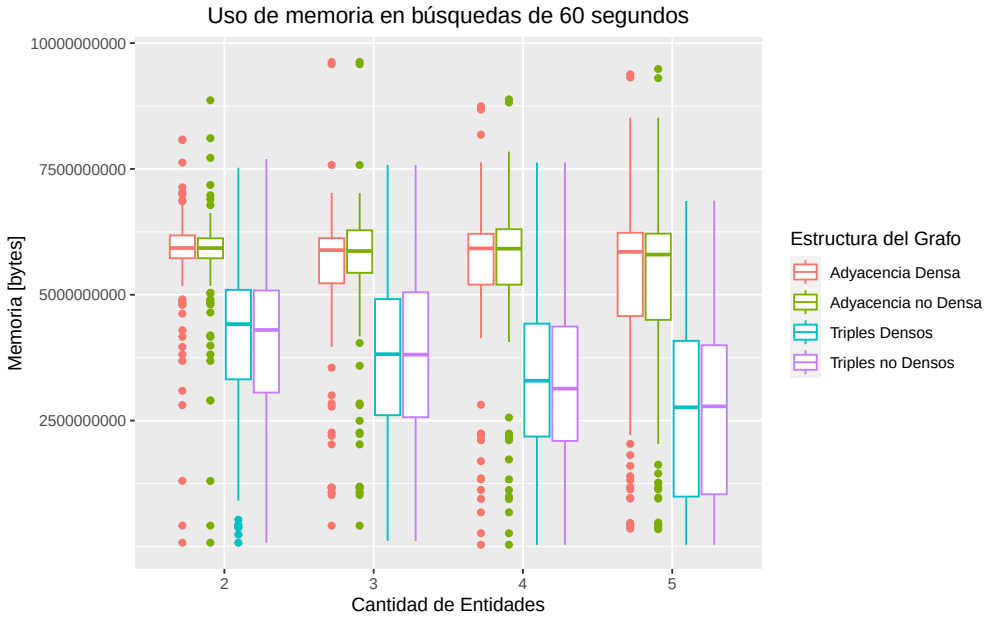


Figura 4.5: Memoria usada en búsquedas de caminos durante 60 segundos para cada grupo de entidades.

En los resultados las estructuras Adyacencia Densa y Adyacencia no Densa usan más memoria que las estructuras Triples Densos y Triples no Densos. Esto corresponde con la velocidad de obtención de vecinos ya que al ser más rápida, recorre una mayor parte del grafo en el mismo tiempo y con ello logra generar más copias de nodos que las estructuras Triples.

A continuación la Figura 4.6 muestra un gráfico en escala logarítmica de la división *Memoria/AristasEncontradas* en las búsquedas de caminos. Esto permite ver un valor aproximado del peso en bytes de una arista.

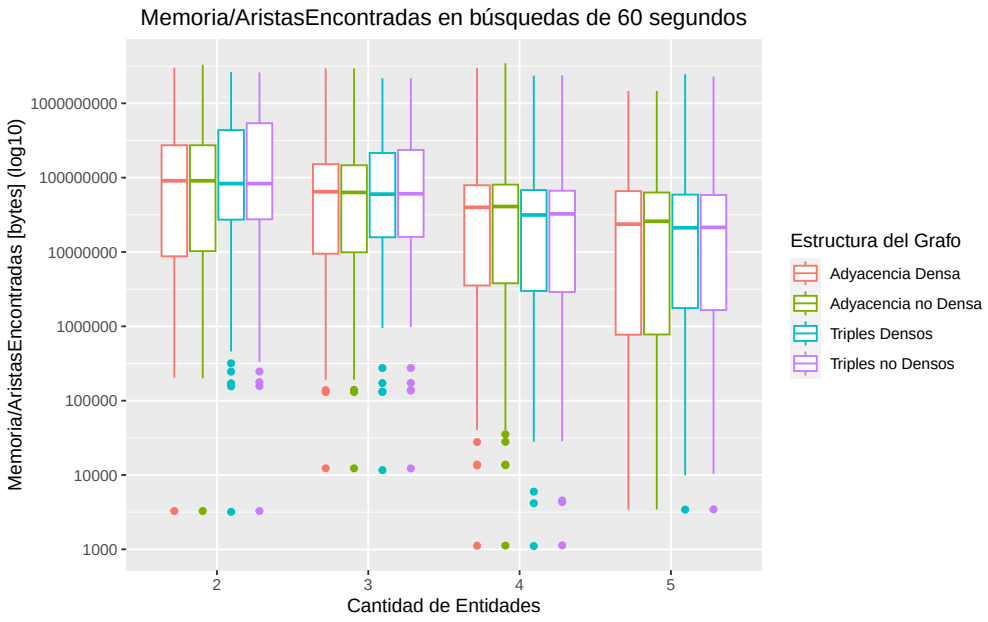


Figura 4.6: División *Memoria/AristasEncontradas* en búsquedas de caminos durante 60 segundos para cada grupo de entidades.

Las tendencias son similares, teniendo medianas muy cercanas. Las variaciones ocurren ya que en la búsqueda también se crean nodos que no necesariamente pertenecen a un camino. Las tendencias más bajas las presentan las estructuras Adyacencia Densa y Adyacencia no Densa.

4.4.3. Búsqueda de Caminos con Límite de Grado

Se agrega la optimización de grado al algoritmo de búsqueda de caminos. Esta consiste en limitar la búsqueda de modo que si el nodo actual posee más de una cantidad de vecinos, no se continúa buscando caminos con sus vecinos. Este límite se considera con nodos intermedios en los caminos y no iniciales.

El experimento a continuación cuenta la cantidad de aristas pertenecientes a caminos que se obtienen durante 60 segundos variando la cantidad de vecinos usada como límite. En este caso el experimento se lleva a cabo solamente en la estructura Adyacencia no Densa y se varía la cantidad máxima de aristas que puede tener un nodo para que se continúe visitando sus vecinos. En la Figura 4.7 se muestra la cantidad de aristas obtenidas en el experimento.

Los resultados muestran que la concentración de aristas pertenecientes a caminos que se obtienen en general crece hasta el límite de 100000 y luego disminuye, obteniendo sus menores valores cuando no posee límite.

La Figura 4.8 muestra el uso de memoria en el experimento de buscar caminos durante 60 segundos variando el límite de aristas.

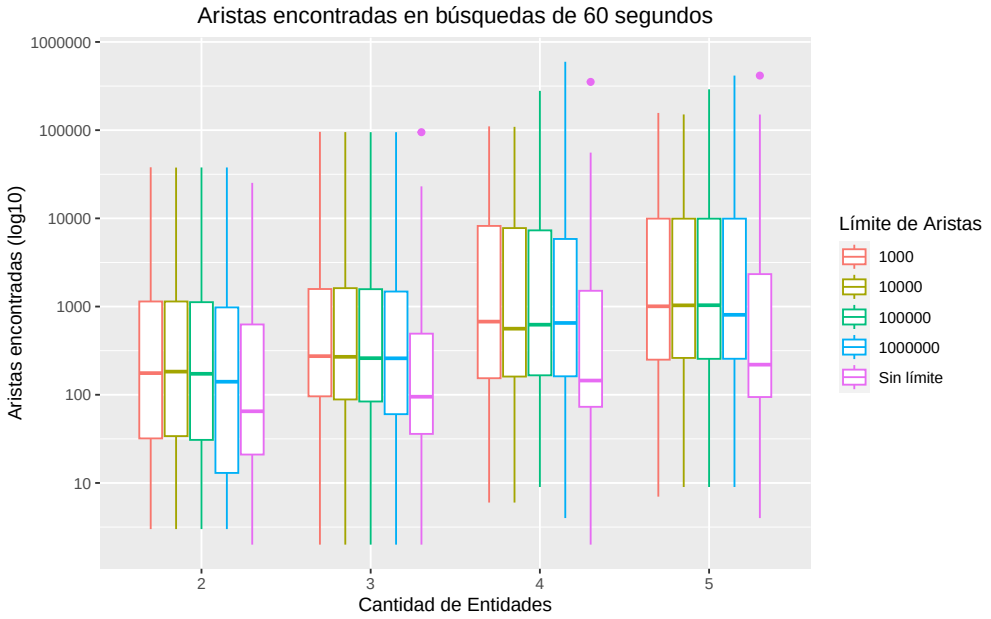


Figura 4.7: Aristas encontradas en escala logarítmica en búsquedas de caminos durante 60 segundos para cada grupo de entidades.

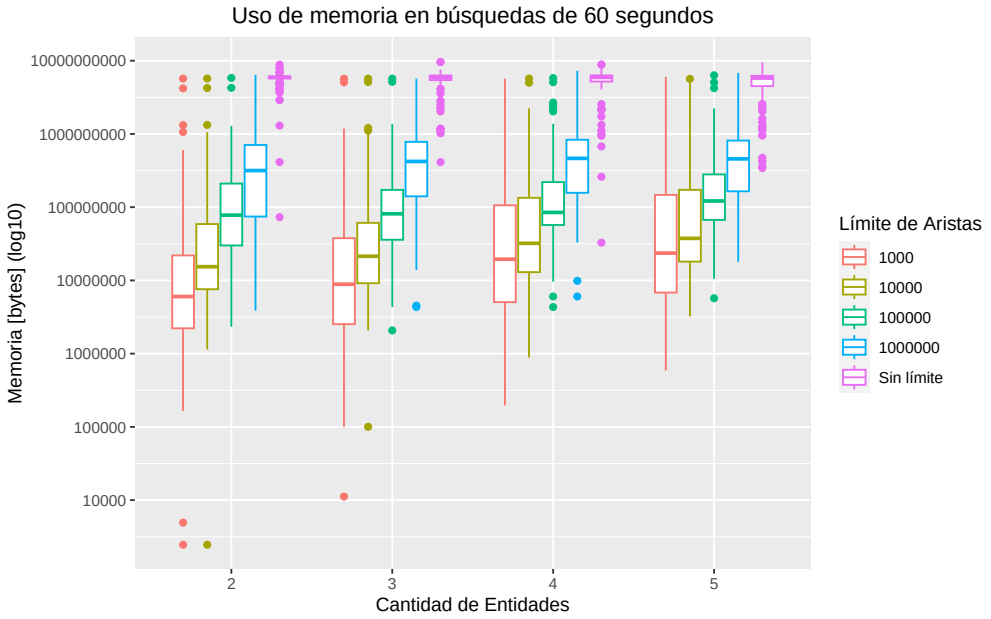


Figura 4.8: Memoria usada en búsquedas de caminos durante 60 segundos para cada grupo de entidades.

Se logra ver que el uso de memoria crece según la cantidad de entidades y también según los límites, alcanzando la menor concentración en el límite de 1000 y la mayor al no usar límite.

La Figura 4.9 muestra la división $Memoria/AristasEncontradas$ en el experimento de buscar caminos durante 60 segundos variando el límite de aristas.

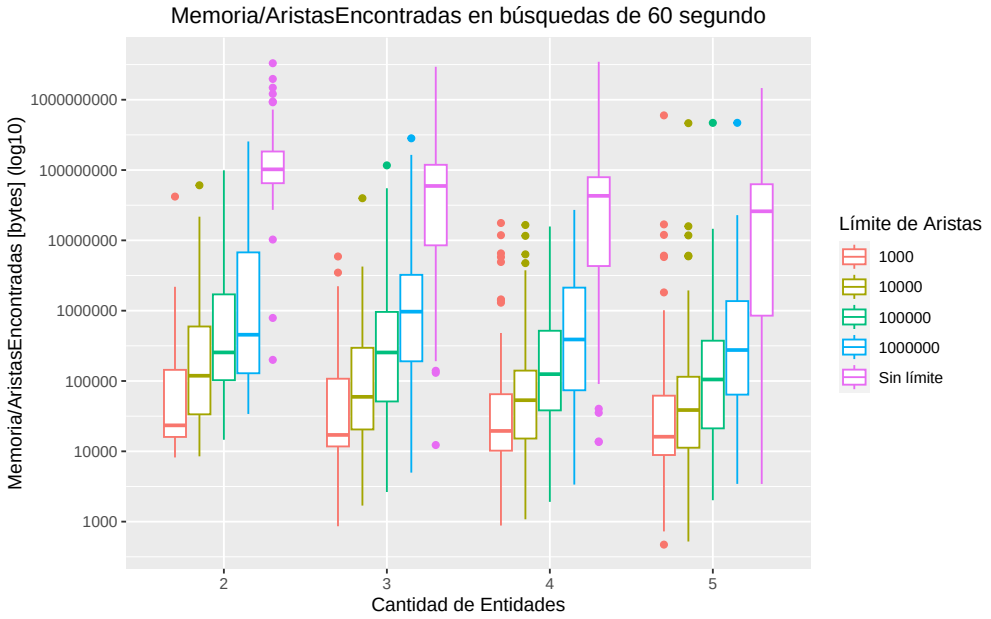


Figura 4.9: División *Memoria/AristasEncontradas* en búsquedas de caminos durante 60 segundos para cada grupo de entidades.

Las tendencias que se obtienen son muy distintas para cada límite. Los menores resultados se obtienen en límites pequeños y el mayor al no usar límite. Esto puede significar que se crean muchos nodos que no pertenecen a caminos entre los nodos iniciales.

Sin considerar los resultados, existe un *trade-off* entre el límite y la memoria usada. Al disminuir el límite se instancian menos nodos, lo que implica un menor uso de la memoria. Además existe un *trade-off* entre el límite y las aristas pertenecientes a caminos obtenidos. Al considerar disminuir el límite, se instancian menos nodos y con ello hay menos caminos posibles.

En los experimentos se asigna un tiempo máximo de ejecución del algoritmo. El *trade-off* que se mantiene es con respecto a la memoria, ya que con límites menores podría instanciar menos nodos. Sin embargo la cantidad de aristas pertenecientes a caminos puede variar. Esto ya que un límite pequeño podría no incluir nodos pertenecientes a caminos, lo que implica obtener menos aristas. Un límite mayor podría incluir nodos que no pertenecen a caminos, esto implica seguir revisando sus nodos vecinos sin obtener aristas nuevas, en lugar de continuar con otro nodo que sí pueda pertenecer a algún camino.

4.5. Decisión Final

El estudio anterior permite una decisión fundamentada de qué estructura se debe usar. Además los experimentos incluyendo límites de aristas orientan la decisión a una respuesta más apta para su inclusión en el servidor de una aplicación web.

Se decide usar la estructura **Adyacencia no Densa**. Los experimentos demuestran que es la estructura de menor uso de memoria para almacenar el grafo con los datos de Wikidata.

Es también la estructura que obtiene más rápidamente los vecinos de los nodos.

En cuanto a la búsqueda de caminos, los gráficos muestran que hay correlación entre el uso de memoria y la cantidad de aristas pertenecientes a caminos que se obtienen. Las estructura Adyacencia Densa y Adyacencia no Densa obtuvieron la mayor cantidad de aristas. Entre ambas se decide usar la estructura **Adyacencia no Densa** ya que obtiene más rápidamente los vecinos de los nodos y esto afecta levemente a la velocidad del algoritmo de búsqueda de caminos, lo que hace a esta estructura levemente superior a la otra.

En cuanto a la optimización de la búsqueda de caminos, se decide usar un límite de **10000** aristas. Este límite permite obtener una gran cantidad de resultados, mostrando en general su concentración en los mayores resultados, en comparación a los demás límites. Además el uso de memoria es de valor intermedio, esto permite resolver más consultas en paralelo sin agotar la memoria disponible.

4.6. Usabilidad

Para evaluar la usabilidad de la aplicación se recurre a usuarios que entregan su retroalimentación y apreciaciones a partir de un formulario. El formulario posee tres secciones.

1. Caracterización:

   Se incluye una pequeña caracterización que pregunta a la persona encuestada si conoce Wikidata.

2. Navegación Guiada:

   A través de instrucciones se guía el uso de la aplicación. El objetivo es llevar a cabo una consulta predeterminada y, sí el usuario encuentra un camino interesante o inesperado entre las entidades, puede comentar acerca de los resultados.

3. Navegación no Guiada, Encuesta de Usabilidad y Comentarios:

   Finalmente se le solicita al usuario que navegue dentro de WoolNet sin ser guiado. A partir del uso debe responder una serie de preguntas entre las que están: ¿entre cuántas entidades busca caminos? y otras enfocadas en la usabilidad mediante la percepción del usuario. Finalmente el usuario puede entregar comentarios generales acerca del sistema.

4.6.1. Grupo encuestado

Esta encuesta es voluntaria y totalmente anónima. Ya que la aplicación busca el uso de todo tipo de usuario, se publica en dos foros de la Universidad de Chile. El primero es el foro de Estudiantes del DCC (Departamento de Ciencias de la Computación) de la Universidad de Chile, en ella el público incluye estudiantes de la carrera de Ingeniería Civil en Computación y académicos de esta misma área. El otro foro es Facultad de Cs. Físicas y Matemáticas de la Universidad De Chile; en este se encuentran personas que estudian en esta facultad y académicos de la misma. A partir de ambos foros la encuesta recibió un total de 80 respuestas.

4.6.2. Caracterización

La caracterización busca clasificar a los usuarios que probaron la aplicación mediante su conocimiento acerca de Wikidata. La pregunta que se debe responder es: **¿Usted conoce Wikidata?**. Las respuestas que se permiten son: **Sí**, **Un poco** y **No**.

En la Figura 4.10 se muestra un gráfico que agrupa a las personas encuestadas según su conocimiento acerca de Wikidata.

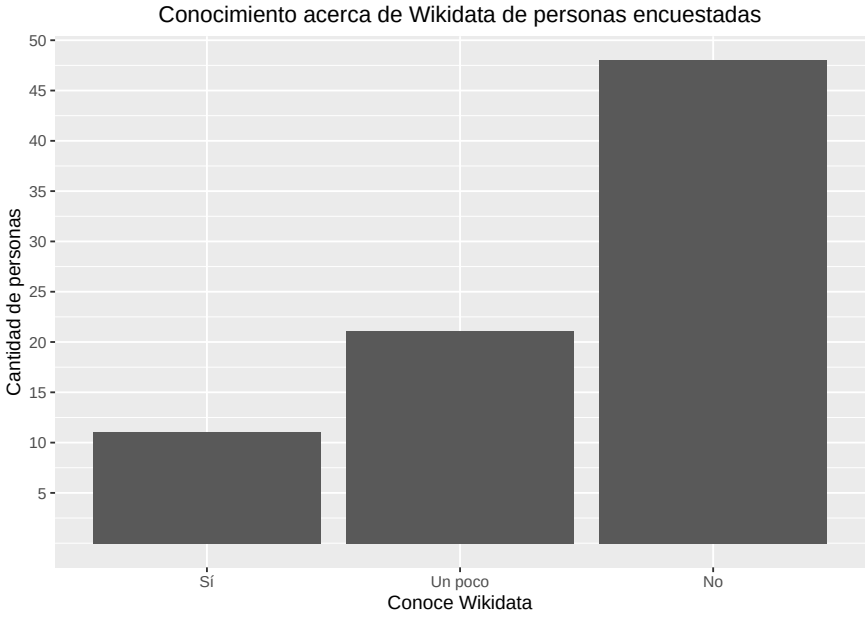


Figura 4.10: Conocimiento acerca de Wikidata de las personas encuestadas.

Se logra ver que en su mayoría son personas que no conocen Wikidata. Esto podría significar que esta aplicación es su primer acercamiento a esta fuente de información.

4.6.3. Navegación Guiada

Los usuarios primero acceden a una navegación guiada. Esta corresponde a una búsqueda de caminos entre entidades ya definidas. Las entidades son **Adolf Hitler**, **Mahatma Gandhi** y **Benito Mussolini**. Se deciden estas entidades ya que se presentan como personalidades opuestas. Además la búsqueda de caminos obtiene muchos resultados, lo que motiva el uso del slider. La búsqueda posee resultados que pueden ser interesantes y motivar el comentario del usuario, asegurando el uso correcto de la aplicación.

La primera pregunta que deben responder es: **¿Encontró usted al menos una conexión interesante o inesperada que relaciona Hitler, Gandhi y/o Mussolini?**. Las respuestas posibles son: **Sí**, **No sé** y **No**. En la Figura 4.11 se muestra un gráfico que agrupa a las personas encuestadas según encontraron relaciones interesantes en la búsqueda guiada.

Se logra ver que una mayor cantidad de personas encuestadas encuentran una conexión interesante entre las entidades. Las personas encuestadas podían comentar voluntariamente acerca de estas conexiones, las respuestas se concentran en las siguientes conexiones:

Figura 4.11: Encuentran algo interesante en la búsqueda las personas encuestadas.

- Nominación al Premio Nobel de la Paz (Hitler, Mussolini y Gandhi).
- Vegetarianismo (Hitler y Gandhi).
- Persona del año (Hitler y Gandhi).
- Muerte por arma de fuego (Hitler, Mussolini y Gandhi).

La búsqueda plantea personalidades opuestas y con ello una relación entre ellas puede ser más interesante. De acuerdo a los resultados, el aspecto más interesante fue que Hitler, Mussolini y Gandhi fueron nominados al Premio Nobel de la Paz. Además, dentro de las relaciones más destacadas, hay algunas que relacionan dos de las tres entidades, y otras que relacionan las tres entidades.

4.6.4. Navegación no Guiada y Comentarios

Se plantea al usuario que realice una búsqueda a su elección, esta vez sin guiarlo; esto permite que busque entidades de su interés. El usuario primero debe responder entre cuantas entidades realiza esta búsqueda.

En la Figura 4.12 se muestra la frecuencia de entre cuántas entidades realizaron una búsqueda de caminos no guiada las personas encuestadas.

Se logra ver una mayor cantidad de búsquedas entre 3 entidades, sin embargo la cantidad varía entre 1 hasta 8 entidades.

El usuario luego debe responder un cuestionario de 10 enunciados de selección múltiple que corresponden con la Escala de Usabilidad del Sistema (*System Usability Scale*), con las

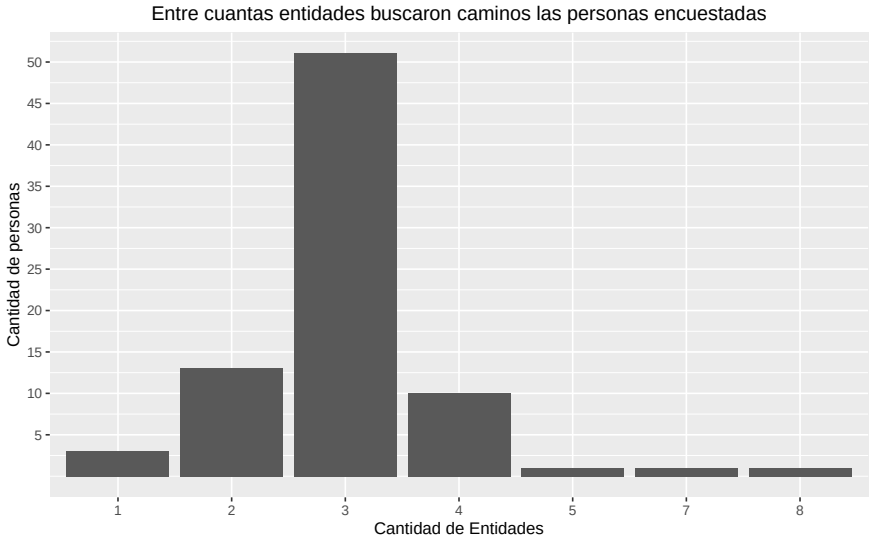


Figura 4.12: Entre cuantas entidades buscaron caminos las personas encuestadas.

siglas *SUS* [13]. Esta es una herramienta confiable y rápida para medir la usabilidad de un sistema mediante la percepción de eficacia (pueden los usuarios lograr con éxito sus objetivos), eficiencia (cuánto esfuerzo y recursos se gastan para lograr esos objetivos) y satisfacción (fue satisfactoria la experiencia). El usuario posee 5 alternativas para cada enunciado que van desde **en desacuerdo** (1) a **de acuerdo** (5). A continuación se muestran los 10 enunciados que se deben responder:

1. Creo que usaría este sistema con frecuencia.
2. Encontré el sistema innecesariamente complejo.
3. El sistema me parece fácil de usar.
4. Creo que necesitaría el apoyo de un técnico para poder utilizar este sistema.
5. Considero que las diversas funciones de este sistema estaban bien integradas.
6. Pensé que había demasiada inconsistencia en este sistema.
7. Me imagino que la mayoría de la gente aprendería a usar este sistema muy rápidamente.
8. Encontré el sistema muy engorroso de usar.
9. Me sentí muy confiado usando el sistema.
10. Necesitaba aprender muchas cosas antes de poder empezar con este sistema.

En la Tabla 4.4 se muestra la frecuencia de los resultados para cada uno de los 10 enunciados.

Para interpretar los resultados se requiere su normalización y conversión a puntaje *SUS*. Las reglas para esta conversión son las siguientes:

Tabla 4.4: Frecuencia de las respuestas para cada enunciado.

Pregunta	R1	R2	R3	R4	R5
1	7	25	20	16	12
2	40	19	16	4	1
3	0	3	5	35	37
4	59	13	7	1	0
5	1	2	11	28	38
6	53	22	2	2	1
7	2	5	9	22	42
8	43	27	4	5	1
9	0	3	13	23	41
10	65	11	2	2	0

- En las respuestas; **en desacuerdo** posee el valor 1 y crece con cada opción hasta **de acuerdo** que posee el valor de 5.
- Los enunciados impares (1, 3, 5, 7, 9) corresponden a apreciaciones positivas. Al valor de la selección se le debe restar 1.
- Los enunciados pares (2, 4, 6, 8, 10) corresponden a apreciaciones negativas. A 5 se le debe restar el valor de la selección.
- Se suman estos valores y se multiplican por 2.5, el resultado corresponde al puntaje SUS obtenido por esa encuesta.

A partir de lo anterior, se calculan los puntajes que resultan en el valor promedio de **81,19**. En la Figura 4.13 se muestra la distribución de los puntajes de todas las encuestas.

Sauro [9] mediante el estudio de más de 500 sistemas obtiene un puntaje SUS promedio de **68**. Un resultado menor a este valor indica que hay varios aspectos a corregir. Dado que el sistema obtuvo un puntaje promedio que supera ese valor, se considera usable.

Finalmente las personas encuestadas podían agregar un comentario acerca de la aplicación. Se obtuvieron **58** comentarios que incluyen apreciaciones positivas acerca de la aplicación y mejoras en cuanto a usabilidad. A continuación se resumen los comentarios:

- *Luego de un tiempo aparecen muchas relaciones entre entidades, esto genera superposición de aristas lo que dificulta la lectura, planteando la posibilidad de filtrar más caminos.*
- *A veces sucede que la aplicación sigue buscando caminos y no encuentra nuevos.*
- *Usar una paleta de colores más llamativa, mejorar la compatibilidad en navegadores de celulares, redistribución de la posición de los botones y usar otras opciones en los iconos.*
- *Es una aplicación interesante, divertida, entretenida y de fácil uso.*

[9] Measuring Usability with the System Usability Scale (SUS)

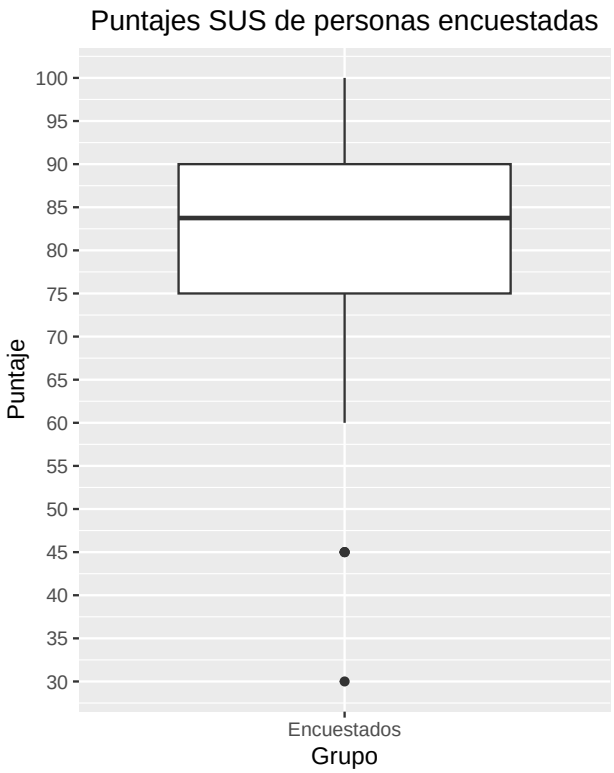


Figura 4.13: Puntajes SUS obtenidos en las encuestas.

- *La aplicación obtiene rápidamente caminos entre entidades y puede ser muy útil para llevar a cabo búsquedas de información sobre conceptos.*

En base a lo anterior, se puede entender que WoolNet es una aplicación útil y usable, sin embargo posee aspectos visuales que pueden mejorarse. Una idea que se planteó es la de seguir explorando formas de filtrar caminos para evitar la sobrecarga visual. Hasta ahora se tienen los límites de aristas para los nodos por parte del algoritmo de búsqueda de caminos y los sliders inferiores en aplicación.

Capítulo 5

Conclusiones

En esta sección se presentan las conclusiones de la memoria. Inicia con un resumen del trabajo desarrollado. A continuación se hará una recapitulación de los objetivos, describiendo por qué fueron logrados. Luego se describe la relevancia e impacto de este trabajo. Se continúa con lecciones aprendidas a lo largo del desarrollo de la memoria y se finaliza describiendo trabajos futuros.

5.1. Resumen

En esta memoria se trabajó en tres áreas: estructuras de almacenamiento de grafos, algoritmos en grafos y el desarrollo de aplicaciones web. Estas tres se unen en el desarrollo de un sistema visual para explorar subgrafos temáticos en Wikidata.

Se desarrollaron cinco estructuras de almacenamiento de datos en un grafo; objetos de Java, Triples Densos, Triples no Densos, Adyacencia Densa y Adyacencia no Densa. Estas estructuras fueron evaluadas de acuerdo con el uso de memoria, rapidez en la obtención de vecinos de los nodos y obtención de caminos de largo máximo 3 entre nodos. La estructura que obtuvo los mejores resultados, en términos de caminos obtenidos en menor tiempo y uso de memoria, fue Adyacencia no Densa.

Se desarrolló un algoritmo basado en BFS para la obtención de los caminos entre nodos. El algoritmo fue evaluado midiendo la cantidad de aristas pertenecientes a caminos que se obtienen en la búsqueda. Esto permite asegurar la confiabilidad en la rápida obtención de caminos entre entidades.

Finalmente se desarrolló una aplicación web para el despliegue de los subgrafos. Esta permite mostrar los caminos entre los nodos que elija el usuario. La usabilidad de la aplicación fue evaluada obteniendo en su mayoría resultados positivos.

5.2. Objetivos

El objetivo general contempla el diseño e implementación una herramienta web que sea un sistema visual para explorar subgrafos temáticos en Wikidata, incluyendo los caminos entre las entidades que desee el usuario, manteniendo un enfoque en la usabilidad y accesibilidad para los usuarios no expertos en el uso de Wikidata y eficiente en la obtención de relaciones.

De acuerdo a la experimentación y encuesta realizada a los usuarios de la aplicación desarrollada, este objetivo fue logrado (desarrollando una aplicación accesible y probada por usuarios). La aplicación está disponible en https://woolnet.dcc.uchile.cl/.

Con respecto a los objetivos específicos, estos se centran en el desarrollo de un algoritmo de obtención de caminos entre nodos en un grafo, con un enfoque en la obtención rápida de caminos. Esto se logró implementando un algoritmo basado en BFS el cual es usado por la aplicación. Este se evaluó midiendo las aristas pertenecientes a caminos que se obtienen en un minuto, notando que se obtiene una gran cantidad en poco tiempo.

Dentro de los objetivos específicos, también se incluye la implementación del sistema visual que despliegue las relaciones y entidades obtenidas en las búsquedas de los usuarios. Esto fue logrado mediante el desarrollo de una aplicación web; esta hace uso del algoritmo de búsqueda planteado y despliega los resultados al usuario según entre qué entidades quiere buscar.

Por último, hubo una parte importante del trabajo que no fue planteada como objetivo. Esto corresponde al estudio y desarrollo de una estructura de almacenamiento del grafo. Esto ya que no se contemplaba como un punto importante al suponer en un inicio que los datos podrían ser soportados por una estructura de grafo basada solamente en objetos de Java. Al requerir de una estructura que pueda almacenar los datos completamente y obtenerlos rápidamente se hizo necesario su estudio y comparación con otras estructuras de almacenamiento de grafos.

5.3. Reflexión acerca de la relevancia/impacto

En este trabajo se logra desarrollar una aplicación web que acerca Wikidata a personas que no la conocen mediante el despliegue de subgrafos. De acuerdo con las respuestas recibidas en la evaluación de la aplicación, se logra este acercamiento. Esto propone una gran relevancia de este trabajo a los usuarios que no conocían Wikidata. La aplicación les entrega una herramienta para llevar a cabo una búsqueda exploratoria acerca de cualquier concepto. Sin embargo, para que el impacto sea superior se requiere alcanzar más gente, lo que trae consigo nuevos desafíos; aumentar el poder de procesamiento del servidor y buscar vías para acercar la aplicación a más personas.

5.4. Lecciones Aprendidas

Este trabajo presenta un fuerte enfoque en el diseño y estudio de estructuras de almacenamiento de grafos, lo que en un principio no se consideró como un objetivo o parte del trabajo.

Esto ya que se suponía que los datos de Wikidata podrían ser soportados por la estructura que usa objetos de Java.

Lo anterior ocurrió por la falta de conocimiento en cuanto a la máquina que se usaría para almacenar los datos y del peso real de los datos que se iban a usar.

La lección principal es el estudio y medición de la incertidumbre en los proyectos computacionales. En el caso descrito anteriormente se planteó esta incertidumbre asignando un mayor tiempo de trabajo al algoritmo, lo que abarcaba su desarrollo y el trabajo de la carga de los datos.

5.5. Trabajo Futuro

El trabajo actual posee un estudio de estructuras para almacenar el grafo de Wikidata. Un trabajo futuro que se plantea es experimentar con otras formas de almacenamiento para grafos, recurriendo a estructuras más complejas que puedan comprimir la información y con esto usar menor cantidad de memoria al almacenar los datos. Ya que el uso de los datos es solo de lectura, una línea de estudio que se propone para llevarlo a cabo es el uso de estructuras compactas.

La arquitectura actual almacena los datos en una sola máquina. Un trabajo futuro consiste en cambiar cómo se almacenan, distribuyendo los datos en varias máquinas, por ejemplo, en el caso de no poder cargar en memoria la estructura completa del grafo. Otra idea consiste en usar un almacenamiento híbrido, que permita almacenar los datos en disco, pero incluir un caché con resultados de consultas ya procesadas para acelerar las consultas que más se puedan repetir.

Otro trabajo futuro puede considerar un mayor enfoque en los aspectos visuales (UI/UX). El trabajo actual aborda esto desarrollando una aplicación agradable para los usuarios, sin embargo, no posee un estudio profundo de esta área. Se plantea como trabajo futuro una evaluación más robusta con usuarios y un trabajo multidisciplinario con profesionales en el área de diseño.

Finalmente, se puede abordar un trabajo futuro enfocado en la arquitectura del servidor. La solución actual usa una sola máquina; esto implica que su memoria principal contiene: la carga de los datos, la aplicación web y los grafos que se forman en las búsquedas de caminos. Este factor no permite muchos usuarios en simultáneo llevando a cabo búsquedas muy largas. Para esto sería interesante investigar la posibilidad de tener más máquinas que almacenen los grafos que se forman en las búsquedas y una que almacene la información del grafo de Wikidata. Esta última puede funcionar como una API.

Bibliografía

[1] Vrandecic, D. y Krötzsch, M., "Wikidata: a free collaborative knowledgebase," Commun. ACM, vol. 57, no. 10, pp. 78–85, 2014, doi:10.1145/2629489.

[2] Cyganiak, R., Wood, D., y Lanthaler, M., "RDF 1.1 Concepts and Abstract Syntax." W3C Recommendation, 2014. https://www.w3.org/TR/rdf11-concepts/.

[3] Prud'hommeaux, E. y Seaborne, A., "SPARQL query language for RDF," W3C recommendation, W3C, 2008.

[4] Malyshev, S., Krötzsch, M., González, L., Gonsior, J., y Bielefeldt, A., "Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph," en The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II (Vrandecic, D., Bontcheva, K., Suárez-Figueroa, M. C., Presutti, V., Celino, I., Sabou, M., Kaffee, L., y Simperl, E., eds.), vol. 11137 de Lecture Notes in Computer Science, pp. 376–394, Springer, 2018, doi:10.1007/978-3-030-00668-6_23.

[5] Everitt, T. y Hutter, M., "A topological approach to meta-heuristics: Analytical results on the BFS vs. DFS algorithm selection problem," CoRR, vol. abs/1509.02709, 2015, http://arxiv.org/abs/1509.02709.

[6] Lee, L., Siek, J. G., y Lumsdaine, A., "The generic graph component library," en Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1999, Denver, Colorado, USA, November 1-5, 1999 (Hailpern, B., Northrop, L. M., y Berman, A. M., eds.), pp. 399–414, ACM, 1999, doi:10.1145/320384.320428.

[7] Reid-Miller, M., "Lecture 10—Shortest Weighted Paths I,", https://www.cs.cmu.edu/afs/cs/academic/class/15210-s12/www/lectures/lecture10.pdf.

[8] Then, M., Kaufmann, M., Chirigati, F., Hoang-Vu, T., Pham, K., Kemper, A., Neumann, T., y Vo, H. T., "The more the merrier: Efficient multi-source graph traversal," Proc. VLDB Endow., vol. 8, no. 4, pp. 449–460, 2014, doi:10.14778/2735496.2735507.

[9] Heim, P., Hellmann, S., Lehmann, J., Lohmann, S., y Stegemann, T., "Relfinder: Revealing relationships in RDF knowledge bases," en Semantic Multimedia, 4th International Conference on Semantic and Digital Media Technologies, SAMT 2009, Graz, Austria, December 2-4, 2009, Proceedings (Chua, T., Kompatsiaris, Y., Mérialdo, B., Haas, W., Thallinger, G., y Bailer, W., eds.), vol. 5887 de Lecture Notes in Computer Science, pp. 182–187, Springer, 2009, doi:10.1007/978-3-642-10543-2_21.

[10] Heim, P., Lohmann, S., y Stegemann, T., "Interactive relationship discovery via the semantic web," en The Semantic Web: Research and Applications, 7th Extended Semantic

Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30 - June 3, 2010, Proceedings, Part I (Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., y Tudorache, T., eds.), vol. 6088 de Lecture Notes in Computer Science, pp. 303–317, Springer, 2010, doi:10.1007/978-3-642-13486-9_21.

[11] Tartari, G. y Hogan, A., "Wisp: Weighted shortest paths for RDF graphs," en Proceedings of the Fourth International Workshop on Visualization and Interaction for Ontologies and Linked Data co-located with the 17th International Semantic Web Conference, VOILA@ISWC 2018, Monterey, CA, USA, October 8, 2018 (Ivanova, V., Lambrix, P., Lohmann, S., y Pesquita, C., eds.), vol. 2187 de CEUR Workshop Proceedings, pp. 37–52, CEUR-WS.org, 2018, http://ceur-ws.org/Vol-2187/paper4.pdf.

[12] Inostroza Guerrero, B. H., "Interfaz para el curso"la web de datos"," 2020.

[13] Bangor, A., Kortum, P. T., y Miller, J. T., "An empirical evaluation of the system usability scale," Intl. Journal of Human–Computer Interaction, vol. 24, no. 6, pp. 574–594, 2008.