Proyecto Fin de Carrera

DISEÑO E IMPLEMENTACIÓN DE UNA HERRAMIENTA PARA LA COMPARACIÓN DE ALGORITMOS DE DETECCIÓN DE COMUNIDADES EN GRAFOS

Design and Implementation of a Tool for the Comparison of Community Detection Algorithms in Graphs

Para acceder al título de

INGENIERO EN INFORMÁTICA

Camilo Palazuelos Calderón

Julio de 2012

UNIVERSIDAD DE CANTABRIA

INGENIERÍA EN INFORMÁTICA

CALIFICACIÓN DEL PROYECTO FIN DE CARRERA

Realizado por: Camilo Palazuelos Calderón
Director del PFC: Marta Elena Zorrilla Pantaleón
Título: Diseño e implementación de una herramienta para la comparación de algoritmos de detección de comunidades en grafos
Title: Design and Implementation of a Tool for the Comparison of Community Detection Algorithms in Graphs

Presentado a examen el día:

para acceder al título de

INGENIERO EN INFORMÁTICA

Composición del Tribunal

Presidente: González Harbour, Michael
Secretario: Martínez Fernández, María del Carmen
Vocal: Menéndez de Llano Rozas, Rafael
Vocal: Zorrilla Pantaleón, Marta Elena
Vocal: Sanz Gil, Roberto

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente

Fdo.: El Secretario

Fdo.: Vocal

Fdo.: Vocal

Fdo.: Vocal

Fdo.: El Director del PFC

A mi abuelo, Albert Calderón,
por enseñarme a amar la informática

Agradecimientos

Ahora que tocan a su fin, estos últimos cinco años se me antojan como un largo viaje en tren. Muchos han sido los paisajes contemplados desde mi asiento, algunos tan bellos que consiguieron mantener mi cara pegada al cristal por un instante o dos; muchas las personas que han ido subiendo en una estación u otra, algunas tan maravillosas que desearía no perderlas nunca por el camino; y muchos los momentos vividos junto a ellas, algunos tan inolvidables que no quiero finalizar estas líneas sin hacerles llegar mi gratitud.

En primer lugar, me gustaría darle las gracias a la Dra. Marta Elena Zorrilla Pantaleón que, subiendo a este tren hace dos años, me mostró uno de esos paisajes que consiguieron emocionarme: la investigación. Gracias por tu cercanía, honestidad, profesionalidad y rigurosidad, así como por permitirme llegar hasta aquí.

En segundo lugar, quiero expresar mi afecto hacia mi familia, porque sin ellos ni siquiera habría tomado este tren. Gracias de todo corazón a mi madre, mi abuela, mi abuelo, mi hermana y mi hermano, que hacen mágica mi vida.

Por último, y más importante, mis gracias más sinceras a Francisco Calatayud, por razones que me permitirían llenar las páginas de un libro. El siguiente tren es sólo para nosotros.

Resumen

En la última década, la aparición de servicios como Facebook o Twitter ha dado como resultado un renovado interés en el análisis de redes sociales, siendo la detección de comunidades uno de los principales problemas que se han abordado. La detección de comunidades consiste en organizar los vértices de un grafo en grupos densamente conectados entre sí. A pesar de que se han propuesto decenas de algoritmos y varios generadores de grafos para comprobar su eficacia, la prueba de los mismos no ha recibido gran atención en la literatura: ésta suele limitarse a la aplicación del algoritmo propuesto a un conjunto de grafos cuya estructura es conocida de antemano o a la selección de los parámetros de un generador de grafos que permitan obtener redes estructuralmente sencillas. Esto supone un gran problema ya que no se puede afirmar qué método es mejor, por lo que, en la práctica, la elección del algoritmo a usar vendrá determinada por factores que nada tienen que ver con su eficiencia (por ejemplo, su popularidad o la reputación de su autor).

Por ello, este Proyecto Fin de Carrera ha diseñado e implementado una aplicación que permite comparar algoritmos de detección de comunidades de manera imparcial. Ésta se ha diseñado de tal manera que los usuarios pueden añadirle algoritmos, generadores de grafos y medidas de evaluación de resultados, para lo cual se ha hecho uso de un lenguaje multiplataforma de propósito general, como Java. La aplicación obtiene los grafos a partir de los generadores suministrados y se los envía a los algoritmos para que éstos le devuelvan la estructura de comunidades detectada. Así, con las medidas de evaluación oportunas, puede determinar qué algoritmo se comporta mejor. Asimismo, se ha implementado implementar un mecanismo para que la ejecución del código de los componentes suministrados sea segura, de manera que un usuario malintencionado no pueda ejecutar código que sea capaz de afectar a la seguridad de la máquina. El diseño e implementación de esta aplicación se han llevado a cabo siguiendo la metodología MÉTRICA en desarrollo orientado a objetos.

Abstract

In the last decade, the appearance of services such as Facebook or Twitter has allowed for a renewed interest in social network analysis, being community detection one of the main problems tackled. Community detection consists in organizing the vertices of a graph in groups that permit them to be densely connected between each other. Despite the fact that many different algorithms and graph generators to test the efficiency of those algorithms have been proposed, that testing has not been duly treated in the literature: it usually is limited to the application of the proposed algorithm on a set of graphs whose structure is known in advance or the selection of the parameters of a graph generator that permit obtaining structurally simple networks. This becomes a great problem due to the fact that it cannot be ascertained what method is best, thus in practice choosing what algorithm to use will become conditioned by factors that have nothing to do with its efficiency (e.g., its popularity or the reputation of its author).

For this reason, this Final Degree Project designed and implemented an application that permits comparing community detection algorithms in graphs in an impartial fashion. It was developed in such a manner that users can add algorithms, graph generators and measures for comparing results, for which a multipurpose and multiplatform programming language was used, in this case Java. The application obtains graphs from the generators included in it and sends them to the algorithms for these to return the community structure detected. Thus, with the appropriate measures, it is able to determine what algorithm does best. Also, a mechanism was implemented so as to ensure that the execution of these modules is safe, so that a malicious user cannot execute code that potentially puts the machine security at risk. The design and implementation of this application were done using the MÉTRICA methodology for object-oriented developments.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Este capítulo expone el contexto en que se enmarca el presente Proyecto Fin de Carrera introduciendo el área de investigación de la detección de comunidades en grafos y presentando las carencias detectadas en dicha área que motivaron su desarrollo. Asimismo, incluye una breve descripción de los objetivos a lograr, así como una sucinta introducción a la metodología seguida para alcanzarlos.

1.1. Antecedentes y motivación

En la última década, la aparición de servicios como Facebook o Twitter ha dado como resultado un renovado interés en el análisis de redes sociales, siendo la detección de comunidades uno de los principales problemas que se han abordado. La detección de comunidades tiene como objetivo organizar los vértices de un grafo en grupos densamente conectados entre sí (Girvan y Newman, 2002). En la práctica, los vértices de una comunidad comparten características comunes o tienen un rol similar dentro del conjunto de vértices del grafo, lo que permite descubrir el tipo de relaciones que los mantienen unidos y entender su comportamiento. Actualmente, es un área de investigación muy activa y, aunque tiene su origen en sociología (Rice, 1927), la detección de comunidades ha sido, y continúa siendo, muy útil en otras disciplinas, como biología (Chen y Yuan, 2006; Palla et al., 2005) o computación (Newman, 2004a; Rosvall y Bergstrom, 2008).

A pesar de que se han propuesto decenas de algoritmos y varios generadores de grafos para comprobar su eficacia (para un resumen completo véase Fortunato, 2010), la prueba de los mismos no ha recibido gran atención en la literatura: ésta suele limitarse a la aplicación del algoritmo propuesto a un conjunto de grafos cuya estructura es conocida de antemano o a la selección de los parámetros adecuados de un generador de grafos que permitan obtener redes[1] estructuralmente sencillas. Esta falta de rigurosidad explica la abundante aparición de técnicas de detección de comunidades en los últimos años. De la misma manera, desde el primer análisis comparativo, llevado a cabo por Danon et al. (2005), apenas han surgido trabajos con comparativas empíricas formales de algoritmos de detección de comunidades (Fan et al., 2007; Lancichinetti y Fortunato, 2009b; Sawardecker et al., 2009). Esto supone un gran problema ya que, dado un conjunto arbitrario de algoritmos, no se puede afirmar qué método es mejor, por lo que, en la práctica, la elección del algoritmo a usar vendrá determinada por factores que nada tienen que ver con su eficiencia (por ejemplo, su popularidad o la reputación de su autor).

[1] En este documento, los términos *grafo* y *red* se usan indistintamente.

1.2. Objetivos

A la luz de las limitaciones expuestas en la Sección 1.1, este Proyecto Fin de Carrera propone el diseño e implementación de una aplicación, Graphite, que permita llevar a cabo comparaciones empíricas de algoritmos de detección de comunidades de manera imparcial y exhaustiva. Para ello, deberá diseñarse de tal manera que los usuarios puedan añadirle, mediante un mecanismo de extensiones o *plug-ins*, tres tipos de implementaciones: (i) algoritmos, (ii) generadores de grafos, y (iii) medidas de evaluación para comparar los resultados obtenidos por los algoritmos con los indicados por los generadores de grafos. Asimismo, se le podrán proporcionar grafos individuales escritos en ficheros con formato DGML[2].

Dada la naturaleza extensible de la aplicación, así como la heterogeneidad de los sistemas operativos con que sus usuarios se desenvuelven, el lenguaje de programación seleccionado para desarrollarla e implementar sus extensiones será un lenguaje multiplataforma de propósito general, como Java. Asimismo, se deberá desarrollar un mecanismo de seguridad que permita ejecutar el código de las extensiones suministradas por los usuarios de forma segura. Para ello, dichas extensiones se ejecutarán en un entorno seguro o *sandbox* en que los recursos y privilegios a su disposición serán muy reducidos. De esta manera, se evita que un usuario malintencionado sea capaz de ejecutar código malicioso que pueda comprometer la integridad de la máquina.

El usuario indicará a la aplicación el conjunto inicial de grafos a utilizar proporcionándoselos de manera explícita a través de ficheros con formato DGML, o bien solicitando su creación a los generadores de grafos suministrados. La aplicación enviará dicho conjunto de grafos a los algoritmos que el usuario desee comparar para que cada uno de ellos le devuelva la estructura de comunidades obtenida para cada grafo del conjunto. Por último, con las medidas de evaluación que se hayan seleccionado, Graphite será capaz de determinar cuál de los algoritmos se comporta mejor. Esto supone un gran avance con respecto al tipo de análisis comparativos que se han llevado a cabo hasta ahora, ya que, al conocer todos los detalles involucrados en el proceso de comparación (incluyendo, por ejemplo, los parámetros utilizados en la generación de los grafos) y poder guardarlos y recuperarlos en cualquier momento, se hace posible que la comparativa sea reproducible, favoreciendo así la objetividad y credibilidad de la misma.

1.3. Metodología

Una vez definido el conjunto de objetivos que regirá este Proyecto Fin de Carrera, es preciso establecer el procedimiento para alcanzarlos de manera satisfactoria. Por ello, se ha considerado adecuado el uso de la metodología MÉTRICA versión 3 en desarrollo orientado a objetos para sistematizar las actividades que darán soporte al ciclo de vida de Graphite, la aplicación a desarrollar.

MÉTRICA es una metodología propuesta por el Ministerio de Hacienda y Administraciones Públicas del Gobierno de España con el fin de lograr una planificación, desarrollo y mantenimiento de sistemas de información dentro de su ámbito regulador. Todo contrato establecido entre el Gobierno de España y cualquier empresa de desarrollo de software impone el uso de esta metodología, justificándose esta restricción mediante las siguientes ventajas:

[2]Siglas de *Directed Graph Markup Language*. Véase http://schemas.microsoft.com/vs/2009/dgml/

- Define sistemas de información que ayudan a alcanzar los objetivos de la empresa mediante el establecimiento de un marco estratégico para el desarrollo de los mismos.
- Da una mayor importancia a la captura y análisis de requisitos, lo cual se traduce en productos software que satisfacen las necesidades de los usuarios en mayor medida.
- Proporciona mayor capacidad de adaptación a los cambios haciendo uso de la reutilización de software en caso de ser posible.
- Simplifica el mantenimiento y uso del producto software desarrollado, lo cual tiene un impacto muy positivo en la productividad de la empresa.

En su versión 3, MÉTRICA hace uso tanto de los métodos de desarrollo habituales como de los estándares de ingeniería de software más novedosos teniendo en cuenta, además, la interacción entre los usuarios y sus versiones anteriores. Así, esta metodología abarca dos tipos de desarrollo: (i) estructurado, y (ii) orientado a objetos, siendo el último el elegido para regular el proceso de desarrollo de Graphite.

MÉTRICA se basa en la definición de procesos, al igual que ISO/IEC 12.207, dividiendo la construcción de software en los procesos de planificación, desarrollo y mantenimiento. El segundo, a su vez, está formado por los cinco siguientes: (i) estudio de viabilidad, (ii) análisis, (iii) diseño, (iv) construcción, e (v) implantación y aceptación, siendo los cuatro primeros los que se han plasmado en este documento.

Por último, MÉTRICA incorpora interfaces para la definición de las actividades de soporte al proceso de desarrollo del producto software, las cuales están orientadas a mejorar la ejecución de los procesos de esta metodología. Estas interfaces son: (i) gestión de proyectos, (ii) gestión de configuración, (iii) aseguramiento de la calidad, y (iv) seguridad.

Capítulo 2

Detección de comunidades en grafos

Este capítulo ofrece una visión general del estado del arte de la detección de comunidades en grafos haciendo un recorrido a través de las aportaciones más importantes recogidas en la literatura. Previamente, introduce las nociones básicas y expone los diversos enfoques propuestos a lo largo de la última década para el avance de esta área de investigación.

2.1. Definición de comunidad

La detección de comunidades tiene como objetivo identificar, utilizando simplemente la topología de un grafo, grupos de vértices densamente conectados entre sí y que compartan características comunes o tengan un rol similar dentro del conjunto de vértices de dicho grafo. Aunque su popularidad actual tiene su origen en el trabajo de Girvan y Newman (2002), este problema ya había sido esbozado con anterioridad en otras disciplinas, considerándose la publicación de Rice (1927) la primera aplicación de técnicas de detección de comunidades[3]. Décadas más tarde, le seguirían los trabajos de Homans (1950) y Weiss y Jacobson (1955), cuyos métodos han servido de inspiración a numerosos algoritmos actuales de detección de comunidades (para un resumen completo véase Hastie et al., 2001).

A pesar de que su objetivo parece intuitivo a primera vista, la detección de comunidades adolece de un grave problema: no existe una definición de comunidad universalmente aceptada más allá de la noción de que debe haber más aristas entre los vértices de una comunidad que con los vértices de otras comunidades (Fortunato, 2010). De hecho, la mayor parte de los algoritmos desarrollados en la última década posee su propia definición, la cual depende, en gran medida, del fenómeno bajo estudio. Esto, unido a la falta de rigurosidad en lo relativo a la prueba de los algoritmos propuestos y la escasez de análisis comparativos de los mismos (Danon et al., 2005; Fan et al., 2007; Lancichinetti y Fortunato, 2009b; Sawardecker et al., 2009), explica la abundante aparición de técnicas en los últimos años y limita el avance de esta área de investigación, haciendo prácticamente imposible establecer una definición formal de comunidad.

No obstante, algunos trabajos se han centrado en especificar propiedades comunes, basadas en criterios de cohesión, que toda definición de comunidad debe cumplir (Scott, 2000). Una de estas propiedades es la introducida por Moody y White (2003), que establece que la eliminación de un vértice cualquiera de una comunidad no puede provocar la disolución

[3] En dicho trabajo, Rice perseguía identificar grupos políticos de personas de acuerdo con la similitud de sus patrones de voto.

de la misma. Asimismo, Wasserman y Faust (1994) presentaron dos criterios de cohesión de comunidades íntimamente relacionados: (i) mutualidad completa, y (ii) alcanzabilidad. El primero define las comunidades como subgrafos cuyos vértices son todos adyacentes entre sí (este tipo de estructuras son conocidas como cliques en teoría de grafos); el segundo permite relajar la noción de clique definiendo las comunidades como k-cliques, siendo un k-clique un subgrafo tal que la distancia entre cada uno de sus vértices es a lo sumo k (Luce, 1950). Mientras que la mutualidad completa, debido a su descripción tan estricta, es rara vez usada (Palla et al., 2005), la alcanzabilidad aparece como definición de comunidad en un mayor número de algoritmos (Rees y Gallagher, 2010). A pesar de que actualmente no existe consenso en cuanto a la definición de comunidad, la mayor parte de los algoritmos propuestos en la última década hace uso de, al menos, uno de los criterios presentados en esta sección.

2.2. Algoritmos

La inexistencia de una definición de comunidad consistente justifica la heterogeneidad de las propuestas para dar solución al problema de la detección de comunidades, surgiendo, de esta manera, la necesidad de catalogar los algoritmos en base al enfoque empleado. Los algoritmos pueden clasificarse, además de por la técnica utilizada para llevar a cabo la identificación de las comunidades, de acuerdo con tres criterios comunes a todos ellos. El primero de tales criterios hace referencia al determinismo del algoritmo, es decir, puede ser determinista o no determinista. El segundo criterio considera los tipos de grafos para los que el algoritmo es capaz de detectar comunidades, atendiendo a la dirección de sus aristas, es decir, puede manipular grafos dirigidos, no dirigidos o ambos; y a su peso, es decir, puede manipular grafos ponderados, no ponderados o ambos. Por último, el tercer criterio contempla los tipos de comunidades que genera el algoritmo, es decir, detecta comunidades solapadas[4], no solapadas o ambas. A continuación, se introducen las tres categorías que permiten clasificar la mayor parte de los algoritmos propuestos en la última década.

2.2.1. Algoritmos divisivos

La filosofía subyacente en los algoritmos divisivos consiste en eliminar las aristas que conectan vértices pertenecientes a diferentes comunidades, de manera que éstas queden aisladas unas de otras. La principal dificultad de este enfoque reside en identificar tales aristas. Habitualmente, las comunidades detectadas por este tipo de algoritmos se representan mediante dendrogramas.

El algoritmo divisivo más conocido es el algoritmo de Girvan y Newman, el cual determina las aristas que conectan vértices pertenecientes a diferentes comunidades a partir de una extensión de la intermediación (Girvan y Newman, 2002; Newman y Girvan, 2004). La intermediación es una medida de centralidad que indica la influencia de un vértice de un grafo en base al número de caminos mínimos, entre todos los pares de vértices, que pasen por dicho vértice (Freeman, 1977). Para su algoritmo, Girvan y Newman extendieron la definición de intermediación para contemplar el cálculo de la influencia de las aristas en base al número de caminos mínimos, entre todos los pares de vértices, que pasan por cada arista. En los últimos años, se han propuesto modificaciones de este algoritmo para la detección de comunidades solapadas (Gregory, 2007, 2008; Pinney y Westhead, 2006).

[4]En la detección de comunidades solapadas, los vértices de un grafo pueden clasificarse no sólo en una, sino en más de una comunidad.

2.2.2. Algoritmos basados en modularidad

La modularidad es una función, propuesta por Girvan y Newman (2002) como criterio de parada de su algoritmo, que indica la calidad de las comunidades detectadas; cuanto mayor es la modularidad, mejor es el resultado obtenido por el algoritmo. En los años sucesivos, se popularizó la optimización de la modularidad y se extendió su uso, llegando a ser el componente fundamental de muchos algoritmos de detección de comunidades.

Dependiendo de la armonía entre complejidad computacional y grado de optimización, se distinguen tres tipos de técnicas basadas en modularidad: (i) algoritmos voraces con un grado bajo de optimización (Clauset et al., 2004; Danon et al., 2006; Newman, 2004b), (ii) algoritmos de complejidad elevada con una precisión mayor (Guimerà et al., 2004; Massen y Doye, 2005; Medus et al., 2005), y (iii) algoritmos que equilibran la complejidad computacional y el grado de optimización (Duch y Arenas, 2005; Newman, 2006).

Actualmente, la modularidad parece haber perdido importancia debido, principalmente, a los dos defectos de los que adolece: (i) su límite de resolución, y (ii) la diversidad estructural de los conjuntos de comunidades con alta modularidad (Fortunato y Barthélemy, 2007; Lancichinetti y Fortunato, 2011). El primer inconveniente se refiere a la imposibilidad de detectar comunidades que sean pequeñas en comparación con el tamaño del grafo al que pertenecen, incluso aunque éstas sean comunidades bien definidas (por ejemplo, cliques); el segundo inconveniente establece que el conjunto óptimo de comunidades puede no coincidir con el más intuitivo.

2.2.3. Algoritmos basados en dinámica social

El declive de la importancia de la modularidad ha llevado a los investigadores a tener que buscar otros enfoques que incorporen la manera en que se desarrollan las comunidades. La aparición de servicios como Facebook o Twitter ha brindado la oportunidad de observar el proceso natural de creación y evolución de las comunidades. De esta manera, los algoritmos actuales de detección de comunidades comienzan a adoptar un planteamiento más cercano a los fenómenos que acontecen en los servicios de redes sociales.

Cazabet et al. (2010) propusieron un algoritmo basado en dos principios: (i) naturaleza intrínseca de las comunidades, y (ii) detección longitudinal. Dado un número natural k, el primer principio establece que la detección de comunidades no debería estar limitada a la identificación de un cierto k-clique, es decir, el algoritmo puede encontrar comunidades de diversos tamaños en el mismo grafo; el segundo principio tiene por objetivo recoger la dinámica del grafo, esto es, el momento en que se crea un vértice, una arista o una comunidad. Palazuelos y Zorrilla (2011, 2012) desarrollaron un algoritmo de detección de comunidades solapadas basado en la idea intuitiva de amistad entre los miembros de una comunidad, donde algunos de estos miembros se comportan como líderes del grupo. El algoritmo no sólo es capaz de identificar las comunidades solapadas de un grafo dado, sino que permite revelar la estructura jerárquica de sus vértices. Para ello, los autores definieron una nueva medida de centralidad: el grado extendido. Siguiendo un enfoque muy similar al anterior, Stanoev et al. (2011) propusieron un algoritmo de detección de comunidades solapadas. Asimismo, el algoritmo es capaz de indicar la influencia de los vértices del grafo contando, para ello, el número de triángulos que cada vértice comparte con sus vértices adyacentes; cuanto mayor es este número, más influyente es su vértice asociado.

2.3. Grafos clásicos

Al igual que en otras disciplinas, en el área de la detección de comunidades existe un conjunto de resultados experimentales que se considera fiable para medir la efectividad de los métodos desarrollados. Dado que las redes sociales constituyen el ejemplo por antonomasia de grafos con estructura de comunidades, la mayor parte de estos resultados experimentales proviene de investigaciones sociológicas relativamente modernas de técnicas de análisis de redes sociales. En esta sección, se ofrecen tres ejemplos clásicos de redes reales utilizadas de manera asidua en la evaluación y comparación de gran parte de los algoritmos propuestos en la última década (para un resumen completo véase Fortunato, 2010).

El primer ejemplo es el club de kárate de Zachary (1977). Esta red presenta 78 vínculos sociales entre 34 miembros de una asociación de kárate de una universidad americana en la década de 1970. De manera accidental, en algún momento del último de los tres años que Zachary dedicó al estudio de la dinámica social del club, el presidente y el profesor tuvieron una disputa que provocó la escisión de la asociación en dos grupos más pequeños centrados, precisamente, en el presidente y el profesor. La Figura 2.1 muestra las comunidades no solapadas publicadas originalmente por Zachary, donde los vértices 1 y 34 representan al presidente y al profesor, respectivamente.

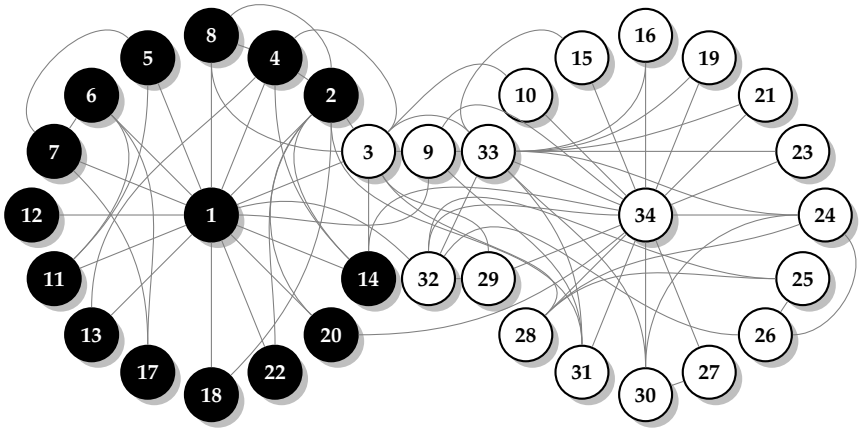


Figura 2.1: Comunidades originales del club de kárate de Zachary

Los delfines mulares estudiados por Lusseau et al. (2003) constituyen otro de los ejemplos clásicos de redes reales utilizadas en la evaluación y comparación de algoritmos de detección de comunidades. Esta red describe 159 asociaciones entre 62 delfines mulares de Doubtful Sound, Nueva Zelanda, recopiladas por Lusseau et al. tras siete años de investigación. Los delfines se separaron en dos grupos después de que uno de ellos se alejara de Doubtful Sound por un tiempo. Así, la red se divide de manera natural en dos comunidades fácilmente reconocibles, en las cuales puede detectarse, incluso, la presencia de cliques.

Finalmente, el último ejemplo es la red de fútbol americano construida por Girvan y Newman (2002). Esta red reúne 616 partidos entre 115 equipos de la División I de la liga universitaria americana en la temporada del año 2000. De acuerdo con las reglas de la liga, cada equipo participante debía inscribirse en una de las 12 conferencias establecidas a tal efecto, lo cual proporciona una aproximación a la estructura de comunidades a detectar.

2.4. Generadores de grafos

A comienzos de la última década, los grafos clásicos se convirtieron en el instrumento fundamental de evaluación de los algoritmos de detección de comunidades propuestos debido, principalmente, a que su estructura y tamaño eran similares a los de las redes sociales bajo estudio. Años más tarde, la creciente popularidad de servicios como Facebook o Twitter trajo consigo la aparición de redes cada vez más extensas cuya estructura ya no se asemejaba a la presentada por los grafos clásicos. De esta manera, se hicieron necesarios la investigación y desarrollo de algoritmos que permitieran generar grafos con características semejantes a las de las nuevas redes sociales.

Los generadores de grafos comparten con los algoritmos de detección de comunidades dos criterios de acuerdo con los cuales pueden clasificarse[5]. El primero considera los tipos de grafos que el generador es capaz de producir, atendiendo a la dirección de sus aristas, es decir, puede generar grafos dirigidos, no dirigidos o ambos; y a su peso, es decir, puede generar grafos ponderados, no ponderados o ambos. El segundo criterio contempla los tipos de comunidades en que el generador clasifica los vértices generados, es decir, los organiza en comunidades solapadas, no solapadas o ambas.

El primer generador de grafos con estructura de comunidades conocido es el introducido por Condon y Karp (2001). Este generador es capaz de producir grafos no dirigidos y no ponderados de n vértices clasificándolos en c comunidades de n/c vértices cada una. La probabilidad de que una arista conecte dos vértices de una misma comunidad se denota p_{in} mientras que la probabilidad de que una arista conecte dos vértices de diferentes comunidades se denota p_{out}, siendo siempre $p_{\text{in}} > p_{\text{out}}$. La expresión del grado promedio de los vértices $\langle k \rangle$ permite relacionar todos los parámetros del generador

$$\langle k \rangle = p_{\text{in}} \left(\frac{n}{c} - 1 \right) + p_{\text{out}} \frac{n\,(c-1)}{c} \quad . \tag{2.1}$$

En los últimos años, se han propuesto modificaciones de este generador de grafos. Girvan y Newman (2002) contemplaron un caso especial fijando el número de vértices $n = 128$ y el número de comunidades $c = 4$, así como el grado promedio de los vértices $\langle k \rangle = 16$. Asimismo, Fan et al. (2007) lo extendieron para generar grafos ponderados.

A pesar de que el generador de grafos de Condon y Karp está basado en ideas intuitivas, los grafos que produce exhiben dos propiedades que las redes reales no parecen abrazar: (i) todos los vértices poseen aproximadamente el mismo grado, y (ii) todas las comunidades tienen exactamente el mismo tamaño. Para solventar este problema, Lancichinetti et al. (2008) introdujeron el generador de grafos LFR, el cual, actualmente, se considera estándar en la evaluación y comparación de algoritmos de detección de comunidades. LFR es una extensión del generador de grafos clásico propuesto por Girvan y Newman que permite producir grafos en que cada uno de sus vértices comparte una fracción $1-\mu$ de sus aristas con vértices de su propia comunidad y una fracción μ con vértices de otras comunidades, siendo μ un número real entre 0 y 1 denominado parámetro de mezcla. Cuanto más pequeño es el parámetro de mezcla μ, más claras son las estructuras de comunidades que presentan los grafos generados. Posteriormente, se extendió LFR para generar grafos dirigidos y ponderados con comunidades solapadas (Lancichinetti y Fortunato, 2009a).

[5] El tercer criterio de los algoritmos de detección de comunidades, el determinismo, no tiene cabida aquí ya que, por definición, los generadores de grafos son algoritmos no deterministas.

2.5. Medidas de evaluación

El proceso de evaluación de un algoritmo de detección de comunidades implica la aplicación del mismo a un conjunto de grafos cuya estructura de comunidades sea conocida de antemano. Siendo $\mathcal{A} = \{A_1, A_2, \ldots\}$ el conjunto de comunidades detectadas por el algoritmo a evaluar y $\mathcal{G} = \{G_1, G_2, \ldots\}$ la estructura de comunidades del grafo bajo estudio, el cometido de una medida de evaluación es determinar cuán similares son $\mathcal{A}$ y $\mathcal{G}$. En esta sección, se ofrecen tres ejemplos clásicos de métodos utilizados de manera asidua en la evaluación y comparación de gran parte de los algoritmos propuestos en la última década (para un resumen completo véase Meilă, 2007).

La primera medida de evaluación de algoritmos de detección de comunidades conocida es la introducida por Girvan y Newman (2002). Esta medida es capaz de indicar cuán semejantes son $\mathcal{A}$ y $\mathcal{G}$ mediante el recuento de vértices clasificados correctamente por el algoritmo a evaluar, considerando que un vértice v de una comunidad de $\mathcal{A}$ está clasificado adecuadamente si en esa misma comunidad también se encuentra, al menos, la mitad de los vértices de la comunidad de $\mathcal{G}$ a la que pertenece v. Una vez calculado el número de vértices clasificados correctamente, éste se divide entre el número total de vértices del grafo, de manera que esta medida arroja un número real entre 0 y 1; cuanto mayor es este número, mayor es la similitud entre $\mathcal{A}$ y $\mathcal{G}$.

Debido a la arbitrariedad de su definición, la medida de evaluación de Girvan y Newman pronto cayó en desuso. Una de las primeras estrategias que la sucedieron fue el recuento de pares, cuya filosofía subyacente consiste en calcular el número de pares de vértices que pertenecen a la misma comunidad tanto en $\mathcal{A}$ como en $\mathcal{G}$. El índice de Rand (1971), medida de evaluación que sigue esta estrategia, se define como la proporción del número de pares de vértices que pertenecen o bien a la misma comunidad, o bien a diferentes comunidades tanto en $\mathcal{A}$ como en $\mathcal{G}$ entre el número total de pares de vértices, es decir,

$$R(\mathcal{A}, \mathcal{G}) = \frac{p_{11} + p_{00}}{p_{11} + p_{10} + p_{01} + p_{00}} \ , \tag{2.2}$$

donde p_{11} indica el número de pares de vértices que se encuentran en la misma comunidad tanto en $\mathcal{A}$ como en $\mathcal{G}$, p_{00} el número de pares de vértices que pertenecen a diferentes comunidades en ambos conjuntos, y p_{10} y p_{01} el número de pares de vértices clasificados en la misma comunidad en $\mathcal{A}$ y $\mathcal{G}$, respectivamente, y en diferentes comunidades en los conjuntos opuestos en cada caso.

En los últimos años, se ha hecho muy popular el uso de la información mutua normalizada (Strehl y Ghosh, 2003) como medida de evaluación de algoritmos de detección de comunidades, la cual sigue un enfoque basado en teoría de la información. La información mutua normalizada se basa en la idea de que si dos conjuntos de comunidades son similares, entonces se necesitará muy poca información para obtener un conjunto a partir del otro. En términos matemáticos, esto es

$$I(\mathcal{A}, \mathcal{G}) = \frac{\displaystyle\sum_{i=1}^{|\mathcal{A}|} \sum_{j=1}^{|\mathcal{G}|} |A_i \cap G_j| \log \left(n \frac{|A_i \cap G_j|}{|A_i|\,|G_j|} \right)}{\sqrt{\left(\displaystyle\sum_{i=1}^{|\mathcal{A}|} |A_i| \log \frac{|A_i|}{n} \right) \left(\displaystyle\sum_{j=1}^{|\mathcal{G}|} |G_j| \log \frac{|G_j|}{n} \right)}} \ . \tag{2.3}$$

Capítulo 3

Captura y análisis de requisitos

Este capítulo recoge la captura y análisis de requisitos de Graphite con el fin de determinar las condiciones y necesidades que deberán satisfacerse en su diseño e implementación posteriores. Para ello, aborda aspectos de la aplicación que describen tanto funcionalidades como restricciones sobre el proceso de desarrollo de la misma.

3.1. Requisitos funcionales

Los requisitos funcionales definen el comportamiento específico de la aplicación, el cual puede expresarse a través de los servicios que ésta debe proporcionar o las tareas que precisa realizar. Para poder describir los requisitos funcionales de una aplicación de manera adecuada, ha de conocerse la interacción de sus usuarios con ella. En el caso de Graphite, los usuarios podrán llevar a cabo dos tipos de interacciones con la aplicación: (i) evaluar y comparar algoritmos de detección de comunidades, y (ii) gestionar las extensiones suministradas por ellos mismos. A continuación, se presentan los diagramas de casos de uso, así como los requisitos funcionales, asociados a estos dos escenarios.

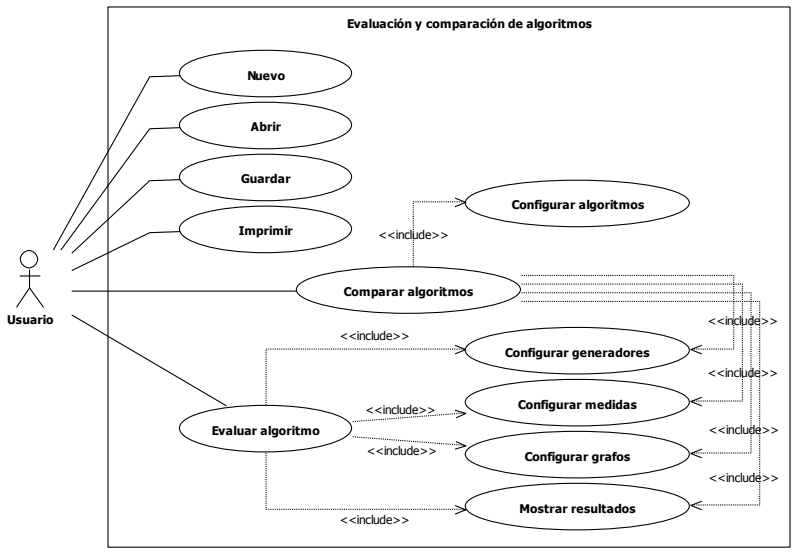


Figura 3.1: Diagrama de casos de uso para la evaluación y comparación de algoritmos

La Figura 3.1 muestra las funcionalidades relativas a la evaluación y comparación de algoritmos de detección de comunidades que Graphite proporcionará a sus usuarios. A partir de este diagrama, se pueden describir los siguientes requisitos funcionales:

- **Evaluación de un algoritmo de detección de comunidades:** Los usuarios podrán elegir uno de los algoritmos instalados en la aplicación para determinar su eficiencia. Después, tendrán la posibilidad de elegir qué grafos individuales, generadores de grafos y medidas de evaluación intervendrán en el proceso, así como definir la configuración con que éstos se ejecutarán.
- **Comparación de dos o más algoritmos de detección de comunidades:** Los usuarios podrán elegir uno de los algoritmos instalados en la aplicación para compararlo con, al menos, otro algoritmo. Después, tendrán la posibilidad de elegir con cuál o cuáles se llevará a cabo la comparación y qué grafos individuales, generadores de grafos y medidas de evaluación intervendrán en el proceso, así como definir la configuración con que éstos se ejecutarán.
- **Almacenamiento, recuperación y representación de la información:** Los usuarios podrán guardar las evaluaciones y comparaciones llevadas a cabo, así como abrirlas posteriormente. Asimismo, tendrán la posibilidad de iniciar una nueva evaluación o comparación en cualquier momento e imprimir los resultados de las mismas.

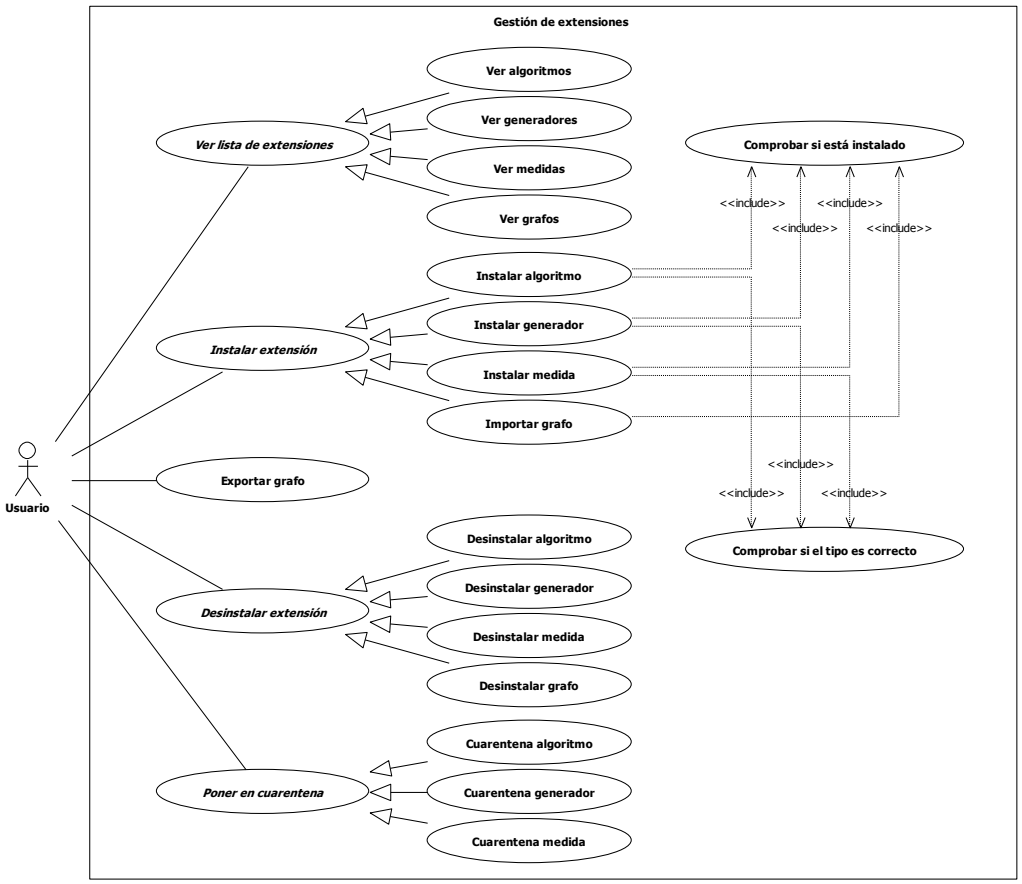


Figura 3.2: Diagrama de casos de uso para la gestión de extensiones

La Figura 3.2 muestra las funcionalidades relativas a la gestión de las extensiones suministradas por los usuarios que Graphite les proporcionará. A partir de este diagrama, se pueden describir los siguientes requisitos funcionales:

- **Instalación de extensiones:** Los usuarios podrán instalar nuevos algoritmos, generadores de grafos y medidas de evaluación, así como importar grafos individuales escritos en ficheros con formato DGML. Durante el proceso de instalación, Graphite comprobará si la extensión seleccionada ya está instalada y si su tipo se corresponde con el tipo de extensión que se desea instalar.
- **Desinstalación de extensiones:** Los usuarios tendrán la posibilidad de desinstalar algoritmos, generadores de grafos, medidas de evaluación y grafos individuales previamente instalados en la aplicación.
- **Visualización de extensiones:** Los usuarios podrán consultar la información asociada a cada uno de los algoritmos, generadores de grafos, medidas de evaluación y grafos individuales instalados en la aplicación.
- **Puesta en cuarentena:** Los usuarios tendrán la posibilidad de aislar, sin llevar a cabo su eliminación, aquellos algoritmos, generadores de grafos o medidas de evaluación que muestren un comportamiento sospechoso durante su ejecución.
- **Almacenamiento de grafos generados:** Los usuarios podrán exportar los grafos producidos por los generadores de grafos en los procesos de evaluación y comparación de algoritmos a ficheros con formato DGML.

3.2. Requisitos no funcionales

Los requisitos no funcionales establecen las restricciones a tener en cuenta durante el proceso de desarrollo mediante la definición de los atributos o propiedades que la aplicación deberá haber adquirido al finalizar éste. De esta manera, se presentan los requisitos no funcionales asociados a Graphite:

- **Extensibilidad:** La aplicación deberá diseñarse de tal manera que pueda ampliarse fácilmente para contemplar la resolución de problemas similares a la detección de comunidades en grafos.
- **Portabilidad:** Para asegurar la compatibilidad de la aplicación con el mayor número posible de sistemas operativos, ésta, al igual que sus extensiones, deberá desarrollarse en Java, lenguaje multiplataforma de propósito general.
- **Rendimiento:** La aplicación deberá contemplar el uso de sistemas de gestión de bases de datos para llevar a cabo un almacenamiento y recuperación eficientes de los grafos producidos por los generadores de grafos.
- **Seguridad:** Para garantizar que el código de las extensiones suministradas no vulnere la seguridad de la máquina, la aplicación deberá poder ejecutarlas en un entorno seguro en que los recursos y privilegios a su disposición serán muy reducidos.
- **Usabilidad:** La aplicación deberá interaccionar con los usuarios a través de una interfaz gráfica de usuario que sea capaz de mostrar los resultados de los procesos de evaluación y comparación de algoritmos de manera clara y concisa.

Capítulo 4

Diseño e implementación

Este capítulo ofrece una visión pormenorizada del diseño e implementación de Graphite haciendo un recorrido a través de las partes fundamentales que lo componen. Previamente, esboza el diseño general de la arquitectura de la aplicación introduciendo las estrategias seguidas para llevar a cabo la implementación de la misma.

4.1. Diseño general de la arquitectura

Graphite es una herramienta que permite realizar comparaciones empíricas de algoritmos de detección de comunidades de manera imparcial y exhaustiva. Se ha diseñado de manera que los usuarios puedan añadirle, mediante un mecanismo de extensiones o *plug-ins*, cuatro tipos de elementos: (i) algoritmos, (ii) generadores de grafos, (iii) medidas de evaluación, y (iv) grafos individuales. Debido a esta naturaleza extensible de la aplicación, así como a la heterogeneidad de los sistemas operativos con que sus usuarios se desenvuelven, el lenguaje de programación seleccionado para desarrollarla e implementar sus extensiones ha sido Java, lenguaje multiplataforma de propósito general.

A pesar de que actualmente la aplicación cubre por completo los procesos de evaluación y comparación de algoritmos de detección de comunidades, cabe la posibilidad de que en el futuro se extienda para contemplar la resolución de problemas relativamente similares (por ejemplo, algoritmos de agrupamiento o *clustering*). Por ello, se consideró adecuado el uso del paradigma Modelo–Vista–Controlador (MVC) en el diseño de la misma. MVC es un patrón de arquitectura de software cuya ventaja fundamental reside en desligar la lógica de negocio, la interfaz gráfica de usuario y la gestión de eventos en tres capas diferentes, de manera que la modificación de alguna de ellas no involucre cambios en las demás (Reenskaug, 1979). Como consecuencia de este desacoplamiento, MVC facilita el mantenimiento posterior del sistema. El modelo representa la información con que la aplicación se desenvuelve; la vista presenta el modelo de forma visual posibilitando la interacción con los usuarios; y el controlador es el encargado de detectar las acciones de dichos usuarios y notificárselas al modelo y a la vista para que tomen las acciones oportunas. La Figura 4.1 muestra el diseño general de la aplicación clasificando cada una de sus partes de acuerdo con el paradigma MVC.

Por una parte, la capa relativa a la vista de la aplicación está formada por el paquete de interfaz gráfica de usuario, el cual representa el estado del modelo de forma visual. Éste hace uso de los paquetes de interfaces para proveedores de extensiones y de detectores de eventos, pertenecientes al modelo y al controlador de la aplicación, respectivamente. Por otra parte,

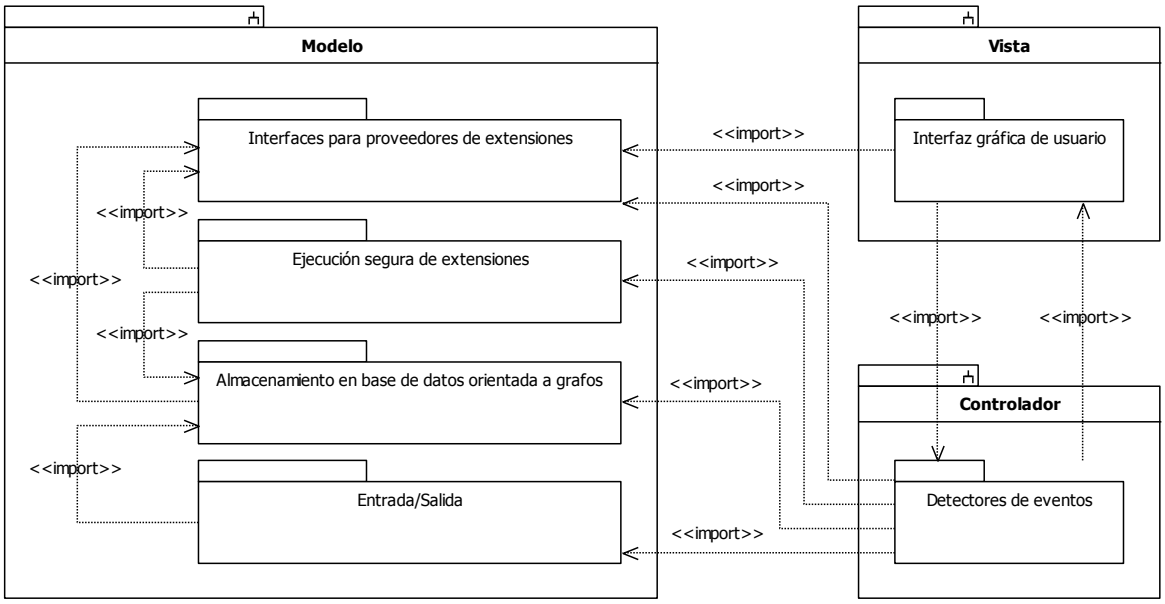


Figura 4.1: Diagrama de paquetes de la aplicación

la capa correspondiente al controlador de la aplicación está compuesta por el paquete de detectores de eventos, el cual registra las acciones de los usuarios y se las notifica al modelo y a la vista. Dado que actúa como intermediario entre las otras dos capas, este paquete hace uso de todos los paquetes de la aplicación.

De forma somera, se presenta la descripción de los paquetes que constituyen la capa correspondiente al modelo de la aplicación. En las secciones subsiguientes, se precisa el contenido de cada uno de estos paquetes a través de diagramas de clases. Para garantizar la claridad de la información presentada, estos diagramas de clases rehúsan recoger elementos con visibilidad privada (salvo en el caso de atributos privados representados en relaciones de asociación entre clases) y relaciones de dependencia entre clases si ya quedan indicadas de manera explícita por sus métodos.

- **Paquete de interfaces para proveedores de extensiones:** Conjunto de interfaces públicas que modela de forma completa todos los elementos que intervienen en los procesos de evaluación y comparación de algoritmos de detección de comunidades.
- **Paquete de almacenamiento en base de datos orientada a grafos:** Conjunto de clases que proporciona un protocolo de comunicación entre los grafos y generadores de grafos suministrados y la base de datos subyacente. Hace uso del paquete de interfaces para proveedores de extensiones.
- **Paquete de ejecución segura de extensiones:** Conjunto de clases que permite ejecutar cada tipo de extensión en un entorno seguro en que los recursos y privilegios a su disposición son muy reducidos. Hace uso de los paquetes de interfaces para proveedores de extensiones y de almacenamiento en base de datos orientada a grafos.
- **Paquete de entrada/salida:** Conjunto de clases que permite leer y escribir grafos en ficheros con formatos compatibles con la aplicación mediante el acceso a la base de datos. Hace uso del paquete de almacenamiento en base de datos orientada a grafos.

4.2. Interfaces para proveedores de extensiones

El diseño extensible de Graphite permite que los usuarios puedan añadirle varios tipos de extensiones. Para hacerlo posible, ha sido necesario definir un conjunto de interfaces públicas que modelen de forma completa todos los elementos que intervienen en los procesos de evaluación y comparación de algoritmos de detección de comunidades (véase la Figura 4.2). Esto proporciona un modelo estándar que simplifica dramáticamente el desarrollo de extensiones, las cuales únicamente deben importar este paquete, una vez compilado como biblioteca, e implementar las interfaces correspondientes contenidas en el mismo.

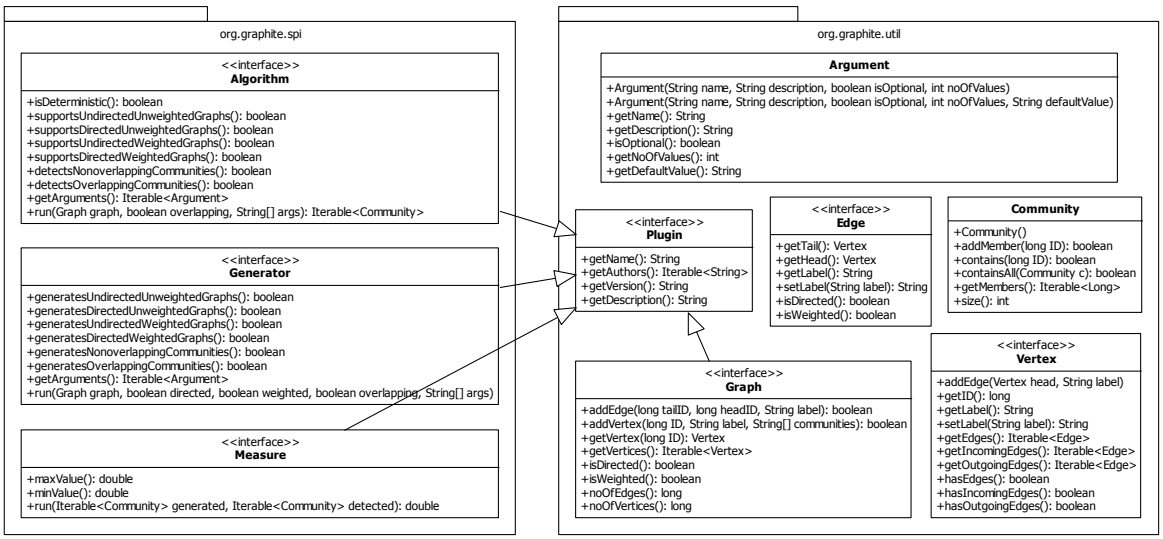


Figura 4.2: Diagrama de clases de interfaces para proveedores de extensiones

Para construir el mecanismo de extensiones, se ha utilizado la clase `ServiceLoader` del paquete `java.util`, disponible en Java SE 6 y posteriores, la cual es capaz de buscar y cargar extensiones que implementen un conjunto de interfaces dado en el directorio que se le especifique. Una vez cargadas, las extensiones pueden ser utilizadas por la aplicación como instancias de las interfaces que implementan. Para desarrollar una extensión compatible con Graphite, basta con proporcionar una implementación de la interfaz correspondiente y crear un fichero de configuración en el directorio `META-INF/services` del fichero JAR que la contiene. El nombre de dicho fichero de configuración debe ser nombre completo de la interfaz implementada (por ejemplo, `org.graphite.spi.Algorithm`) y su contenido debe ser el nombre completo de la clase que la implemente (por ejemplo, `paquete.subpaquete.Clase`). A continuación, se describe la funcionalidad de cada una de las interfaces y clases contenidas en este paquete.

Tabla 4.1: Métodos de la interfaz `Plugin`

Nombre	Descripción
getName	Retorna el nombre de la extensión.
getAuthors	Retorna los autores de la extensión.
getVersion	Retorna la versión de la extensión.
getDescription	Retorna la descripción de la extensión.

La interfaz `Plugin` representa una extensión compatible con Graphite. Una extensión contiene información acerca de su nombre, sus autores, su versión actual y una descripción que la define. La Tabla 4.1 recoge el nombre y descripción de cada uno de los métodos de esta interfaz.

Tabla 4.2: Métodos de la interfaz `Algorithm`

Nombre	Descripción
isDeterministic	Indica si el algoritmo es determinista o no determinista.
supportsUndirectedUnweightedGraphs	Indica si el algoritmo puede detectar comunidades en grafos no dirigidos y no ponderados.
supportsDirectedUnweightedGraphs	Indica si el algoritmo puede detectar comunidades en grafos dirigidos y no ponderados.
supportsUndirectedWeightedGraphs	Indica si el algoritmo puede detectar comunidades en grafos no dirigidos y ponderados.
supportsDirectedWeightedGraphs	Indica si el algoritmo puede detectar comunidades en grafos dirigidos y ponderados.
detectsNonoverlappingCommunities	Indica si el algoritmo es capaz de detectar comunidades no solapadas.
detectsOverlappingCommunities	Indica si el algoritmo es capaz de detectar comunidades solapadas.
getArguments	Retorna información acerca de los argumentos del algoritmo.
run	Retorna las comunidades detectadas por el algoritmo.

La interfaz `Algorithm` representa un algoritmo de detección de comunidades definido por los criterios presentados en la Sección 2.2. Extiende la interfaz `Plugin` y hace uso de (i) la interfaz `Graph` en la detección de comunidades, (ii) la clase `Argument` en la definición de sus argumentos, y (iii) la clase `Community` para retornar las comunidades detectadas. La Tabla 4.2 recoge el nombre y descripción de cada uno de los métodos de esta interfaz.

Tabla 4.3: Métodos de la interfaz `Generator`

Nombre	Descripción
generatesUndirectedUnweightedGraphs	Indica si el generador de grafos puede producir grafos no dirigidos y no ponderados.
generatesDirectedUnweightedGraphs	Indica si el generador de grafos puede producir grafos dirigidos y no ponderados.
generatesUndirectedWeightedGraphs	Indica si el generador de grafos puede producir grafos no dirigidos y ponderados.
generatesDirectedWeightedGraphs	Indica si el generador de grafos puede producir grafos dirigidos y ponderados.
generatesNonoverlappingCommunities	Indica si el generador de grafos es capaz de producir comunidades no solapadas.
generatesOverlappingCommunities	Indica si el generador de grafos es capaz de producir comunidades solapadas.
getArguments	Retorna información acerca de los argumentos del generador de grafos.
run	Genera un nuevo grafo.

La interfaz `Generator` representa un generador de grafos definido por los criterios presentados en la Sección 2.4. Extiende la interfaz `Plugin` y hace uso de (i) la interfaz `Graph` en la generación del grafo, y (ii) la clase `Argument` en la definición de sus argumentos. La Tabla 4.3 recoge el nombre y descripción de cada uno de los métodos de esta interfaz.

La interfaz `Measure` representa una medida de evaluación. Extiende la interfaz `Plugin` y hace uso de la clase `Community` en la comparación de los conjuntos de comunidades dados para indicar cuán similares son. La Tabla 4.4 recoge el nombre y descripción de cada uno de los métodos de esta interfaz.

Tabla 4.4: Métodos de la interfaz `Measure`

Nombre	Descripción
maxValue	Retorna el valor máximo que la medida de evaluación puede tomar.
minValue	Retorna el valor mínimo que la medida de evaluación puede tomar.
run	Indica cuán similares son los conjuntos de comunidades dados.

La interfaz `Graph` representa un grafo cuyas aristas pueden tener dirección y peso. Extiende la interfaz `Plugin` y hace uso de la interfaz `Vertex` cuando consulta el conjunto de vértices del grafo. La Tabla 4.5 recoge el nombre y descripción de cada uno de los métodos de esta interfaz.

Tabla 4.5: Métodos de la interfaz `Graph`

Nombre	Descripción
addEdge	Añade una nueva arista al grafo.
addVertex	Añade un nuevo vértice al grafo.
getVertex	Retorna el vértice con el identificador dado.
getVertices	Retorna todos los vértices del grafo.
isDirected	Indica si el grafo es dirigido.
isWeighted	Indica si el grafo es ponderado.
noOfEdges	Retorna el número de aristas del grafo.
noOfVertices	Retorna el número de vértices del grafo.

La interfaz `Vertex` representa un vértice de un grafo. Hace uso de la interfaz `Edge` al consultar los conjuntos de aristas (entrantes, salientes o ambos) que conectan con él. La Tabla 4.6 recoge el nombre y descripción de cada uno de los métodos de esta interfaz.

Tabla 4.6: Métodos de la interfaz `Vertex`

Nombre	Descripción
addEdge	Añade una nueva arista entre el vértice y el vértice dado con la etiqueta especificada.
getID	Retorna el identificador único del vértice.
getLabel	Retorna la etiqueta del vértice.
setLabel	Reemplaza la etiqueta del vértice.
getEdges	Retorna todas las aristas conectadas al vértice.
getIncomingEdges	Retorna todas las aristas entrantes conectadas al vértice.
getOutgoingEdges	Retorna todas las aristas salientes conectadas al vértice.
hasEdges	Indica si hay aristas conectadas al vértice.
hasIncomingEdges	Indica si hay aristas entrantes conectadas al vértice.
hasOutgoingEdges	Indica si hay aristas salientes conectadas al vértice.

La interfaz `Edge` representa la relación entre dos vértices de un grafo. Hace uso de la interfaz `Vertex` al indicar cuáles son el vértice origen y el vértice destino de dicha relación. La Tabla 4.7 recoge el nombre y descripción de cada uno de los métodos de esta interfaz.

Tabla 4.7: Métodos de la interfaz `Edge`

Nombre	Descripción
getTail	Retorna el vértice origen de la arista.
getHead	Retorna el vértice destino de la arista.
getLabel	Retorna la etiqueta de la arista.
setLabel	Reemplaza la etiqueta de la arista.
isDirected	Indica si la arista es dirigida.
isWeighted	Indica si la arista es ponderada.

La clase `Argument` representa un argumento de una extensión. Un argumento contiene información acerca de su nombre, su descripción, si es opcional, su número de valores requerido (0 ó 1) y su valor por defecto. La Tabla 4.8 recoge el nombre y descripción de cada uno de los métodos de esta clase.

Tabla 4.8: Métodos de la clase `Argument`

Nombre	Descripción
Argument	Crea un nuevo argumento con o sin valor por defecto.
getName	Retorna el nombre del argumento.
getDescription	Retorna la descripción del argumento.
isOptional	Indica si el argumento es opcional.
getNoOfValues	Retorna el número valores requerido por el argumento.
getDefaultValue	Retorna el valor por defecto del argumento.

La clase `Community` representa un grupo de vértices densamente conectados entre sí y que comparten características comunes o tienen un rol similar dentro del conjunto de vértices del grafo. La Tabla 4.9 recoge el nombre y descripción de cada uno de los métodos de esta clase.

Tabla 4.9: Métodos de la clase `Community`

Nombre	Descripción
Community	Crea una nueva comunidad.
addMember	Añade el vértice dado a la comunidad.
contains	Indica si la comunidad contiene el vértice dado.
containsAll	Indica si la comunidad contiene todos los vértices de la comunidad dada.
getMembers	Retorna los identificadores asociados a los vértices de la comunidad.
size	Retorna el número de vértices de la comunidad.

4.3. Almacenamiento en base de datos orientada a grafos

La rigurosidad en los procesos de evaluación y comparación de algoritmos de detección de comunidades implica la generación de una cantidad significativa de grafos de gran escala, lo cual hace inviable, en la mayor parte de los casos, su almacenamiento en memoria principal. Para acometer con éxito la gestión de tan vasta cantidad de datos, Graphite hace uso de Neo4j[6], un sistema de gestión de bases de datos orientadas a grafos de código abierto que permite almacenar la información en disco en forma de grafos. Las bases de datos orientadas a grafos hacen uso de tres elementos intrínsecos de los mismos para representar y almacenar la información: (i) vértices, (ii) aristas, y (iii) propiedades. El primero permite modelar entidades, el segundo simboliza las relaciones entre ellas y el tercero representa la información contenida en los dos primeros[7]. Dado que gestionan la información a través de grafos, este tipo de bases de datos ofrece un enfoque adecuado en situaciones en que los grafos constituyen la representación natural de la información o las relaciones entre los datos son tan importantes como los propios datos (por ejemplo, redes sociales o redes de interacciones proteína–proteína).

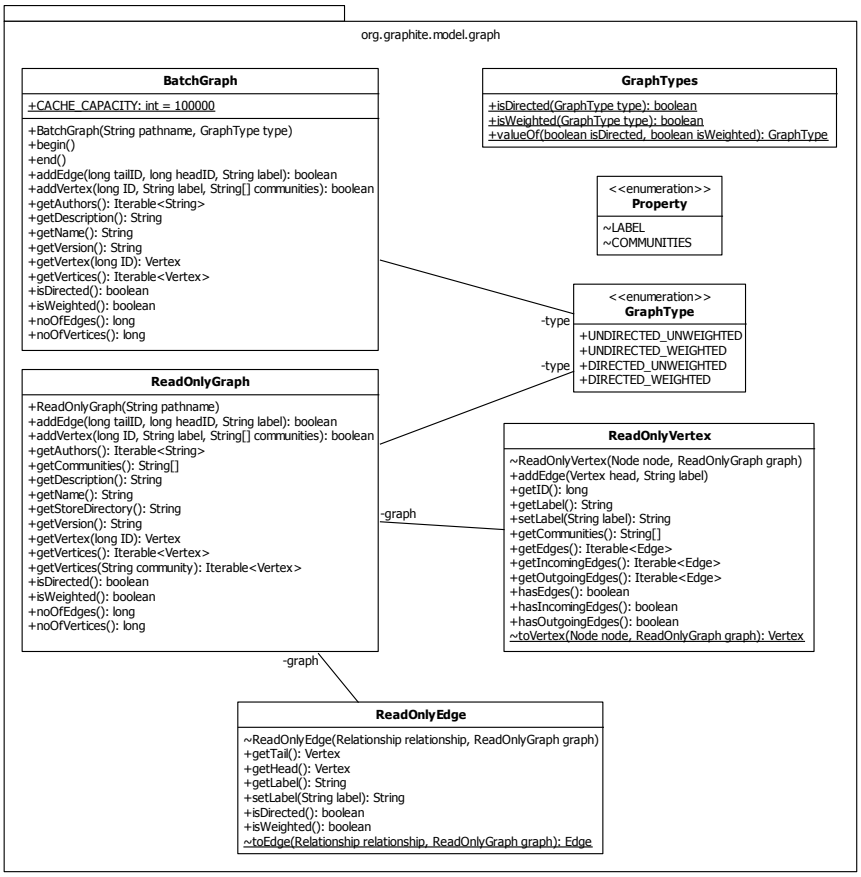


Figura 4.3: Diagrama de clases de almacenamiento en base de datos orientada a grafos

[6] Véase http://neo4j.org/

[7] Por ejemplo, en un servicio de correo electrónico, los vértices modelarían las cuentas de usuario y las aristas simbolizarían los mensajes intercambiados entre dichas cuentas. Así, la dirección de correo electrónico podría definirse como una de las propiedades de los vértices, mientras que las aristas podrían contemplar el contenido de los mensajes intercambiados como una de las suyas.

Neo4j es un sistema de gestión de bases de datos orientadas a grafos con persistencia transaccional que permite almacenar la información en disco en forma de grafos. Actualmente, es uno de los sistemas de este tipo más implantados en entornos de producción. A pesar de que existe un nutrido número de alternativas, se eligió Neo4j entre todas ellas por dos razones primordiales: (i) está escrito en Java, y (ii) puede ser embebido por la aplicación que lo utiliza. Graphite hace uso de Neo4j para gestionar los grafos producidos por los generadores de grafos, así como los proporcionados en ficheros con formato DGML. Para facilitar la traducción entre las estructuras de datos usadas por Graphite y Neo4j, ha sido necesario definir un conjunto de clases que actúe como capa intermedia y proporcione un protocolo de comunicación entre ambos (véase la Figura 4.3). A continuación, se describe la funcionalidad de cada una de estas clases.

La clase `BatchGraph` implementa un grafo, cuyas aristas pueden tener dirección y peso, que ofrece soporte para inserciones masivas de datos suprimiendo la concurrencia y las transacciones. Dado que los generadores de grafos pueden necesitar llevar a cabo inserciones masivas de datos, Graphite les proporciona esta implementación de la interfaz `Graph` a través del método *run* de la interfaz `Generator`.

La clase `ReadOnlyGraph` implementa un grafo, cuyas aristas pueden tener dirección y peso, que únicamente ofrece soporte para operaciones de lectura. Dado que los algoritmos de detección de comunidades no precisan realizar modificaciones sobre el grafo que se les proporciona, Graphite les ofrece esta implementación de la interfaz `Graph` a través del método *run* de la interfaz `Algorithm`. La Tabla 4.10 muestra qué métodos de la interfaz `Graph` han sido implementados por las clases `BatchGraph` y `ReadOnlyGraph` y cuáles no, en cuyo caso lanzan la excepción `UnsupportedOperationException` del paquete `java.lang`.

Tabla 4.10: Métodos de la interfaz `Graph` implementados por `BatchGraph` y `ReadOnlyGraph`

Método	`BatchGraph`	`ReadOnlyGraph`
addEdge	Sí	No
addVertex	Sí	No
getAuthors	No	Sí
getDescription	No	Sí
getName	No	Sí
getVersion	No	Sí
getVertex	No	Sí
getVertices	No	Sí
isDirected	No	Sí
isWeighted	No	Sí
noOfEdges	No	Sí
noOfVertices	No	Sí

La clase `ReadOnlyVertex` implementa un vértice de un grafo que únicamente ofrece soporte para operaciones de lectura. Dado que los algoritmos de detección de comunidades no precisan realizar modificaciones sobre los vértices del grafo que se les proporciona, Graphite les ofrece esta implementación de la interfaz `Vertex`. La Tabla 4.11 muestra qué métodos de la interfaz `Vertex` han sido implementados por esta clase.

Tabla 4.11: Métodos de la interfaz `Vertex` implementados por la clase `ReadOnlyVertex`

Método	Implementado
addEdge	No
getID	Sí
getLabel	Sí
setLabel	No
getEdges	Sí
getIncomingEdges	Sí
getOutgoingEdges	Sí
hasEdges	Sí
hasIncomingEdges	Sí
hasOutgoingEdges	Sí

La clase `ReadOnlyEdge` implementa una arista de un grafo que únicamente ofrece soporte para operaciones de lectura. Dado que los algoritmos de detección de comunidades no precisan realizar modificaciones sobre las aristas del grafo que se les proporciona, Graphite les ofrece esta implementación de la interfaz `Edge`. La Tabla 4.12 muestra qué métodos de la interfaz `Edge` han sido implementados por esta clase.

Tabla 4.12: Métodos de la interfaz `Edge` implementados por la clase `ReadOnlyEdge`

Método	Implementado
getTail	Sí
getHead	Sí
getLabel	Sí
setLabel	No
isDirected	Sí
isWeighted	Sí

El tipo enumerado `Property` representa las propiedades de un vértice o una arista de un grafo. Graphite es compatible con las siguientes propiedades: (i) `LABEL`, y (ii) `COMMUNITIES`. Definida en un vértice, la primera hace referencia al nombre de éste, mientras que en una arista simboliza el peso de la misma; la segunda representa el conjunto de comunidades en que un vértice está clasificado[8].

El tipo enumerado `GraphType` representa los tipos de un grafo. Graphite es compatible con los siguientes tipos de grafos: (i) `UNDIRECTED_UNWEIGHTED` o no dirigidos y no ponderados, (ii) `DIRECTED_UNWEIGHTED` o dirigidos y no ponderados, (iii) `UNDIRECTED_WEIGHTED` o no dirigidos y ponderados, y (iv) `DIRECTED_WEIGHTED` o dirigidos y ponderados. Por último, la clase `GraphTypes` contiene varios métodos útiles para los tipos de grafos. Estos métodos son:

- *isDirected:* Indica si el tipo de grafo dado es dirigido.
- *isWeighted:* Indica si el tipo de grafo dado es ponderado.
- *valueOf:* Retorna el tipo de grafo que se corresponde con la dirección y peso dados.

[8]Las aristas no pueden contener la propiedad `COMMUNITIES`.

4.4. Ejecución segura de extensiones

Uno de los desafíos principales a los que se enfrenta un mecanismo de extensiones como el que posee Graphite es impedir que, a través de una extensión, un usuario malintencionado sea capaz de ejecutar código malicioso que pueda comprometer la integridad de la máquina. Para evitarlo, ha sido necesario definir un conjunto de clases que permita ejecutar cada tipo de extensión en un entorno seguro o *sandbox* en que los recursos y privilegios a su disposición sean muy reducidos (véase la Figura 4.4). A continuación, se describe la funcionalidad de cada una de estas clases.

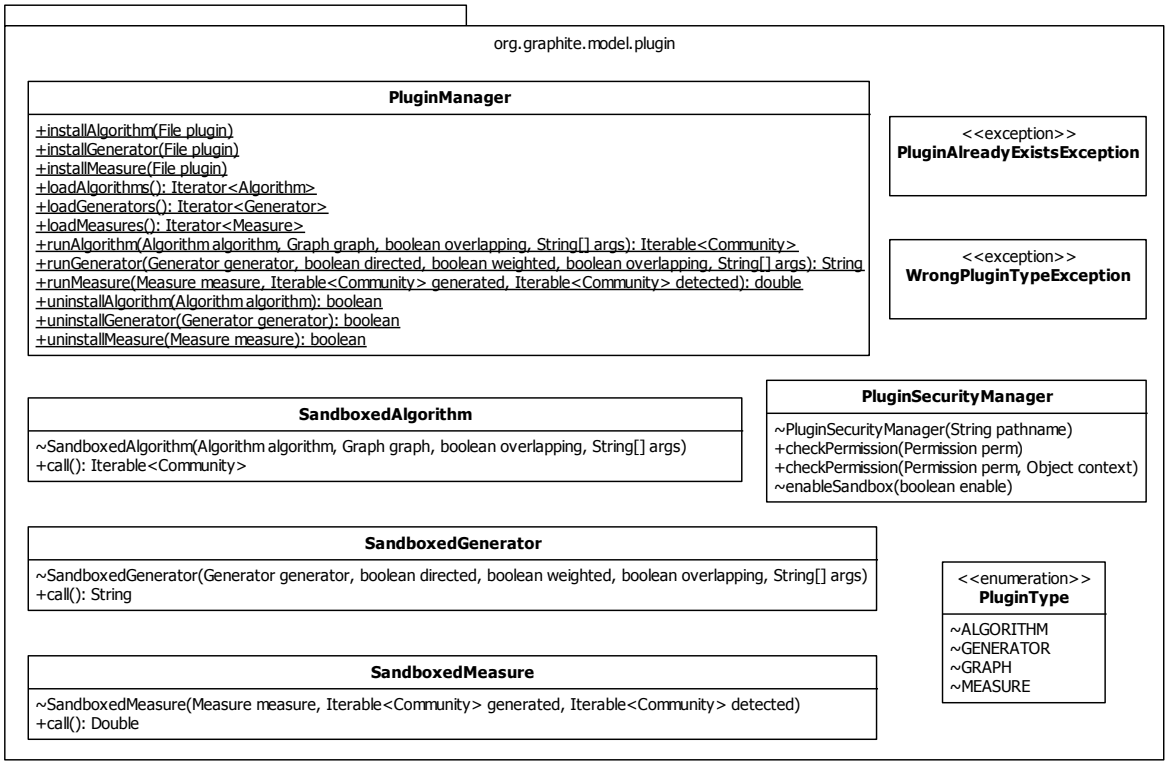


Figura 4.4: Diagrama de clases de ejecución segura de extensiones

Para construir el entorno seguro de ejecución, se ha utilizado la clase `SecurityManager` del paquete `java.lang`, la cual es capaz de determinar cuál es la operación que la aplicación se dispone a realizar e indicar si va a llevarse a cabo en un contexto en que su ejecución esté permitida. La clase `PluginSecurityManager` extiende la clase `SecurityManager` y sobrescribe los métodos *checkPermission* de la última para conceder los siguientes permisos a las extensiones suministradas:

- **Operaciones sobre ficheros:** Se admite la lectura de ficheros almacenados en el directorio de instalación de Graphite o en el directorio temporal del sistema operativo[9]. La escritura y eliminación de ficheros solamente están permitidas en el caso de los generadores de grafos, ya que el proceso de creación de la base de datos requiere de estos permisos.

[9] Esto es debido a que los generadores de grafos almacenan los grafos producidos en subdirectorios del directorio temporal del sistema operativo creados a tal efecto.

- **Operaciones sobre trazas de ejecución:** Se admite el control sobre la configuración de trazas de ejecución del paquete `java.util.logging` en el caso de los generadores de grafos, ya que el proceso de creación de la base de datos requiere de este permiso.
- **Operaciones reflexivas:** Las operaciones reflexivas son aquéllas que proporcionan la habilidad de acceder a atributos o invocar métodos de visibilidad no pública. Se debe extremar la precaución al conceder este permiso, por lo que se admite únicamente en el caso de los generadores de grafos, ya que el proceso de creación de la base de datos requiere de este permiso.

Una vez sobrescritos los métodos *checkPermission* concediendo los permisos anteriores, la clase `PluginSecurityManager` puede utilizarse como entorno seguro de ejecución de extensiones efectuando su activación o desactivación mediante el método *enableSandbox*. Mientras que Graphite se ejecuta en un contexto en que tiene a su disposición todos los recursos disponibles, las clases `SandboxedAlgorithm`, `SandboxedGenerator` y `SandboxedMeasure` hacen uso de la clase `PluginSecurityManager` para proporcionar un entorno seguro de ejecución a los algoritmos, generadores de grafos y medidas de evaluación suministrados, respectivamente. Asimismo, estas clases realizan la ejecución de las extensiones en hilos de ejecución independientes para garantizar su suspensión en caso de mostrar un comportamiento que, a pesar de considerarse seguro, sea anómalo (por ejemplo, bucles infinitos).

La clase `PluginManager` ofrece métodos para instalar y desinstalar, así como cargar y ejecutar de forma segura, extensiones que implementen las interfaces definidas en el subpaquete `org.graphite.spi` del paquete de interfaces para proveedores de extensiones (véase la Figura 4.2). Éstos son los métodos que invoca Graphite internamente para llevar a cabo la gestión de las extensiones.

El tipo enumerado `PluginType` representa los tipos de extensiones. Como ya se ha aseverado en varias ocasiones a lo largo de este documento, Graphite es compatible con los siguientes cuatro tipos de extensiones: (i) `ALGORITHM` o algoritmos de detección de comunidades, (ii) `GENERATOR` o generadores de grafos, (iii) `GRAPH` o grafos, y (iv) `MEASURE` o medidas de evaluación.

Por último, se definen dos excepciones que son susceptibles de ser lanzadas durante la instalación de una extensión: (i) `PluginAlreadyExistsException`, y (ii) `WrongPluginTypeException`. La primera se lanza cuando se intenta instalar una extensión cuyo nombre y versión son iguales a los de otra previamente instalada; la segunda se lanza cuando se intenta instalar una extensión cuyo tipo no es el esperado por la aplicación.

4.5. Entrada/Salida

Conocer todos los detalles involucrados en el proceso de comparación de algoritmos de detección de comunidades, pudiendo guardarlos y recuperarlos en cualquier momento, hace posible que la comparativa sea reproducible, favoreciendo así la objetividad y credibilidad de la misma. El paquete de entrada/salida de Graphite se encarga tanto de importar grafos individuales escritos en ficheros con formato DGML como de exportar los producidos por los generadores de grafos a este formato. DGML fue introducido por Microsoft en Visual Studio 2010 como lenguaje de definición de grafos. Está basado en XML y no sólo permite describir vértices y aristas, sino también asociar propiedades a ellos.

Por una parte, la clase `DGMLReader` del paquete de entrada/salida proporciona un mecanismo de traducción entre grafos escritos en ficheros con formato DGML y la interfaz `Graph` de Graphite. Para ello, hace uso de SAX[10], biblioteca de tratamiento de ficheros XML en Java, para leer los grafos definidos en ficheros con formato DGML y de la clase `BatchGraph` del paquete de almacenamiento en base de datos orientada a grafos para llevar a cabo la inserción de los vértices y aristas de dichos grafos. Por otra parte, la clase `DGMLWriter` del mismo paquete ofrece la funcionalidad inversa: permite exportar los grafos producidos por los generadores de grafos durante el proceso de comparación de algoritmos a ficheros con formato DGML. Para ello, hace uso de la clase `ReadOnlyGraph` del paquete de almacenamiento en base de datos orientada a grafos para llevar a cabo la lectura de los vértices y aristas de los grafos a exportar.

La Figura 4.5 muestra un ejemplo de fichero con formato DGML junto con su representación en forma de grafo. A pesar de que Graphite es compatible con el subconjunto de etiquetas mostrado en este ejemplo, impone la restricción de que el `Id` o identificador único de los vértices debe ser un número entero no negativo salvo en el caso del vértice con identificador `-1`, que indica la dirección y peso de las aristas del grafo. El grafo definido en este ejemplo consta de cuatro vértices, tres aristas y dos comunidades, declarados por las etiquetas `Node`, `Link` y `Category`, respectivamente. El resto de etiquetas es prácticamente consistente con la nomenclatura utilizada en este documento[11].

```
<?xml version="1.0" encoding="utf-8"?>
<DirectedGraph Title="Example">
   <Nodes>
      <Node Id="-1" Label="UNDIRECTED_WEIGHTED"  />
      <Node Id="0"  Label="v_0" Category="Black" />
      <Node Id="1"  Label="v_1" Category="White" />
      <Node Id="2"  Label="v_2" Category="Black" />
      <Node Id="3"  Label="v_3">
         <Category Ref="Black" />
         <Category Ref="White" />
      </Node>
   </Nodes>
   <Links>
      <Link Source="0" Target="1" Label="1" />
      <Link Source="1" Target="2" Label="3" />
      <Link Source="2" Target="3" Label="5" />
   </Links>
</DirectedGraph>
```

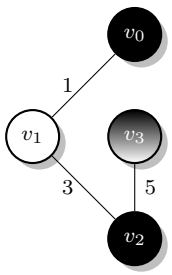


Figura 4.5: Ejemplo de fichero con formato DGML

4.6. Evaluación y pruebas

Una vez descritos el diseño e implementación de Graphite, es preciso establecer un conjunto de pruebas que permita llevar a cabo su evaluación verificando que satisface los requisitos que han guiado estas fases de su desarrollo. Los fallos detectados por este conjunto de pruebas han dado lugar a modificaciones progresivas en el diseño e implementación de la aplicación culminando con la ausencia de errores en el código de la misma. En esta sección, se presentan los tipos de pruebas realizados sobre la arquitectura de Graphite (para la evaluación de la interfaz gráfica de usuario véase el Capítulo 5).

[10] Siglas de *Simple API for XML*. Véase `http://www.saxproject.org/`

[11] La especificación completa puede encontrarse en `http://schemas.microsoft.com/vs/2009/dgml/`

En base al enfoque empleado, las pruebas realizadas para llevar a cabo la evaluación de la arquitectura de la aplicación se pueden catalogar en:

- **Pruebas de caja negra:** Se basan en proporcionar un conjunto de parámetros de entrada significativo a una funcionalidad determinada de la aplicación y comprobar si los valores obtenidos son consistentes con los esperados. Llevadas a la práctica de manera adecuada, permiten demostrar que la funcionalidad está bien implementada.
- **Pruebas de caja blanca:** Se centran en los detalles de implementación de las funciones internas de un módulo concreto de la aplicación, por lo que su diseño está íntimamente relacionado con ésta. Una de las técnicas más utilizadas es la cobertura de sentencias, la cual comprueba que cada instrucción del módulo se ejecute, al menos, una vez.

En el caso de las pruebas de caja negra, se utilizó JUnit[12], conjunto de bibliotecas para hacer pruebas unitarias en Java, para comprobar la funcionalidad de cada uno de los paquetes descritos en este capítulo. Como ya se ha aseverado al principio de esta sección, la aparición de fallos debida a la aplicación de estas pruebas dio lugar a modificaciones progresivas en el diseño e implementación de Graphite que culminaron con la ausencia de errores en su código.

Finalmente, en el caso de las pruebas de caja blanca, se utilizó EclEmma[13], extensión del entorno de desarrollo integrado Eclipse para hacer pruebas de cobertura, para conocer qué porcentaje de instrucciones de cada uno de los paquetes descritos en este capítulo se ejecutaba, al menos, una vez. El resultado de estas pruebas fue más que satisfactorio, arrojando un porcentaje muy cercano al 100 % para el conjunto de clases e interfaces de la aplicación completa.

[12] Véase `http://www.junit.org/`
[13] Véase `http://www.eclemma.org/`

Capítulo 5

Interfaz gráfica de usuario

Este capítulo describe la fase de evaluación de la interfaz gráfica de usuario con el fin de verificar que la aplicación ofrece, a través de ella, una interacción adecuada con los usuarios. Previamente, introduce los elementos más importantes de dicha interfaz, así como los aspectos de usabilidad que justifican su diseño.

5.1. Usabilidad

El requisito de usabilidad recogido en la Sección 3.2 establecía que, una vez implementada, la aplicación debería interaccionar con los usuarios a través de una interfaz gráfica de usuario que, ofreciendo la posibilidad de llevar a cabo cada una de las tareas descritas en la Sección 3.1, fuera capaz de mostrar los resultados de los procesos de evaluación y comparación de algoritmos de manera clara y concisa. Esta sección introduce los elementos más importantes de la interfaz gráfica de usuario, así como los aspectos que justifican su diseño.

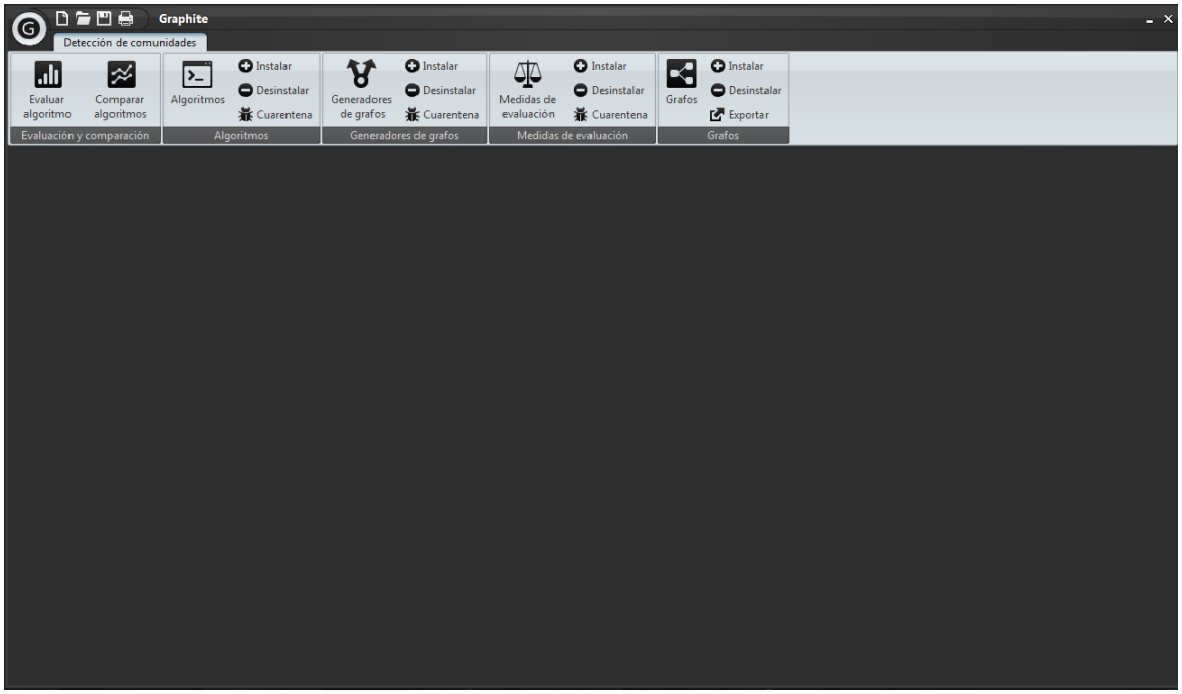


Figura 5.1: Ventana principal de Graphite

Figura 5.2: Cinta de opciones de Graphite

La Figura 5.1 muestra la ventana principal de Graphite una vez iniciado. Esta ventana consta de dos elementos bien diferenciados: (i) la cinta de opciones, situada en la parte superior, y (ii) el área de datos, que ocupa el resto de la misma. La cinta de opciones, que puede observarse detalladamente en la Figura 5.2, ofrece la posibilidad de llevar a cabo cada una de las tareas descritas en la Sección 3.1. Su diseño está basado en el introducido por Microsoft en Office 2007, el cual se beneficia de las siguientes ventajas:

- **Facilidad de aprendizaje:** En una aplicación con un número de opciones moderado, las cintas de opciones permiten condensar y presentar las opciones disponibles de manera sencilla y ordenada reduciendo la curva de aprendizaje.
- **Facilidad de uso:** La disposición estratégica de la información en las cintas de opciones hace posible que los usuarios tengan que realizar un conjunto de pasos reducido para llevar a cabo una tarea determinada.
- **Flexibilidad:** La ausencia de rigidez en el diseño de las cintas de opciones posibilita su adaptación a una gran variedad de escenarios de interacción entre los usuarios y la aplicación.
- **Robustez:** Las cintas de opciones permiten facilitar el cumplimiento de los objetivos de los usuarios mediante la adecuación de su contenido a la tarea concreta que éstos estén desempeñando.

Por otra parte, el área de datos muestra la información correspondiente a las extensiones instaladas en la aplicación acomodándose dinámicamente al contenido seleccionado por los usuarios. Su diseño se inspira en el lenguaje de diseño Metro, utilizado en varios productos de Microsoft como el futuro Windows 8, que hace posible mostrar la información de manera clara y concisa. En las secciones subsiguientes, se ofrecen dos conjuntos de pruebas que verifican que la interfaz gráfica de usuario ofrece una interacción adecuada con los usuarios.

5.2. Gestión de extensiones

La gestión de extensiones constituye el primer escenario de interacción entre Graphite y sus usuarios (véase la Sección 3.1), siendo la instalación de extensiones el requisito de mayor complejidad. A continuación, se presenta de manera visual el proceso de evaluación llevado a cabo para probar la funcionalidad y robustez de esta tarea.

Para instalar una extensión, el usuario deberá hacer clic en el botón Instalar asociado a alguno de los tipos de extensiones presentados en la cinta de opciones. Así, se le abrirá el selector de ficheros mostrado en la Figura 5.3. Éste filtra los ficheros con formato JAR, de manera que la selección de un fichero con otro formato dará lugar a un error en la instalación.

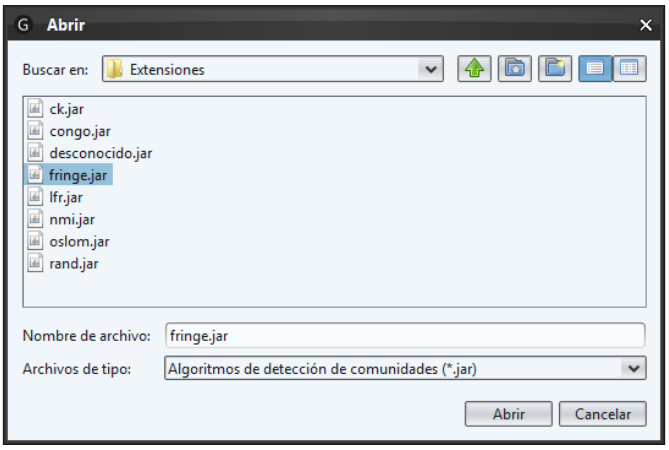


Figura 5.3: Instalación de una extensión

Seguidamente, Graphite comprobará si la extensión seleccionada ya está instalada y si su tipo se corresponde con el tipo de extensión que se desea instalar. En caso de que alguno de los resultados de estas comprobaciones entre en conflicto con los esperados por Graphite, la aplicación exhibirá el mensaje de error mostrado en la Figura 5.4 y cancelará la instalación.

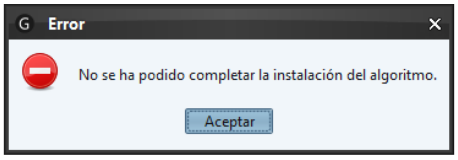


Figura 5.4: Error durante la instalación de una extensión

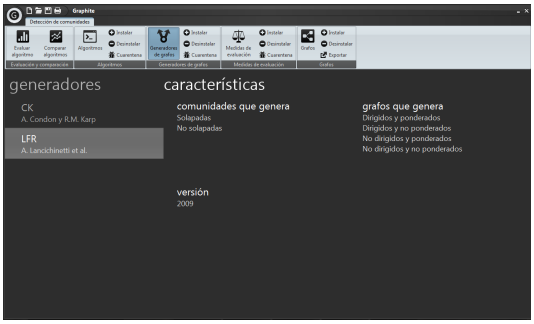
Una vez instaladas las extensiones, el área de datos correspondiente a cada tipo de extensión se actualizará con la información pertinente. La Figura 5.5 muestra el aspecto del área de datos de los algoritmos instalados, el cual presenta cuatro características asociadas a ellos: (i) comunidades que detecta, (ii) grafos que soporta, (iii) determinismo, y (iv) versión. Asimismo, en la parte izquierda, se detalla la lista de algoritmos instalados junto con su nombre y los apellidos de sus autores. La Figura 5.6 muestra las áreas de datos del resto de tipos de extensiones.

El área de datos de los generadores de grafos instalados presenta tres características asociadas a ellos: (i) comunidades que genera, (ii) grafos que genera, y (iii) versión. Asimismo, en la parte izquierda se detalla la lista de generadores de grafos instalados junto con su nombre y los apellidos de sus autores.

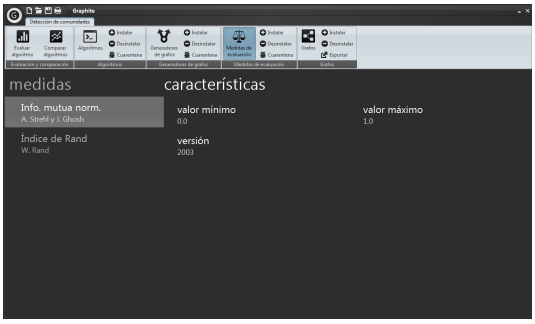
De manera análoga, el área de datos de las medidas de evaluación instaladas presenta tres características asociadas a ellas: (i) valor mínimo, (ii) valor máximo, y (iii) versión. Al igual que en los casos anteriores, en la parte izquierda se detalla la lista de medidas de evaluación instaladas junto con su nombre y los apellidos de sus autores.

Por último, el área de datos de los grafos importados presenta cuatro características asociadas a ellos: (i) dirección de sus aristas, (ii) peso de las mismas, (iii) número de vértices, y (iv) número de aristas. Al contrario que en casos anteriores, en la parte izquierda se detalla la lista de grafos importados junto con, únicamente, su nombre.

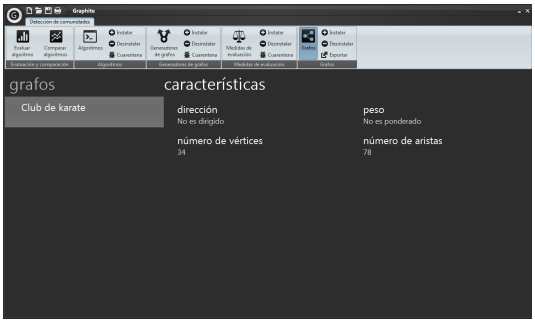
Figura 5.5: Área de datos de los algoritmos instalados



(a) Generadores de grafos



(b) Medidas de evaluación



(c) Grafos

Figura 5.6: Áreas de datos del resto de tipos de extensiones

5.3. Evaluación y comparación de algoritmos

La evaluación y comparación de algoritmos constituyen el segundo y último escenario de interacción entre Graphite y sus usuarios (véase la Sección 3.1). A continuación, se presenta de manera visual el proceso de evaluación llevado a cabo para probar la funcionalidad y robustez de estas tareas.

Figura 5.7: Configuración del algoritmo FRINGE

Para evaluar un algoritmo, el usuario deberá seleccionar uno de los ofrecidos en el área de datos de los algoritmos instalados (por ejemplo, el algoritmo FRINGE de Palazuelos y Zorrilla, 2012) y, después, hacer clic en el botón Evaluar algoritmo de la cinta de opciones. Así, la aplicación exhibirá el cuadro de diálogo mostrado en la Figura 5.7. Éste solicita al usuario el tipo de comunidades a detectar, así como el tipo de los grafos a producir por los generadores de grafos, teniendo en cuenta las características del algoritmo. Seguidamente, el usuario deberá introducir los argumentos del algoritmo en caso de que éste lo precise.

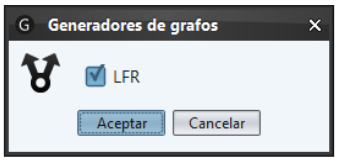


Figura 5.8: Elección de los generadores de grafos

A continuación, el usuario deberá seleccionar los generadores de grafos con que desee producir los grafos que participarán en el proceso de evaluación del algoritmo (véase la Figura 5.8, en que se elige el generador de grafos LFR de Lancichinetti y Fortunato, 2009a). Posteriormente, el usuario deberá establecer el número de grafos que desea generar indicando la cantidad de veces que se ejecutará el generador de grafos, así como los argumentos de éste en caso de que lo requiera (véase la Figura 5.9).

Figura 5.9: Configuración del generador de grafos LFR

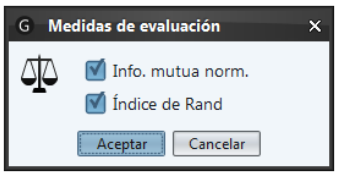


Figura 5.10: Elección de las medidas de evaluación

Por último, el usuario deberá indicar qué medidas de evaluación determinarán la calidad de las comunidades detectadas por el algoritmo. En este caso, se eligen el índice de Rand (1971) y la información mutua normalizada de Strehl y Ghosh (2003) (véase la Figura 5.10). Así, después de que la aplicación gestione los procesos de generación de grafos, detección de comunidades y evaluación de resultados, exhibirá una ventana similar a la mostrada en la Figura 5.11.

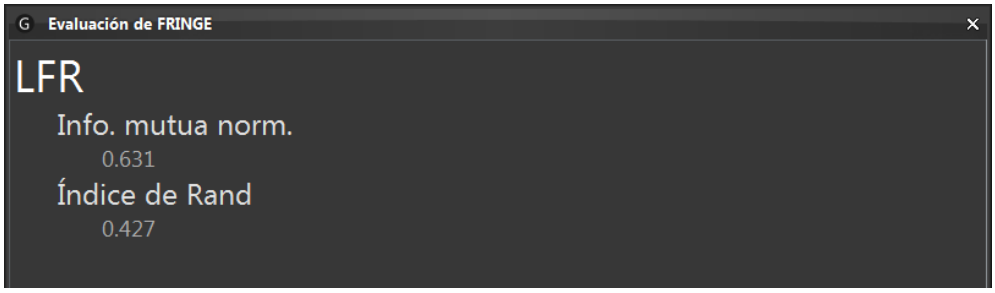


Figura 5.11: Resultados de la evaluación del algoritmo FRINGE

La ejecución de todas las extensiones involucradas en la evaluación de un algoritmo se lleva a cabo en un entorno seguro en que los recursos y privilegios a su disposición son muy reducidos. Para comprobar el funcionamiento de dicho entorno, se desarrolló un algoritmo, denominado Desconocido, cuyo método *run* invoca el método *exit* de la clase `java.lang.System`, considerado peligroso por Graphite por ser capaz de interrumpir su ejecución. De esta manera, si el usuario, en vez de FRINGE, hubiera elegido evaluar el algoritmo Desconocido, la aplicación habría cancelado su ejecución y exhibido el mensaje de error mostrado en la Figura 5.12.

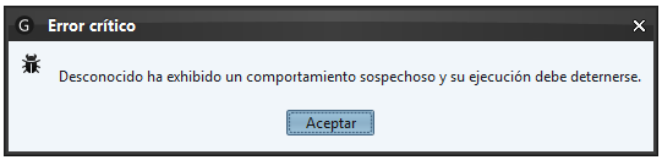


Figura 5.12: Fallo de seguridad en la ejecución de una extensión

Finalmente, para comparar dos o más algoritmos, el usuario deberá seleccionar uno de los ofrecidos en el área de datos de los algoritmos instalados (por ejemplo, el algoritmo FRINGE de nuevo) y, después, hacer clic en el botón Comparar algoritmos de la cinta de opciones. Así, la aplicación exhibirá el cuadro de diálogo mostrado en la Figura 5.13 teniendo en cuenta la compatibilidad del algoritmo a comparar con los contenidos en dicho cuadro. En este caso, se eligen los algoritmos CONGO de Gregory (2008) y OSLOM de Lancichinetti et al. (2011).

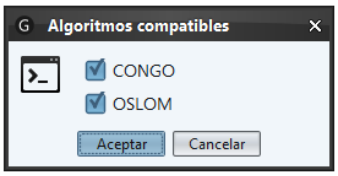


Figura 5.13: Algoritmos con que comparar FRINGE

A continuación, el usuario deberá introducir los argumentos de aquellos algoritmos que lo precisen. La Figura 5.14 muestra la configuración de CONGO. A partir de este punto, se lleva a cabo el proceso de evaluación descrito en esta sección para cada uno de los algoritmos elegidos. Así, después de que la aplicación gestione los procesos de generación de grafos, detección de comunidades y evaluación de resultados, exhibirá una ventana similar a la mostrada en la Figura 5.15.

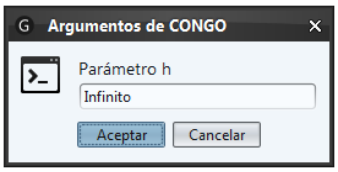


Figura 5.14: Configuración del algoritmo CONGO

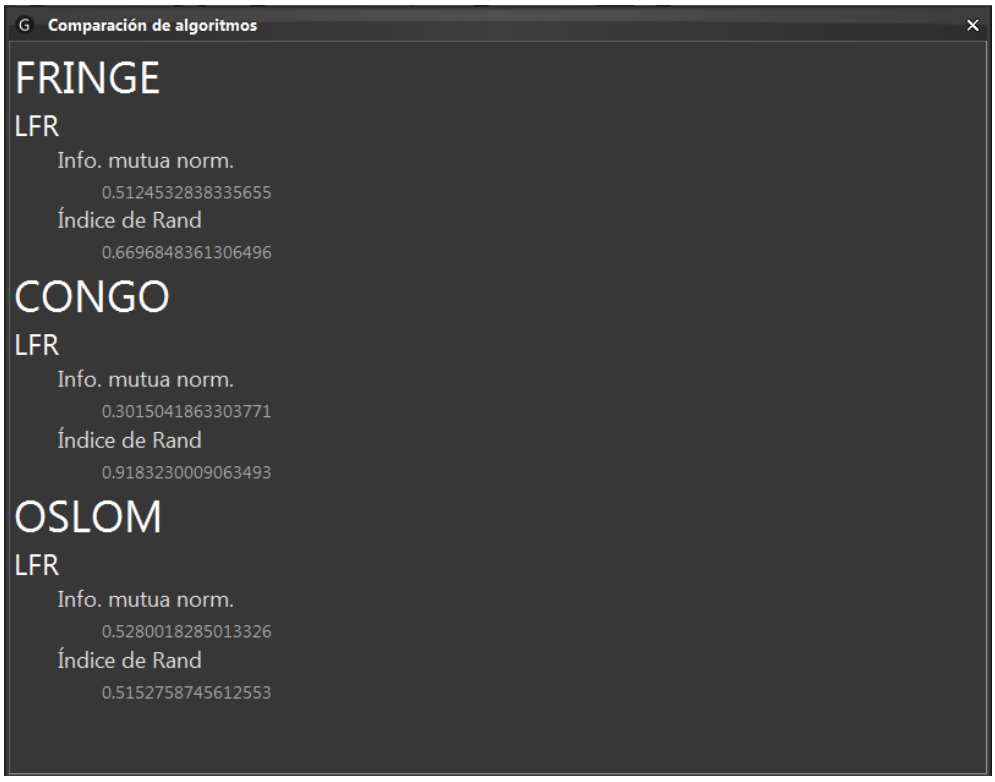


Figura 5.15: Resultados de la comparación de los algoritmos elegidos

Capítulo 6

Conclusiones y trabajos futuros

Este capítulo aporta una perspectiva general de los objetivos alcanzados por la aplicación desarrollada en el presente Proyecto Fin de Carrera, así como de sus contribuciones al área de investigación de la detección de comunidades en grafos. Asimismo, describe las posibles mejoras en dicha aplicación perfilando, de esta manera, las líneas de trabajo futuras.

6.1. Conclusiones

El objetivo fundamental de este Proyecto Fin de Carrera consistía en diseñar e implementar una aplicación multiplataforma, Graphite, que permitiera llevar a cabo comparaciones empíricas de algoritmos de detección de comunidades de manera imparcial y exhaustiva. Para lograr este objetivo, se han realizado la captura y análisis de requisitos, el diseño e implementación, y la evaluación y pruebas de la aplicación desarrollada, pudiendo concluir que se ha alcanzado dicho objetivo de manera satisfactoria. En esta sección, se introducen las características más importantes implementadas en la aplicación.

Graphite se ha diseñado de manera que los usuarios puedan añadirle, mediante un mecanismo de extensiones, cuatro tipos de elementos: (i) algoritmos, (ii) generadores de grafos, (iii) medidas de evaluación, y (iv) grafos individuales escritos en ficheros con formato DGML. Para hacerlo posible, ha sido necesario definir un conjunto de interfaces públicas que modelen de forma completa todos los elementos que intervienen en los procesos de evaluación y comparación de algoritmos de detección de comunidades. Esto proporciona un modelo estándar que simplifica dramáticamente el desarrollo de extensiones.

Para llevar a cabo un almacenamiento y recuperación eficientes de los grafos producidos por los generadores de grafos, así como de los suministrados en ficheros con formato DGML, Graphite hace uso de Neo4j, un sistema de gestión de bases de datos orientadas a grafos con persistencia transaccional que permite gestionar la información en forma de grafos. Por ello, ha sido necesario definir un conjunto de clases que actúe como capa intermedia entre las estructuras de datos usadas por Graphite y Neo4j, de forma que proporcione un protocolo de comunicación entre ambos.

Graphite es capaz de impedir que, a través de una extensión, un usuario malintencionado pueda ejecutar código malicioso que comprometa potencialmente la integridad de la máquina. Para evitarlo, ha sido necesario definir un conjunto de clases que permita ejecutar cada tipo de extensión en un entorno seguro en que los recursos y privilegios a su disposición sean muy reducidos.

Graphite interacciona con sus usuarios a través de una interfaz gráfica de usuario que, además de ofrecer la posibilidad de llevar a cabo cada una de las tareas implementadas en la aplicación, es capaz de mostrar los resultados de los procesos de evaluación y comparación de algoritmos de manera clara y concisa. En su implementación, se han tenido en cuenta cuatro aspectos primordiales relativos a su usabilidad: (i) facilidad de aprendizaje, (ii) facilidad de uso, (iii) flexibilidad, y (iv) robustez.

Con todo, la aportación más importante de Graphite al área de investigación de la detección de comunidades es que supone un gran avance con respecto al tipo de análisis comparativos que se han llevado a cabo hasta ahora, ya que, al poder guardar y recuperar los resultados del proceso de comparación de algoritmos, incluyendo todos los detalles involucrados en el mismo, se hace posible que la comparativa sea reproducible, favoreciendo así la objetividad y credibilidad de la misma.

6.2. Trabajos futuros

A pesar de que actualmente la aplicación cubre por completo los procesos de evaluación y comparación de algoritmos de detección de comunidades, en un futuro inmediato será necesario implantar una serie de mejoras que garantice el éxito de su fase de despliegue. Así, se proponen las siguientes líneas de trabajo futuras:

- **Documentación y ayuda integradas:** Actualmente, se está empezando a desarrollar un sistema de documentación y ayuda integrado en la aplicación que permita a Graphite orientar a los usuarios a través de los procesos de evaluación y comparación de algoritmos de detección de comunidades.
- **Lenguajes de definición de grafos:** Graphite permite importar grafos individuales escritos en ficheros con formato DGML, así como exportar los grafos producidos por los generadores de grafos a este formato. En el futuro, sería beneficioso extender esta funcionalidad para contemplar el uso de un número mayor de formatos.
- **Publicación:** La fase de despliegue incluirá conversaciones con investigadores del área de la detección de comunidades para conocer su grado de interés en Graphite, así como su disposición a adaptar sus algoritmos para ser compatibles con él. Para facilitar esta interacción, se desarrollará un sitio web que permita descargar la aplicación.
- **Representación gráfica de resultados:** Graphite muestra los resultados de las evaluaciones y comparaciones de algoritmos en forma de texto. En el futuro, sería apropiado añadir la posibilidad, sin dejar de ofrecer la representación textual, de presentar dichos resultados a través de distintos tipos de gráficas.

Bibliografía

Cazabet, R., Amblard, F., y Hanachi, C. Detection of Overlapping Communities in Dynamical Social Networks. En *Proc. of the 2010 IEEE International Conference on Social Computing (SocialCom)*, páginas 309–314, 2010.

Chen, J. y Yuan, B. Detecting Functional Modules in the Yeast Protein–Protein Interaction Network. *Bioinformatics*, 22(18):2283–2290, 2006.

Clauset, A., Newman, M., y Moore, C. Finding Community Structure in Very Large Networks. *Physical Review E*, 70(6):066111, 2004.

Condon, A. y Karp, R. Algorithms for Graph Partitioning on the Planted Partition Model. *Random Structures and Algorithms*, 18(2):116–140, 2001.

Danon, L., Díaz-Guilera, A., Duch, J., y Arenas, A. Comparing Community Structure Identification. *Journal of Statistical Mechanics: Theory and Experiment*, 2005(09):P09008, 2005.

Danon, L., Díaz-Guilera, A., y Arenas, A. The Effect of Size Heterogeneity on Community Identification in Complex Networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2006(11):P11010, 2006.

Duch, J. y Arenas, A. Community Detection in Complex Networks using Extremal Optimization. *Physical Review E*, 72(2):027104, 2005.

Fan, Y., Li, M., Zhang, P., Wu, J., y Di, Z. Accuracy and Precision of Methods for Community Identification in Weighted Networks. *Physica A: Statistical Mechanics and its Applications*, 377(1):363–372, 2007.

Fortunato, S. Community Detection in Graphs. *Physics Reports*, 486(3-5):75–174, 2010.

Fortunato, S. y Barthélemy, M. Resolution Limit in Community Detection. *Proceedings of the National Academy of Sciences*, 104(1):36, 2007.

Freeman, L. A Set of Measures of Centrality based on Betweenness. *Sociometry*, 40(1):35–41, 1977.

Girvan, M. y Newman, M. Community Structure in Social and Biological Networks. *Proceedings of the National Academy of Sciences*, 99(12):7821, 2002.

Gregory, S. An Algorithm to Find Overlapping Community Structure in Networks. En *Proc. of the 2007 European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, páginas 91–102, 2007.

Gregory, S. A Fast Algorithm to Find Overlapping Communities in Networks. En Daelemans, W., Goethals, B., y Morik, K., editores, *Machine Learning and Knowledge Discovery in Databases*, volumen 5211 de *Lecture Notes in Computer Science*, páginas 408–423. Springer, 2008.

Guimerà, R., Sales-Pardo, M., y Amaral, L. Modularity from Fluctuations in Random Graphs and Complex Networks. *Physical Review E*, 70(2):025101, 2004.

Hastie, T., Tibshirani, R., y Friedman, J. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics, 2001.

Homans, G. *The Human Group*. Harcourt, Brace & Co., 1950.

Lancichinetti, A. y Fortunato, S. Benchmarks for Testing Community Detection Algorithms on Directed and Weighted Graphs with Overlapping Communities. *Physical Review E*, 80 (1):016118, 2009a.

Lancichinetti, A. y Fortunato, S. Community Detection Algorithms: A Comparative Analysis. *Physical Review E*, 80(5):056117, 2009b.

Lancichinetti, A. y Fortunato, S. Limits of Modularity Maximization in Community Detection. *Physical Review E*, 84(6):066122, 2011.

Lancichinetti, A., Fortunato, S., y Radicchi, F. Benchmark Graphs for Testing Community Detection Algorithms. *Physical Review E*, 78(4):046110, 2008.

Lancichinetti, A., Radicchi, F., Ramasco, J., y Fortunato, S. Finding Statistically Significant Communities in Networks. *PLoS ONE*, 6(4):e18961, 2011.

Luce, R. Connectivity and Generalized Cliques in Sociometric Group Structure. *Psychometrika*, 15(2):169–190, 1950.

Lusseau, D., Schneider, K., Boisseau, O., Haase, P., Slooten, E., y Dawson, S. The Bottlenose Dolphin Community of Doubtful Sound Features a Large Proportion of Long-lasting Associations. *Behavioral Ecology and Sociobiology*, 54(4):396–405, 2003.

Massen, C. y Doye, J. Identifying Communities within Energy Landscapes. *Physical Review E*, 71(4):046101, 2005.

Medus, A., Acuña, G., y Dorso, C. Detection of Community Structures in Networks via Global Optimization. *Physica A: Statistical Mechanics and its Applications*, 358(2):593–604, 2005.

Meilă, M. Comparing Clusterings – An Information based Distance. *Journal of Multivariate Analysis*, 98(5):873–895, 2007.

Moody, J. y White, D. Structural Cohesion and Embeddedness: A Hierarchical Concept of Social Groups. *American Sociological Review*, 68(1):103–127, 2003.

Newman, M. Coauthorship Networks and Patterns of Scientific Collaboration. *Proceedings of the National Academy of Sciences of the United States of America*, 101(Suppl. 1):5200, 2004a.

Newman, M. Fast Algorithm for Detecting Community Structure in Networks. *Physical Review E*, 69(6):066133, 2004b.

Newman, M. Modularity and Community Structure in Networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, 2006.

Newman, M. y Girvan, M. Finding and Evaluating Community Structure in Networks. *Physical Review E*, 69(2):026113, 2004.

Palazuelos, C. y Zorrilla, M. FRINGE: A New Approach to the Detection of Overlapping Communities in Graphs. En Murgante, B., Gervasi, O., Iglesias, A., Taniar, D., y Apduhan, B., editores, *Computational Science and its Applications - ICCSA 2011*, volumen 6784 de *Lecture Notes in Computer Science*, páginas 638–653. Springer, 2011.

Palazuelos, C. y Zorrilla, M. Analysis of Social Metrics in Dynamic Networks: Measuring the Influence with FRINGE. En *Proc. of the 2012 EDBT/ICDT Workshops*, páginas 9–12, 2012.

Palla, G., Derényi, I., Farkas, I., y Vicsek, T. Uncovering the Overlapping Community Structure of Complex Networks in Nature and Society. *Nature*, 435(7043):814–818, 2005.

Pinney, J. y Westhead, D. Betweenness-based Decomposition Methods for Social and Biological Networks. *Interdisciplinary Statistics and Bioinformatics*, páginas 87–90, 2006.

Rand, W. Objective Criteria for the Evaluation of Clustering Methods. *Journal of the American Statistical Association*, 66(336):846–850, 1971.

Reenskaug, T. Thing–Model–View–Editor: An Example from a Planningsystem. *Xerox PARC Technical Note*, 1979.

Rees, B. y Gallagher, K. Overlapping Community Detection by Collective Friendship Group Inference. En *Proc. of the 2010 International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, páginas 375–379, 2010.

Rice, S. The Identification of Blocs in Small Political Bodies. *The American Political Science Review*, 21(3):619–627, 1927.

Rosvall, M. y Bergstrom, C. Maps of Random Walks on Complex Networks Reveal Community Structure. *Proceedings of the National Academy of Sciences*, 105(4):1118, 2008.

Sawardecker, E., Sales-Pardo, M., y Amaral, L. Detection of Node Group Membership in Networks with Group Overlap. *The European Physical Journal B-Condensed Matter and Complex Systems*, 67(3):277–284, 2009.

Scott, J. *Social Network Analysis: A Handbook*. SAGE Publications, 2000.

Stanoev, A., Smilkov, D., y Kocarev, L. Identifying Communities by Influence Dynamics in Social Networks. *Physical Review E*, 84(4):046102, 2011.

Strehl, A. y Ghosh, J. Cluster Ensembles – A Knowledge Reuse Framework for Combining Multiple Partitions. *Journal of Machine Learning Research*, 3(1):583–617, 2003.

Wasserman, S. y Faust, K. *Social Network Analysis: Methods and Applications*. Structural Analysis in the Social Sciences. Cambridge University Press, 1994.

Weiss, R. y Jacobson, E. A Method for the Analysis of the Structure of Complex Organizations. *American Sociological Review*, 20(6):661–668, 1955.

Zachary, W. An Information Flow Model for Conflict and Fission in Small Groups. *Journal of Anthropological Research*, 33(4):452–473, 1977.