UIMP
Universidad Internacional
Menéndez Pelayo

Formalización del problema de la partición del grafo

Laura Rodríguez Navas
rodrigueznavas@posgrado.uimp.es

Junio 2020
Resolución de problemas con metaheurísticos

Resumen

Dentro del amplio campo de la Teoría de Grafos, en este informe se considera el problema de la partición del grafo: dado un grafo no dirigido cuyo número de vértices es par y cuyas aristas tienen asociado un peso, se trata de encontrar la partición del conjunto de vértices en dos subconjuntos de tamaño similar de manera que se minimice la suma de los pesos de aquellas aristas que unen vértices que están en diferentes conjuntos.

En el capítulo inicial se presenta una breve introducción y descripción del problema. En el segundo capítulo, se describen tres codificaciones para este problema, señalando sus ventajas y desventajas. Concretamente, se describen estas codificaciones que son implementaciones en Python de los algoritmos basados en técnicas metaheurísticas: Kernighan-Lin, Spectral Bisection y Multilevel Spectral Bisection. En el tercer capítulo, se hace una comparativa entre las diferentes codificaciones. Finalmente, en el último capítulo se presentan algunas conclusiones.

Las codificaciones que se describen en este informe se encuentran en el repositorio:

`https://github.com/lrodrin/masterAI/tree/master/A2`

Tabla de Contenidos

1. Introducción

1.1 Partición de Grafos

Los grafos son una forma eficiente de representar estructuras de datos complejas y contienen todo tipo de información. Cuando un grafo se acerca o incluso excede los límites de memoria, se vuelve más difícil y muy costoso de procesar si no está particionado.

La solución es dividir el grafo de manera que las particiones se puedan cargar en la memoria. Al particionar un grafo, los nodos y las aristas se deben dividir uniformemente para tener un buen equilibrio en la distribución de las recursos computacionales. Esto es necesario para mantener la máxima eficiencia.

La partición de grafos es una disciplina ubicada entre las ciencias computacionales y la matemática aplicada. La mayoría del trabajo previo en esta disciplina fue realizada por matemáticos que se enfocaron principalmente en lograr el equilibrio en la distribución de los recursos computacionales repartiendo uniformemente estos recursos entre las particiones.

En los últimos años los grafos han sido ampliamente utilizados, en consecuencia, el problema de la partición de grafos también ha sido ampliamente estudiado. El problema de la partición de grafos es un problema muy conocido ya que se deriva de situaciones del mundo real que tiene aplicaciones en muchas áreas.

La primera aplicación del problema fue en la partición de componentes de un circuito en un conjunto de tarjetas, que juntas realizaban tareas en un dispositivo electrónico. Las tarjetas tenían un tamaño limitado, de tal manera que el dispositivo no llegara a ser muy grande, y el número de elementos de cada tarjeta estaba restringido. Si el circuito era demasiado grande podía ser dividido en varias tarjetas las cuales estaban interconectadas, sin embargo, el coste de la interconexión era muy elevado por lo que el número de interconexiones debía ser minimizado.

La aplicación descrita fue presentada en [1], en la cual se define un algoritmo eficiente para encontrar buenas soluciones. En la aplicación, el circuito es asociado a un grafo y las tarjetas como subconjuntos de una partición. Los nodos del grafo son representados por los componentes electrónicos y las aristas forman las interconexiones entre los componentes y las tarjetas.

Los algoritmos de partición de grafos utilizados son a menudo complejos. El uso de tales algoritmos da como resultado un procesamiento computacionalmente intenso del grafo que lleva una cantidad considerable de tiempo. Se pueden utilizar algoritmos menos complejos para reducir este tiempo, a costa de una distribución de los recursos menos equilibrada.

El enfoque del informe se centra en la división de grafos en dos particiones. Se ha elegido la partición en dos partes porque es la forma más sencilla de comparar las particiones con un método de *"bisección"* (ver Definición 1).

Además, asumiremos que la forma más eficiente de particionar todos los tipos de grafos no se limita a un solo algoritmo de partición. La codificación de tres algoritmos se ha utilizado para examinar el problema de partición de grafos. Estos algoritmos son: Kernighan-Lin, Spectral Bisection y Multilevel Spectral Bisection (ver sección 2).

En este informe se muestra inicialmente el uso de grafos más pequeños para describir los tres algoritmos anteriores, y la hora de compararlos se han usado grafos más grandes para probar si las codificaciones siguen produciendo computacionalmente buenos resultados.

Definición 1. *En matemáticas, el método de bisección es un algoritmo de resolución numérica de ecuaciones no lineales que trabaja dividiendo un subconjunto a la mitad y seleccionando el subconjunto que genera la solución.*

1.2 Descripción del problema de Partición de Grafos

El problema de partición de grafos puede formularse como un problema de programación lineal. La programación lineal se dedica a maximizar o minimizar (optimizar) una función lineal, denominada función objetivo, de tal forma que las variables de dicha función están sujetas a una serie de restricciones expresadas mediante un sistema de ecuaciones o inecuaciones también lineales.

Concretamente, el problema de partición de grafos puede formularse como un problema de programación lineal entera porqué los pesos de las aristas normalmente toman valores enteros.

Los problemas de programación lineal entera generalmente están relacionados directamente con problemas de optimización combinatoria, esto genera que al momento de resolver los problemas de programación lineal entera se encuentren restricciones dado el coste computacional de resolverlos. Por ese motivo los algoritmos que buscan soluciones a este problema no pueden garantizar que la solución encontrada sea la óptima.

El problema de partición de grafos ha sido denominado como un problema NP-completo[2], lo que implica que las soluciones para él no pueden ser encontradas en tiempos razonables. Entonces, en lugar de encontrar la solución óptima para el problema, recurriremos a algoritmos que pueden no ofrecer la solución óptima pero que dan una buena solución, al menos la mayor parte del tiempo.

La definición formal del problema tiene como objetivo principal, dividir un grafo en *"k"* subgrafos. La única restricción trascendental para resolver el problema es que tiene que haber un número mínimo y máximo de nodos pertenecientes a cada subgrafo. Todos los subgrafos deben tener incorporados al menos 2 nodos y como máximo el doble de los nodos que podrían existir para cada subgrafo si se divide el grafo total en *"k"* subgrafos con la misma cantidad de nodos. En definitiva, que las aristas que unen los subgrafos pesen lo menos posible y de esa manera poder evitar una cierta *"dependencia trascendental"* entre ellos.

La función objetivo es la que pretende minimizar las aristas que unen a los subgrafos. Para ello se analiza su peso. Esta función es a la que le incorporamos la restricción comentada anteriormente, el número mínimo y máximo de nodos por subgrafo.

A continuación, planteamos el problema de partición de grafos desde un punto de vista más formal.

Definición 2. *Un grafo G es un par ordenado $G = (V, E)$, donde V es un conjunto de vértices y E es un conjunto de aristas.*

Dado el grafo $G = (V, E)$ que está formado por un conjunto de n vértices V y un conjunto de m aristas E, el problema de partición de grafos busca asignar a cada vértice $v \in V$ un entero $p(v) \in \{1, 2, ..., k\}$ tal que:

- $p(v) \neq p(u) \, \forall \{u, v\} \in E$
- k sea mínimo

De las particiones sobre el conjunto de vértices V, una solución S es representada por un subconjunto de k clases de pesos, $S = \{S_1, ...S_k\}$. Para que la solución S sea factible, es necesario que se cumplan las siguientes restricciones a la vez que se minimice el número de clases de k.

$$\bigcup_{i=1}^{k} S_i = V \tag{1.1}$$

$$S_i \cap S_j = \emptyset \, (1 \leq i \neq j \leq k) \tag{1.2}$$

$$\forall u, v \in S_i, \{u, v\} \notin E \, (1 \leq i \leq k) \tag{1.3}$$

Las restricciones (1.1) y (1.2) establecen que la solución S debe ser una partición del subconjunto de vértices V , mientras que la restricción (1.3) obliga a que ningún par de vértices adyacentes sean asignados a la misma clase k, es decir, que todas las clases de k pesos en la solución deben formar subconjuntos independientes.

2. Algoritmos

Debido a la *intratabilidad* computacional del problema de la partición de grafos, como hemos comentado en la Introducción (ver sección 1), no existe un algoritmo concreto que permita obtener en tiempo polinómico una solución óptima a la partición de cualquier grafo. Es por ello por lo que, los algoritmos codificados que describimos en este informe se basan en la metaheurística, con el objetivo de obtener soluciones de buena calidad en tiempos computacionales aceptables, a pesar de que los algoritmos metaheurísticos no garantizan que se vaya a obtener una solución óptima al problema. Puede que hasta no obtengamos una solución.

En la siguiente sección se describen los algoritmos Kernighan-Lin[1] (ver sección 2.1), Specrtal Bisection (ver sección 2.2) y Multilevel Spectral Bisection (ver sección 2.3). Tres algoritmos que se diseñaron específicamente para la resolución del problema de la partición de grafos. Cualquiera de estos algoritmos proporciona una solución factible al problema, pudiendo ser esta óptima o no. Como hemos comentado anteriormente, el enfoque del informe radica en la partición de diferentes grafos en dos particiones usando estos tres algoritmos.

Además de describir cada uno de los algoritmos, también se describirán algunos detalles sobre su codificación. De ejemplo se ha utilizado un grafo inicial (ver Figura 2.1). Este grafo es un grafo no dirigido (ver Definición 3) de diez nodos con pesos en sus aristas.

Definición 3. *Un grafo no dirigido es un grafo (ver Definición 2) donde:* $V \neq \emptyset$ *y* $E \subseteq \{x \in \mathcal{P}(V) : |x| = 2\}$ *es un subconjunto de pares no ordenados de elementos de* V*. Un par no ordenado es un subconjunto de la forma* $\{a\ ,\ b\}$*, de manera que* $\{a,\ b\} = \{b,\ a\}$*. En los grafos no dirigidos, los subconjuntos de pares no ordenados pertenecen al conjunto potencial de* V*, denotado* $\mathcal{P}(V)$*, y son de cardinalidad 2.*

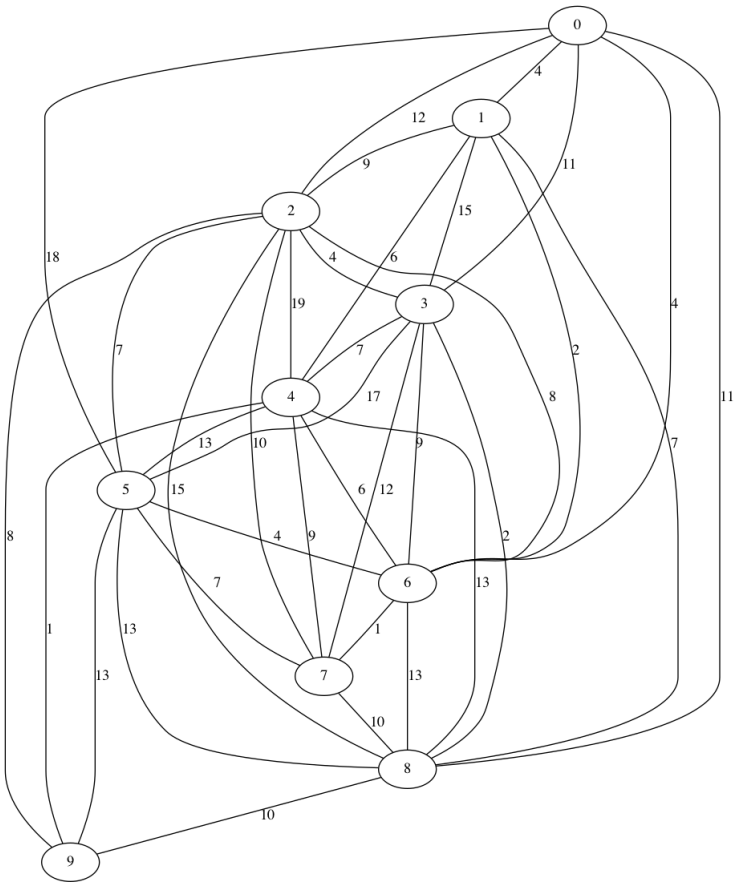


Figura 2.1: Grafo inicial.

2.1 Kernighan-Lin

El algoritmo Kernighan-Lin[1], a menudo abreviado como K-L, es uno de los primeros algoritmos de partición de grafos y fue desarrollado originalmente para optimizar la colocación de circuitos electrónicos en tarjetas de circuito impreso para minimizar el número de conexiones entre las tarjetas (circuitos VLSI[1][3]).

El algoritmo se caracteriza por ser:

- **Iterativo**. Porque el grafo inicialmente ya está particionado, pero la aplicación del algoritmo intentará mejorar u optimizar la partición inicial.
- **Voraz**. Porque el algoritmo hará cambios si hay un beneficio inmediato sin considerar otras formas posibles de obtener una solución óptima. Para ello, elige la opción óptima en cada paso con la esperanza de llegar a una solución óptima general.
- **Determinista**. Porque se obtendrá el mismo resultado cada vez que se aplique el algoritmo.

El algoritmo K-L, como acabamos de comentar, no crea la partición inicial de un grafo, sino que mejora iterativamente las particiones creadas a partir de esta partición inicial. La idea original era tomar una partición aleatoria y aplicarle Kernighan-Lin[1]. Esto se repetiría varias veces y se elegiría el mejor resultado. Mientras que para grafos pequeños esto ofrece resultados razonables, es bastante ineficiente para tamaños de grafos más grandes.

Hoy en día, el algoritmo se usa para mejorar las particiones encontradas por otros algoritmos, tratando de mejorar las particiones mediante el intercambio de nodos vecinos. Por tanto, complementa muy bien otro tipo de algoritmos como por ejemplo los que utilizan un método de bisección espectral (ver sección 2.2).

2.1.1 Descripción

El objetivo del algoritmo es dividir el conjunto de vértices V de un grafo no dirigido (ver Definición 3) en dos subconjuntos disjuntos (ver Definición 4) A y B de igual tamaño, de una manera que minimice la suma T de los pesos del subconjunto de aristas que cruzan de A a B.

El algoritmo mantiene y mejora una partición en cada iteración, usando un algoritmo voraz (ver Definición 5). Después empareja los vértices de A con los vértices de B, de modo que al mover los vértices emparejados de una partición a la otra se maximiza la ganancia, que mide cuánta "información" nos brinda la función objetivo. Una vez ha emparejado los vértices y maximizado la ganancia, crea un subconjunto de los pares de los vértices elegidos para tener el mejor efecto sobre la calidad de la solución T. Dado un grafo con n vértices, cada iteración del algoritmo se ejecuta en el tiempo $O(n^2 \log n)$.

Definición 4. *A y B son dos subconjuntos disjuntos si $A \cup B$ y $A \cap B \neq 0$.*

Definición 5. *Dado un conjunto finito C, un algoritmo voraz devuelve un subconjunto S tal que $S \subseteq C$ y que además cumple con las restricciones del problema inicial. A cada subconjunto S que satisfaga las restricciones y que logra que la función objetivo se minimice o maximice (según corresponda) diremos que S es una solución óptima.*

Más formalmente, sea a un elemento del subconjunto A y b un elemento del subconjunto B, para cada $a \in A$, I_a será el coste interno de a, es decir, la suma de los pesos de las aristas entre a y otros nodos en A. Y E_a será el coste externo de a, es decir, la suma de los pesos de las aristas entre a y los nodos en B. De manera similar, se define I_b y E_b para cada $b \in B$.

Conceptualmente ahora podemos decir que existe una diferencia D entre la suma de los costes externos e internos s:

$$D_s = E_s - I_s$$

Y entonces si a y b se intercambian, la ganancia $(T_{antigua} - T_{nueva})$ se calcula como:

$$T_{antigua} - T_{nueva} = D_a + D_b - 2c_{a,b}$$

donde $c_{a,b}$ es el número de conexiones (coste) de las aristas posibles entre a y b.

A partir de esto, el algoritmo intenta encontrar una solución óptima de iteraciones de intercambio entre los elementos de A y de B que maximice la ganancia. Luego ejecuta las iteraciones, dividiendo el grafo en dos particiones de A y B.

Con estos últimos conceptos, podemos describir la primera iteración del algoritmo usando el pseudocódigo en [3].

Given two subpartitions $\mathcal{V}_A, \mathcal{V}_B$

Compute the diff–value of all vertices.

Unmark all vertices.

Let k_0 = cut-size.

For $i = 1$ **to** $\min(|\mathcal{V}_A|, |\mathcal{V}_B|)$ **do**

- Among all unmarked vertices, find the pair (a_i, b_i) with the biggest gain (which might be negative).
- mark a_i and b_i.
- **For** each neighbor v of a_i or b_i **do**

  Update diff(v) as if a_i and b_i had been swapped, i.e.,

$$\text{diff}(v) := \text{diff}(v) + \begin{cases} 2w_e(e_{v,a_i}) - 2w_e(e_{v,b_i}) & \text{for } v \in \mathcal{V}_A \\ 2w_e(e_{v,b_i}) - 2w_e(e_{v,a_i}) & \text{for } v \in \mathcal{V}_B \end{cases}$$

- $k_i = k_{i-1} + \text{gain}(a_i, b_i)$, i.e., k_i would be the cut-size if $a_1, a_2, \ldots, a_i$ and $b_1, b_2, \ldots, b_i$ had been swapped.

Find the smallest j such that $k_j = \min_i k_i$.

Swap the first j pairs, i.e.,

$$\mathcal{V}_A := \mathcal{V}_A - \{a_1, a_2, \ldots, a_j\} \cup \{b_1, b_2, \ldots, b_j\}$$
$$\mathcal{V}_B := \mathcal{V}_B - \{b_1, b_2, \ldots, b_j\} \cup \{a_1, a_2, \ldots, b_j\}$$

Figura 2.2: Primera iteración de Kernighan-Lin.

Entonces, en cada iteración, el algoritmo K-L intercambia pares de vértices para maximizar la ganancia, y continúa haciéndolo hasta que se intercambian todos los vértices de la partición más pequeña.

C. Fiduccia y R. Mattheyses realizaron importantes avances prácticos en [4], quienes mejoraron el algoritmo de K-L de tal manera que, cada iteración se ejecuta en $O(n^2)$, en lugar de $O(n^2 \log n)$. La reducción del coste computacional se logra en parte eligiendo nodos individuales para intercambiar en lugar de parejas de vértices.

Una gran ventaja del algoritmo es que no se detiene incluso aceptando ganancias negativas, sigue esperando que las ganancias posteriores sean mayores y que el tamaño de la suma de los pesos de las aristas se reduzca hasta que se intercambian todos los vértices de la partición más pequeña. Esta capacidad es una característica crucial, aunque también cuenta con unas cuantas desventajas.

Podemos decir que las principales desventajas de K-L son:

- Los resultados son aleatorios porque el algoritmo comienza con una partición aleatoria.
- Computacionalmente es un algoritmo lento.
- Solo se crean dos particiones del mismo tamaño.
- Las particiones deben tener el mismo tamaño para que el algoritmo no intente encontrar soluciones óptimas que ya existan.
- No resuelve muy bien los problemas con las aristas ponderadas.
- La solución dependerá en gran medida de la primera partición.

2.1.2 Ejemplo de codificación

Existen muchas variaciones del algoritmo K-L que a menudo intercambian el tiempo de ejecución con la calidad, o generalizan el algoritmo a más de dos particiones. En este caso, para la codificación del algoritmo se ha utilizado la librería de Python: NetworkX. El algoritmo implementado divide un grafo en dos particiones usando el algoritmo Kernighan-Lin[1]. Es decir, divide un grafo en dos subconjuntos intercambiando iterativamente pares de nodos para reducir el peso de las aristas entre los dos subconjuntos.

Definición 6. *NetworkX es un paquete de Python para la creación, manipulación y estudio de la estructura, dinámica y funciones de los grafos.*

Por ejemplo, en una ejecución de la codificación, donde la entrada es el grafo de la Figura 2.1, y después de crear una partición inicial, podemos obtener los subconjuntos: A = {3, 4, 6, 8, 9} y B = {0, 1, 2 , 5, 7}. Estos subconjuntos son del mismo tamaño y sus elementos están ordenados.

En esta ejecución de ejemplo, los pasos que ha seguido el algoritmo son los siguientes:

- Dibuja una línea que separa el grafo en dos particiones con el mismo número de vértices en cada partición.
- Cuenta la cantidad de aristas que cruzan esa línea. Esta cantidad se denomina *"cut-size"* y el objetivo es disminuirla, quiere decir, disminuir el número de conexiones entre los vértices del grafo.
- Encuentra el coste de todos los vértices en el grafo, buscando el número de conexiones que cada vértice tiene dentro de su propia partición, y resta eso del número de conexiones que cada vértice tiene con los vértices de la otra partición.
- Determina la ganancia máxima intercambiando dos nodos. La ecuación de ganancia es la que se muestra en la sección 2.1.1.
- Intercambia los dos nodos con la ganancia máxima. Si se han calculado todas las ganancias de emparejamiento de todos los nodos y la ganancia máxima es igual a cero o negativa, los nodos con la ganancia más alta aún deberán intercambiarse.
- Resta la ganancia del valor de *"cut-size"* original para obtener el nuevo valor de *"cut-size"*.
- Cambia los nodos que se han intercambiado.
- Repite estos pasos hasta que la ganancia máxima es cero o negativa.

Los pasos que sigue el algoritmo que acabamos de describir se pueden visualizar en la siguiente figura:

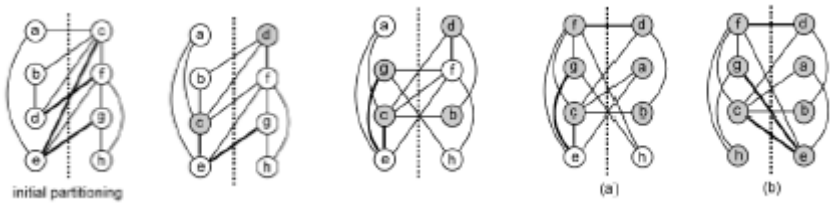


Figura 2.3: Pasos de Kernighan-Lin.

2.2 Spectral Bisection

El algoritmo de bisección espectral que se describe en la siguiente sección es bastante diferente al algoritmo que acabamos de describir. Ya que no utiliza el grafo en sí, sino una representación matemática del mismo, una matriz.

2.2.1 Descripción

El método de bisección espectral aborda el problema de la partición de grafos con un estrategia desarrollada por Pothen, Simon y Liou[5]. El método se basa en el cálculo del *eigenvector*, denominado vector de Fiedler[6], correspondiente al segundo *eigenvalue* más pequeño de la matriz laplaciana $L(G = A_{i,j})$ del grafo G que se define:

$$A_{ij} = \begin{cases} -1 & \text{if } i \neq j \, and \, e_{i,j} \in E \\ 0 & \text{if } i \neq j \, and \, e_{i,j} \notin E \\ deg(v_i) & \text{if i = j} \end{cases}$$

Se ve fácilmente que la matriz laplaciana se define como $L(G) = D - A$, donde A es la matriz de adyacencia (ver Definición 2.5) del grafo y D es la diagonal de la matriz que representa el *eigenvector* para cada *eigenvalue* del grafo. En el caso de que el grafo tenga pesos en las aristas, W_e, el grafo G se define:

$$A_{ij} = \begin{cases} -W_e(e_{i,j}) & \text{if } i \neq j \, and \, e_{i,j} \in E \\ 0 & \text{if } i \neq j \, and \, e_{i,j} \notin E \\ \sum_{k=1}^{n} W_e(e_{i,j}) & \text{if i = j} \end{cases}$$

El principal objetivo del algoritmo de bisección espectral, como acabamos de comentar, es el cálculo del vector de Fiedler, además de minimizar el coste del número de aristas del grafo. En tal escenario, el *eigenvalue* más pequeño (λ_2) de L produce un coste óptimo (c) tal que $c \geqslant \frac{\lambda_2}{n}$, donde n es le número de nodos del grafo. Así, el *eigenvector* que corresponde a λ_2 es el vector que se denomina vector de Fiedler.

En general, esto produce una buena partición (pero no necesariamente la mejor). La partición obtenida por el segundo *eigenvalue* puede ser potencialmente mejorada tratando de maximizar

el *eigenvalue* mínimo de la matriz L al encontrar un cambio apropiado en los elementos de la diagonal D. Sin embargo, esta técnica requiere del cálculo de todos los *eigenvalues* de la matriz L. Esto no es muy factible computacionalmente para los grafos con muchos nodos.

La implementación original de este algoritmo utiliza el algoritmo de Lanczos (ver Definición 7) modificado para calcular el vector de Fiedler.

Definición 7. *El algoritmo de Lanczos[7] es un algoritmo iterativo creado por Cornelius Lanczos, que es una adaptación de los métodos iterativos para encontrar los eigenvalues más útiles y el eigenvector de una matriz de dimensión NxN realizando un número de operaciones M, donde el valor de M será más pequeño que N.*

Para dividir un grafo en un número de particiones de potencia de dos, el algoritmo de bisección espectral aplica los siguientes pasos:

1. Calcula el vector de Fiedler utilizando el algoritmo Lanczos sin factorizar.
2. Ordena los vértices de acuerdo con los tamaños de los componentes del vector Fiedler.
3. Asigna la mitad de los vértices a cada subconjunto.
4. Aplica los pasos 1-3 a cada subconjunto hasta obtener el número deseado de particiones.

La principal etapa de este algoritmo es el primer paso, el cálculo del vector de Fiedler, que nos da una idea de cómo funciona el algoritmo. A continuación, se muestra el pseudocódigo del algoritmo y luego se intenta explicar cómo funciona en la siguiente sección.

Given a connected graph $\mathcal{G}$, number the vertices in some way and form the Laplacian matrix $L(\mathcal{G})$.

Calculate the second-smallest eigenvalue λ_2 and its eigenvector u.

Calculate the median m_u of all the components of u.

Choose $\mathcal{V}_1 := \{v_i \in \mathcal{V} \mid u_i < m_u\}$, $\mathcal{V}_2 := \{v_i \in \mathcal{V} \mid u_i > m_u\}$, and, if some elements of u equal m_u, distribute the corresponding vertices so that the partition is balanced.

Figura 2.4: Spectral Bisection.

2.2.2 Ejemplo de codificación

La partición de grafos usando un método de partición espectral se basa la álgebra lineal. Sin embargo, los métodos espectrales para la partición de grafos son algoritmos computacionalmente caros. El algoritmo utilizado en la codificación calcula la matriz laplaciana L del grafo de entrada inicial (ver Figura 2.1). Pero antes de describir la codificación introduciremos los conceptos de matriz de adyacencia (ver Definición 8) y matriz laplaciana (ver Definición 9), ya nombrados anteriormente en la sección anterior.

Definición 8. *Una propiedad importante de los nodos de un grafo es su grado. El grado de un nodo es el número de aristas que están conectadas a él. Si para dos nodos A y B hay una arista uniéndolos, decimos que A y B son adyacentes. Así, se puede describir un grafo en una matriz de adyacencia. Esta es una matriz en la que se describe la conectividad de todos los nodos en el grafo. Una matriz de adyacencia A de un grafo G con N nodos es una matriz con el tamaño de N x N. La posición dentro de la matriz A_{ij} representa la conectividad del nodo j con el nodo i. Si hay una arista entre el nodo j y el nodo i, entonces el valor en la posición de matriz A_{ij} será uno, en todos los demás casos será cero.*

Esto se puede describir como tal:

$$A_{ij} = \begin{cases} 1 \text{ si hay una arista que va del nodo } j \text{ el nodo } i \\ 0 \text{ otros casos} \end{cases}$$

En un grafo no dirigido (el caso estudiado en este informe), el valor en la posición de la matriz de adyacencia A_{ij} será el mismo que el valor en la posición A_{ij}. Ver ejemplo de una matriz de adyacencia en la Figura 2.5.

Definición 9. *La matriz de grados es una matriz que se puede construir a partir de la matriz de adyacencia (ver Definición 8). Esto se puede hacer construyendo un vector de grados 1 x N. Ver ejemplo de vector de grados en la Figura 2.5. Este vector contiene en el elemento i la suma de la fila del número de columna j de la matriz de adyacencia. La matriz de grados D se construye multiplicando el vector de grados con la matriz de adyacencia de tamaño N x N. Con la matriz de grados D y la matriz de adyacencia A, se puede construir la matriz laplaciana L. Esto se puede hacer restando A de D.*

$$L_{ij} = D_{ij} - A_{ij}$$

Así, la matriz laplaciana L se puede definir mediante el siguiente conjunto de reglas:

$$A_{ij} = \begin{cases} k(i), & \text{if } i = j \\ -1, & \text{if } \exists \ e(i,j) \\ 0, & \text{otros casos} \end{cases}$$

A continuación, describimos como se obtienen los denominados *eigenvalues* y *eigenvectors* de la matriz laplaciana L mediante álgebra lineal directa. Los *eigenvalues* se ordenan en función de la magnitud: $\lambda_0 = 0 \leq \lambda_1 \leq \lambda_{n-1}$. El eigenvector correspondiente al *eigenvalue* más pequeño consiste exclusivamente a un vector de unos. Este vector no tiene ningún uso práctico ya que no se puede recuperar información útil de él. Sin embargo, el segundo eigenvector más pequeño contiene la información necesaria para hacer la partición espectral. Este vector es el vector de Fiedler y se define formalmente:

$$v^{-T} \cdot L(G) \cdot \vec{v} = \textstyle\sum_{i,j \in E} (x_i - x_j)^2$$

$$\lambda_1 = \frac{min}{\vec{v} \perp (1,\ldots,1)} \left(\frac{v^{-T} \cdot L(G) \cdot \vec{v}}{v^{-T} \cdot \vec{v}} \right)$$

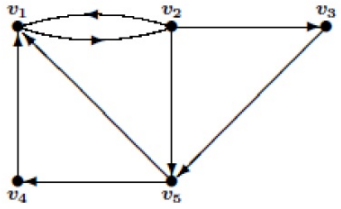
Cuando se calcula el *eigenvalue* correspondiente al vector de Fiedler de la matriz laplaciana L, se puede ver si el grafo está conectado o no. El *eigenvalue* también se conoce como el valor de conectividad algebraica [6]. El grafo tiene que estar conectado ya que la partición espectral no podría ser efectiva. Esto se debe a que la partición espectral se centra en minimizar el coste (número de conexiones) de las aristas. En un gráfico desconectado, esto sería cero.

Una vez que se encuentra el vector de Fiedler, se calcula el valor medio (pivote):

$$Pivote = \frac{\sum(\text{componentes de Fiedler})}{\text{Número de componentes}}$$

Con este pivote, los nodos que se corresponden con los *eigenvalues* se pueden dividir en dos partes. Todos los nodos con los *eigenvalues* correspondientes debajo del pivote se colocarán en la partición A y todos los demás nodos se colocarán en la partición B.

Este método se ha utilizado para la codificación del algoritmo de partición espectral. Probando con una ejecución de ejemplo podemos obtener los subconjuntos: A = {1, 2, 3, 4, 6, 9} y B = {0, 8, 5, 7}. En este caso, los subconjuntos no son del mismo tamaño, y tampoco se encuentran ordenados. Es una de las diferencias más significativas entre las codificaciones de Kernighan-Lin y Multilevel Spectral Bisection con esta codificación. Tampoco nos crea particiones perfectamente equilibradas (con el mismo número de nodos).



	V1	V2	V3	V4	V5	
V1	0	1	0	0	0	1
V2	1	0	1	0	1	3
V3	0	0	0	0	1	1
V4	1	0	0	0	0	1
V5	1	0	0	1	0	2

Figura 2.5: Ejemplo de matriz de adyacencia y vector de grados.

2.3 Multilevel Spectral Bisection

El particionamiento de grafos por *multilevel* es un método moderno, que reduce el tamaño de las particiones del grafo con la combinación de los vértices y las aristas sobre varios niveles, creando particiones del grafo cada vez más pequeñas y extensas, con muchas variaciones y combinaciones de diferentes métodos. Tampoco utiliza el grafo en sí, sino una representación matemática del mismo, una matriz. La técnica *multilevel* calcula el vector de Fiedler[6] de manera más eficiente.

2.3.1 Descripción

El algoritmo de bisección espectral *multilevel* se basa en la construcción sucesiva de una serie de pequeñas matrices, cada una de las cuales se aproxima a su predecesora más grande. En algún momento, la matriz es tan pequeña que el algoritmo de Lanczos[7] puede calcular el vector Fiedler[6] de la matriz laplaciana en un período de tiempo insignificante. Este vector se intercambia en el siguiente nivel, produciendo una aproximación al vector de Fiedler para la matriz laplaciana correspondiente a este nivel. El vector intercambiado es entonces mejorado usando el algoritmo de iteración del cociente de Rayleigh (ver Definición 10). El proceso continúa hasta que se obtiene el vector de Fiedler para la matriz original.

Definición 10. *En matemáticas, dado una matriz M y un vector distinto de cero x, el cociente de Rayleigh, se define como $R(M, x)$, tal que:*

$$R(M, x) = \frac{x \cdot Mx}{x \cdot x}$$

La idea de la técnica *multilevel* es la de reducir la magnitud de un grafo, calcular una partición de este grafo reducido, y finalmente proyectar esta partición en el grafo original. Dado un grafo inicial, intentamos reducirlo con un nuevo grafo que en cierto sentido es similar, pero con menos vértices, más pequeño. Ya que el nuevo grafo es más pequeño, será más barato calcular el vector de Fiedler del nuevo grafo. El cálculo del vector de Fiedler propio para MSB es se muestra en la Figura 2.6.

Function *Fiedler*($\mathcal{G}$)

If $\mathcal{G}$ is small enough **then**

Calculate $f = u_2$ using the Lanczos algorithm

else

Construct the smaller graph $\mathcal{G}'$ and (implicitly) the corresponding Laplacian L'.
f′ = *Fiedler*($\mathcal{G}'$)
Use f′ to find an initial guess $f^{(0)}$ for f.
calculate f from the initial guess $f^{(0)}$.

endif

Return f

Figura 2.6: Cálculo del vector de Fiedler para MSB.

En la Figura 2.6, se han omitido tres detalles bastante importantes: cómo construir el grafo más pequeño (fase de contracción), cómo encontrar una función inicial $f^{(0)}$ para f (fase de intercalación) y cómo calcular eficientemente f usando $f^{(0)}$ (fase de refinamiento). El algoritmo requiere que se agreguen estos tres elementos al nivel único básico del algoritmo de bisección espectral, descrito en la sección 2.2. Las tres fases se describen brevemente a continuación.

- **Contracción**. Se basa en la construcción de una serie de grafos más pequeños, que conserven la estructura del grafo original. En esta primera fase el grafo se reduce mediante la fusión de sus vértices. La fusión de los vértices se realiza de forma iterativa: de un grafo se crea un nuevo grafo más pequeño y de este nuevo grafo más pequeño se crea un grafo aún más pequeño. Esto se hace hasta que una pequeña cierta magnitud es alcanzada. De este modo se crean grafos con diferentes magnitudes.
- **Intercalación**. Dado un vector de Fiedler de un grafo más pequeño que el original, se basa en intercalar este vector al siguiente grafo más grande de manera que proporcione una buena aproximación al siguiente vector de Fiedler, mientras se crean grafos cada vez más grandes hasta llegar al tamaño del grafo original. Así se crea una partición del grafo más grande con la magnitud más pequeña del grafo.
- **Refinamiento**. La partición calculada iterativamente en la fase de intercalación se proyecta de nuevo al grafo original. En cada iteración se aplica un refinamiento usando el algoritmo del cociente de Rayleigh (ver Definición 10). El refinamiento se usa para asegurar que el cálculo del vector de Fiedler sea más preciso de manera eficiente. En este proceso se usa el algoritmo del cociente de Rayleigh (ver Figura 2.7).

Given an initial guess $(f^{(0)}, \lambda^{(0)})$ for an eigenpair (f, λ), set $v = f^{(0)}/\|f^{(0)}\|$ and

repeat

$\phi := v^T L v$
Solve $(L - \phi I)x = v$
$v := x/\|x\|$

until ϕ is "close enough" to λ

Figura 2.7: Algoritmo del cociente de Rayleigh.

Con estas mejoras añadidad respecto al algoritmo de bisección espectral, la técnica *multilevel* mejorará significativamente los resultados, tanto en términos de calidad y tiempo de ejecución (ver sección 3).

2.3.2 Ejemplo de codificación

Para la codificación del algoritmo MSB se ha usado uno de los algoritmos más exitosos para el problema de partición de grafos, el algoritmo METIS[9]. METIS es un algoritmo que se enfoca en minimizar el número de aristas cruzadas de cada partición y distribuir el número de nodos de manera uniforme entre las particiones. La Figura 2.8 muestra el algoritmo:

Function *ML-Partition*($\mathcal{G}$)

  If $\mathcal{G}$ is small enough **then**

    Find partition $(\mathcal{V}_1, \mathcal{V}_2)$ of $\mathcal{G}$ in some way.

  else

    Construct a smaller approximation $\mathcal{G}'$
    $(\mathcal{V}'_1, \mathcal{V}'_2) = ML\text{-}Partition(\mathcal{G}')$
    $(\tilde{\mathcal{V}}_1, \tilde{\mathcal{V}}_2)$ = Project-partition-up $(\mathcal{V}'_1, \mathcal{V}'_2)$
    $(\mathcal{V}_1, \mathcal{V}_2)$ = Refine-partition $(\tilde{\mathcal{V}}_1, \tilde{\mathcal{V}}_2)$

  endif

Figura 2.8: Algoritmo METIS.

Con METIS, como algoritmo MSB, los grafos se dividen en tres fases (ver sección 2.3.1). Aunque ene este caso describiremos las tres fases que hemos visto anteriormente, adaptadas para el algoritmo METIS. La primera fase la llamamos fase de engrosamiento, la segunda es la fase de partición y la tercera y última fase, la fase de no engrosamiento (ver Figura 2.8 de todas las fases).

A continuación, se muestra una breve explicación de estas fases dentro del algoritmo METIS.

La fase de engrosamiento. El objetivo principal de la fase de engrosamiento dentro del algoritmo METIS es reducir el grafo original a un grafo más pequeño, que aún contiene suficiente información para hacer una buena partición. Para ello se utiliza un algoritmo de coincidencia. En la literatura encontramos varios algoritmos de coincidencia. Los más destacados son el algoritmo de coincidencia pesada y el algoritmo de coincidencia aleatoria. En el algoritmo de coincidencia pesada inicialmente coinciden los pesos de las aristas más pesadas. En cambio, en el algoritmo de coincidencia aleatoria se seleccionan aleatoriamente los vértices más conectados (con más conexiones entre ellos).

La fase de partición. El objetivo principal de la fase de partición dentro del algoritmo METIS es dividir el grafo en dos partes balanceadas mientras se minimiza el número de conexiones entre los vértices. En esta fase normalmente se utilizan algoritmos de partición de grafos para una bisección inicial, como Spectral Bisection (ver sección 2.2) o Kernighan-Lin (ver sección 2.1). El algoritmo Kernighan-Lin es muy utilizado en esta fase.

La fase de no engrosamiento. El objetivo principal de la fase de no engrosamiento, o también denominada fase de descomposición, dentro del algoritmo METIS, es el reajuste de las particiones creadas por la fase de partición. Este proceso se realizará varias veces donde se aplica el algoritmo de refinamiento (ver Figura 2.7) para garantizar un buen equilibrio en cada partición.

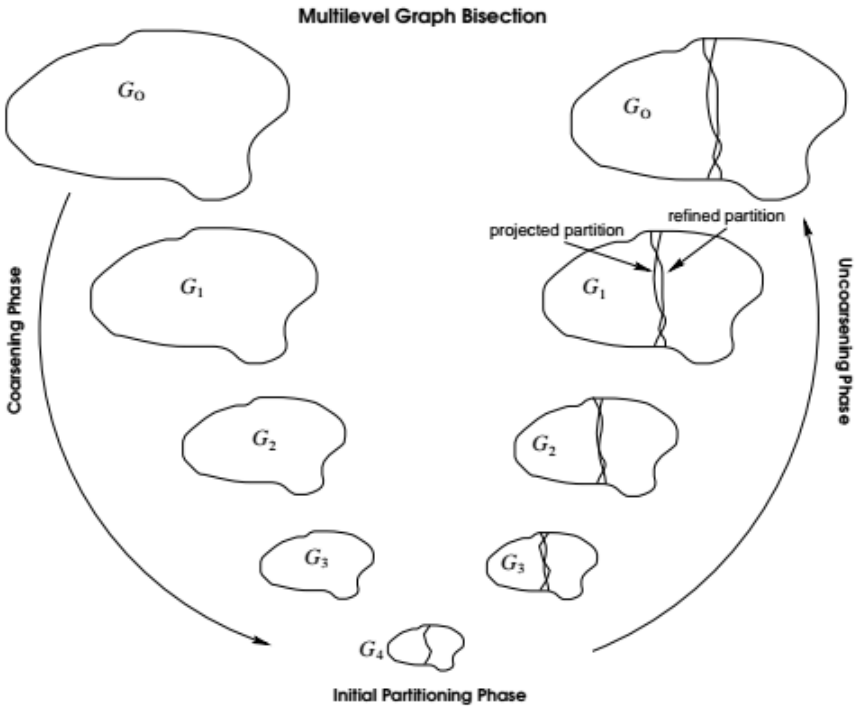


Figura 2.9: Fases de MSB.

Las características clave del algoritmo METIS son las siguientes:

- Proporciona particiones de alta calidad. Los experimentos con una gran cantidad de nodos muestran que METIS produce particiones que son consistentemente mejores que las producidas por otros algoritmos ampliamente utilizados.
- Es extremadamente rápido. Los experimentos en una amplia gama de grafos han demostrado que el algoritmo METIS es de uno a dos órdenes de magnitud más rápido que otros algoritmos de partición ampliamente utilizados.

Para la codificación del algoritmo descrita en este informe se escogió la librería de Python: NetworkX-METIS. *NetworkX-METIS* es un complemento para la librería de *NetworkX* que usa el algoritmo METIS para la partición de grafos. Por ejemplo, en una ejecución de la codificación, donde la entrada es el grafo de la Figura 2.1, se han creado los subconjuntos: A = {0, 1, 3, 7, 8} y B = {2, 4, 5, 6, 9}. Estos subconjuntos tienen el mismo tamaño y sus elementos están ordenados, igual que en la sección 2.1.2.

En la próxima sección que comparamos los tres algoritmos veremos que es muy eficiente.

3. Comparativa entre los algoritmos

En esta sección, veremos una comparativa entre los algoritmos que hemos descrito en las secciones previas, tras ser aplicados sobre diferentes grafos aleatorios, también conocidos como grafos de Erdös y Renyi[11] o grafos binomiales. Pero antes de ver la comparativa, en primer lugar, definimos el concepto de grafo aleatorio.

Se dice que un grafo es aleatorio si la presencia y colocación de sus aristas siguen una distribución aleatoria. Por tanto, un grafo aleatorio no puede diseñarse bajo ningún criterio concreto. Los grafos aleatorios también son conocidos como grafos de Erdös y Renyi porqué utilizan el modelo de Erdös y Renyi. El nombre de este modelo proviene de los matemáticos Paul Erdős y Alfréd Rényi, quienes lo presentaron por primera en 1959.

El modelo de Erdös y Renyi (a veces abreviado como modelo ER), es el modelo que se ha empleado para la generación de los grafos aleatorios de prueba que se usan en la comparativa de las codificaciones. Para ello, los grafos se han construido mediante la conexión de los nodos al azar. Cada arista se incluye en el grafo con una probabilidad de p independiente de las otras aristas del grafo. De manera equivalente, todos los grafos con n nodos y m aristas tienen la misma probabilidad.

La probabilidad p se define:

$$p^m(1-p)^{\binom{n}{2}-m}$$

En la implementación de los grafos aleatorios se ha utilizado la librería de Python: NetworkX. Concretamente el método que se ha utilizado nos devuelve un grafo no dirigido, donde se elige cada una de las aristas posibles con probabilidad $p = 0.7$, según el número de nodos de entrada n. En particular, el caso $p = 0.7$ se corresponde con el caso en el que los n vértices se eligen con mayor probabilidad ($p > 0.5$). Los pesos de las aristas también se han generado aleatoriamente dentro de un intervalo de 1 a 20.

Podemos observar la codificación acabada de describir, que se ejecuta en tiempo $O(n^2)$.

```
G = nx.erdos_renyi_graph(n, 0.7)

for u, v in G.edges():
  if u != v:
    G[u][v]['label'] = random.randrange(1, 20)
```

Y en la siguiente tabla, se presenta la comparación entre los algoritmos codificados sobre los grafos aleatorios generados en este informe, en términos de tiempo computacional. Se muestra el tiempo (en segundos) en completar las ejecuciones de los algoritmos sobre los grafos aleatorios para distintos números de vértices n. Para ello se ha utilizado un MacBook Pro, con un procesador de cuatro núcleos de 2.8 GHz y con 16 GB de memoria RAM. Es de esperar que, a mayor número de vértices, los algoritmos hayan tenido mayor tiempo de ejecución.

n	307	552	861
Kernighan-Lin	0.0117	0.0214	0.0451
Spectral Bisection	0.0021	0.0031	0.0060
MSB	0.0025	0.0031	0.0037

Tabla 3.1: Tabla comparativa de los algoritmos.

Después de la comparación, podemos ver como los algoritmos Kernighan-Lin, Spectral Bisection y Multilevel Spectral Bisection (MSB) tienen unos tiempos de ejecución similares cuando el número de vértices n es más pequeño. Fenómeno que cambia cuando casi se duplican el número de vértices a 552.

Además, si nos fijamos en el algoritmo Kernighan-Lin, cuando el número de vértices casi se duplica a 552, el tiempo de ejecución del algoritmo también se duplica. Parece que el algoritmo va aumentando el tiempo de ejecución a medida que aumenta el número de vértices de manera proporcionada.

En cambio, el algoritmo Spectral Bisection obtiene unos resultados muy parecidos con el algoritmo Multilevel Spectral Bisection (MSB), pero cuando el número de vértices empieza a ser muy elevado, la eficiencia de este algoritmo baja drásticamente. Que no es el caso del algoritmo Multilevel Spectral Bisection (MSB).

Como conclusión general de esta breve comparación entre los algoritmos podemos decir que el algoritmo Multilevel Spectral Bisection (MSB) es el más eficiente, con tiempos de ejecución más pequeños, y el algoritmo Kernighan-Lin es el algoritmo menos eficiente, con tiempos de ejecución más grandes. También podemos observar que los tiempos de ejecución no son muy grandes, ya que solo se ha medido el tiempo de ejecución de los algoritmos y además el número de vértices totales no es muy grande. Aun así, nos ha permitido ver las diferencias en tiempo de ejecución de las tres codificaciones que se han implementado y descrito en este informe.

La elección del tiempo de ejecución frente a la calidad de las particiones para la comparación de los algoritmos ha dependido del hecho que el problema de partición de grafos sea un problema NP-completo[2]. Recordemos que esto quiere decir que los tres algoritmos puede que no nos ofrezcan una solución óptima en cada ejecución, pero sí que nos dan una buena solución, al menos la mayor parte del tiempo. Por esto, como es difícil comparar la calidad de las particiones entre los diferentes algoritmos, se elijó la medida del tiempo de ejecución de cada algoritmo. Por otro lado, bajo el contexto de matriz-vector de los algoritmos Spectral Bisection y Multilevel Spectral Bisection (MSB), también solo nos interesaba el tiempo de ejecución total.

Concretamente en este caso, podría ser que, si solo utilizamos una vez una determinada matriz, el algoritmo que entrega solo particiones de calidad media en general podría ser más rápido que el algoritmo más lento con particiones de mejor calidad. Pero si usamos la misma matriz (o diferentes matrices con el mismo grafo) a menudo, el algoritmo más lento podría ser preferible. De hecho, hay incluso más factores a considerar. Hasta ahora hemos querido que las diferentes particiones tuvieran del mismo tamaño. Pero como hemos visto en la sección 2.2, podría ser una ventaja aceptar particiones de un tamaño ligeramente diferente para lograr una mejor solución. Todo esto debería demostrar que no existe un mejor algoritmo único para todas las situaciones y que los diferentes algoritmos descritos en las anteriores secciones pueden tener diferentes aplicaciones.

Aunque en general, los algoritmos de particionamiento complejos tardan más en calcular las particiones, ya que crean particiones que a menudo son más equilibradas y/o minimizan con mayor número las conexiones entre particiones, podemos considerar que algoritmos de partición menos complejos sean más eficientes, porqué el objetivo principal de los algoritmos de partición siempre es reducir el tiempo de cálculo en una ejecución. En consecuencia, en algunos casos un algoritmo menos complejo sería preferible a uno más complejo. También nos hemos encontrado con que la eficiencia cambia con el número de particiones y que los algoritmos producen resultados diferentes dependiendo de los atributos del grafo de entrada inicial.

4. Conclusiones

En este informe se ha intentado describir teóricamente las codificaciones de los algoritmos que se han implementado para el trabajo final de la asignatura. También como parte de una pequeña investigación sobre el problema de la partición de grafos y algunas codificaciones existentes.

La elección de los algoritmos para su codificación se ha basado en mostrar los algoritmos más conocidos y utilizados para el problema de la partición de grafos. Se han implementado en Python porqué es un lenguaje de programación cuya filosofía hace hincapié en la legibilidad de su código, que es un lenguaje muy utilizado en Inteligencia Artificial, y que además es un lenguaje interpretado, dinámico y multiplataforma.

Al final de este proceso hemos utilizado diferentes grafos aleatorios no dirigidos para la comparación de los tres algoritmos descritos y codificados. Estos grafos no son muy grandes, ya que lo difícil de este trabajo final era la implementación de los tres algoritmos y se la ha dado más peso en este caso. Aun así, al comparar los algoritmos hemos podido observar su comportamiento, como por ejemplo que el algoritmo Multilevel Spectral Bisection es el algoritmo más eficiente que se ha codificado y que el algoritmo Kernighan-Lin es el menos eficiente. También hemos observado una diferencia importante durante la comparación, que es que el algoritmo de bisección espectral nos ha creado dos particiones no equilibradas.

Como cualquier de las codificaciones nos puede dar una solución óptima, en la comparación de los algoritmos solo se ha tenido en cuenta su tiempo de ejecución. En ese tiempo de ejecución, no se ha añadido el tiempo empleado en la creación de los grafos, antes y después de las particiones.

Otro motivo es que cada vez que hemos ejecutado uno de los algoritmos, las soluciones han cambiado. Así que, por eso también los hemos comparado por tiempo de ejecución. Como sabemos, al ser algoritmos metaheurísticos, todas las soluciones pueden ser óptimas y válidas. Eso quiere decir que, aunque un algoritmo sea más eficiente, no obtendrá siempre la mejor solución. Porqué como hemos visto, a menudo no existe una relación entre el tiempo de ejecución y la calidad de la solución. Algunos algoritmos funcionan muy rápido pero solo encuentran una solución de calidad media, mientras que otros que tardan mucho tiempo pueden ofrecen soluciones excelentes. La elección del tiempo frente a la calidad se ha descrito más detalladamente en la sección 3.

Para finalizar, comentar que en las secciones anteriores siempre se ha hablado de minimizar el tamaño de las conexiones entre los vértices de los grafos en ciertas condiciones. Pero para todos menos algunos gafos bien estructurados o pequeños (casos no triviales), la minimización real no es factible porque tomaría mucho tiempo. En pocas palabras, hemos comprobado que no hay un algoritmo conocido que sea mucho más rápido que probar todas las combinaciones posibles y que hay pocas esperanzas de que se encuentre alguno. Un poco más exacto (pero aún ignorando algunos fundamentos de la teoría NP), significa por ejemplo, que la bisección del grafo cae en una gran clase de problemas, todos los cuales pueden transformarse entre sí y para los cuales no hay algoritmos conocidos que puedan resolver el problema en tiempo polinómico, con un tamaño del grafo de entrada muy grande. Entonces, en lugar de encontrar la solución óptima, por eso hemos recorrido a la metaheurísitica.

Bibliografia

[1] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970. 3, 6, 7, 9

[2] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990. 4, 18

[3] C. P. Ravikumar and George W. Zobrist. *Parallel Methods for VLSI Layout Design*. Greenwood Publishing Group Inc., USA, 1995. 7, 8

[4] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. In *19th Design Automation Conference, Las Vegas, NV, USA, 14-16 June, 1982*. IEEE, 1982. 8

[5] H. Simon A. Pothen and K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *IAM Journal of Matrix Analysis*, 3(11):430–452, 1990. 10

[6] Hongyuan Zha Ming Gu Chris HQ Ding, Xiaofeng He and Horst D Simon. A min-max cut algorithm for graph partitioning and data clustering. In *In Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference*. IEEE, 2001. 10, 13

[7] C Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards*, 45(4):255, 1950. 11, 13

[8] B. Hendrickson and R. Leland. The chaco user's guide, version 2.

[9] G. Karypis and V. Kumar. Metis: Unstrctured graph partitioning and sparse matrix ordering system, version 2.0. 15

[10] A. Gupta. Wgpp: Watson graph partitioning (and sparse matrix ordering) package.

[11] P. Erdős and A. Rényi. On random graphs. *Publicationes Mathematicae*, 6:290–297, 1959. 17