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, las codificaciones 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 los recursos. 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 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 en las particiones.

El enfoque del informe se centra en la división de grafos en dos partes. Se ha elegido la partición en dos partes porque es la forma más sencilla de comparar las particiones con un método de *"bisection"* (que siempre crea dos particiones). 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 utiliza para examinar el problema de partición de grafos: Kernighan-Lin, Spectral Bisection y Multilevel Spectral Bisection (ver sección 2).

Su objetivo se centra en particionar diferentes grafos usando los algoritmos anteriores. Inicialmente se han usado grafos más pequeños para describir los algoritmos de partición y la hora de compararlos se han usado grafos más grandes para probar si el código sigue produciendo computacionalmente buenos resultados.

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, es decir, se intenta maximizar el peso de las aristas que pertenecen a un subgrafo. 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 riguroso.

Definición 1. *Un grafo G es un par ordenado $G = (V, E)$, donde V es un conjunto de vértices y E es un conjunto de aristas.*

Sea $G = (V, E)$ un grafo formado por un conjunto de n vértices V y un conjunto de m aristas E. Dado el grafo G, 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 es mínimo

De las particiones sobre el conjunto de vértices V, una solución S es representada por un conjunto 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 conjunto de vértices V , mientras que la restricción (1.3) obliga a que ningún par de vértices adyacentes sean asignados a la misma clase, es decir, que todas las clases de pesos en la solución deben formar conjuntos independientes.

2. Algoritmos

Debido a la *"intratabilidad"* del problema de la partición de grafos, como hemos comentado, 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 para 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.

En el siguiente apartado se describen los algoritmos Kernighan-Lin[1] (ver apartado 2.1), Specrtal Bisection (ver apartado 2.2) y Multilevel Spectral Bisection (ver apartado 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. 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 para describir cada codificación se ha utilizado un grafo inicial (ver Figura 2.1). Este grafo es un grafo no dirigido (ver Definición 2) de diez nodos con pesos en sus aristas.

Definición 2. *Un grafo no dirigido es un grafo (ver Definición 1) donde:* $V \neq \emptyset$ *y* $E \subseteq \{x \in \mathcal{P}(V) : |x| = 2\}$ *es un conjunto de pares no ordenados de elementos de* V*. Un par no ordenado es un conjunto de la forma* $\{a\ ,\ b\}$*, de manera que* $\{a,\ b\} = \{b,\ a\}$*. En los grafos, los subconjuntos de pares no ordenados pertenecen al conjunto potencial de* V*, denotado* $\mathcal{P}(V)$*, y son de cardinalidad 2.*

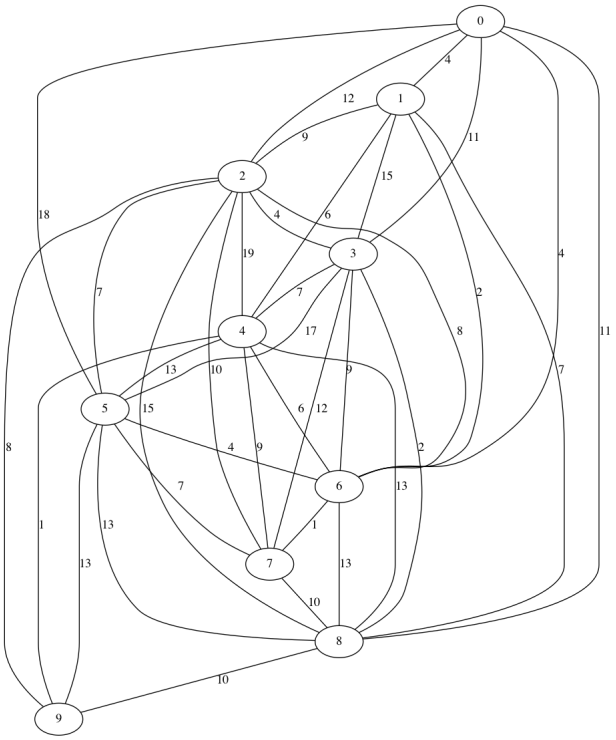


Figura 2.1: Grafo inicial.

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 es:

- **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 general óptima.
- **Determinista**. Porque se obtendrá el mismo resultado cada vez que se aplique el algoritmo.

El algoritmo K-L, como acabamos de describir, no crea particiones, sino que las mejora iterativamente. 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 realizan una partición 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 2) en dos subconjuntos disjuntos (ver Definición 3) 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 4), 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. Después de emparejar los vértices y maximizar 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 3. *A y B son dos subconjuntos disjuntos si $A \cup B$ y $A \cap B \neq 0$.*

Definición 4. *Dado un conjunto finito C, un algoritmo voraz devuelve un conjunto S tal que $S \subseteq C$ y que además cumple con las restricciones del problema inicial. A cada conjunto 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 detalladamente, sea a un elemento del subconjunto A y b un elemento del subconjunto B, para cada $a \in A$, I_a es 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 es 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 $b \in B$.

Así podemos decir que existe una diferencia D entre la suma de los costes externos e internos s. Si formalizamos obtenemos:

$$D_s = E_s - I_s$$

Y entonces si a y b se intercambian, la ganancia se calcula como:

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

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

El algoritmo intenta encontrar una solución óptima de operaciones de intercambio entre los elementos de A y B que maximice la ganancia ($T_{antigua} - T_{nueva}$) y luego ejecuta las operaciones, produciendo una partición del grafo en A y B.

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

```
function Kernighan-Lin(G(V, E)) is
 determine a balanced initial partition of the nodes into sets A and B

 do
  compute D values for all a in A and b in B
  let gv, av, and bv be empty lists
  for n := 1 to |V| / 2 do
    find a from A and b from B, such that gain is maximal
    remove a and b from further consideration in this pass
        add g to gv, a to av, and b to bv
        update D values for the elements of A = A \ a and B = B \ b
  end for
  find k which maximizes g_max, the sum of gv[1], ..., gv[k]
  if g_max > 0 then
    Exchange av[1], av[2], ..., av[k] with bv[1], bv[2], ..., bv[k]
  until (g_max <= 0)

return G(V, E)
```

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.

Una gran ventaja del algoritmo es que no se detiene incluso aceptando ganancias negativas, sigue esperando que las ganancias posteriores sean más grandes 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 esta capacidad sea crucial, el algoritmo tiene unas cuantas desventajas:

- 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.

C. Fiduccia y R. Mattheyses realizaron importantes avances prácticos en [4], quienes mejoraron el algoritmo de Kernighan-Lin[1] de tal manera que, cada partición se ejecuta en $O(n^2)$, en lugar de $O(n^2 \, log \, n)$. La reducción se logra en parte eligiendo nodos individuales para intercambiar en lugar de parejas de vértices.

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 libraría de Python: NetworkX. El algoritmo 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 5. *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 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, los pasos que sigue 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 la línea. Este número se llama *"cut-size"* y el objetivo es disminuirlo, es 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 restando eso del número de conexiones que cada vértice tiene con vértices en la otra partición.
- Determina la ganancia máxima intercambiando dos nodos. La ecuación de ganancia es la que se muestra en 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 *"cut-size"* original para obtener el nuevo *"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,

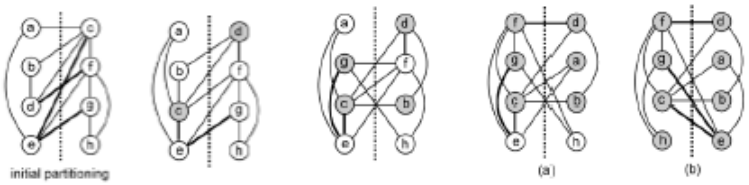


Figura 2.2: Kernighan-Lin.

2.2 Spectral Bisection

2.2.1 Descripción

2.2.2 Ejemplo de codificación

La partición de grafos usando un método de partición espectral se basa 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 6) y matriz laplaciana (ver Definición 7).

Definición 6. *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:*

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

En un grafo no dirigido (el caso estudiado en este informe), el valor en la posición en la matriz de adyacencia de A_{ij} será el mismo que el valor en la posición A_{ij}. Ver ejemplo de una matriz de adyacencia en la Figura 2.3.

Definición 7. *La matriz de grados es una matriz que se puede construir a partir de la matriz de adyacencia (ver Definición 6). Esto se puede hacer construyendo un vector de grado de 1 x N, como ejemplo podemos verlo en la Figura 2.3. 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 se construye multiplicando el vector de grados con la matriz de identidad 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.*

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

La matriz laplaciana se puede definir mediante el siguiente conjunto de reglas:

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

La matriz laplaciana tiene la entrada -1 donde la matriz de adyacencia tiene un valor distinto de cero. Los grados se representan en diagonal. Todos los demás valores son cero.

A continuación, describimos como se obtienen los denominados *eigenvalues* y *eigenvectors* de la matriz laplaciana. Se obtienen mediante álgebra lineal directa. Los *eigenvalues* (y sus *eigenvectors*

correspondientes) 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 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 necesitaba para hacer la partición espectral. Este vector también se conoce como el vector de Fiedler[5] y se define formalmente:

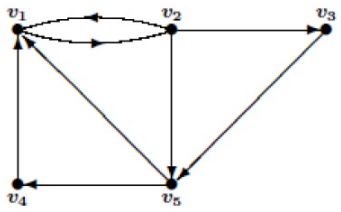
$$v^{-T} \cdot L(G) \cdot \vec{v} = \sum_{i,j \in E} (x_i - x_j)^2$$

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

Cuando se calcula el *eigenvalue* correspondiente del vector de Fiedler de la matriz laplaciana, se puede ver si el grafo está conectado o no. Este número también se conoce como la conectividad algebraica [6]. Esto es útil ya que la partición espectral solo puede ser efectiva si el grafo está conectado. Esto se debe a que la partición espectral se centra en minimizar el peso de las aristas. En un gráfico desconectado, esto sería cero. Una vez que se encuentra el vector de Fiedler, se calcula el valor medio:

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

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 particionan 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. Que después de ejecutar un partición 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 (Spectral Bisection). El algoritmo de partición espectral se llama de forma recursiva para crear las dos particiones. Esto indica que el número de particiones se ha limitado expresamente a dos. Sin embargo, el algoritmo no es perfecto, como vemos, no 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 2.3: Matriz de adyacencia.

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.

2.3.1 Descripción

El algoritmo Multilevel Spectral Bisection (MSB) descrito en la sección 2.3.1 ha resultado ser muy eficiente (ver sección 3). En MSB, se transfiere información aproximada sobre el segundo *eigenvector* del grafo original entre niveles, mejorando esta información en cada nivel y finalmente, se usa otra vez el *eigenvector* para dividir el grafo.

Se muestra el pseudocódigo del algoritmo a continuación:

```
Function ML_Partition(G)

If G is small enough then
Find partition (V1, V2) of G in some way

else

Construct a smaller approximation G'
(V1', V2') = ML_Partition(G')
(V1", V2") = Project-partition-up (V1', V2')
(V1, V2)   = Refine-partition (V1", V2")

endif

Return (V1, V2)
```

La misma idea se puede usar para dividir el grafo en p partes a la vez. En realidad, hay toda una familia de algoritmos que puede dividir grafos en p partes a la vez. Algunos de estos algoritmos se encuentran implementados en paquetes de software como Chaco[6], METIS[7] o WGPP[8].

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.

METIS[7] 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. Con METIS, los grafos se dividen en tres fases. La primera fase es la fase de engrosamiento, la segunda la fase de partición y la tercera y última fase la fase de no engrosamiento (ver Figura 2.4). A continuación, se muestra una breve explicación de estas fases.

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. Esto sucede con el uso de un algoritmo de coincidencia. Hay varios algoritmos de coincidencia, los más destacados son el algoritmo de coincidencia pesada y el algoritmo de coincidencia aleatoria. El algoritmo de coincidencia pesada inicialmente coincide con los pesos de las aristas más pesados. En cambio, el algoritmo de coincidencia aleatoria selecciona aleatoriamente los vértices más conectados (con más conexiones entre dos vértices).

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 se utilizan algoritmos de partición de grafos para una bisección inicial, como Spectral Bisection (ver sección 2.2), Kernighan-Lin (ver sección 2.1), etc.

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

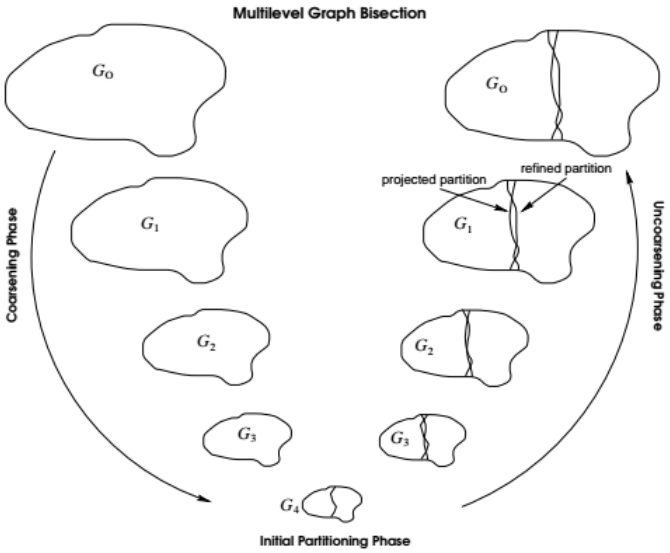


Figura 2.4: Matriz de adyacencia.

Las características clave del algoritmo 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. Las particiones producidas por el algoritmo METIS son consistentemente de un 10% a un 50% mejores que las producidas por algoritmos de partición espectral.
- 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 en este informe se escogió la librería de Python: NetworkX-METIS. *NetworkX-METIS* es un complemento para la libraría 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.

3. Comparativa entre los diferentes algoritmos

Para finalizar, veremos una comparativa entre los algoritmos que hemos analizado en las secciones previas, tras ser aplicados sobre grafos aleatorios, también conocidos como grafos de Erdös y Renyi o grafos binomiales. Pero, 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 con ningún criterio concreto. La noción del modelo de un grafo aleatorio incluye el límite entre cada par de nodos con igual probabilidad, independientemente de los extremos. Concretamente del modelo se denomina: modelo de Erdös y Renyi. El nombre del modelo proviene de los matemáticos Paul Erdős y Alfréd Rényi[9], quienes lo presentaron por primera en 1959.

El modelo Erdös y Renyi (a veces nombrado en la literatura abreviado como modelo ER), es el método que se ha empleado en la generación de los grafos aleatorios. Para ello, un grafo se construye 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 de:

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

Para 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 todos los grafos en n vértices se eligen con mayor probabilidad. Los pesos de las aristas también se han generado aleatoriamente dentro de un intervalo de pesos de 1 a 20.

Podemos observar la codificación utilizada a continuación, 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)
```

Finalmente, en la tabla siguiente, se presenta una comparación entre los algoritmos codificados sobre los grafos aleatorios de este trabajo en términos de tiempo computacional. Se muestra el tiempo (en segundos) en completar ejecuciones sobre conjuntos de 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 tengan mayor tiempo de ejecución.

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

Podemos ver como los algoritmos Kernighan-Lin, Spectral Bisection y Multilevel Spectral Bisection 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.

Si nos fijamos con el algoritmo Kernighan-Lin, cuando el número de vértices inicialmente casi se dobla, el tiempo de ejecución también lo hace. 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 Multilevel Spectral Bisection, pero cuando el número de vértices empieza a ser muy elevado, la eficiencia baja drásticamente. Que no es el caso del algoritmo Multilevel Spectral Bisection.

Como conclusión global a la comparación entre los algoritmos podemos decir que el algoritmo Multilevel Spectral Bisection es el más eficiente para grafos no dirigidos que se ha codificado, y el algoritmo Kernighan-Lin es el menos eficiente.

En las secciones anteriores siempre hemos hablado de minimizar el tamaño de partición en ciertas condiciones. Pero para todos menos algunos gafos bien estructurados o pequeños, la minimización real no es factible porque tomaría mucho tiempo. Encontrar la solución óptima para el problema de partición de grafos es para todos los casos no triviales que se sabe que son NP-completos[2].

En pocas palabras, esto significa que no hay un algoritmo conocido que sea mucho más rápido que probar todas las combinaciones posibles y hay pocas esperanzas de que se encuentre uno. Un poco más exacto (pero aún ignorando algunos fundamentos de la teoría NP) significa 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 en el tamaño de la entrada.

Entonces, en lugar de encontrar la solución óptima, recurrimos a la heurística. Es decir, tratamos de usar algoritmos que pueden no ofrecer la solución óptima cada vez, pero que darán una buena solución al menos la mayor parte del tiempo. Como veremos, a menudo hay una compensación entre el tiempo de ejecución y la calidad de la solución. Algunos algoritmos se ejecutan bastante rápido, pero solo encuentran una solución de calidad media, mientras que otros tardan mucho tiempo pero ofrecen soluciones excelentes e incluso otros se pueden ajustar entre ambos extremos.

La elección del tiempo frente a la calidad depende de la aplicación prevista. Para el diseño VLSI o el diseño de la red, puede ser aceptable esperar mucho tiempo porque una solución aún mejor puede ahorrar dinero real.

Por otro lado, en el contexto de la multiplicación escasa de matriz-vector, solo nos interesa el tiempo total. Por lo tanto, el tiempo de ejecución para la partición del gráfico debe ser menor que el tiempo ahorrado por la multiplicación de matriz-vector más rápida. Si solo utilizamos una determinada matriz una vez, un algoritmo rápido que entrega solo particiones de calidad media en general podría ser más rápido que un algoritmo más lento con particiones de mejor calidad. Pero si usamos la misma matriz (o diferentes matrices con el mismo gráfico) a menudo, el algoritmo más lento podría ser preferible.

De hecho, hay incluso más factores a considerar. Hasta ahora queríamos que las diferentes particiones tuvieran el mismo peso. Como veremos en la sección 2.2, podría ser una ventaja aceptar particiones de un tamaño ligeramente diferente para lograr un mejor

tamaño de corte Todo esto debería demostrar que no existe un mejor algoritmo único para todas las situaciones y que los diferentes algoritmos descritos en los siguientes capítulos tienen sus aplicaciones.

Después de crear las particiones, se calcula la distribución de nodos y bordes dentro de las particiones y entre las particiones. Esto ofrece una idea de cómo las proporciones de partición se desvían entre sí. Estas particiones se utilizan en dos algoritmos de validación diferentes, Pagerank y Breadth First Search, para ver la diferencia en la comunicación entre las particiones. Este proyecto se centrará en particionar usando los valores de partición en los nodos del gráfico completo. Los algoritmos de partición y los algoritmos distributivos se realizarán inicialmente para gráficos más pequeños. Luego, se utilizarán tipos de gráficos más grandes y diferentes con diferentes atributos para probar si el código sigue produciendo resultados precisos.

además la eficiencia cambia con el número de particiones.

Los algoritmos de partición probablemente producirán resultados diferentes dependiendo de los atributos del gráfico de entrada.

Los algoritmos de particionamiento complejos tardan más en calcular las particiones. Sin embargo, crean particiones que a menudo son más equilibradas y / o minimizan la comunicación entre particiones mejor que los algoritmos de partición menos complejos. En general, el objetivo es reducir el tiempo de cálculo de un proceso. En consecuencia, en algunos casos un algoritmo menos complejo sería preferible a uno más complejo

Considere un ejemplo para elaborar este punto un poco más. El tiempo para calcular un gráfico grande y bien dividido con una buena distribución de la carga de trabajo y conexiones de borde mínimas podría ser bastante alto, digamos 3 horas. Estimando el tiempo de 30 minutos para calcular los resultados con el uso de estas particiones, esto suma un total de 3.5 horas. Si, por otro lado, usamos un método de particionamiento menos refinado, el proceso de particionamiento en sí mismo puede demorar una hora y el tiempo posterior para calcular resultados 1.5 horas con un tiempo total de 2.5 horas. En este caso, sería rentable utilizar el método de partición menos refinado. Sin embargo, si queremos hacer múltiples cálculos utilizando los gráficos particionados, un método más refinado sería mejor. Un enfoque ampliamente utilizado para particionar gráficos grandes es el complejo algoritmo METIS.

4. Conclusiones

Hemos utilizado el mismo grafo aleatorio para los tres ejemplos descritos de las codificaciones de los algoritmos.

El algoritmo Multilevel Spectral Bisection es el más eficiente que se ha codificado, y el algoritmo Kernighan-Lin es el menos eficiente.

Para la comparativa solo se ha tenido en cuenta el tiempo de ejecución de los algoritmos. No se ha añadido el tiempo de ejecución empleado en la creación de los grafos, antes y después de la partición.

Cada vez que ejecutamos un algoritmo las soluciones cambian. Así que, por eso, los hemos comparado por tiempo de ejecución. Ya que, al ser algoritmos metaheurísticos, todas las soluciones pueden ser óptimas y válidas. Eso quiere decir, que, aunque un algoritmo sea más eficiente, no obtendrá la solución óptima.

Como hemos visto, a menudo 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 medio calidad mientras que otros tardan mucho tiempo pero ofrecen soluciones excelentes e incluso otros se puede sintonizar entre ambos extremos.

La elección del tiempo frente a la calidad.

Las particiones pequeñas en general podrían ser más rápidas que un algoritmo más lento con mejor calidad particiones Pero si usamos la misma matriz (o diferentes matrices con el mismo gráfico) a menudo, el algoritmo más lento podría ser preferible. tamaño de corte Todo esto debería demostrar que no existe un mejor algoritmo único para todas las situaciones. y que los diferentes algoritmos descritos en los siguientes capítulos tienen todos sus aplicaciones.

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, 16

[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. 9

[5] 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. 12

[6] B. Hendrickson and R. Leland. The chaco user's guide, version 2. 13

[7] G. Karypis and V. Kumar. Metis: Unstrctured graph partitioning and sparse matrix ordering system, version 2.0. 13

[8] A. Gupta. Wgpp: Watson graph partitioning (and sparse matrix ordering) package. 13

[9] P. Erdős and A. Rényi. On random graphs. *Publicationes Mathematicae*, 6:290–297, 1959. 15