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. (PENDIENTE DE TERMINAR TRAS FINALIZAR LA MEMORIA)

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. Una de las áreas más interesantes que utiliza como base este tipo de redes es la detección de comunidades, 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.

La motivación de este proyecto nace de la idea de crear una solución heurística diferente a las preexistentes (normalmente basadas en algun algun algoritmo destructivo), optando por un enfoque destructivo.

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.

Capítulo 3

Tecnologías, Herramientas y Metodologías

3.1. Tecnologías

3.1.1. Java

Figura 3.1: Logo Java

Java (Figura 3.1) 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.

3.2. Herramientas

3.2.1. Control de versiones: Git

Figura 3.2: Logo Git

Git[1] (Figura 3.2) 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.

3.2.2. Entorno de desarrollo: IntelliJ

Figura 3.3: Logo IntelliJ

IntelliJ[2] (Figura 3.3) es un IDE para Java desarollado por JetBrains ideado para mejorar la productividad del programador.

3.3. Metodologías

Dado que pretendemos obtener la mejor versión de nuestro algoritmo, el desarrollo seguirá una metodología iterativa-incremental[3], dónde en cada iteración se propondrán y evaluarán diferentes mejoras sobre el algoritmo.

[1] https://git-scm.com/
[2] https://www.jetbrains.com/idea/
[3] https://proyectosagiles.org/desarrollo-iterativo-incremental/

Capítulo 4

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.

4.1. Requisitos

4.1.1. 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.

4.1.2. 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].

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 [-.5,1)

4.2. Diseño e implementación

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

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

Grafo conexo

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

Para estimar cómo de buena es cada versión del algoritmo usaremos un grafo de ejemplo representativo que llamaremos *Instancia de Prueba* (214 nodos y 1954 aristas).

4.2.1. 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 aleatorio 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
```

Modularidad para Instancia de Prueba: -0.007

4.2.2. 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 los siguiente (Ver junto al Algoritmo 2):

1. Comenzamos con una sola comunidad que contiene todos los nodos.

2. Se busca el nodo peor conectado (menor número de aristas) ó el primero que encuentre una arista.

3. 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).

4. Repetimos los pasos 1-3 con el clúster B hasta que no se generen nuevos clusters.

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

Modularidad para Instancia de Prueba: 0.096

4.2.3. ConstDividerGreedy: Segunda aproximación

Podemos observar como la mejora en cuanto a modularidad no es destacable (recordemos que la modularidad toma un valor entre -0.5 y 1.0). Es necesario utilizar otro enfoque. El problema de este algortimo es que una mala elección del primer nodo puede provocar que no se formen correctamente las comunidades. Aleatorizar este la elección de este primer nodo nos abre la puerta a realizar múltiples iteraciones comenzando desde distintos nodos. Los pasos a seguir en este algoritmo serían (Ver junto al Algoritmo 3):

- 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 4.2.3), 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:

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

Para probar el algoritmo se han utilizado los siguientes valores de α: 0.0, 0.25, 0.5, 0.75, 1.

4.2.4. ConstDividerGreedy + Local Search: Última versión

La modularidad obtenida trás aplicar un enfoque voraz mejora de forma considerable. A pesar de ello, intentaremos aumentarla aún más partiendo de un heurístico no contemplado hasta ahora, la búsqueda local. Esta 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
```

Modularidad para Instancia de Prueba (1000 iteraciones y α = 0.0): 0.251

Para probar el algoritmo se han utilizado los siguientes valores de α: 0.0, 0.25, 0.5, 0.75, 1.

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
```

Modularidad para Instancia de Prueba (1000 iteraciones y α = 0.25): 0.3295

Observamos que para $\alpha = 0{,}25$ obtenemos un resultado notablemente mejor que con el anterior algoritmo y consideramos *ConstDividerGreedy + Local Search* la última versión de nuestro algoritmo.

4.2.5. Paralelización

Una vez obtenida una modularidad aceptable, 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 del algoritmo.

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 16.04 con 16GB de RAM y 4 procesadores (sin *hyper-threading*).

Tiempos de ejecucción (en milisegundos) para 1000 iteraciones:

- 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 5

Estudio comparativo

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

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

Capítulo 6

Conclusiones del proyecto y trabajos futuros

Prueba cita [1].

Bibliografía

[1] J. Sánchez-Oro and A. Duarte. Iterated greedy algorithm for performing community detection in social networks. 2018.