UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

Inteligencia Relacional en Twitter con Electron, Gephi y Vis.js

TRABAJO FIN DE MÁSTER

Máster Big Data Analytics

Autor: Marco Antonio Melero Gallo
Tutor: Francisco Manuel Rangel Pardo

Curso 2015-2016

Índice general

Índice general III
Índice de figuras V
Índice de tablas V
Índice de programas VI

1 Introducción 1
1.1 Motivación . . . 1
1.2 Objetivos . . . 3
1.3 Estructura de la memoria . . . 3
2 Marco tecnológico 5
2.1 Fuente de información . . . 6
2.2 Almacenamiento de información . . . 7
2.3 Procesamiento de información . . . 8
2.4 Interfaz de usuario . . . 9
2.4.1 Atom y Electron . . . 9
2.4.2 HTLM5 y CSS . . . 11
2.4.3 Bootstrap . . . 13
2.4.4 jQuery . . . 14
2.4.5 Librería vis.js . . . 14
2.5 Enlace entre interfaz de usuario y backend . . . 15
3 Procesamiento de la información 19
3.1 Modelo de datos . . . 19
3.2 Construcción del grafo . . . 21
3.3 Análisis y preproceso del grafo . . . 24
4 Explotación y visualización del grafo 27
4.1 Lectura y marcado de la información del grafo . . . 27
4.2 Visualización del grafo . . . 29
4.3 Opciones de configuración de la red . . . 30
4.4 Opciones de control de la interfaz . . . 31
5 Casos de uso 35
6 Conclusiones y trabajos futuros 37
Bibliografía 39

Apéndices
A Creación del grafo 41
B Opciones de visualización de vis.js 43

Índice de figuras

1.1 Metodología de Autoritas para la transformación digital 2
1.2 Herramientas de Autoritas para la transformación digital 2

2.1 Esquema de tecnologías empleadas 5
2.2 Modelo de relaciones entre usuarios de Twitter 7
2.3 Proceso de indexación con Lucene 8
2.4 Ejemplo de visualizaciones con Gephi 9
2.5 Editor Atom 10
2.6 Gráfica de descargas de Electron 10
2.7 Aplicaciones desarrolladas con Electron 11
2.8 Integración de Bootstrap en nuestro proyecto 13

3.1 Perfil de Twitter de la ETSINF 20
3.2 Ejemplo de filtrado de un grafo 23
3.3 Esquema de la función PrepareGraphJSON 26

4.1 Opciones de visualización del grafo 29
4.2 Interfaz de usuario 31
4.3 Menú lateral 32
4.4 Búsqueda avanzada 32

5.1 Visualización de un grafo con 20 nodos visibles 35
5.2 Visualización de un grafo con comunidades 36
5.3 Visualización del grafo completo 36

Índice de tablas

3.1 Listado de campos de la tabla Census 20
3.2 Listado de campos de la tabla Graph 21

4.1 Tabla de patrones de separación 28

Índice de programas

2.1 Estructura básica de una página HTML 12
2.2 Función de enlace con el backend 16
2.3 Ejemplo de enlace con el backend 17
2.4 Función inicial de enlace con el backend 17
3.1 Definición de la clase *Graph* 22
3.2 Algoritmo para crear el grafo *Graph* (sin filtros) 22
A.1 Algoritmo para crear el grafo *Graph* (con filtros) 41
B.1 Opciones de visualización de grafo 43

CAPÍTULO 1
Introducción

1.1 Motivación

Desde el nacimiento de Internet en los años 60, la red de redes no ha parado de crecer y evolucionar. Lo que hoy conocemos como internet, nada tiene que ver con lo que era en sus inicios. Nacida como una pequeña red de ordenadores interconectados, ésta fue expandiéndose lentamente y ampliando el número de ordenadores que a ella se adherían.

A finales de 1990, Tim Berners-Lee crea la primera página web, lo que conocemos como la Worl Wide Web (www). En sus inicios, las páginas web eran estáticas, de solo lectura, en las que el usuario no podía interactuar, y el acceso estaba totalmente limitado al Webmaster de la misma, que era el único que podía subir contenidos. Esto es lo que se conoce como Web 1.0.

Poco a poco, la Web 1.0 fue evolucionando dando paso a webs más dinámicas, tanto en su aspecto como en su contenido. Sitios diversos como blogs, wikis, foros y chats (entre otros) fueron surgiendo para dar un papel protagonista a los usuarios. Éstos ya no son meros esperactores y consumidores de información, sino que se convierten en *prosumidores*[1]. Es decir, además de obtener información de la web, la generan de manera activa. Es lo que conocemos como Web 2.0 o Web Social, en la que la comunicación es bidireccional.

El término Web 2.0, acuñado por Tim O'Reilly en 2004[2], no hace referencia solamente a un cambio tecnológico sino también a la manera de utilizar la web y relacionarse. Se fomenta la colaboración y el intercambio ágil de información entre los usuarios de una comunidad o red social. Es por ello que las redes sociales son un claro ejemplo de este nuevo paradigma colaborativo. Los usuarios de las redes sociales generan una enorme cantidad de contenidos de todo tipo y se comunican con otros usuarios. Es por ello que tanto las redes sociales como Internet son una fuente inagotable de información de la que surgen innumerables oportunidades de todo tipo.

Muchas marcas, empresas y negocios en general se encuentran en las redes sociales en busca de nuevas oportunidades de negocio. Para poder obtener una ventaja competitiva se requiere algo más que poder recoger información de las redes sociales. Los datos por sí solos no aportan valor alguno, sin embargo, esto cambia totalmente si se establecen metodologías adecuadas que permitan transformar esta información en inteligencia.

[1] https://es.wikipedia.org/wiki/Prosumidor

[2] Término definido por Dale Dougherty de O'Really Media en una tormenta de ideas con Craig Cline de MediaLive para la conferencia organizada en Octubre del 2004, bajo el lema La Web como plataforma.

Autoritas Consulting[3] es una empresa que se dedica a la consultoría estratégica, global y de negocio que se centra en la generación de inteligencia para las organizaciones. A partir del conocimiento del cliente en sus distintos ámbitos y mediante la utilización de una metodología propia, permite ayudarlos en la toma de decisiones.

La metodología propuesta por Autoritas (Sapere-Agile [1]) es la que podemos ver en la Figura 1.1, donde se establecen distintas fases de intervención y donde cada una de estas fases se representa con un color. En cada una de estas fases, se utilizan una serie de herramientas. Dichas herramientas aparecen dibujadas en la Figura 1.2 con el mismo color de la fase de intervención en la que se utilizan.

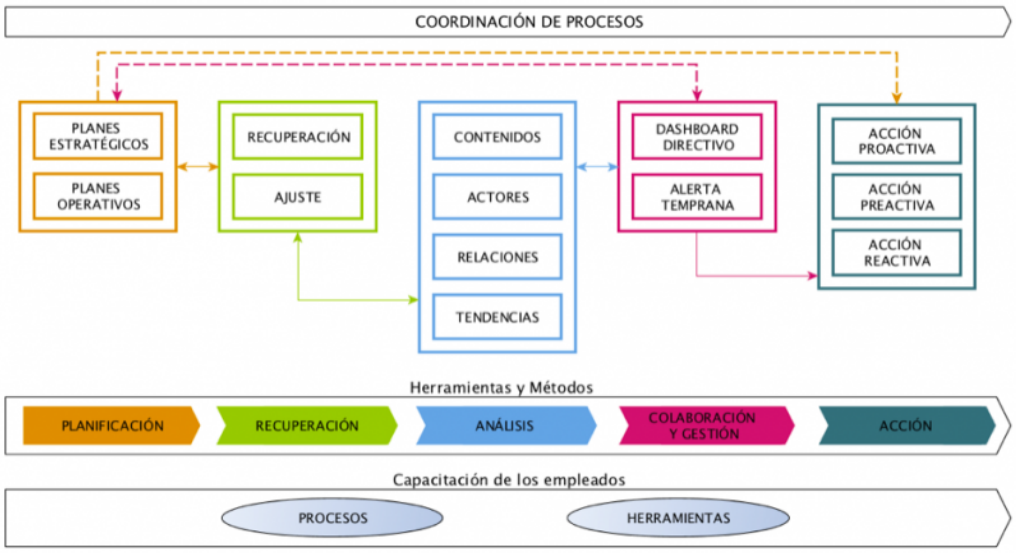

Figura 1.1: Metodología de Autoritas para la transformación digital

Figura 1.2: Herramientas de Autoritas para la transformación digital

[3] http://autoritas.net

El presente trabajo se enmarca dentro de la herramienta *Social Census TW*, en las fases de recuperación y análisis. Dicha herramienta permite, entre otras cosas, el censado y etiquetado de usuarios de Twitter.

Es importante indicar, que aunque el desarrollo se haya realizado dentro de una solución propia de la empresa Autoritas, el modelado y la solución implementadas podrían ser extrapoladas a cualquier otra herramienta para el análisis de Inteligencia Relacional en Twitter.

1.2 Objetivos

El objetivo principal de este proyecto ha sido el desarrollo de una herramienta de visualización de grafos mediante la modelización de la estructura de relaciones de la red social Twitter.

Dicha herramienta debe de cumplir las siguientes características:

- Ser sencilla y de fácil manejo.
- Tener un buen tiempo de respuesta.
- Permitir el análisis visual del grafo detectando comunidades de usuarios y sus *influencers*.
- Permitir el filtrado de información para una mejor explotación de la información existente.
- Permitir exportar los resultado en algún formato concreto para poder ser utilizados en otros aplicativos.

1.3 Estructura de la memoria

Este proyecto se ha estructurado en las siguientes partes:

- **Capítulo 1. Introducción.** En este capítulo se realiza una breve introducción de la evolución de la web y de las redes sociales, viendo las oportunidades que nos brindan si se saben analizar del modo adecuado.
- **Capítulo 2. Marco tecnológico.** En este capítulo se describen las distintas tecnologías empleadas para llevar a cabo este proyecto y se explica la modelización de la estructura relacional de Twitter que se ha realizado.
- **Capítulo 3. Procesamiento de la información.** En este capítulo se explica detalladamente todo el procesado de la información que se ha realizado, desde su recuperación de la red social Twitter hasta su visualización por pantalla.
- **Capítulo 4. Explotación y visualización del grafo.** En este capítulo se describen las distintas opciones desarrolladas en la interfaz de usuario.
- **Capítulo 5. Casos de uso.** En este capítulo presentamos algunos ejemplos prácticos realizados con la interfaz de usuario desarrollada.

- **Capítulo 6. Conclusiones y trabajos futuros.** En este capítulo presentamos las conclusiones al trabajo que hemos llevado a cabo, comentando lo aprendido durante la realización del mismo, analizando si se han conseguido los objetivos planteados inicialmente, y finalmente, comentando los posibles trabajos futuros.

CAPÍTULO 2
Marco tecnológico

En este apartado describiremos las diversas y variadas tecnologías empleadas en el marco de este proyecto. Para ello, comenzaremos explicando qué es Twitter, cómo se estructura y cómo puede obtenerse información de esta fuente. Para nosotros se corresponde con una fuente inagotable de información de la que podemos y debemos sacar el máximo partido.

Además de Twitter, veremos las tecnologías empleadas en el desarrollo de este proyecto. Éstas se encuentran diferenciadas en dos grupos, las utilizadas para el desarrollo en el lado del servidor (*tecnologías backend*), y las utilizadas para el desarrollo en el lado del cliente (*tecnologías frontend*). En la Figura 2.1 podemos ver un pequeño esquema de todas ellas, agrupadas por sus distintos ámbitos de uso.

A continuación detallaremos el papel de cada una de estas tecnologías.

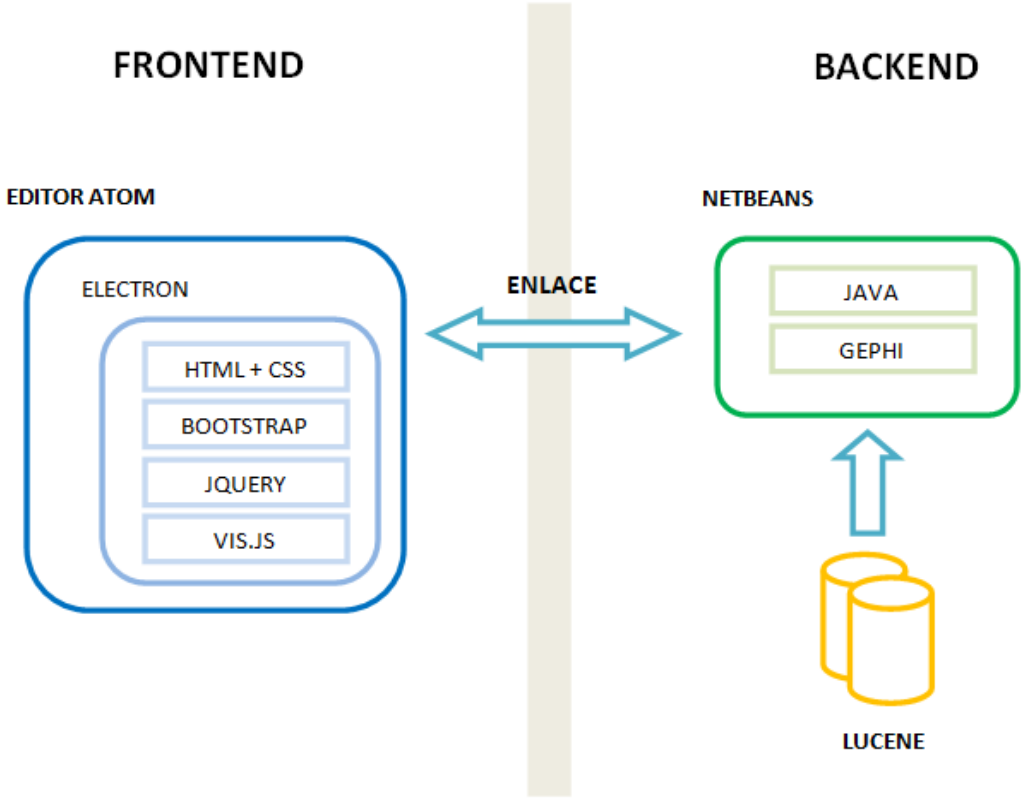

Figura 2.1: Esquema de tecnologías empleadas

2.1 Fuente de información

Twitter es una de las redes sociales más extendidas a nivel mundial. Desde su creación en 2006 su número de usuarios no ha parado de crecer hasta alcanzar los más de 500 millones de usuarios que existen actualmente. Con un máximo de 140 caracteres por mensaje y una gran facilidad de uso, se ha abierto hueco en el mundo digital [1].

Podríamos considerarla como una gran ventana al mundo desde la cual puede obtenerse una amplia gama informativa y de muy diversa índole (noticias de actualidad, conversaciones y opiniones entre usuarios, publicidad, etc.). «Su éxito se basa en el uso de hastags, palabras claves precedidas de una almohadilla si se trata de un tema o de una @ si se trata de una persona, que facilitan la monitorización de temas de esta red social» [2].

De toda esta gran cantidad de información ofrecida por Twitter, **¿qué es lo que necesitamos para generar Inteligencia Relacional?**

Lo primero que necesitamos es poder identificar a los actores que intervienen, los contenidos que aparecen y las relaciones que se dan entre todos ellos. A continuación debemos saber cómo almacenar esta información. La mejor manera para poder visualizar estas relaciones es mediante un grafo porque permite modelar la interacción entre los distintos elementos exitentes. «Un grafo se define como un conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas o arcos. Matemáticamente un grafo G=(V,E) es una pareja ordenada en la que V es un conjunto no vacío de vértices y E es un conjunto de aristas»[2].

Una vez conocida la forma en que se visualizarán las relaciones de Twitter, distinguiremos tres áreas de estudio en función de la interacción de los distintos componentes presentes en nuestro escenario. Estas áreas son las siguientes:

- *Relaciones entre usuarios*: consiste en analizar las relaciones existentes entre los diferentes usuarios para lograr determinar cuáles de ellos son más influyentes (*influencer*) y qué comunidades se forman. Estas relaciones, tanto a nivel de influencer como de comunidad, pueden ser muy diversas, ya que las influencias ejercidas, tanto por los usuarios como las comunidades que se forman, pueden depender de las relaciones directas entre los usuarios (es seguido/sigue a), de las conversaciones que mantienen los usuarios a través de menciones, retwuits o favoritos, o incluso de la compartición de contenidos mediante hashtags.

- *Relaciones entre contenidos y usuarios*: en este caso, los contenidos de Twitter pueden ser varios; temas, URLs, hashtags, categorías, etiquetas y palabras. En todos ellos se analiza el grado de relación de los usuarios con los distintos contenidos. En función del contenido, el grado de relación permite mostrar los contenidos más leídos y/o compartidos por los usuarios (en el caso de las URLs), la distribución de *influencers* respecto a los hashtags o ver el grado de conversación generado por los usuarios cuando se analizan las categorías.

- *Relaciones entre contenidos*: consiste en analizar las relaciones entre los contenidos que son categorías, etiquetas o palabras. Se basa en la posible vinculación de aparición entre los contenidos. La representación gráfica de éstos suele venir dada por el número de apariciones de las mismas, y el grosor de las aristas que relacionan los contenidos por el número de veces que quedan relacionados los nodos.

[1] https://about.twitter.com/es/company
[2] https://es.wikipedia.org/wiki/Grafo

De manera genérica, el modelo de relaciones para la red social Twitter queda detallado y explicado en [3]. Este modelo es el que tomamos como referencia en este proyecto, seleccionando un submodelo que trate de explicar las distintas relaciones entre los usuarios.

El submodelo que vamos a utilizar representa la relación de "amistad" entre los usuarios, en la que un determinado usuario puede seguir a otro usuario, y a su vez, le pueden seguir otros usuarios. En la Figura 2.2 vemos la representación de este modelo. En él observamos dos relaciones distintas; una relación en la que el usuario A sigue al usuario B, y otra en la que el usuario B es seguido por el usuario A. El modelo se ha realizado de la manera más sencilla posible, pero ello no significa que estas dos relaciones tengan que darse obligatoriamente entre dos nodos, podrían darse entre otros muchos.

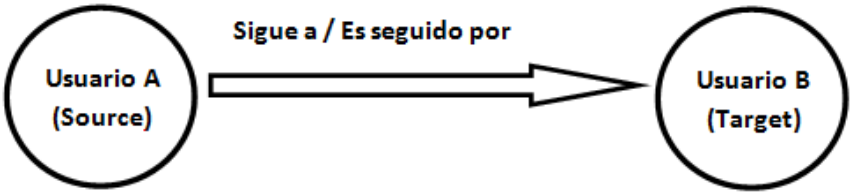

Figura 2.2: Modelo de relaciones entre usuarios de Twitter

¿Cómo obtenemos la información de Twitter?

La información de Twitter se obtiene a través de su API (*Application Programming Interface*)[3]. Dicha API[4] nos da acceso a la lectura y escritura de los datos de Twitter, pero no de manera ilimitada. Existen algunas limitaciones a la hora de obtener estos datos. Algunos ejemplos de estas limitaciones pueden ser la realización de un número máximo de peticiones a la API durante una ventana temporal de 15 minutos, o la obtención de un número máximo de tweets en cada una de las peticiones. No profundizaremos en estos aspectos porque no son objeto del estudio de este proyecto, pero es importante conocer su existencia.

2.2 Almacenamiento de información

El proceso de almacenamiento de la información extraída de Twitter se lleva a cabo sobre Lucene[5]. Lucene es una librería de código abierto (con licencia de Apache) desarrollada enteramente sobre Java. Además, dispone de otras versiones para otros lenguajes como por ejemplo Delphi, Perl, C#, C++, Python, Ruby y PHP.

Una de las grandes ventajas de Lucene es que, además de ser una librería escalable de alto rendimiento, permite la indexación y búsqueda de documentos, entendiendo por documento cualquier información que pueda representarse de manera textual. Por este motivo, es una herramienta tremendamente potente que permite la extracción de información, independientemente de los formatos de fichero utilizados.

[3]«API ,del inglés Application Programming Interface, conocida en castellano como interfaz de programación de aplicaciones, es un conjunto de subrutinas, funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción»

[4]`https://dev.twitter.com/rest/public`

[5]`http://lucene.apache.org/core/index.html`

Durante el proceso indexación se extrae toda la información necesaria de las distintas fuentes origen (en nuestro caso desde la red social Twitter), se realiza un proceso de análisis, y finalmente se almancenan los índices según determinadas estructuras de datos que permiten un rápido acceso a esta información. En la Figura 2.3 vemos con mayor claridad el proceso que se lleva a cabo.

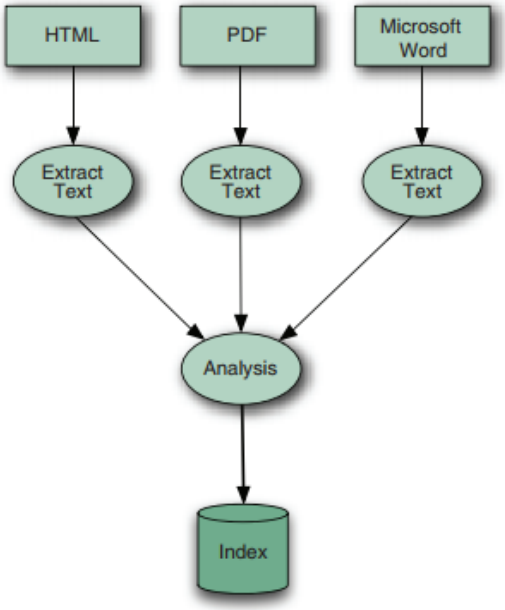

Figura 2.3: Proceso de indexación con Lucene. Tomada de [4], pág.34

El proceso de búsqueda no es más que la consulta de nuestra información a través de los índices creados previamente. Para ello se utilizan determinadas clases Java, mediante las cuales se lanzan *Querys* específicas sobre la información almacenada. El resultado de estas búsquedas se devuelve en un objeto *Hit* que contiene los documentos seleccionados.

2.3 Procesamiento de información

El procesamiento de nuesto grafo se llevará a cabo mediante la API de Gephi. «Gephi es una herramienta de código abierto desarrollada en Java que permite la visualización y el análisis de grandes gráficos de red»[6]. Gephi permite trabajar con diferentes tipos de archivos de grafos: .graphml, .gexf, .gdf, .gephi, etc. e importar desde archivos .csv la información necesaria para la lectura de los nodos y las aristas que definen el grafo.

Gephi, ampliamente utilizado en proyectos de investigación académica y periodismo de datos, es ideal para el análisis de las redes sociales ya que permite mostrarlos en tiempo real, explorarlos, analizarlos, filtrarlos y clusterizarlos. Por este motivo podemos decir que se trata de una herramienta muy potente que nos ofrece una amplia cantidad de opciones para el análisis de nuestra red.

En la Figura 2.4 podemos ver un par de ejemplos de visualizaciones realizadas con Gephi.

[6] https://gephi.org/

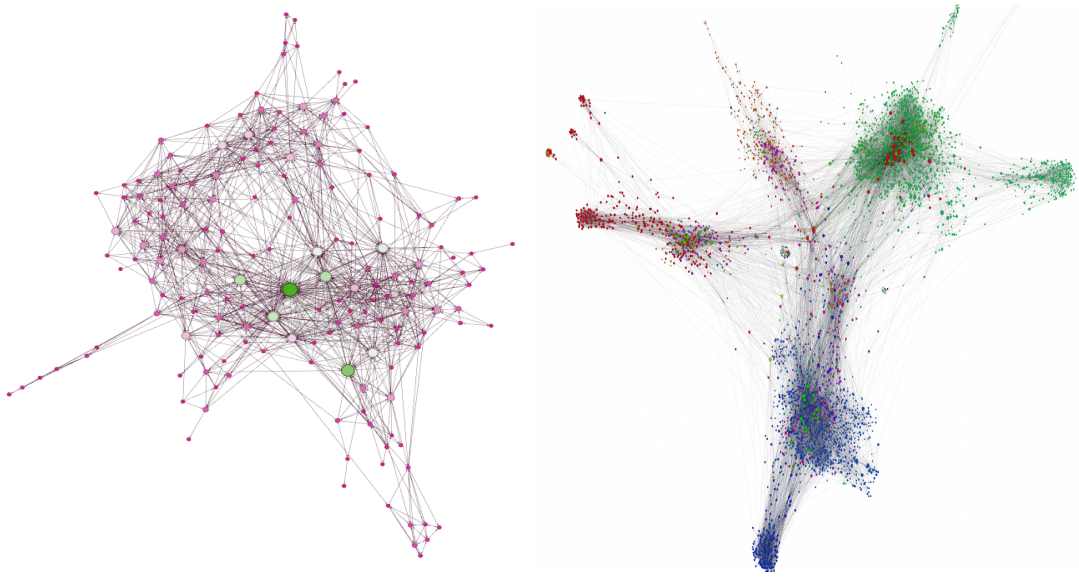

Figura 2.4: Ejemplos de visualizaciones de grafos con Gephi

2.4 Interfaz de usuario

El objetivo de nuestro proyecto es desarrollar una interfaz de usuario que permita, de manera clara y sencilla, la extracción de inteligencia a partir de las relaciones y la interacción de los usuarios de la red social Twitter. Para lograrlo, hemos tenido que aunar gran variedad de tecnologías que describiremos a continuación.

2.4.1. Atom y Electron

El desarrollo de nuestra interfaz de usuario se ha llevado a cabo mediante la utilización de Atom. Atom es un editor de código abierto para sistemas operativos Windows, Linux y macOS. Ha sido desarrollado por el equipo de GitHub[7] y está basado en tecnologías web (escrito enteramente en HTML, CSS, Javascript y Node.js). Se trata de un editor muy flexible, ya que disponde de muchos temas diferentes para su personalización, así como de plugins que lo dotan de funcionalidad extra. Además de estas extensiones, Atom es fácilmente *hackeable*, es decir, cualquier programador con los conocimientos adecuados podría modificar el código del editor y adaptarlo a sus gustos y necesidades.

Hemos utilizado la versión de Atom que se ejecuta bajo el sistema operativo Windows. Para su utilización, descargamos el ejecutable de su página web[8] y lo instalamos, siguiendo los pasos indicados en el instalador.

En la Figura 2.5 vemos una imagen del editor Atom. A la izquierda de la pantalla se encuentra el área de navegación a través de los directorios de nuestro proyecto. A la derecha están los distintos ficheros que se han ido abriendo con el uso de Atom.

[7] https://github.com/
[8] https://atom.io/

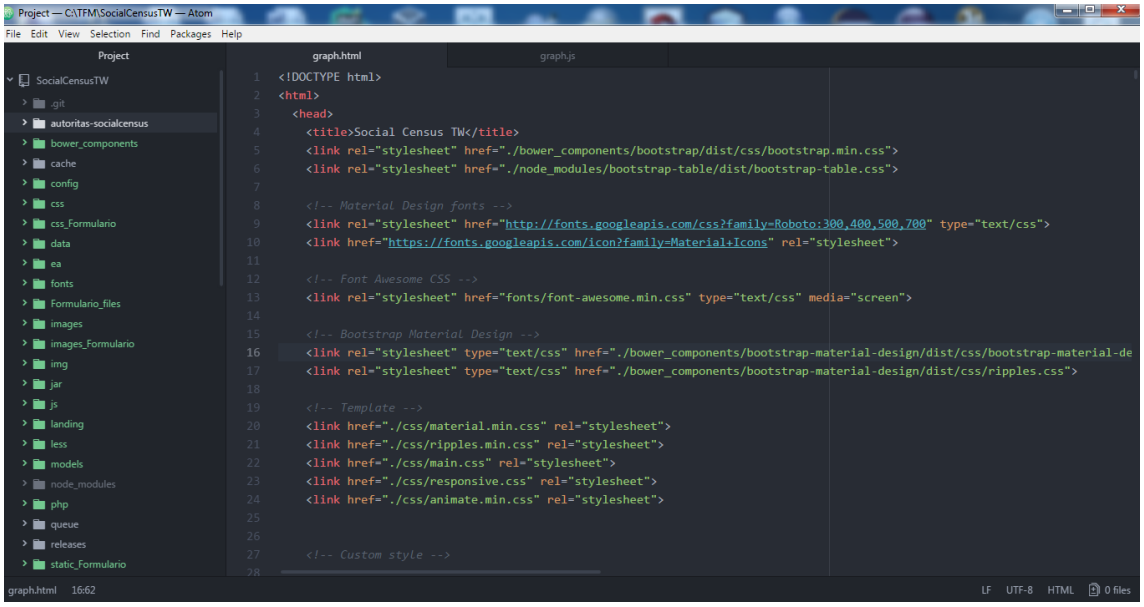

Figura 2.5: Editor Atom

Una vez conocido el editor empleado, veremos el framework utilizado para el desarrollo de la interfaz. Éste ha sido Electron, y en él se englobarán el resto de *tecnologías frontend* utilizadas (HTML5, CSS, jQuery, Bootstrap y vis.js). Electron[9] es un framework de código abierto utilizado para desarrollar aplicaciones de escritorio multiplataforma (MAC, Windows y Linux). Está basado en Node.js[10] y Chromium[11], y nos permite desarrollar aplicaciones utilizando javascript, HTML y CSS. Utiliza páginas web como interfaz gráfica de usuario, donde cada página se ejecuta sobre su propio proceso (utiliza la arquitectura multi-proceso de Chrome). Podríamos considerarlo un navegador Chromium controlado por Javascript. Por estos y otros motivos, Electron se trata de una herramienta rápida, cómoda y de fácil uso que no necesita aprendizaje de nuevos lenguajes de programación. Creado por GitHub, y anteriormente conocido por Atom Shell, Electron dispone de un extenso grupo de activos colaboradores procedentes de su amplia comunidad [5].

Tal y como muestra la Figura 2.6 el número de descargas de Electron ha ido en aumento desde su creación, llegando hasta los 1,2 millones en mayo del 2016.

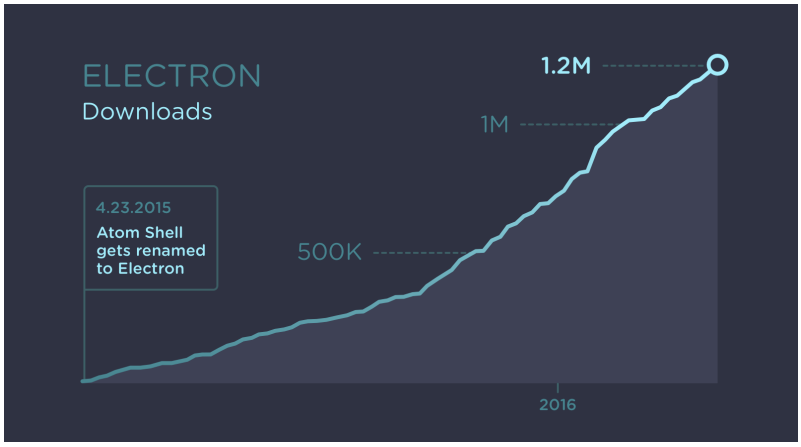

Figura 2.6: Descargas de Electron entre abril del 2015 y mayo del 2016

[9] https://electron.atom.io
[10] https://nodejs.org/en/
[11] https://www.chromium.org/Home

Veamos algunos ejemplos de aplicaciones desarrolladas en Electron en la Figura 2.7, entre las que destacan Slack[12] y el editor de código gratuito Visual Studio Code[13].

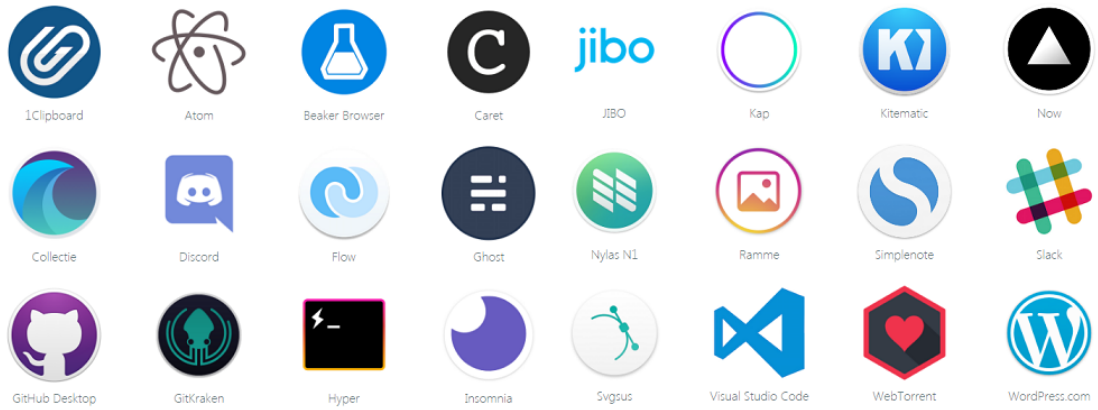

Figura 2.7: Aplicaciones desarrolladas con Electron

Para empezar a utilizar Electron primero debemos instalar Io.js[14] y Node.js. Nosotros hemos utilizado la versión 6.9.1 de Node.js mediante el instalador de Windows. Posteriormente, procedimos a la instalación de Electron mediante el paquete npm y el siguiente comando:

npm install electron-prebuilt -g

Por último, instalamos electron-packager-interactive (***epi***). Para ello utilizamos el comando que puede verse bajo. *Epi* es una herramienta interactiva de empaquetado de las aplicaciones generadas por Electron.

npm install -g electron-packager-interactive

2.4.2. HTLM5 y CSS

«HTML, sigla en inglés de *HiperText Markup Language* (Lenguaje de Marcado de Hipertexto), hace referencia al lenguaje de marcado para la elaboración de páginas web. Es un estándar que sirve de referencia del software que conecta con la elaboración de páginas web en sus diferentes versiones, define una estructura básica y un código (denominado código HTML) para la definición de contenido de una página web, como texto, imágenes, videos, juegos, entre otros. Es el estándar que se ha impuesto en la visualización de páginas web y el que todos los navegadores actuales han adoptado»[15].

Toda la información de las páginas web creadas con HTML se crean a partir de etiquetas que vienen predefinidas. Estas etiquetas se indican a través de corchetes (por ej.: <body>). La estructura básica de una página web está compuesta por una cabecera (<*head*>) con un título (<*title*>), y el cuerpo de la página (<*body*>) donde iría el resto del contenido de la página.

[12] https://slack.com/
[13] https://code.visualstudio.com/
[14] A partir de la versión 4 de Node.js, sus instaladores llevan incluido Io.js
[15] https://es.wikipedia.org/wiki/HTML

En el código fuente 2.1 presentamos un ejemplo.

```
<html>
    <head>
        <title>Título de la página</title>
    </head>
    <body>
        Aquí iría el contenido de la página
    </body>
</html>
```

Código fuente 2.1: Estructura básica de una página HTML

Los principales rescursos del lenguaje HTML son:

- Etiquetas: permiten la inclusión de imágenes, tablas, etc.
- Hipertexto: es la forma en la que te puedes desplazar dentro de una web, ya sea de manera local o remota.
- Hiperlinks: también conocido como hipervínculo o enlace, es un elemento que hace referencia a otro recurso, pudiendo ser éste otro documento o cualquier tipo de recurso.

Las principales ventajes de la utilización de HTML son su facilidad de comprensión y su uso extendido. Concretamente, en nuestro proyecto hemos utilizado la versión 5 de este lenguaje (HTML5), que proporciona una serie de ventajas adicionales a las ya existentes en el lenguaje HTML [6]. Algunas de estas ventajas son las siguientes:

- Nueva estructura de etiquetas mejoradas. Muchas de estas nuevas etiquetas son más semánticas y favorecen el posicionamiento (SEO) y la accesibilidad.
- Incluye una nueva etiqueta de dibujo sobre la página web llamada *canvas*.
- Incluye dos nuevas etiquetas de audio y vídeo que permiten la ejecución de audio y vídeo sin la necesidad de incluir código flash.
- Permite el acceso y uso de las páginas web sin necesidad de estar conectados a internet.
- Disminuye el tiempo de carga de las páginas, haciéndolas más ligeras.
- Tiene una mayor compatibilidad con navegadores de dispositivos móviles y facilita el desarrollo adaptativo a estos dispositivos.
- Facilita la geolocalización.

CSS (*Cascading Style Sheet*) son las siglas del nombre en inglés de *Hojas de estilo en cascada*. Es un lenguaje que permite definir la presentación de los documentos HTML. En él podemos definir las fuentes a utilizar, los colores, márgenes, imágenes de fondo, posicionamiento de los distintos elementos de las páginas, etc.

Su utilización en este proyecto, al igual que en otras muchas webs, está motivada por las multiples ventajas que conlleva. Por ejemplo:

- Permite la definición de todos los estilos en uno o más ficheros que son independientes de la estructura HTML. De esa manera, la capa de presentación puede cambiarse en cualquier momento sin necesidad de modificar el documento HTML.

- La utilización de hojas de estilo CSS ayuda a tener un código más limpio y organizado.
- Mejora la velocidad de carga de la web y por tanto el posicionamiento SEO.
- Permite la utilización de distintas hojas de estilo en función del dipositivo que se esté utilizando para la visualización.

2.4.3. Bootstrap

Bootstrap[16] es un framework de código abierto utilizado para el diseño adaptativo de aplicaciones web y de móviles. Contiene plantillas de diseño creadas con HTML, CSS y Javascript. Dichas plantillas están formadas por tipografías, formularios, botones, menús de navegación y otros elementos de diseño preestablecidos. Esto nos pemite diseñar páginas web para todo tipo de dispositivo de una manera fácil, rápida y sencilla [7].

Inicialmente fue creado por Mark Otto y Jacbod Thornton como una solución interna de Twitter. Posteriormente, en agosto de 2011 su código fue liberado en Github, convirtiéndose hoy en día en uno de sus repositorios más populares.

Existen varias maneras de comenzar a utilizar Bootstrap en nuestras aplicaciones. Podríamos instalarlo mediante la utilización de cualquiera de estos gestores de paquetes, npm[17], RubyGems[18], Composer[19], Bower[20] o NutGet[21]. En nuestro caso, hemos decidido descargarlo de su web[22], incluir las carpetas en nuestro proyecto, y enlazar los archivos de Bootstrap desde nuestras páginas para que funcione correctamente (véase la Figura 2.8). Hemos utilizado la versión 3.3.6 de Bootstrap.

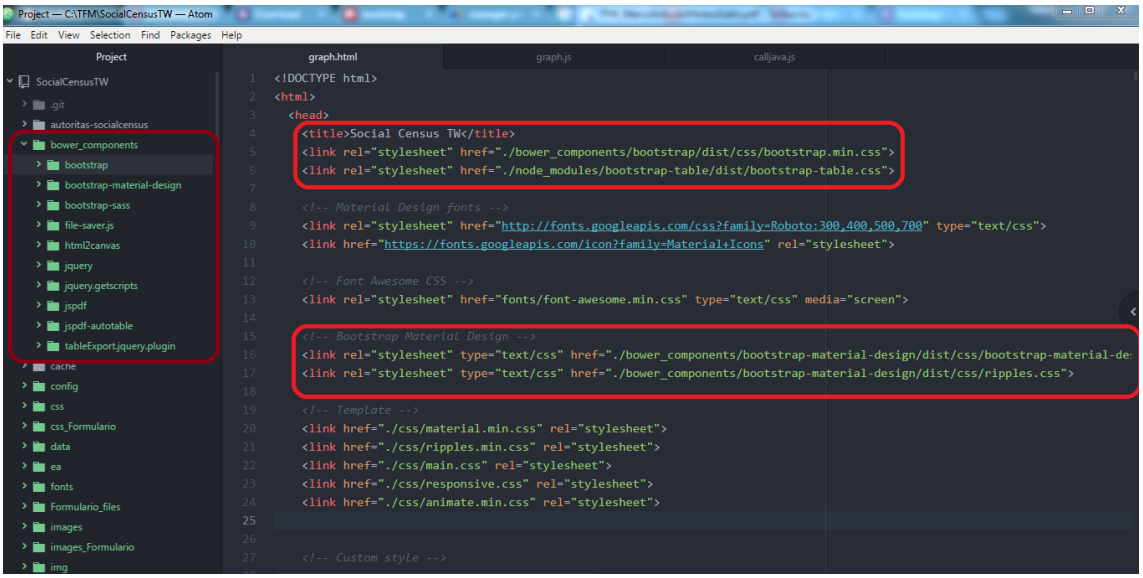

Figura 2.8: Integración de Bootstrap en nuestro proyecto

Como hemos comentado anteriormente, Boostrap se compone de plantillas de diseño. Las hay de muchos tipos, algunas gratuitas y otras de pago. En nuestro proyecto nos

[16] http://getbootstrap.com/
[17] https://www.npmjs.com/
[18] https://rubygems.org/
[19] https://getcomposer.org/
[20] https://bower.io/
[21] https://www.nuget.org/
[22] http://blog.getbootstrap.com/2015/11/24/bootstrap-3-3-6-released/

hemos decantado por una gratuita, que hemos descargado de esta web[23] y donde pueden verse los distintos elementos de diseño que están a nuestra disposición.

En la plantilla que hemos utilizado no hay ninguna referencia a los elementos conocidos como *sliders*. Hemos referenciado otra plantilla específica[24] en la que solo se desarrolla este tipo de elemento. De esa manera, utilizando ambas plantillas, hemos desarrollado el diseño de nuestra aplicación.

2.4.4. jQuery

jQuery [25] es un librería Javascript de código abierto distribuida bajo licencia del MIT (*Massachusetts Institute of Technology*) [26]. La sencillez de su código simplifica la tarea de programar en Javascript y permite agregar interactivad a los sitios web. La limpieza de su código y la efectividad a la hora de obtener resultados la convierten en una librería fundamental para el desarrollo web [8].

Entre sus muchas ventajas podemos destacar las siguientes:

- El código desarrollado es independiente del navegador que estemos utilizando.
- Es flexible y rápido para el desarrollo web.
- Es un producto estable, bien documentado y con un buen soporte (hay actualizaciones frecuentes para la corrección de errores debido a su amplia comunidad de desarrolladores).
- Dispone de una gran cantidad de plugins (interfaces de usuario, galerías de fotos, carrusel de imágenes, efectos varios, etc.) que facilitan y mucho el desarrollo de aplicaciones.
- Tiene una excelente integración con AJAX[27].

Todas estas ventajas, unidas a la sencillez de su código, nos permiten desarrollar aplicaciones muy potentes con un alto grado de interactividad, razones que nos han llevado a utilizarla en nuestro proyecto.

De la misma manera que ocurre con Bootstrap, jQuery puede instalarse, bien mediante la utilización de los gestores de paquetes npm y Bower, bien con la descargar local de los ficheros necesarios, referenciándolos después. En nuestro proyecto hemos optado por la segunda opción, tal y como hicimos con Bootstrap.

2.4.5. Librería vis.js

Para la visualización de los grafos de nuestra aplicación utilizamos vis.js. «Vis.js es una librería de visualización dinámica y basada en navegador. Está diseñada para ser fácil de usar, manejar grandes cantidades de datos dinámicos y habilitar la manipulación e interacción de los mismos»[28]. La librería dispone de 5 tipos de componentes:

[23] http://fezvrasta.github.io/bootstrap-material-design/bootstrap-elements.html
[24] https://refreshless.com/nouislider/
[25] https://jquery.com/
[26] «MIT (*Massachusetts Institute of Technology*), son las iniciales del nombre en inglés del *Instituto Tecnológico de Massachusetts*. Es una universidad privada localizada en Cambridge, Massachusetts (Estados Unidos) cuya página web es la que puede verse a continuación http://web.mit.edu/»
[27] https://es.wikipedia.org/wiki/AJAX
[28] http://visjs.org/

- Network: Permite la visualización de redes dinámicas, personalizables y que se organizan de manera automática. Hay muchas y variadas opciones de personalización, tanto de nodos como de ejes, como de la propia estructura de la red. Se puede personalizar las formas, los colores, los tamaños, las imágenes, etc. Un amplio repertorio de opciones que convierten a este componente en una opción muy versátil para la representación de todo tipo de redes.

- Timeline: Componente utilizado para representar datos sobre líneas temporales de manera interactiva. Para ello se puede crear, modificar y borrar elementos de la línea temporal, aplicar zoom o alejar la vista teniendo total libertad sobre la misma, establecer rangos de periodos visualizables y otra serie de opciones completamente configurables.

- Graph2d: Permite la visualización de gráficos en dos dimensiones y diagramas de barras completamente interactivos. Para su dibujado se utiliza el HTML DOM (*HTML Document Objet Model*[29]) y SVG (*Scalable Vector Graphics*[30]). Además es fácilmente personalizable con estilos CSS.

- Graph3d: De manera análoga a lo que sucede con el componente Graph2d, con este componente podemos crear gráficos animados e interactivos pero en este caso en 3 dimensiones. Para ello utiliza el *HTML canvas*[31]. Algunos elementos que pueden utilizarse son superfícies, líneas, puntos, etc. pudiendo llegar a dibujar varios miles de puntos de datos con fluidez.

- DataSet: Se utiliza para la gestión de datos desestructurados en formato JSON. Se pueden añadir, modificar y borrar los datos que se deseen, así como detectar cambios sobre los mismos.

En nuestro caso de visualización del grafo de relaciones de Twitter, solamente necesitaremos dos de sus componentes:

- *DataSet* para el almacenamiento de los nodos y las relaciones existentes entre los nodos del grafo a representar.

- *Network* para la visualización y manipulación gráfica de nuestro grafo.

En los apartados 4.2 y 4.3 profundizaremos en el uso de estos dos componentes.

2.5 Enlace entre interfaz de usuario y backend

Como hemos comentado al comienzo del apartado 2, el desarrollo de nuestra interfaz gráfica se compone de dos partes. Una parte gráfica y visible para el usuario (frontend), y otra (no visible) que gestiona la interacción con los datos almacenados en nuestras bases de datos Lucene (backend). En este apartado explicaremos cómo se han integrado ambas partes y qué problemas han surgido durante esta integración.

Por un lado, toda la interfaz gráfica ha sido desarrollada en Electron y otras tecnologías dedicadas al desarrollo web, de las que hemos hablado en apartados anteriores. Por otro lado, el backend ha sido desarrollado en Java con NetBeans[32].

[29] https://es.wikipedia.org/wiki/Document_Object_Model
[30] https://es.wikipedia.org/wiki/Scalable_Vector_Graphics
[31] https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API
[32] https://netbeans.org/

NetBeans es un IDE (*Integrated Development Environment*)[33] libre, gratuito y de código abierto. Permite el desarrollo de aplicaciones de escritorio, móviles y web con Java, Javascript, HTML5, PHP, C/C++ y otros, aunque principalmente está orientado al uso de Java.

Mediante NetBeans hemos generado el fichero SocialCensusLib.jar que contiene toda la lógica de nuestro backend, y éste se ha dejado en la ruta \SocialCensusTW\jar de nuestro proyecto en Electron. Además de este fichero, y para que todo funcione correctamente, ha sido necesario dejar en la ruta \SocialCensusTW\jar\lib las librerías externas utilizadas en la implementación en Java, donde cabe destacar la librería de Gephi.

Para realizar el enlace entre ambas partes, creamos la función javascript que puede verse en el código fuente 2.2. Dicha función recibe tres parámetros que son los siguientes:

- *jar_name*: nombre del fichero jar que se quiere ejecutar.
- *function_name*: nombre de la función a la que se quiere invocar.
- *arguments*: parámetros que recibirá la función a la que vamos a invocar.

```
function javaMethodSyncString (jar_name, function_name, arguments) {

    const spawnSync = require('child_process').spawnSync;
    var path = __dirname;
    var child = spawnSync('java', ['-jar', '"' + __dirname + '/jar/' +
        jar_name + '"', function_name, '"' + path + '"', arguments], {shell:
        true, encoding: 'utf8'});
    var response = child.stdout.toString();

    return response;

}
```

Código fuente 2.2: Función de enlace con el backend

[33]IDE, del inglés Integrated Development Enviroment, conocido en castellano como entorno integrado de desarrollo, es una aplicación informática que está compuesta por un conjunto de herramientas útiles para el desarrollador

En el código fuente 2.3, podemos ver un ejemplo en el que se observa con más claridad cómo se hace la llamada a esta función. En él se muestra una llamada al proceso de recálculo del grafo (*PrepareGraphJSON*).

```
function RecalculateGraph(origin) {

    var jar_name = "SocialCensusLib.jar";
    var function_name = "PrepareGraphJSON";

    var visibleData = 0;

    if (origin=="PORCENTAJE"){
      visibleData = Math.floor(sliderPercentageNodes.noUiSlider.get());
    } else {
      visibleData = $('#visibleNodes').val();
    }

    var fs = require('fs');
    var project = fs.readFileSync(__dirname+'/config/proyectolista.txt', '
        utf8');
    var arguments ='"'+project+'" "'+origin+'" "'+visibleData+'"';

    var result = javaMethodSyncString(jar_name, function_name, arguments);

    return result;

}
```

Código fuente 2.3: Ejemplo de enlace con el backend

Una de las principales dificultades que nos hemos encontrado durante la integración de estos dos entornos ha sido la de depurar los errores de una manera sencilla. Esto se debía a que no se podía realizar una depuración conjunta de ambos entornos. Es decir, si se producía un error en una función Java durante la ejecución de la aplicación, debíamos simular la llamada a ésta desde NetBeans con los parámetros adecuados. En ningún caso podíamos utilizar las herramientas de depuración de Atom porque solamente nos permitía ver los parámetros de entrada, y los valores de salida de la función.

El error más importante que hemos encontrado durante esta fase de integración fue el tamaño del valor de respuesta de una función Java. Nos referimos al grafo en formato json que devuelve una de las funciones. Inicialmente, la función que hemos visto en 2.2 era ligeramente distinta. El código original es el que puede verse en 2.4.

```
function javaMethodSyncString (jar_name, function_name, arguments) {

    const spawnSync = require('child_process').spawnSync;
    var path = __dirname;
    var child = spawnSync('java', ['-jar', '"' + __dirname + '/jar/' +
        jar_name + '"', function_name, '"' + path + '"', arguments], {shell:
        true});
    var response = child.stdout.toString();

    return response;

}
```

Código fuente 2.4: Función inicial de enlace con el backend

Como puede observarse, el código es prácticamente idéntico, con la salvedad de la llamada a la función *spawnSync*. Al llamar a esta función lo que se consigue es que el programa no continúe hasta que el proceso hijo no haya finalizado toda la tarea, o hasta que se le haya enviado una señal de finalización[34]. En nuestro caso concreto, no estaba sucediendo ninguna de las dos cosas y, sin embargo, el proceso no devolvía los mismos datos que se obtenían al lanzar la función desde NetBeans. Finalmente, nos percatamos de que la función spawnSync utiliza por defecto buffers, y éstos tienen una limitación máxima de 200*1024 bytes. Si esta limitación se excede, el proceso hijo termina y, por lo tanto, devuelve el control. Así que la solución pasó por utilizar Strings en lugar de buffers y, de esa manera, quedó solucionado nuestro problema.

[34] https://nodejs.org/api/child_process.html#child_process_synchronous_process_creation

CAPÍTULO 3
Procesamiento de la información

En este capítulo analizaremos el proceso de tratamiento de nuestra información, desde que obtenemos la información de Twitter y la almacenamos en las bases de datos Lucene, hasta su visualización en nuestro aplicativo.

Trataremos tres aspectos: el modelo de datos, la construcción del grafo y el análisis y preproceso del mismo.

En el **modelo de datos** se explica cómo queda almacenada la información en las bases de datos Lucene. La **construcción del grafo** es la vista en detalle del proceso diseñado para extraer la información de la base de datos, y la construcción del grafo que se visualizará en fases posteriores. Finalmente, una vez hemos creado nuestro grafo sobre la estructura adecuada, realizamos un **análisis y preproceso del grafo** para conseguir su correcta visualización en nuestro aplicativo.

3.1 Modelo de datos

Hemos extraído toda la información de la red social Twitter. Mediante la utilización de su API, descargamos un conjunto de usuarios, sus seguidores (*followers*) y sus amigos (*friends*)[1], que quedan almancenados en dos bases de datos Lucene.

Por un lado, tenemos una base de datos, a la que llamaremos *Census*, con toda la información detallada de cada uno de los usuarios de Twitter que hemos descargado.Por otro lado, tenemos otra base de datos distinta, a la que llamaremos *Graph*. Esta base de datos modela la relación entre dos de los usuarios descargados, y se corresponde con el modelo de información explicado previamente en el capítulo 2, apartado 2.1. En la tabla vemos que existen unos campos relacionados con el usuario origen (*source*), y otros que referencian al usuario destino (*target*). Los campos que empiezan por la letra *S* hacen referencia al usuario origen, y aquellos que empiezan por la letra *T* hacen referencia al usuario destino.

[1]No es objeto de este proyecto saber cómo descargar los tweets mediente su API, ni qué tweets se van a descargar.

Veamos los distintos campos de la tabla Census y el significado de cada uno de ellos (Tabla 3.1).

ATRIBUTO	SIGNIFICADO
user	Usuario de Twitter
fullname	Nombre completo del usuario
bio	Descripción del perfil del usuario (Biografía)
location	Localización del usuario
url	Dirección web del usuario
photo	Dirección web donde se encuentra la foto de perfil del usuario
followers	Usuarios que siguen el usuario
friends	Usuarios a quien sigue el usuario
lists	Identificador de las listas del usuario
searchuser	Usuario de Twitter formateado para una correcta búsqueda y recuperación desde Lucene (e.g. pase a minúsculas)
tags	Etiquetas de Social Census Twitter

Tabla 3.1: Listado de campos de la tabla Census

En la Figura 3.1 presentamos el ejemplo del perfil de Twitter de la ETSINF y, en dicho perfil, hacemos referencia a los campos que forman la tabla Census.

Figura 3.1: Perfil de Twitter de la ETSINF

A continación, tal y como hemos hecho con la tabla Census, veamos los distintos campos de la tabla Graph y el significado de cada uno de ellos (Tabla 3.2).

ATRIBUTO	SIGNIFICADO
suser	Usuario de Twitter que representa el origen de la relación
sfullname	Nombre completo del usuario origen
sbio	Descripción del perfil del usuario origen (Biografía)
slocation	Localización del usuario origen
surl	Dirección web del usuario origen
sphoto	Dirección web donde se encuentra la foto de perfil del usuario origen
sfollowers	Usuarios que siguen el usuario origen
sfriends	Usuarios a quien sigue el usuario origen
lsists	Identificador de las listas del usuario origen
ssearchuser	Usuario de Twitter (que representa el origen de la relación) formateado para una correcta búsqueda y recuperación desde Lucene (e.g. pase a minúsculas)
stags	Etiquetas de Social Census Twitter del usuario origen
tuser	Usuario de Twitter que representa el destino de la relación
tfullname	Nombre completo del usuario destino
tbio	Descripción del perfil del usuario destino (Biografía)
tlocation	Localización del usuario destino
turl	Dirección web del usuario destino
tphoto	Dirección web donde se encuentra la foto de perfil del usuario destino
tfollowers	Usuarios que siguen el usuario destino
tfriends	Usuarios a quien sigue el usuario destino
tlists	Identificador de las listas del usuario destino
tsearchuser	Usuario de Twitter (que representa el destino de la relación) formateado para una correcta búsqueda y recuperación desde Lucene (e.g. pase a minúsculas)
ttags	Etiquetas de Social Census Twitter del usuario destino

Tabla 3.2: Listado de campos de la tabla Graph

NOTA: Las tablas *Census* y *Graph* no están normalizadas. Por este motivo, los campos que tratan de modelar la relación entre dos usuarios aparecen duplicados en la tabla *Graph*. Hemos tomado esta decisión para optimizar los tiempos de respuesta cuando recuperamos el grafo en base a filtros.

3.2 Construcción del grafo

Después de explicar como queda almancenada la información obtenida de Twitter, describiremos el tratamiento que hacemos de la misma.

Para construir nuestro grafo de relaciones hemos definido la clase *Graph*. Tal y como puede verse en el código fuente 3.1, el grafo está formado por dos hashtable[2], una para almacenar los valores de los nodos, y otra para las relaciones entre dos nodos. El motivo de elegir un hashtable tanto para los nodos como para las relaciones ha sido su velocidad

[2] https://docs.oracle.com/javase/7/docs/api/java/util/Hashtable.html

de acceso a la hora de realizar búsquedas, puesto que hemos realizado una gran cantidad de operaciones de búsquedas sobre ellas y éstas tienen coste constante O(1).

```
public class Graph {
  public Hashtable<String,String> Nodes = new Hashtable<String,String>();
  public Hashtable<String,Integer> Relations = new Hashtable<String,Integer>();
}
```

Código fuente 3.1: Definición de la clase *Graph*

Las hashtable están formadas por un par <K,V>, donde K representa la clave de búsqueda sobre la estructura, y la V el valor asociado a dicha clave. En el caso de los nodos, tanto la clave como su valor se corresponderán con el nombre del usuario de Twitter. En el caso de las relaciones, la clave será un String con la siguiente estructura *sSource:::sTarget*, donde el *sSource* es el nombre del usuario de Twitter donde tiene origen la relación, y *sTarget* es el nombre del usuario destino. El valor almancenado para dicha clave es el número de apariciones de dicha relación en nuestro grafo. Por lo tanto, si en nuestro grafo aparecen 5 relaciones donde el nodo origen es "X" y el destino "Y", existiría un campo relación con el par {"X:::Y", 5}.

Una vez conocida la estructura que utilizaremos para nuestro grafo, veamos el procedimiento seguido para el montaje del mismo. En el código fuente 3.2 vemos el fragmento de código Java que hemos utilizado para ello.

Recorremos todas las relaciones existentes en la base de datos *Graph*. Obtenemos los valores de *suser* y *tuser*, y comprobamos si ya están dados de altas como nodos en nuestro grafo. En el caso de que no sea así, los creamos. Finalmente, creamos la clave de la relación con el valor *sSource:::sTarget* y comprobamos si ésta existe. En caso afirmativo, obtenemos el número de apariciones y la incrementamos en uno. En caso contrario, la insertaremos por primera vez añadiendo el valor uno. Este es el procedimiento seguido cuando no se utiliza ningún filtro.

```
    ir = GetGraphReader(project);
    for (int i=0; i<ir.maxDoc(); i++) {
        if (ir.isDeleted(i)) continue;
        Document doc = ir.document(i);

        String sSource = doc.getValues("suser")[0];
        String sTarget = doc.getValues("tuser")[0];

        if (!oGraph.Nodes.containsKey(sSource))
        oGraph.Nodes.put(sSource, sSource);

        if (!oGraph.Nodes.containsKey(sTarget))
        oGraph.Nodes.put(sTarget, sTarget);

        String key = sSource + ":::" + sTarget;
        int freq = 0;
        if (oGraph.Relations.containsKey(key)) {
            freq = oGraph.Relations.get(key);
        }

        oGraph.Relations.put(key, ++freq);

    }
```

Código fuente 3.2: Algoritmo para crear el grafo *Graph* (sin filtros)

Por el contrario, si se utiliza alguno de los filtros establecidos, como usuario, bio, localización o etiquetas (tags), el código utilizado es algo distinto (véase el código fuente A.1, apéndice A). En dicho caso, lo primero que hacemos es el filtrado de nodos que cumplen las condiciones seleccionadas de la base de datos *Census*. Una vez que tenemos dichos nodos, los almacenamos en un hashtable llamado *FilteredNodes*. A partir de aquí, el procedimiento seguido es prácticamente idéntico al anterior, salvo por un detalle que explicaremos a continuación.

El siguiente paso del proceso es recorrer toda la base de datos de relaciones *Graph*. Durante este recorrido, se crearán los nodos y la relación existente entre ambos, siempre y cuando, ambos nodos de la relación se encuentren dentro del hashtable *FilteredNodes*. Es decir, se crearán tanto los nodos como la relación existente entre ellos, siempre que ambos nodos cumplan la condición del filtro seleccionado. En la Figura 3.2 vemos un pequeño ejemplo de cómo se aplica este filtrado. Dado un grafo de 3 nodos con sus correspondientes aristas, supongamos que se le aplica un filtro. El resultado del filtro es que solamente el nodo A y B cumplen las condiciones seleccionadas. En tal caso, el grafo resultante es el que puede verse a la derecha de la imagen.

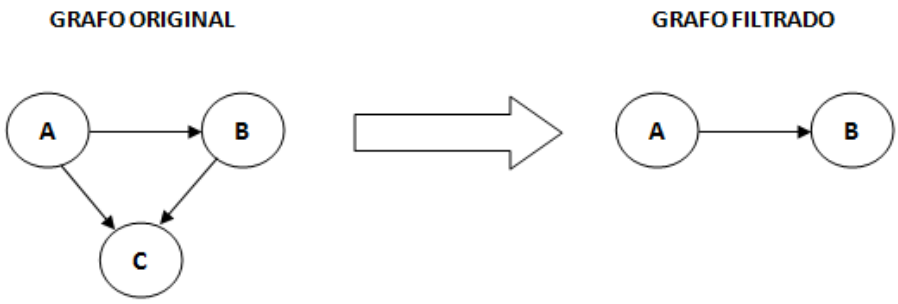

Figura 3.2: Ejemplo de filtrado de un grafo

Como último paso de esta fase de creación de nuestro grafo, exista o no filtrado, lo almacenaremos en un fichero con extensión *gexf*, y con la estructura adecuada que deben de tener este tipo de ficheros.

«GEXF (*Graph Exchange XML Format*) es un lenguaje para la definición de estructuras de redes complejas»[3]. Nacido en 2007 del proyecto Gephi ha ido evolucionando progresivamente hasta alcanzar un alto nivel de madurez. GEXF se ha convertido en un formato altamente extendido y abierto. Hemos almacenado nuestro grafo en este tipo de ficheros por dos motivos:

- Se pretendía modularizar parte del código para que éste no tuviera que ejecutarse siempre. De esta manera, se ha intentado que este código se ejecute el menor número de veces.

- Se quería descargar un poco la memoria de nuestro programa, al mismo tiempo que se dejaba en un formato adecuado para su posterior tratamiento antes de la visualización por pantalla.

[3] https://gephi.org/gexf/format/

3.3 Análisis y preproceso del grafo

Con nuestro grafo guardado en el fichero *gexf*, debemos realizar, a continuación, el preproceso del mismo mediante la librería Gephi. Lo que se pretende con este preproceso es reducir la dimensionalidad del grafo para que su visualización por pantalla resulte mucho más eficiente. Normalmente, los grafos obtenidos de redes sociales están compuestos por un elevadísimo número de nodos y, especialmente, de relaciones. Esto hace muy dificultosa su representación gráfica, tanto a nivel visual como a nivel computacional (puede llegar a provocar el colapso de la máquina si se consumen todos sus recursos físicos).

Inicialmente pretendíamos realizar nuestro preproceso sobre los siguientes aspectos de los grafos:

- *Eigen vector*. Se trata de una medida de centralidad que mide la influencia de un nodo sobre el grafo [9]. Podemos afirmar que los nodos que tienen un valor alto de eigen vector están conectados con muchos nodos, que también están bien conectados. Por lo tanto, consideramos que este tipo de nodos tienen mucha influencia. Por este motivo, el eigen vector es una de las medidas que utilizaremos para realizar filtros sobre nuestro grafo original. Además, nos permite ver fácilmente qué nodos son los más influyentes sobre las consultas realizadas. Esta medida irá asociada al tamaño del nodo del grafo, esto es, cuanto mayor sea éste, mayor eigen vector tendrá.

- *Modularidad*. Es una medida que permite analizar la estructura del grafo, es decir, que sirve para detectar grupos o comunidades dentro de un grafo. Esta medida se basa en la alta y fuerte conectividad de los nodos que se encuentran dentro de una misma comunidad, también en la baja interconexión entre nodos de distintas comunidades [10]. La representación de las distintas comunidades del grafo se hará mediante colores, siendo cada color de un nodo el indicador de pertenencia a una comunidad.

- *Componente gigante*. El componente gigante de un grafo es el componente conectado del grafo, que contiene una fracción fija de los vértices del grafo completo [11], es decir, son el conjunto de nodos conectados entre sí que engloban la mayoría del grafo. Con esta medida se intenta reducir la dimensionalidad eliminando aquellos nodos que se encuentran menos conectados.

Finalmente, nos decantamos únicamente por el cálculo del eigen vector y el cálculo de comunidades porque empíricamente hemos comprobado que el tipo de grafos de amistad con el que trabaja Social Census TW, está altamente relacionado y conectado, y es poco probable que se pudieran eliminar nodos mediante la utilización del componente gigante. Por lo tanto, para evitar ralentizar el sistema con cálculos innecesarios, hemos desestimado la utilización de este indicador que nos facilita Gephi.

¿Cómo realizamos el preproceso para mostrar los nodos y relaciones visibles? Veamos los pasos que hemos seguido:

- Leemos el fichero *gexf*, en el que previamente se ha almacenado el grafo.

- Inicializamos la librería de Gephi.

- Realizamos el cálculo del eigen vector y la modularidad con las librerías de Gephi.

- Ordenamos ascendentemente los valores del eigen vector para cada uno de los nodos obtenidos en el paso anterior.
- Obtenemos o calculamos el número de nodos visibles (*iNumVisibleNodes*). En el caso de que se haya facilitado el porcentaje en la interfaz, calcularemos los nodos visibles. En caso contrario, directamente ya tendremos su valor.
- Obtenemos el valor del eigen vector que se encuentra en la posición indicada por *iNumVisibleNodes*[4].
- Guardamos los nodos que tienen un valor de eigen vector mayor o igual al que hemos seleccionado en el paso anterior (filtrado de los nodos).
- Guardamos el número de nodos visibles.
- Guardamos las relaciones, en las que el nodo origen y destino se encuentren entre los seleccionados en el filtrado de nodos (filtrado de relaciones).
- Guardamos el número de relaciones visibles.
- Montamos el grafo (en formato json) con los nodos y relaciones filtrados.
- Devolvemos el grafo por la salida estándar, junto con el número de nodos y relaciones visibles.
- Mostramos en la interfaz gráfica los valores calculados y permitimos al usuario visualizar el grafo en pantalla.

[4]La variable *iNumVisibleNodes* no indica exactamente el número de nodos visibles, sino la posición del valor del eigen vector que utilizaremos como valor mínimo de corte para aplicar el filtro. Por ese motivo, y en el caso de que haya valores de eigen vector repetidos, es posible que el número de nodos visible sea mayor al indicado por el usuario en la interfaz.

Todos los pasos que acabamos de mencionar, salvo el último, se engloban dentro de la función *PrepareGraphJSON* de nuestro backend. En la Figura 3.3 mostramos un pequeño esquema de todo lo que engloba esta función.

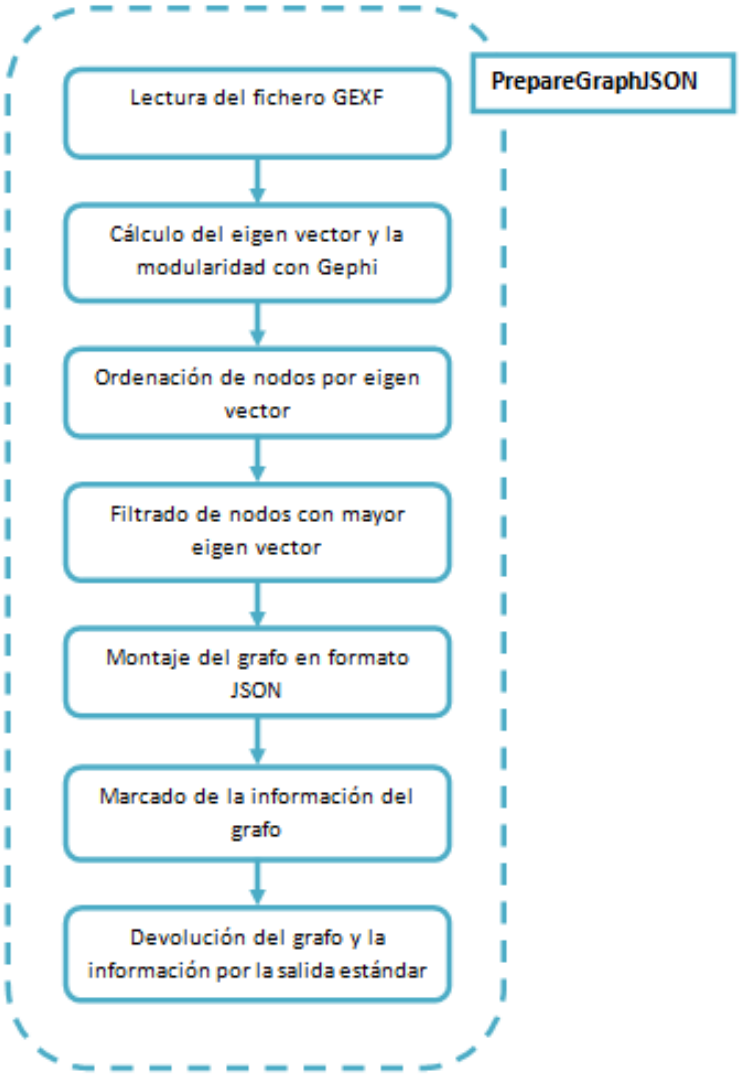

Figura 3.3: Esquema de la función PrepareGraphJSON

CAPÍTULO 4
Explotación y visualización del grafo

En este capítulo, describiremos detalladamente toda la parte gráfica de nuestra aplicación. Profundizaremos sobre cuestiones tales como:

- Lectura y marcado de la información del grafo: detalle del proceso realizado para extraer y marcar la información tras el preproceso de la misma.
- Visualización del grafo: descripción detallada del proceso seguido para la visualización de éste mediante vis.js
- Opciones de configuración de la red: detalle de la configuración específica de vis.js.
- Opciones de control de la interfaz: detalle de las distintas opciones implementadas en la interfaz y de cómo funcionan.

4.1 Lectura y marcado de la información del grafo

Tal y como explicamos en el apartado 3.2, la información obtenida de Twitter pasa necesariamente por una serie de procesos secuenciales previos a la visualización del grafo. Primero se obtiene de las dos bases de datos Lucene los datos necesarios para montar el grafo y guardarlo en un fichero *gexf*. Con este fichero, se realizan una serie de cálculos con la librería Gephi, y finalmente se devuelven unos datos en un formato concreto. Toda la información devuelta en este último paso se vuelca sobre la salida estándar, que es capturada desde nuestra capa de presentación.

Toda la información volcada sobre la salida estándar ha pasado por un proceso previo de marcado de la información. Es decir, todos y cada uno de los datos que se devuelven, se encuentran situados entre unos patrones de inicio y fin para poder localizarlos fácilmente.

En la tabla 4.1 presentamos tres columnas. La primera de ellas indica el dato que se obtiene. Las siguientes columnas indican los patrones de inicio y fin utilizados para identificar el dato de la primera columna. En el caso de que la columna esté vacía, significa que el delimitador es un salto de línea.

DATOS	PATRONES DE INICIO	PATRONES DE FIN
Grafo	::::	::::
Nodos totales	# Nodes loaded:	
Nodos visibles	# Nodos visibles:	
Relaciones totales	# Edges loaded:	
Relaciones visibles	# Relaciones visibles:	##_VISIBLE_EDGES_##
% Nodos visibles	# Porcentaje nodos visibles:	##_PERCENTAGE_VISIBLE_NODES_##

Tabla 4.1: Tabla de patrones de separación

¿Por qué capturamos la información de la salida estándar?

Porque durante la realización de los cálculos con la librería Gephi, una de las funciones a las que llamamos, siempre vuelca el número de nodos y de relaciones totales del grafo. Después, tras la realización del montaje del grafo, éste lo devolvíamos por la salida estándar. Así que cuando obteníamos el grafo en formato json, se producía un error.

Depurando la aplicación vimos que realmente no solo volvía nuestro grafo, sino también cualquier traza o información volcada en dicha salida. Por ese motivo nos vimos obligados a devolver el grafo entre un patrón de inicio y de fin para poder identificarlo, y desechar la información innecesaria. Aprovechando este hecho y, ante la necesidad de obtener más información desde nuestro backend, se ha intentado estandarizar la manera de recuperarla. Es decir, que todo dato que nos sea de utilidad para dibujar el grafo se vuelque en la salida estándar, delimitada con ciertos patrones. Así evitamos llamadas innecesarias a nuestra librería, y el proceso se realiza de manera más ágil.

4.2 Visualización del grafo

Con todos los datos necesarios a nuestra disposición, el siguiente paso es actualizar la pantalla con ellos. De esa forma, el usuario está en todo momento informado de qué usuarios y relaciones existen, los que se están visualizando, así como el porcentaje total de usuarios que podrá ver por pantalla.

En la Figura 4.1 se pueden observar los campos a los que hacemos referencia.

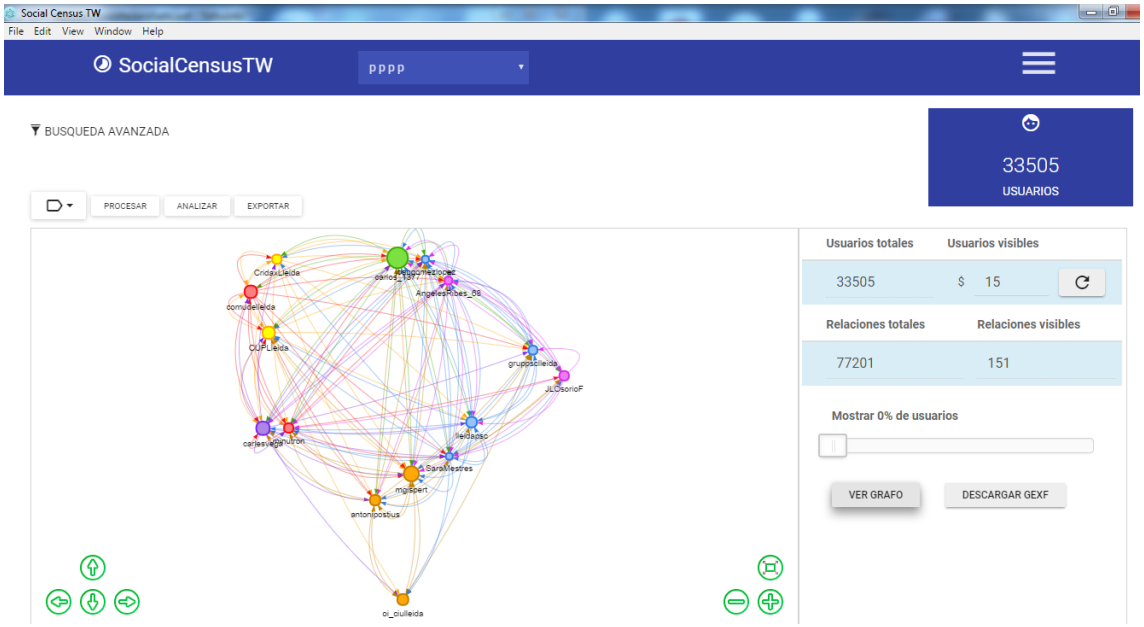

Figura 4.1: Opciones de visualización del grafo

Con los datos ya actualizados en pantalla, el usuario puedo solicitar ver el grafo en cualquier momento. Para ello, solamente deberá hacer clic sobre el botón *Ver grafo*. Este botón está programado para llamar a la función *DrawGraph()*. Esta es la función principal que contiene toda la lógica de dibujado de nuestro gráfico.

¿Qué pasos seguimos al utilizar la función *DrawGraph()* para dibujar nuestro grafo?

- Borrado del área de dibujado. Se comprueba si existe un grafo en pantalla, y de ser así se procede a su borrado.
- Transformación del grafo. Se pasa del formato de texto a un objeto JSON para su utilización en la librería vis.js.
- Creación de una nueva red (*Network*) de vis.js.
- Inicialización de los *DataSets* de vis.js con los datos del grafo.
- Inicialización de las opciones de visualización de vis.js.
- Visualización del grafo por pantalla mediante vis.js.

4.3 Opciones de configuración de la red

La inicialización de las distintas opciones existentes para la visualización de nuestro grafo requiere una especial atención. Estas opciones son muchas y muy variadas. Por ejemplo, nos permite modificar la configuración de la red y el aspecto de los nodos y las aristas; manipular e interactuar con los elementos de la propia red; modificar el tamaño del área de dibujado, etc. Sin embargo, no creemos conveniente detallar cada una de ellas. Si queremos conocer todas las posibilidades existentes, recomendamos consultar la página web de vis.js[1], en la que se explica cada una de estas opciones.

Las opciones por defecto utilizadas para configurar nuestra red son las siguientes:

- *Layout*: es el elemento que actúa como cámara de visualización sobre el área de dibujado (*canvas*) del grafo. Nos permite acercarnos o alejarnos del área de dibujo, según nuestras necesidades. Se ha desactivado la opción de *improvedLayout* porque no está recomendada para redes de más de 100 nodos. Y este el caso de la mayoría de nuestros grafos.
- *Nodes*: permite definir el aspecto y comportamiento de los nodos. Hemos modificado el aspecto por defecto de los nodos, indicando que sean redondeados, con un tamaño mínimo, un determinado grosor de los bordes y una fuente concreta.
- *Edges*: permite definir el aspecto y comportamiento de las aristas. Hemos indicado el valor por defecto del ancho.
- *Physics*: indica cómo ha de comportarse la red, a la hora de simular los movimientos de la misma y de qué manera han de distribuirse los nodos y aristas en el espacio. Hemos seleccionado la distribución *forceAtlas2Based*, ajustando el resto de parámetros asociados para que la visualización de nuestro grafo quede lo más clara posible.
- *Interaction*: permite definir la interacción con la red. Nuestro grafo permite moverse dentro del área de dibujado mediante la utilización de los botones que aparecen en la parte inferior de dicha área. Además de eso, cuando se mantiene pulsado el botón derecho del ratón sobre la zona de dibujado, desaparecen las relaciones, y quedan visibles solamente los nodos. De esa manera, se pueden ver fácilmente el número de total de nodos y las comunidades a las que pertenecen.

En el Apéndice B.1 pueden consultarse las opciones de configuración que hemos utilizado.

[1] http://visjs.org/docs/network/#options

4.4 Opciones de control de la interfaz

En este apartado, vamos a analizar el funcionamiento de todas las partes de la interfaz de usuario que hemos desarrollado. Para poder seguir un orden concreto, nos basaremos en la Figura 4.2, en la que la interfaz de usuario se ha dividido en partes, con el objetivo de facilitar la identificación de cada una de ellas.

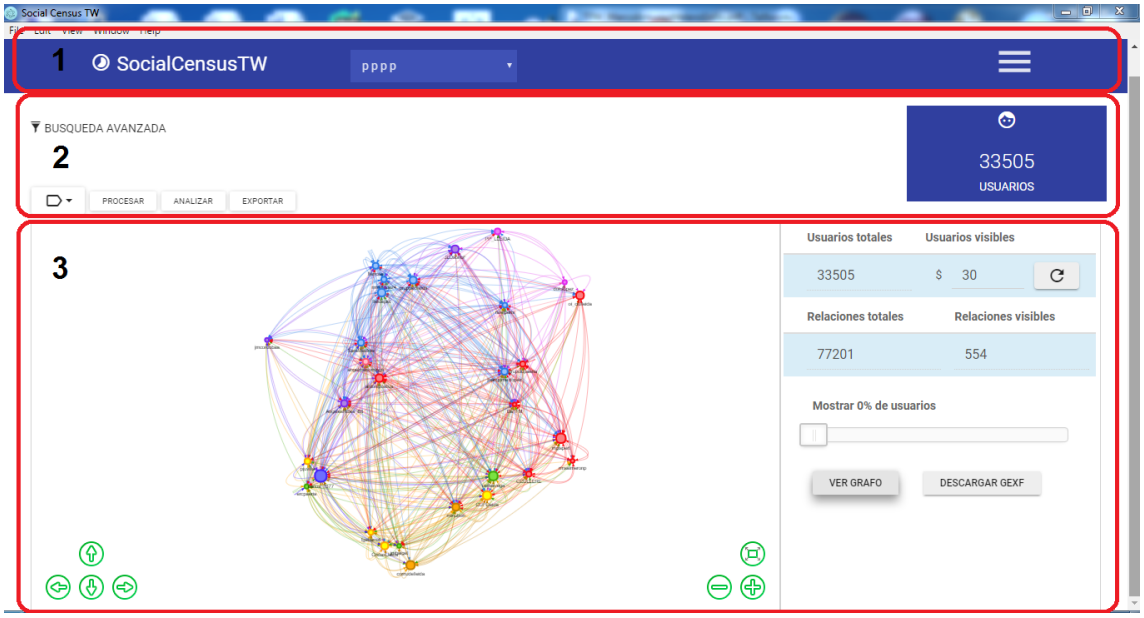

Figura 4.2: Interfaz de usuario

A continuación, vamos a desglosar los elementos que componen la interfaz de usuario.

1. *Cabecera*: es un área común en casi toda la aplicación *Social Census Twitter* (dentro de la que se engloba nuestra interfaz). En ella, además de ver el nombre de la aplicación *SocialCensusTW*, que es un enlace a la página de inicio, podemos ver el proyecto que actualmente está seleccionado, y un icono de unas líneas paralelas en la parte superior derecha. Estas líneas son un icono para poder desplegar un menú lateral y navegar por otras secciones del aplicativo (véase la Figura 4.3).

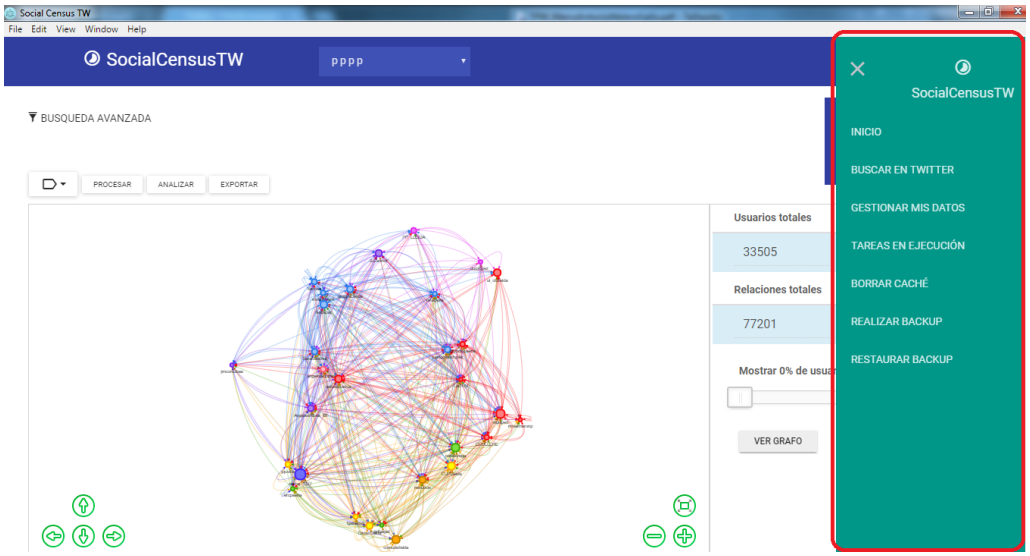

Figura 4.3: Menú lateral

2. *Búsqueda avanzada*: muestra una serie de filtros que permiten la selección de información más específica sobre nuestro grafo. Concretamente, permite el filtrado de etiquetas relacionadas con el usuario, la biografía, localización y etiquetas varias (véase la Figura 4.4 donde aparece desplegado el filtro con las distintas opciones existentes).

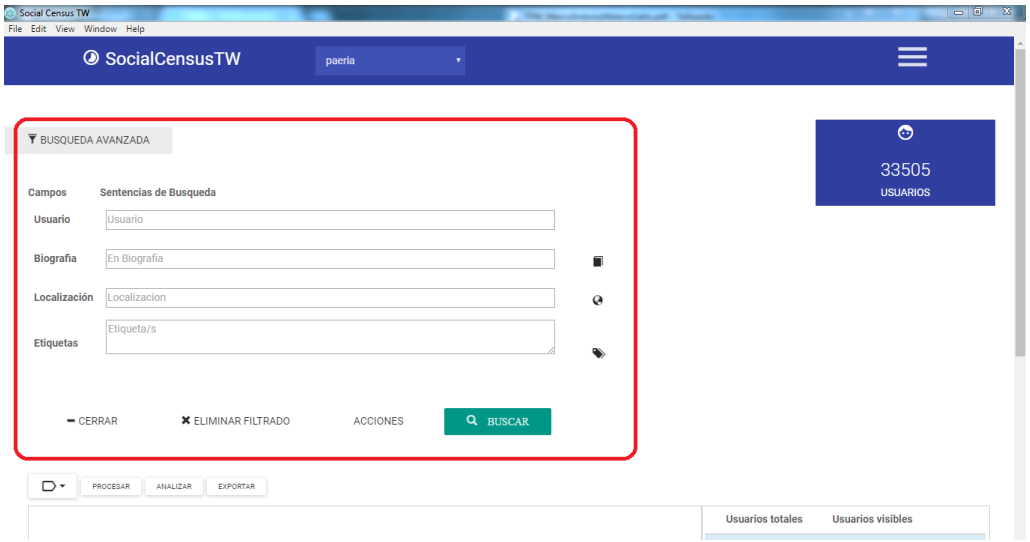

Figura 4.4: Búsqueda avanzada

El aspecto visual de esta búsqueda es el mismo que el que aparece en otras partes del aplicativo. Sin embargo, el funcionamiento interno es distinto. Al visualizar nuestro grafo veremos la relación existente entre un usuario origen y un usuario destino. Cuando aplicamos el filtro desde esta búsqueda avanzada, ambos usuarios (origen y destino) cumplirán las condiciones seleccionadas en el filtro. En el caso de que solamente uno de los usuarios cumpla la condición, no se mostrará.

Aplicamos el filtro porque inicialmente queríamos saber exactamente qué nodos cumplen las condiciones seleccionadas. En trabajos futuros pueden hacerse versiones parametrizables, en las que se indique cómo debe de actuar el filtro, es decir, si ambos usuarios deben cumplir la condición, o si solamente debe cumplirla uno de los usuarios, esto es, origen o destino.

En esta sección de búsqueda avanzada también aparece, en la parte derecha, el número de de usuarios que se encuentran seleccionados en ese momento. Su valor cambiará siempre que cambie el filtro.

Los últimos elementos que se pueden ver en esta sección son los siguientes botones: *Procesar*, *Analizar* y *Exportar*. Estos botones son desplegables y nos guían hacia otras secciones del aplicativo *SocialCensusTW*. Se encuentran en esa zona para mantener la homogeneidad del diseño con otras partes de la aplicación.

3. *Zona central*: está compuesta por dos zonas diferenciadas: una más amplia situada a la izquierda, y otra un poco más estrecha, situada a la derecha. La parte de la izquierda muestra la sección en la que se visualizará el grafo. Inicialmente está vacía, a la espera de que el usuario indique que se dibuje el grafo seleccionado.

   En la parte derecha aparece una sección de control. En esta sección de control se muestra información relativa al grafo que queremos visualizar. Concretamente podemos ver lo siguiente: el número de usuarios totales; el número de usuarios visibles; las relaciones totales; las relaciones visibles; una barra de deslizamiento que indica el porcentaje de nodos que se se están visualizando; y dos botones, uno para ver el grafo, y otro para descargar el fichero *gexf*.

   Inicialmente todos los campos que muestran número de usuarios y relaciones vienen precargados con el valor cero, salvo el número total de usuarios que será el mismo que el que se muestra en la sección de la *Búsqueda avanzada*. El porcentaje de nodos visibles también será del 0 %. Esta inicialización de valores se hace para obligar al usuario a seleccionar un número de usuarios visibles o un determinado porcentaje. De esa manera, inicialmente no se realiza ningún cálculo y la carga de la pantalla es mucho más rápida.

   Los únicos valores del grafo que pueden ser modificados por el usuario son el número de usuarios visibles o el porcentaje. Al cambiar el valor del número de usuarios visibles y pulsar sobre el botón de recarga que se encuentra a su derecha, el resto de valores, incluido el porcentaje, se recalcularán automáticamente en función del número introducido. Sucederá lo mismo cuando se cambie este valor desde la barra de desplazamiento, pero manteniendo el valor del porcentaje seleccionado. Al producirse el recálculo de los valores visibles, lo que se hace internamente es relanzar el preproceso del grafo tal y como se explico en el Capítulo 3.3.

   Finalmente, describiremos el funcionamiento de los botones *Ver grafo* y *Descargar GEXF*. El primero de ellos, como su propio nombre indica, se utiliza para ver el grafo seleccionado, situado en la parte izquierda de la zona central. Esta visualización es la que se realiza mediante la librería vis.js, de la que ya hemos hablado en otros capítulos. El botón *Descargar GEXF* nos permite descargar el fichero del grafo en formato GEXF en nuestro dispositivo. Así, podemos utilizar el grafo seleccionado en otras herramientas ajenas a este aplicativo.

CAPÍTULO 5
Casos de uso

En este capítulo vamos a ver tres ejemplos de utilización de la interfaz desarrollada, así como los beneficios que su uso nos comporta. En los distintos ejemplos, tanto en los que se ven a continuación, como en los que aparecen en otros capítulos, se ha utilizado un grafo que muestra un conjunto de *influencers* de política en Lleida.

En el primero de ellos, vemos cómo se ha realizado un filtrado a través del número de usuarios visibles. Como podemos apreciar en la Figura 5.1, solamente se están mostrando los 20 usuarios con mayor eigen vector de un total de 33505, y 265 relaciones de más de 77 mil exitentes. Esto demuestra el potencial de la herramienta desarrollada ya que, en una sola imagen y en pocos segundos, logramos ver los usuarios más influyentes de nuestro grafo (aquellos que tienen un mayor tamaño) y la relación existente entre ellos. Además, podemos ver los usuarios de las diferentes comunidades con colores distintos. Recordemos que esto no sería posible sin la realización del preproceso de la información que se explicó en el capítulo 3, apartado 3.3.

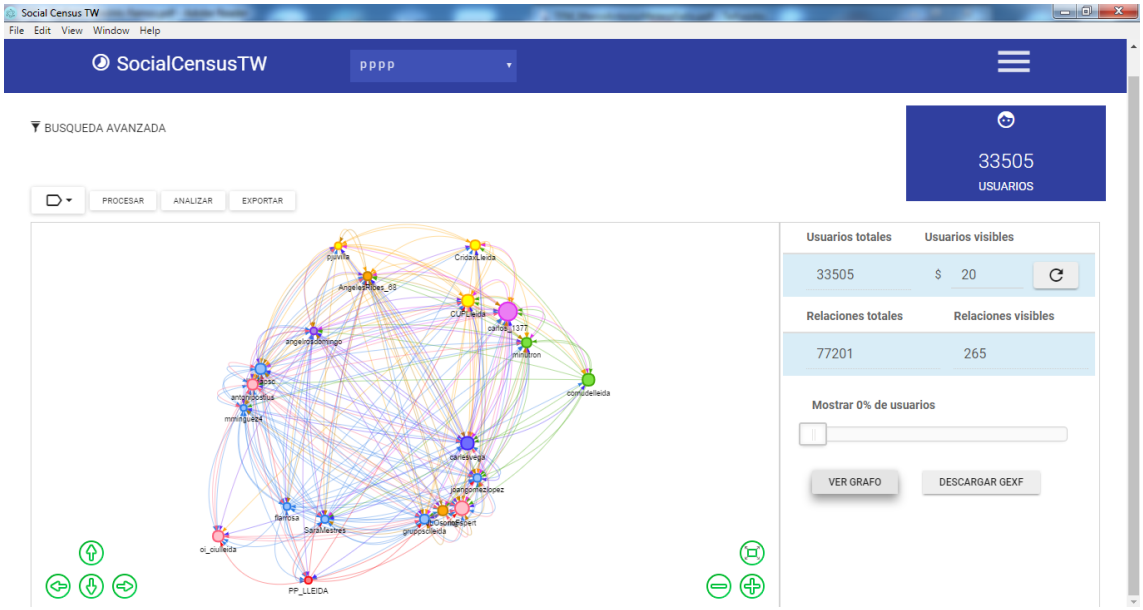

Figura 5.1: Visualización de un grafo con 20 nodos visibles

En la Figura 5.1 no se aprecian las diferentes comunidades existentes porque hay muy pocos usuarios visibles. Sin embargo, en este segundo ejemplo esto se puede ver con mayor claridad en la Figura 5.2, en la que se observan varias comunidades con sus principales *influencers*. Como en el ejemplo anterior, la respuesta visual es de apenas unos segundos, pero la información que nos aporta la imagen es de gran valor para nosotros,

ya que nos permite detectar quién influye sobre el resto de la comunidad en la que se ha aplicado el filtro.

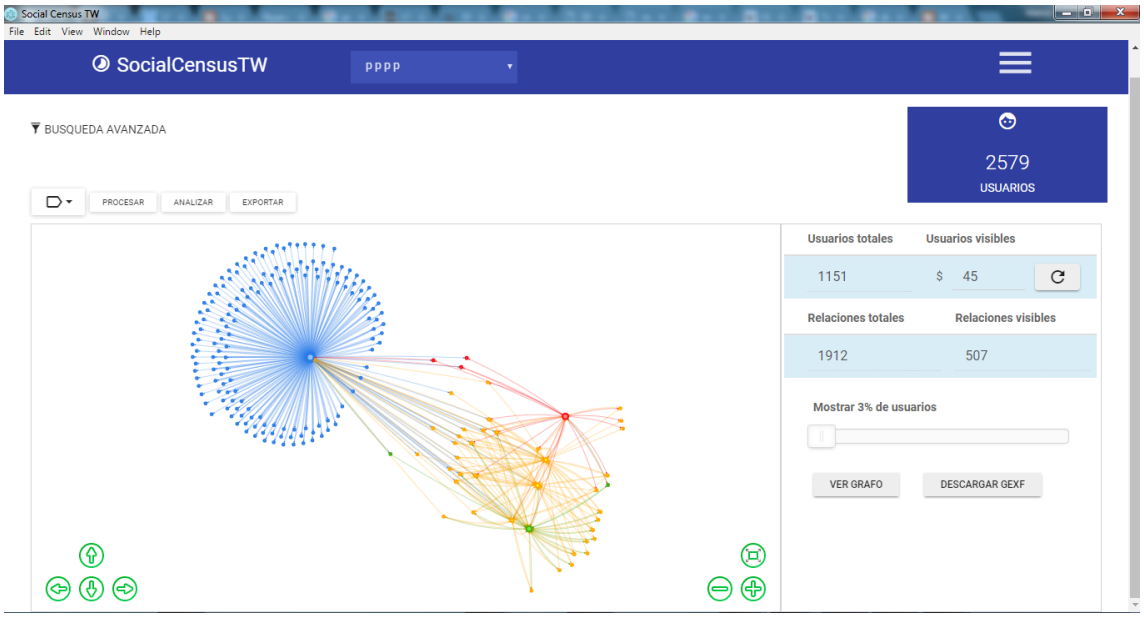

Figura 5.2: Visualización de un grafo donde se aprecian distintas comunidades

Finalmente, en nuestro tercer ejemplo no se ha realizado ningún tipo de filtrado, haciendo que se dibuje el grafo completo que se había descargado. Observando la Figura 5.3 llegamos a la conclusión de que si no permitiéramos el filtrado del mismo a través del eigen vector, sería prácticamente inviable su visualización, y no se podría trabajar ni obtener información. Es más, aunque la aplicación ha permitido la visualización del grafo completo, el tiempo de respuesta ha sido extremadamente lento. De ahí la importancia del filtrado por porcentaje o número de usuarios visibles.

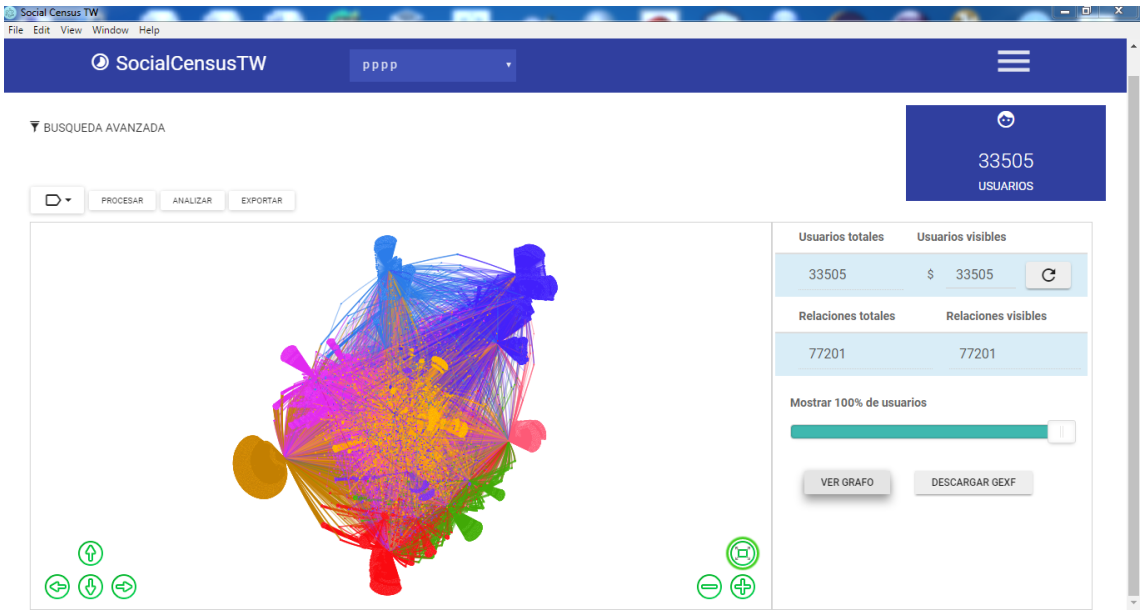

Figura 5.3: Visualización del grafo completo

En el siguiente vídeo se muestra un completo caso de uso: Ver vídeo

CAPÍTULO 6
Conclusiones y trabajos futuros

Hemos desarrollado una aplicación de escritorio que permite, de una manera visual, el análisis de las relaciones entre usuarios de Twitter. El análisis de dichas relaciones ha requerido de mucho esfuerzo para aunar un gran número de tecnologías distintas.

El hecho de programar con un gran volumen de datos exige un conocimiento detallado de las herramientas utilizadas, ya que hay que intentar obtener en todo momento el mayor rendimiento de los recursos que tenemos a nuestra disposición. Una mala gestión de los recursos nos llevaría fácilmente al fracaso de nuestro proyecto por no saber cómo solventar este tipo de problemas. Elegir las estructuras de datos adecuadas, gestionar los recursos eficientemente y tratar de coordinarlo todo no es una tarea fácil en este tipo de proyectos.

A pesar de las dificultades encontradas, hemos logrado alcanzar los objetivos incialmente planteados. A partir de aquí, tomando nuestra aplicación como punto de partida, se abre ante nosotros un gran abanico de posibilidades para mejorar la aplicación y hacerla mucho más interesante.

Algunas de las mejoras que podrían incluirse son las siguientes:

- Optimizar el rendimiento de la aplicación cuando hay un número de nodos elevados (como es habitual en nuestros casos). Aunque el rendimiento es aceptable, y hemos conseguido alcanzar los objetivos previstos, cuando se seleccionan todos los nodos del grafo aumenta el tiempo de respuesta.

- Añadir una pantalla que nos permita parametrizar más elementos de la interfaz, como por el ejemplo, el tamaño de los usuarios más influyentes, los parámetros utilizados, y otra serie de opciones de la herramienta vis.js para modificar el aspecto del grafo representado. También podría permitinos elegir los colores de las distintas comunidades.

- Poder decidir si los nodos que cumplen las condiciones de la búsqueda avanzada deben ser el origen, el destino o ambos. Ahora mismo se ha forzado a que ambos extremos cumplan la condición seleccionada.

- Realizar el filtrado avanzado de los usuarios de una manera mixta, es decir, si el número de usuarios a seleccionar es mayor de una determinada cantidad, utilizaríamos Lucene porque es necesaria una consulta a la base de datos. Si por el contrario es menor de esa cantidad, utilizaríamos los hashmaps.

- Añadir en la pantalla de opciones un check que permita activar o no la frecuencia de aparición en las relaciones. De esa manera, si este check estuviera activado, la

línea que une dos nodos sería más grande cuanta mayor frecuencia de aparición tuviera.

- Seleccionar los usuarios en pantalla para ver los datos de su ficha en Twitter.
- Configurar la barra de desplazamiento (*slider*). Cuando seleccionemos sobre el *slider* el porcentaje de usuarios a mostrar, si excede de un determinado número, que habremos identificado previamente, y vemos que la carga va a ralentizar la aplicación, sugerimos que aparezca una pantalla de confirmación. Esta pantalla preguntará si desea realizar el cálculo, ya que esa operación puede tardar varios minutos, debido al volumen de datos del mismo. En caso afirmativo, funcionaría como hasta ahora, y en caso contrario, dejaría la barra de desplazamiento tal y como estaba originalmente, sin hacer ningún cambio.

Social Census TW es una aplicación propietaria de Autoritas. El desarrollo aquí presentado se encuentra en su versión beta y está siendo probado con clientes reales.

Bibliografía

[1] Llinares Pla, M. *Sapere Aude - Metodología para la Generación de Inteligencial*. Máster Interuniversitario en Analista de Inteligencia. Universidad Rey Juan Carlos - Universidad Carlos III de Madrid (2015). Dirección: Francisco Rangel y Cristina Lucas.

[2] Fernández, S. *Definición metodológica para la generación de inteligencia relacional*. Diploma de especialización en Big Data. Universidad Politécnica de Valencia (2015)

[3] Ramos Pérez, A. *Inteligencia Social de Negocio con Neo4j*. Diploma de especialización en Big Data. Universidad Politécnica de Valencia (2015)

[4] Gospodnetic, O., Hatcher, E. *Lucene in Action*. Manning Publications, second edition (2004).

[5] Kinney, S. *Electron in Action*. Manning Publications, (2016).

[6] Crowther, R., Lennon, J., Blue, A., Wanish, G. *HTML5 in Action*. Manning Publications, (2014).

[7] Spurlock, J. *Bootstrap: Responsive Web Development*. O'Really, first edition (2013).

[8] Bibeault, B., Katz, Y., De Rosa, A. *jQuery in Action*. Manning Publications, third edition (2015).

[9] Bonacich, P. Factoring and weighting approaches to status scores and clique identification. *Journal of Mathematical Sociology* (1972)

[10] Blondel, V. Guillaume, J.L. Lambiotte, R. Lefebvre, E. Fast Unfolding of Communities in Large Networks. *Journal of Statistical Mechanics: Theory and Experiment* (2008)

[11] Charu G. A. *Social Network Data Analytics*. Springer (2014).

APÉNDICE A
Creación del grafo

```
ir = DataMngr.GetDataReader(project);
is = new IndexSearcher(ir);

is.setDefaultFieldSortScoring(true, false);
BooleanQuery q = new BooleanQuery();

QueryParser qp =  null;
if(!user.isEmpty()){
      qp = new QueryParser(Version.LUCENE_30, "user", new StandardAnalyzer(
          Version.LUCENE_30));
      q.add((org.apache.lucene.search.Query) qp.parse(user), BooleanClause.
          Occur.MUST);
}
if(!bio.isEmpty()){
      qp = new QueryParser(Version.LUCENE_30, "bio", new StandardAnalyzer(
          Version.LUCENE_30));
      q.add((org.apache.lucene.search.Query) qp.parse(bio), BooleanClause.
          Occur.MUST);
}
if(!location.isEmpty()){
      qp = new QueryParser(Version.LUCENE_30, "location", new
          StandardAnalyzer(Version.LUCENE_30));
      q.add((org.apache.lucene.search.Query) qp.parse(location),
          BooleanClause.Occur.MUST);
}
if(!tags.isEmpty()){
      qp = new QueryParser(Version.LUCENE_30, "tags", new StandardAnalyzer(
          Version.LUCENE_30));
      q.add((org.apache.lucene.search.Query) qp.parse(tags), BooleanClause.
          Occur.MUST);
}

int iNDocs = is.search(q, 1).totalHits;

if (iNDocs>0) {

    TopDocs hits = is.search(q, iNDocs);
    int Total = hits.scoreDocs.length;
    int freq = 0;

    Hashtable<String, Integer> FilteredNodes = new Hashtable<String,
        Integer>();

    for (int i=0;i<Total;i++) {

        int docId = hits.scoreDocs[i].doc;
        Document doc = is.doc(docId);
```

```
            freq = 0;
            String key = doc.getValues("user")[0];

            if (FilteredNodes.containsKey(key)) {
                freq = FilteredNodes.get(key);
            }
            FilteredNodes.put(key, ++freq);
        }

        irGraph = GetGraphReader(project);
        for (int i=0; i<irGraph.maxDoc(); i++) {

            if (irGraph.isDeleted(i)) continue;
            Document docGraph = irGraph.document(i);

            String sSource = docGraph.getValues("suser")[0];
            String sTarget = docGraph.getValues("tuser")[0];

            if (FilteredNodes.containsKey(sSource) &&
                FilteredNodes.containsKey(sTarget) ) {

                if (!oGraph.Nodes.containsKey(sSource))
                    oGraph.Nodes.put(sSource, sSource);

                if (!oGraph.Nodes.containsKey(sTarget))
                    oGraph.Nodes.put(sTarget, sTarget);

                String key = sSource + ":::" + sTarget;
                freq = 0;
                if (oGraph.Relations.containsKey(key)) {
                    freq = oGraph.Relations.get(key);
                }
                oGraph.Relations.put(key, ++freq);

            }
        }
    }
```

Código fuente A.1: Algoritmo para crear el grafo *Graph* (con filtros)

APÉNDICE B
Opciones de visualización de vis.js

```
var options = {
  layout: {
    improvedLayout: false
  },
  nodes: {
      shape: 'dot',
      size: 30,
      font: {
          size: 24,
          color: '#000'
      },
      borderWidth: 2
  },
  edges: {
      width: 1
  },
  physics: {
      forceAtlas2Based: {
          gravitationalConstant: -26,
          centralGravity: 0,
          springLength: 300,
          springConstant: 0.1
      },
      maxVelocity: 146,
      solver: 'forceAtlas2Based',
      timestep: 0.35,
      stabilization: {
          enabled:true,
          iterations:1000,
          updateInterval:25
      },
  },
  interaction: {
      tooltipDelay: 200,
      hideEdgesOnDrag: true,
      navigationButtons: true,
      keyboard: true
  }
};
```

Código fuente B.1: Opciones de visualización de grafo