MASTER EN DATA SCIENCE

Curso Académico 2018/2019

Trabajo Fin de Master

Divider Greedy algorithm for performing community detection in social networks

Autor : Michel Maes Bermejo

Tutor : Jesús Sánchez-Oro Calvo

Resumen

En este proyecto abordaremos la creación de un algoritmo capaz de detectar comunidades en redes sociales, un problema en el que los algoritmos tradicionales no ofrecen soluciones rápidas. Para ello usaremos un enfoque metaheurístico basado en la división de comunidades e intentaremos mejorar sus prestaciones a lo largo de varias iteraciones. Una vez obtenida la mejor versión de nuestro algoritmo, lo compararemos con los algoritmos existentes para validar su calidad.

Índice general

Capítulo 1

Introducción y motivación

Desde su nacimiento, Internet ha logrado conectar a las personas de forma sencilla. El ejemplo más claro son las redes sociales, que han crecido de forma contundente los últimos años. La rápidez con la que se transmite la información ha provocado que muchas personas las utilicen como medio de información de referencia frente a los tradicionales. Esto ha desencadenado un creciente interés de diferentes marcas e incluso de otros medios de aprovechar este fenómeno para su beneficio, aprovechando la estructura de red (o grafo) en la que se basa.

Desde el punto de vista de un científico de datos, resulta interesante estudiar la estructura que formas estas redes, que constituyen un ejemplo perfecto de un grafo. Esto ha generado un creciente interés en la comunidad investigadora, desarrollandose nuevas lineas de investigación en torno al estudio de las redes sociales. Algunos ejemplos de estas lineas son:

- La recomendación de perfiles a seguir o contenido de otros usuarios afines.
- Detección de información viral
- Detección de comunidades

En este proyecto nos centraremos en la detección de comunidades, una de las áreas más interesantes que cuenta con multitud de aplicaciones.

Actualmente existen multitud de algoritmos que abordan el problema de la detección de comunidades. Los más tradicionales y exactos no son viables para abordar los grandes volumenes de datos que generan las redes sociales, por lo que se ha optado por soluciones heuristicas, no tan exactas, pero mucho más rápidas.

Las soluciones de que ofrecen estos algoritmos por su solapamiento (si permitimos que un nodo del grafo pertenezca a más de una comunidad o no) y su dinamismo (si la red sobre la que trabajamos cambia a lo largo del tiempo). Además, dentro de la detección de comunidades existen dos enfoques contrapuestos; el aglomerativo (todos los nodos comienzan en su propio clúster y se van uniendo) y el divisor (todos los nodos comienzan en el mismo clúster).

La motivación de este proyecto nace de la idea de crear una solución heurística diferente a las preexistentes (normalmente basadas en algun algun algoritmo aglomerativo), optando por un enfoque divisor. La solución propuesta será sin solapes sobre redes estáticas.

Capítulo 2

Objetivos

El objetivo principal de este proyecto será la creación de un algoritmo de detección de comunidades. Para ello utilizaremos un enfoque destructivo (comenzar con todos los nodos del grafo en la misma comunidad e ir haciendo particiones) contrario a un enfoque más utilizado, constructivo (comenzar con cada nodo en su propia comunidad e ir agrupandolos). Durante la construcción, usaremos la modularidad como métrica para evaluar nuestra solución. Realizaremos una serie de iteraciones en las cuales iremos mejorando el algoritmo con el fin de obtener la mejor versión del mismo.

Finalmente, compararemos nuestro algoritmo con otros preexistentes para valorar si es efectivo el enfoque elegido. En esta fase usaremos otras métricas sobre las particiones obtenidas como la conductancia o covertura.

Otros objetivos que perseguiremos serán:

- Aprender a trabajar con metaheurísticas, estrategias genéricas para resolver problemas de computación.
- Aprender a trabajar de manera avanzada con grafos.
- Aprender a solucionar problemas abstractos modelizandolos.
- Ampliar el conocimiento en el area de la detección de comunidades.

Capítulo 3

Descripción algorítmica

En este capítulo se abordará el algoritmo desarrollado así cómo las distintas iteraciones que dieron lugar al mismo.

La detección de comunidades es un problema que se aborda haciendo uso de grafos, estructuras de datos que modelizan una red de elementos.

Definición de Grafo

Un grafo $G = (V, E)$ se define como el conjunto no vacio de n nodos o vértices (V) unidos por un conjunto de m enlaces, denominados aristas o arcos (E). Una arista $(u, v)\epsilon E$ con $u, v\epsilon V$ representa la conexión entre los nodos u y v.

Antes de comenzar, es importante señalar que para la creación de un algoritmo de detección de comunidades es necesario tomar una métrica como referencia, la cual utilizaremos para tomar decisiones en el algoritmo. Entre las métricas más comunes para este tipo de problema se encuentran la cobertura, la conductancia y la modularidad. Para este algoritmo usaremos la modularidad cómo métrica a maximizar.

Modularidad

Las soluciones de los algoritmos de detección de comunidades se construyen máximizando su valor de modularidad, una medida utilizada para medir la fuerza de division de un grafo en clústers. Esta métrica se define cómo la fracción de enlaces que caen dentro de los clústers dados menos el valor esperado que dicha fracción hubiese recibido si los enlaces se hubiesen distribuido al azar. De forma matemática:

$$Md(S,G) = \sum_{j=1}^{max(S)} (e_{jj} - a_j^2)$$

dónde S es la solución y G el grafo, siendo e_{jj} la fracción de enlaces con ambos vértices finales en el mismo grupo:

$$e_{jj} = \frac{|\{(v,u)\epsilon E : S_v = S_u = j\}|}{|E|}$$

y a_j la fracción de enlaces con al menos un extremo en la misma comunidad:

$$a_j = \frac{|\{(v,u)\epsilon E : S_v = j\}|}{|E|}$$

siendo en ambos caso E el conjunto de las aristas de G.

La modularidad toma valor en el intervalo [-0.5,1)

A continuación, se detallará la construcción incremental del algoritmo, explicando cada versión creada.

Las premisas para todos los algoritmos son las siguientes:

- Se comienza con todos en la misma comunidad.
- Se asume un grafo conexo

Grafo conexo

Un grafo conexo $\overline{G}$ es todo grafo G dónde para cada par de vertices (x,y) existe un camino P.

3.1. ConstRandom: Algoritmo aleatorio

En esta primera aproximación (Algoritmo 1) que nos sirve como punto de partida y para posterior comparación, generamos los clústers de manera aleatoria; se crea un número aleatorio de clústers y se distribuyen los nodos de forma aleatoria en ellos.

Algorithm 1 ConsRandom algorithm

```
1: procedure CONSTRANDOM(G)                          ▷ The graph 'G'
2:     S ← EmptySolution(G)
3:     numClusters ← Random(N(g))                    ▷ N returns nº of graph nodes
4:     for i = 0 to numClusters do
5:         createEmptyCluster(S)
6:     for i = 0 to N(S) do
7:         rnd ← Random(numClusters)
8:         AssignToCluster(S, i, rnd)    ▷ Assign node i to cluster rnd in solution S
9:     return S
```

3.2. ConstDivider: Primer aproximación

Este algoritmo supone el primer intento de aplicar un enfoque divisor a la detección de comunidades. Los pasos del algoritmo a seguir son se detallan a continuación (Ver junto al Algoritmo 2).

En el primer paso, comenzamos con una sola comunidad que contiene todos los nodos. A continuación, se busca el nodo peor conectado (menor número de aristas) ó el primero que encuentre de una arista. Se crea un nuevo clúster con ese nodo y se añaden los adyacentes en la medida en la que aumente la modularidad. Se descartan los adyacentes de los nodos los cuales no han mejorado la modularidad. De esta manera, se crearán dos clústers (A y B). Repetimos los el proceso con el clúster B hasta que no se generen nuevos clusters.

3.3. ConstDividerGreedy: Segunda aproximación

El problema del anterior algortimo es que una mala elección del primer nodo puede provocar que no se formen correctamente las comunidades. Para solucionar este problema, usaremos la meta-heurística GRASP.

Algorithm 2 ConstDivider algorithm

procedure CONSTDIVIDER(G) ▷ The graph 'G'
  $S \leftarrow EmptySolution(G)$
  $S' \leftarrow S$
  $c \leftarrow 0$ ▷ Current cluster index
  while $c < Size(S')$ **do**
    $K \leftarrow EmptySet()$ ▷ Checked Nodes
    $S'' \leftarrow S'$
    $createEmptyCluster(S'')$
    $wcn < -GetWorstConnectedNode(S'')$
    $Move(wcn, c+1, S'')$ ▷ Move node wcn to cluster $c+1$ in solution S''
    $A \leftarrow Adyacents(G, wcn)$
    while $Size(A) > 0$ **do**
      $i < -Poll(A)$
      $K < -Append(K, i)$
      $Move(i, c+1, S'')$
      if $Md(S'', G) > Md(S', G)$ **then**
        $S' < -S''$
      for $j \epsilon Adyacents(i)$ **do**
        if $j \notin K \ \ \& \ \ j \notin Cluster(S'', c)$ **then**
          $A < -Remove(A, node)$
          $K < -Append(K, node)$
    if $Md(S', G) > Md(S, G)$ **then**
      $S \leftarrow S'$
    else
      $c \leftarrow c + 1$
  return S

GRASP

GRASP (en inglés *Greedy Randomized Adaptive Search Procedure*) es una metaheurística basada en un procedimiento de búsqueda *greedy* o avaricioso, aleatorio y adaptativo, utilizado en probemas de optimización, que garantiza una solución buena, aunque no necesariamente la óptima. Puede dividirse en 3 fases:

Fase de construcción:

- Generar una lista de candidatos mediante la utilización de una función *greedy*.
- Generar una lista restringida de los mejores candidatos.
- Seleccionar aleatoriamente un elemento de la lista restringida
- Repetir el proceso hasta construir la solución.

Fase de mejora: Hacer un proceso de búsqueda local a partir de la solución construida hasta que no mejore más.

Fase de actualización: Si la solución obtenida mejora la existente, actualizarla.

En esta segunda aproximación, exploraremos el uso de la fase de construcción para elegir el siguiente nodo a mover, dejando fuera la búsqueda local. Aleatorizar la elección del primer nodo nos abre la puerta a realizar múltiples iteraciones comenzando desde distintos nodos. Los pasos a seguir en este algoritmo se detallan a continuación (Ver junto al Algoritmo 3).

Inicialmente, se elige el primer nodo de forma aleatoria, añadiendo, como antes, los nodos adyacentes si aumenta la modularidad. Se crea una lista de candidatos, que contiene a todos los vertices ordenados por el número de aristas que vayan a otro clúster de mayor a menor. A partir de una funcion voraz (Ver Funcion 3.3), obtenemos un umbral mu, que restringe la lista a los vértices con un número de vertices mayor que mu. El siguiente vértice se selecciona al azar de esta lista restringida. Se repite este proceso hasta que no mejora la modularidad.

Función Voraz

Dado g_{max} y g_{min} cómo los valores mayor y menos de nuestra función voraz (el número de aristas) y α un parámetro en el rango [0,1] (que indicará cómo de voraz es nuestro algoritmo, siendo 1 completamente aleatorio y 0 totalmente voraz), el umbral mu se calcula siguiendo la siguiente función:

$$mu = g_{max} - \alpha * (g_{max} - g_{min})$$

3.4. ConstDividerGreedy + Local Search: Última versión

En este caso, aplicaremos la búsqueda local que dejamos fuera en la versión anterior. La búsqueda local utilizará la solución de cada iteración del algortimo anterior (ConstDividerGreedy) e intentará mejorarla cambiando algunos nodos de clúster. La búsqueda local implementada se detalla a continuación (Ver junto al Algoritmo 4).

Para cada clúster, se crea una lista de candidatos a moverse. Esta lista de candidatos estará ordenada por el % de aristas que caen fuera del clúster. Cada nodo de esta lista de candidatos se intenta mover a otro clúster (incluido a un nuevo clúster). Si la modularidad mejora, se actualiza la solución, pero se siguen explorando otros movimientos.

Algorithm 3 ConstDividerGreedy algorithm

```
1:  procedure CONSTDIVIDERGREEDY(G, α)                ▷ The graph 'G' and param 'α'
2:      S ← EmptySolution(G)
3:      S' ← S
4:      c ← 0                                          ▷ Current cluster index
5:      b ← GetRandom(S', c)                           ▷ Base node
6:      refactorIters ← 10
7:      while c < Size(S') & b ≠ −1 & refactorIters > 0 do
8:          K ← EmptySet()                             ▷ Checked Nodes
9:          S'' ← S'
10:         createEmptyCluster(S'')
11:         candidates ← RestrictedList(G, c, α)
12:         if Empty(candidates) then
13:             b ← GetRandom(candidates)
14:         else
15:             b ← GetRandom(S', c)
16:         if b = −1 then
17:             c ← 0
18:             refactorIters ← refactorIters − 1
19:             continue
20:         Move(b, c + 1, S'')                        ▷ Move node b to cluster c + 1 in solution S''
21:         A ← Adyacents(G, wcn)
22:         while Size(A) > 0 do
23:             i < −Poll(A)
24:             K < −Append(K, i)
25:             Move(i, c + 1, S'')
26:             if Md(S'', G) > Md(S', G) then
27:                 S' < −S''
28:             for jϵAdyacents(i) do
29:                 if j ∉ K & j ∉ Cluster(S'', c) then
30:                     A < −Remove(A, node)
31:                     K < −Append(K, node)
32:         if Md(S', G) > Md(S, G) then
33:             S ← S'
34:         c ← c + 1
35:     return S
```

Algorithm 4 LocalSearch algorithm

```
1:  procedure LocalSearch(S, G)                    ▷ The Solution 'S' and Graph 'G'
2:      i ← 0                                      ▷ Current cluster index
3:      while i < Size(S) do
4:          CL ← CandidateList(S, i)
5:          while Size(CL) > 0 do
6:              c ← Poll(CL)                       ▷ Current candidate
7:              S' ← S
8:              createEmptyCluster(S')
9:              j ← 0                              ▷ Target cluster
10:             while j < Size(S) do
11:                 if j = i then
12:                     j ← j + 1
13:                     continue
14:                 Move(c, j, S')                 ▷ Move node c to cluster j in solution S'
15:                 if Md(S', G) > Md(S, G) then
16:                     S < −S'
17:                 Move(c, i, S')                 ▷ Return node c to cluster i in solution S'
18:                 j ← j + 1
19:         i ← i + 1
20:     return S
```

Capítulo 4

Tecnologías, Herramientas y Metodologías

4.1. Tecnologías

4.1.1. Java

Java es un lenguaje de programación de propósito general, concurrente y orientado a objetos. Su sintaxis deriva en gran medida de C y C++. Uno de los principales atractivos de Java es su máquina virtual (JVM) que nos permite ejecutar nuestro código Java en cualquier dispositivo, independientemente de la arquitectura.

4.2. Herramientas

4.2.1. Control de versiones: Git

Git[1] es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente.

4.2.2. Entorno de desarrollo: IntelliJ

IntelliJ[2] es un IDE para Java desarollado por JetBrains ideado para mejorar la productividad del programador.

[1] `https://git-scm.com/`
[2] `https://www.jetbrains.com/idea/`

4.3. Metodologías

Dado que pretendemos obtener la mejor versión de nuestro algoritmo, el desarrollo seguirá una metodología iterativa-incremental[3](Figura 4.1). En cada iteración se tomará el algoritmo previo (si lo hay), se evaluarán sus limitaciones, se mejorará y se evaluará, obteniendo en cada iteración un versión mejorada del mismoo.

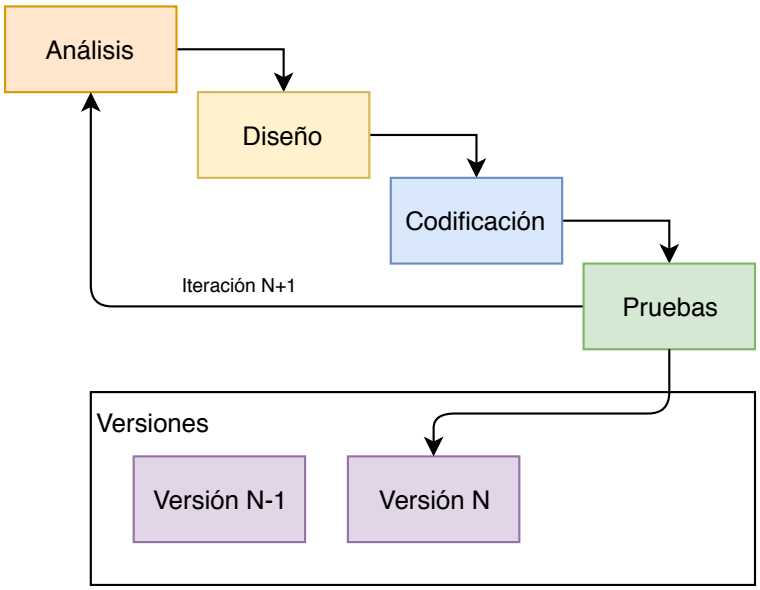

Figura 4.1: Ciclo de la metodología iterativa-incremental

[3]https://proyectosagiles.org/desarrollo-iterativo-incremental/

Capítulo 5

Descripción informática

En este apartado se abordará la construcción del proyecto. Este proyecto consta de una colección de algoritmos de detección de comunidades, que parte desde un algoritmo de detección aleatoria, pasando por la evolución de un algoritmo destructivo hasta obtener la mejor versión del mismo aplicando una búsqueda local. Todo el código generado se encuentra en Java y la documentación para ejecutarlo se encuentra en el Anexo 1.

5.1. Requisitos

5.1.1. Requisitos funcionales

Los requisitos funcionales de la aplicación son simples dado que se trata de un algoritmo:

- Debe poder leer un fichero de entrada (correspondiente al grafo a tratar).
- Dado esa entrada y un algoritmo, debe devolver los clústers resultantes al realizar la detección de comunidades.

5.1.2. Requisitos no funcionales

Los requisitos no funcionales de la aplicación se reducen a la calidad de la solución ofrecida y al tiempo de ejecución:

- Maximizar la modularidad.

- Ofrecer el menor tiempo de cálculo posible[1].

5.2. Implementación

5.2.1. Diseño de clases

Cómo podemos observar en la Figura 5.1, cada algoritmo desarrollado en el capítulo 3 está representado como una clase dentro de nuestro proyecto Java. Todas las clases implementan la interfaz *Constructive* que obliga la implementación de un método que transforma una instancia de partida (el objeto que representa al grafo) en una solución, que contiene la partición en comunidades del grafo.

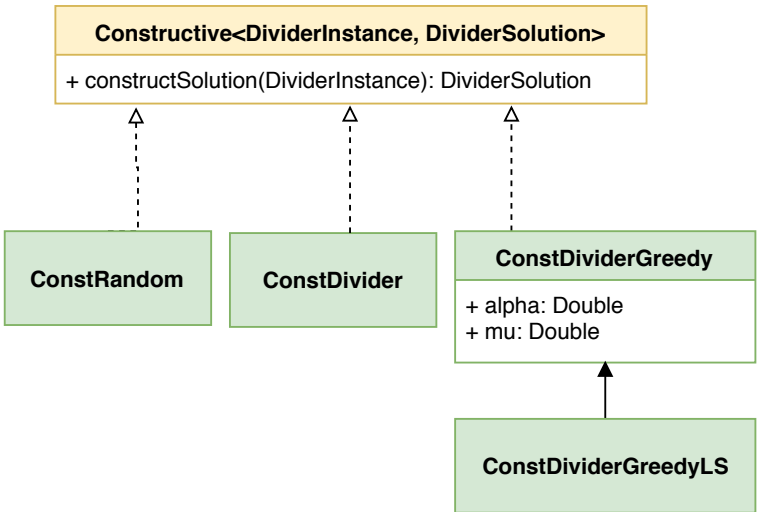

Figura 5.1: Diagrama de clases de la aplicación

La clase *ConstRandom* implementa el Algoritmo 1, dada una instancia devuelve una partición en comunidades generadas de forma aleatoria.

La clase *ConstDivider* implementa el Algoritmo 2, dada una instancia, devuelve una partición en comunidades maximizando la modularidad. Este resultado es determinista.

La clase *ConstDividerGreedy* implementa el Algoritmo 3, dada una instancia, genera una partición en comunidades maximizando la modularidad a partir de una meta-heurística greedy, sin la fase de mejora. Depende de una variable α, que limita la voracidad del algoritmo y de una

[1]Existen algoritmos que ofrecen soluciones óptimas a este problema, pero no son aplicables a grafos grandes dado que requieren demasiado tiempo de cómputo.

valor random, que lo hace no-determinista.

La clase *ConstDividerGreedyLS* implementa el Algoritmo 3. Esta clase genera una solución a partir de la clase *ConstDividerGreedy* y la mejora con una búsqueda local.

Por último, se implementa una clase adicional, *AlgorithmExecutor*, que nos permite ejecutar N veces un algoritmo. Esta clase solo tiene utilidad para los algoritmos no-deterministas, obteniendo la mejor partición en comunidades de las N iteraciones.

5.2.2. Paralelización

Una vez implementada la mejor versión de nuestro algoritmo, terminaremos el desarrollo procurando optimizar el uso de los recursos de la máquina dónde se ejecute el algoritmo. Se hará uso de la libreria estándar de java para el manejo de hilos, $java.util.concurrent$, concretamente de los Ejecutores ($Executors$), que nos permiten gestionar de forma sencilla y óptima la casuística de la programación concurrente. Las tareas a paralelizar de nuestro programa serán las iteraciones que realiza la clase *AlgorithmExecutor* sobre los algoritmos no-deterministas.

Para demostrar la eficacia de usar paralelización en este algoritmo, usaremos de ejemplo la ejecución del algoritmo más lento: $ConstDividerGreedy + LocalSearch$. La siguiente prueba se ha realizado en una máquina Ubuntu 18.04 con 16GB de RAM y 4 procesadores (sin *hyper-threading*).

Tiempos de ejecucción (en milisegundos) para 1000 iteraciones en una instancia de prueba (214 nodos y 1954 aristas):

- Secuencial: 446.187 ms
- Paralelo (4 hilos): 132.753 ms

Podemos observar que se reduce de forma considerable (hasta un 70 %) el tiempo de ejecución, atendiendo a nuestro requisito no funcional de ofrecer el menor tiempo de ejecución posible.

Capítulo 6

Estudio comparativo

El estudio comparativo que se llevará a cabo a continuación constará de dos secciones:

- La primera usará un conjunto de grafos que servirá para comparar los actuales algoritmos junto a los posibles valores de sus parámetros en términos de modularidad.
- La segunda usará otro conjunto de grafos (distintos al primero) que servirán para comparar la mejor versión de nuestro algoritmo con otras soluciones pre-existentes.

Para la realización del estudio, se ejecutaran los diferentes algoritmos en una máquina Ubuntu 18.04 con 16GB de RAM y 4 procesadores (sin *hyper-threading*).

6.1. Comparativa de algoritmos propios

6.1.1. Instancias y Algoritmos

En esta sección analizaremos los resultados de nuestros algoritmos para 5 instancias descritas a continuación en la Tabla 6.1, dónde N es el número de nodos y M el número de aristas de la instancia.

Para los algoritmos $DividerGreedy$ y $DividerGreedyLS$ ejecutaremos 100 iteraciones para cada los siguientes valores de α : 0.25, 0.5 y 0.75.

Instancia	N	M
1000_0.1network.txt	1000	10434
500_0.1network.txt	500	5337
500_0.2network.txt	500	4965
500_0.3network.txt	500	4842
1000_0.2network.txt	1000	10071

Tabla 6.1: Instancias seleccionadas para la primera comparativa

6.1.2. Resultados

Atendiendo al resumen de los resultados ofrecidos en la Tabla 6.2 podemos observar que el mejor algoritmo es $DividerGreedyLS$ con el valor $\alpha = 0{,}25$, el cual ofrece la mejor modularidad para 4 de las 5 instancias, con una desviación típica media muy baja (0.00016). Para ese mismo algoritmo, observamos que para valores de $\alpha = 0{,}5$ y $\alpha = 0{,}75$ la desviación típica media aumenta. Los datos ampliados de esta comparativa pueden encontrarse en la Tabla A.1 del Apéndice A.

Algoritmo	Dev(%)	Mejor modularidad (veces)
Random	1.00264	0/5
Divider	0.54658	0/5
DividerGreedy (α = 0.25)	0.29836	0/5
DividerGreedy (α = 0.5)	0.32129	0/5
DividerGreedy (α = 0.75)	0.31385	0/5
DividerGreedyLS (α = 0.25)	0.00016	4/5
DividerGreedyLS (α = 0.5)	0.00459	2/5
DividerGreedyLS (α = 0.75)	0.00342	0/5

Tabla 6.2: Resumen de la comparativa de algoritmos propios

6.2. Comparativa con algoritmos previos

6.2.1. Instancias y Algoritmos

Una vez hemos encontrado el mejor algoritmo de los desarrollados en este trabajo, en esta sección lo compararemos con los algoritmos previos de detección de comunidades. Los algoritmos seleccionados para la comparativa han sido Edge Betweenness [3], Fast Greedy [2], Walktrap [6], Eigenvectors [5], Spinglass [8], Label Propagation [7] y Multi-level [1], junto al mejor de nuestros algoritmos ($DividerGreedyLS$ con el valor $\alpha = 0{,}25$, ejecutado en 100 ite-

raciones). Para la ejecución de los algoritmos previos se ha hecho uso de la libreria de Python igraph [1]. Las instancias utilizadas para esta comparativa serán las descritas en la Tabla 6.3.

Instancia	**N**	**M**
500_0.4network.txt	500	10338
500_0.5network.txt	500	10310
500_0.6network.txt	500	10312
500_0.7network.txt	500	10232
500_0.8network.txt	500	10194
1000_0.3network.txt	1000	19432
1000_0.4network.txt	1000	20014
1000_0.5network.txt	1000	20770
1000_0.6network.txt	1000	20084
1000_0.7network.txt	1000	20150
1000_0.8network.txt	1000	20640

Tabla 6.3: Instancias seleccionadas para la segunda comparativa

Para esta comparativa, en lugar de hacer uso de la modularidad, se hará uso de la métrica NMI (Normalized Mutual Information). Para el experimento, se usarán las soluciones óptimas para el problema de la detección de comunidades de las instancias seleccionadas, de forma que las soluciones arrojadas por los distintos algoritmos se compararan con ellas obteniendo su NMI, comprobando así la calidad de las soluciones.

Normalized Mutual Information

La información mutua normalizada (en ingles *Normalized Mutual Information*) es una cantidad que mide la dependencia mutua de dos conjuntos. Sin profundizar en su cálculo (que puede consultarse en la literatura [4]), comprobamos que es una buena métrica para evaluar cómo de similares son dos soluciones al problema de la detección de comunidades.

Por ejemplo, dos soluciones al problema (dónde la posición en la lista corresponde al nodo y el valor a la comunidad a la que pertenece) $[0, 0, 1, 1]$ y $[1, 1, 0, 0]$ resultan obtener un valor para esta métrica de 1.0 (son idénticas) a pesar de que usan distintos valores para las etiquetas. Al estar normalizada, nos permite comparar soluciones que cuentan con distinto número de comunidades. Las soluciones $[0, 0, 0, 1, 1, 1]$ y $[1, 1, 1, 2, 2, 0]$ obtienen un valor para la métrica de 0.82, a pesar de que la segunda solución cuenta con una comunidad más.

[1] `https://igraph.org/python/`

6.2.2. Resultados

Atendiendo al resumen de los resultados ofrecidos en la Tabla 6.4 observamos que nuestro algoritmo, a pesar de no encontrar el mejor valor NMI respecto al resto de algoritmos, cuenta con una desviación típica media inferior a 3 de los algoritmos previos, pero lejos de los mejores algoritmos. Es necesario destacar que nuestro algoritmo cuenta con uno de los mayores tiempos de ejecución, con una magnitud similar al Edge Betweeness a pesar de ser ejecutado en paralelo. Los datos ampliados de esta comparativa pueden encontrarse en las Tablas A.2 y A.3 del Apéndice A.

Algoritmo	**Dev(%)**	**Mejor NMI (veces)**
Edge Betweenness	0.09368	3/11
Eigenvectors	0.63170	0/11
Fast Greedy	0.49691	0/11
Label Propagation	0.46733	1/11
Multi-level	0.18881	0/11
Spinglass	0.15059	3/11
Walktrap	0.11338	6/11
DividerGreedyLS	0.33755	0/11

Tabla 6.4: Resumen de la comparativa de algoritmos previos

Capítulo 7

Conclusiones del proyecto y trabajos futuros

Este proyecto ha supuesto un desafío y una fuente de aprendizaje sobre los algoritmos de detección de comunidades, pues hasta ahora solo los había usado sin preocuparme de cómo se implementaban.

Quedan satisfechos los objetivos del proyecto y se ha obtenido un algoritmo de detección de comunidades usable para grandes redes dónde la solución optima nos llevaría demasiado tiempo. A pesar de no obtener un algoritmo revolucionario, hemos conseguido obtener un rendimiento superior en cuanto a las soluciones frente a algunos algoritmos previos.

Sin duda se trata de un algoritmo mejorable, por lo que un trabajo futuro pasaria por encontrar mejores heurísticos que reduzcan el tiempo de ejecucción y mejoren la calidad de las soluciones

Apéndice A

Tablas de resultados

Este apéndice recoge los resultados de los distintos experimentos realizados.

A.1. Resultados del experimento 1

En la Tabla A.1 se ven reflejados los resultados del primer experimento, dónde comparamos las métricas recogidas para cada propuesta en cada uno de las instancias de la Tabla 6.1.1. A continuación se explica cada una de los valores y métricas que se muestran:

- **N**: Este valor indica el número de nodos con el que cuenta la instancia
- **M**: Este valor indica el número de aristas con el que cuenta la instancia
- **Mod**: Esta valor corresponde a la métrica de modularidad arrojada por el algoritmo para una instancia.
- **T(ms)**: Este valor corresponde al tiempo (en milisegundos) que tarda en ejecutar un algoritmo para una instancia.
- **Dev(%)**: Este valor corresponde a la desviación típica de la modularidad de ese algoritmo para una instancia respecto a la mejor modularidad obtenida para la misma instancia.
- **# Best**: Este valor indica si la modularidad ofrecida por ese algoritmo es la mejor para una instancia. Toma el valor 1 en caso afirmativo y 0 en el contrario.

Instance	N	M	Random				Divider			
			Mod	T(ms)	Dev(%)	# Best	Mod	T(ms)	Dev(%)	# Best
1000_0.1network.txt	1000	10434	-0.00062	2	1.00073	0	0.40508	2906	0.51844	0
500_0.1network.txt	500	5337	-0.00274	1	1.00341	0	0.41746	615	0.47964	0
500_0.2network.txt	500	4965	-0.00238	0	1.00334	0	0.34987	451	0.50875	0
500_0.3network.txt	500	4842	-0.00198	0	1.00318	0	0.20573	422	0.66978	0
1000_0.2network.txt	1000	10071	-0.00189	0	1.00253	0	0.33128	2313	0.55629	0
				Avg — Sum	1.00264	0		Avg — Sum	0.54658	0

Instance	N	M	DividerGreedy ($\alpha = 0{,}25$)				DividerGreedy ($\alpha = 0{,}5$)			
			Mod	T(ms)	Dev(%)	# Best	Mod	T(ms)	Dev(%)	# Best
1000_0.1network.txt	1000	10434	0.63597	74689	0.24396	0	0.63300	75142	0.24749	0
500_0.1network.txt	500	5337	0.59792	13909	0.25471	0	0.57846	12973	0.27896	0
500_0.2network.txt	500	4965	0.53244	13854	0.25241	0	0.51089	13096	0.28267	0
500_0.3network.txt	500	4842	0.39405	14059	0.36754	0	0.35768	13271	0.42590	0
1000_0.2network.txt	1000	10071	0.46798	72776	0.37319	0	0.46930	71861	0.37143	0
				Avg — Sum	0.29836	0		Avg — Sum	0.32129	0

Instance	N	M	DividerGreedy ($\alpha = 0{,}75$)				DividerGreedyLS ($\alpha = 0{,}25$)			
			Mod	T(ms)	Dev(%)	# Best	Mod	T(ms)	Dev(%)	# Best
1000_0.1network.txt	1000	10434	0.60782	71093	0.27743	0	0.84120	769637	0	1
500_0.1network.txt	500	5337	0.58249	12158	0.27394	0	0.80227	86615	0	1
500_0.2network.txt	500	4965	0.51672	13316	0.27448	0	0.71221	81108	0	1
500_0.3network.txt	500	4842	0.39707	14375	0.36268	0	0.62252	114188	0.00082	0
1000_0.2network.txt	1000	10071	0.46234	72990	0.38074	0	0.74662	1201514	0	1
				Avg — Sum	0.31385	0		Avg — Sum	0.00016	4

Instance	N	M	DividerGreedyLS ($\alpha = 0{,}5$)				DividerGreedyLS ($\alpha = 0{,}75$)			
			Mod	T(ms)	Dev(%)	# Best	Mod	T(ms)	Dev(%)	# Best
1000_0.1network.txt	1000	10434	0.83649	793103	0.00559	0	0.83700	798806	0.00499	0
500_0.1network.txt	500	5337	0.80227	90651	0	1	0.80051	97795	0.00219	0
500_0.2network.txt	500	4965	0.71071	91511	0.00210	0	0.71109	92985	0.00157	0
500_0.3network.txt	500	4842	0.62304	122239	0	1	0.61997	127982	0.00492	0
1000_0.2network.txt	1000	10071	0.73520	1122416	0.01529	0	0.74178	1177559	0.00647	0
				Avg — Sum	0.00459	2		Avg — Sum	0.00342	0

Tabla A.1: Resultados de la comparativa de algoritmos propios (100 iteraciones)

A.2. Resultados del experimento 2

En las Tablas A.2 y A.3 se ven reflejados los resultados del segundo experimento, dónde comparamos las métricas recogidas para cada algoritmo previo junto a la mejor versión del nuestro en cada una de las instancias de la Tabla 6.3. Los valores que podemos encontrar en la tabla son idénticos a los definidos para el primer experimento a excepción de métrica NMI, que toma el lugar de la modularidad (Mod). Esta métrica muestra el valor de la información mutua normalizada (*Normalized Mutual Information*).

Instance	N	M	Edge Betweeness				Eigenvector			
			NMI	T(ms)	Dev(%)	# Best	NMI	T(ms)	Dev(%)	# Best
1000_0.3network.txt	1000	9716	1	929689	0	1	0.45370	870	0.54630	0
1000_0.4network.txt	1000	10007	0.98833	1293816	0.01167	0	0.46850	719	0.53150	0
1000_0.5network.txt	1000	10385	0.86930	2098782	0.12105	0	0.49608	741	0.49841	0
1000_0.6network.txt	1000	10042	0.79288	2400315	0.14013	0	0.23817	369	0.74171	0
1000_0.7network.txt	1000	10075	0.68024	2522520	0.11684	0	0.22573	377	0.70693	0
1000_0.8network.txt	1000	10320	0.64119	2603402	0	1	0.12856	420	0.79950	0
500_0.4network.txt	500	5169	0.94549	186617	0.04589	0	0.63877	294	0.35541	0
500_0.5network.txt	500	5155	0.84748	254959	0.13958	0	0.36734	174	0.62705	0
500_0.6network.txt	500	5156	0.67682	316304	0.28776	0	0.28475	177	0.70035	0
500_0.7network.txt	500	5116	0.59582	278364	0.16756	0	0.23275	172	0.67482	0
500_0.8network.txt	500	5097	0.60439	278199	0	1	0.14102	215	0.76668	0
				Avg — Sum	0.09368	3		Avg — Sum	0.63170	0

Instance	N	M	Fast Greedy				Label Propagation			
			NMI	T(ms)	Dev(%)	# Best	NMI	T(ms)	Dev(%)	# Best
1000_0.3network.txt	1000	9716	0.75754	80	0.24246	0	1	3	0	1
1000_0.4network.txt	1000	10007	0.68460	82	0.31540	0	0.99734	4	0.0027	0
1000_0.5network.txt	1000	10385	0.61365	89	0.37954	0	0.98214	5	0.0070	0
1000_0.6network.txt	1000	10042	0.51517	114	0.44130	0	0.88736	6	0.0377	0
1000_0.7network.txt	1000	10075	0.29554	120	0.61630	0	0	3	1	0
1000_0.8network.txt	1000	10320	0.12961	145	0.79787	0	0	4	1	0
500_0.4network.txt	500	5169	0.72322	22	0.27019	0	0.93884	1	0.05260	0
500_0.5network.txt	500	5155	0.57974	30	0.41141	0	0.94482	2	0.04075	0
500_0.6network.txt	500	5156	0.42633	30	0.55136	0	0	2	1	0
500_0.7network.txt	500	5116	0.25775	33	0.63990	0	0	2	1	0
500_0.8network.txt	500	5097	0.12068	31	0.80033	0	0	2	1	0
				Avg — Sum	0.49691	0		Avg — Sum	0.46733	1

Tabla A.2: Resultados de la comparativa con algoritmos previos [Edge Betweeness, Eigenvector, Fast Greedy y Label Propagation]

Instance	N	M	Multilevel				Spinglass			
			NMI	T(ms)	Dev(%)	# Best	NMI	T(ms)	Dev(%)	# Best
1000_0.3network.txt	1000	9716	0.94600	75	0.05400	0	0.92004	21189	0.07996	0
1000_0.4network.txt	1000	10007	0.91697	12	0.08303	0	0.89881	18105	0.10119	0
1000_0.5network.txt	1000	10385	0.91981	16	0.06998	0	0.89080	22104	0.09931	0
1000_0.6network.txt	1000	10042	0.85022	18	0.07794	0	0.86342	27019	0.06362	0
1000_0.7network.txt	1000	10075	0.64806	31	0.15861	0	0.77023	31784	0.00000	1
1000_0.8network.txt	1000	10320	0.24340	31	0.62039	0	0.26181	44934	0.59167	0
500_0.4network.txt	500	5169	0.97473	6	0.01639	0	0.97535	8531	0.01576	0
500_0.5network.txt	500	5155	0.93480	6	0.05092	0	0.95089	9949	0.03458	0
500_0.6network.txt	500	5156	0.90370	8	0.04900	0	0.95027	9324	0.00000	1
500_0.7network.txt	500	5116	0.56002	14	0.21758	0	0.71575	19857	0.00000	1
500_0.8network.txt	500	5097	0.19393	13	0.67913	0	0.19921	19883	0.67039	0
				Avg—Sum	0.18881	0		Avg—Sum	0.15059	3

Instance	N	M	Walktrap				DividerGreedyLS			
			NMI	T(ms)	Dev(%)	# Best	NMI	T(ms)	Dev(%)	# Best
1000_0.3network.txt	1000	9716	1	324	0	1	0.85830	1413007	0.14170	0
1000_0.4network.txt	1000	10007	1	289	0	1	0.86577	2096239	0.13423	0
1000_0.5network.txt	1000	10385	0.98902	326	0	1	0.81065	2669170	0.18035	0
1000_0.6network.txt	1000	10042	0.92208	324	0	1	0.66760	3192113	0.27599	0
1000_0.7network.txt	1000	10075	0.71416	355	0.07280	0	0.38896	3830561	0.49501	0
1000_0.8network.txt	1000	10320	0.47410	414	0.26059	0	0.19665	3314237	0.69331	0
500_0.4network.txt	500	5169	0.99097	85	0	1	0.92471	183611	0.06687	0
500_0.5network.txt	500	5155	0.98496	87	0	1	0.84389	236974	0.14322	0
500_0.6network.txt	500	5156	0.82124	93	0.13577	0	0.59652	278820	0.37226	0
500_0.7network.txt	500	5116	0.56647	98	0.20857	0	0.31090	304354	0.56564	0
500_0.8network.txt	500	5097	0.26023	105	0.56943	0	0.21488	310541	0.64448	0
				Avg—Sum	0.11338	6		Avg—Sum	0.33755	0

Tabla A.3: Resultados de la comparativa con algoritmos previos [Multilevel, Spinglass, Walktrap y DividerGreedyLS (100 iteraciones)]

Bibliografía

[1] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[2] A. Clauset, M. E. Newman, and C. Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

[3] M. Girvan and M. E. Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002.

[4] A. F. McDaid, D. Greene, and N. Hurley. Normalized mutual information to evaluate overlapping community finding algorithms. *arXiv preprint arXiv:1110.2515*, 2011.

[5] M. E. Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical review E*, 74(3):036104, 2006.

[6] P. Pons and M. Latapy. Computing communities in large networks using random walks. In *International symposium on computer and information sciences*, pages 284–293. Springer, 2005.

[7] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 76(3):036106, 2007.

[8] J. Reichardt and S. Bornholdt. Statistical mechanics of community detection. *Physical Review E*, 74(1):016110, 2006.