Algoritmia para Problemas Difíciles
Práctica 1: Mínimo Corte

José Daniel Subías Sarrato (NIA: 759533)
Fernando Peña Bes (NIA: 756012)

31 de diciembre de 2020

1. Descripción del problema

Por motivos fiscales *Amazon* ha decidido partir su negocio en r partes donde r es un entero mayor o igual que 2 que se conoce a priori. Como empresa, quiere que esta partición perjudique lo menor posible al conjunto de sus clientes. Esto también implica que el daño que pueda sufrir su volumen de ventas debe ser el mínimo, ya que algunos clientes pueden decidir no realizar una compra cuando deben dividirla en varias compras a distintos proveedores. La partición va a hacerse de manera disjunta, por lo que un producto que esta en una de las nuevas $Amazon_r$ no puede estar ninguna otra.

Como punto de partida, en esta práctica se dispone de un una tabla de productos que los relaciona entre sí. En dicha tabla T, los productos están relacionados de forma que $T[i, j]$ es `true` si los productos indexados por i y j han sido comprados juntos alguna vez y `false` en otro caso. Dicha tabla de información se encuentra en un fichero con un formato específico. La primera linea del fichero corresponde con el número total de productos de *Amazon*. El resto de caracteres son 0 y 1 correspondientes a los valores de la matriz por filas $T[1, 1]$, $T[1, 2]$, $T[2, 2]$, etc.; teniendo en cuenta que los saltos de línea, blancos y tabuladores pueden aparecer en cualquier sitio.

El objetivo será buscar o diseñar un algoritmo que minimice el número de pares de productos que han sido comprados juntos alguna vez y se asignan a distinto proveedor.

Tal y como se pide en el enunciado, se realizará un análisis de costes temporales del algoritmo utilizado. Adicionalmente, aunque no tenga gran relevancia en el análisis del algoritmo, se implementará una estructura de datos para el almacenamiento de los productos.

2. Generación del conjunto de datos

A la hora de realizar análisis asintóticos del coste de un algoritmo, es de gran importancia contar un voluminoso conjunto de datos de prueba. Dicho conjunto ha de contener entradas de altamente variadas para probar la eficiencia del algoritmo en función de diferentes tamaños. Es por esta razón que con anterioridad se entregó un dataset formado por una serie de datos de prueba lo mas variado posible. El procedimiento para la generación de los casos de prueba puede verse en el fichero `LEEME.md` entregado con anterioridad.

3. Almacenamiento de los productos

Como ya se ha comentado, en el enunciado se pide explícitamente que se implemente una estructura de datos para el almacenamiento de los productos de *Amazon*. Dicha información puede estar compuesta por un número variable de atributos, y en muchos casos puede darse la situación de que esta información no esté disponible.

Siguiendo las recomendaciones del enunciado, se optó por implementar una tabla hash para guardar los datos de los productos, puesto que el tamaño del conjunto de productos no varía durante las pruebas y se sabe que esta limitado a menos de 10.000 productos.

Por otro lado, se trata de una estructura de datos que tanto para adición de datos y búsqueda tiene un coste cuasi-constante No obstante, para que esto se cumpla en la práctica, se debe usar una función de dispersión adecuada para el problema concreto. Teniendo en cuenta que la claves de los productos en la tabla hash se podían corresponder con el identificador de los mismos y se trataba de una cadena de caracteres, se pensó en utilizar la función de dispersión PEARSON90. Esta función tiene la ventaja de que, al hacer uso de una tabla auxiliar de números aleatorios, las cadenas muy similares se distribuían en posiciones muy distintas de la tabla. Además soporta anagramas.

A continuación, se muestra el algoritmo de la función hash usada. Su implementación puede verse en método `funcion_dispersion()` del fichero `tabla_hash.hpp`.

```
1: procedure PEARSON90(cadena, n, tablaRand)
2:     h ← 0
3:     for i ∈ {1 . . . n} do
4:         h ← tablaRand[h ⊕ cadena[i]]
5:     return h
```

La tabla hash se implementó de forma abierta, lo que supuso que el manejo de colisiones se gestionara mediante encadenamiento. Esto consistía en que si a dos identificadores de productos se le asignaba la misma posición, en dicha posición se creaba una lista enlazada para almacenar ambos productos. Esto aportaba como ventaja el hecho de que se puedan seguir añadiendo productos a la tabla cuando esta se llena, al ser siempre posible agregar nuevos productos a las listas de colisiones.

Para evitar que las colisiones sean frecuentes, las tablas hash se suelen crear con un tamaño 10 % superior al número de elementos que se estima que se va a almacenar. Puesto que la función de dispersión elegida requiere que el tamaño sea un número primo cercano a una potencia de dos, se optó por usar un tamaño de 16381 ($2^{14} - 3$).

4. Representación del problema

Si representamos la tabla de relaciones T donde se almacenan los productos de *Amazon* como un grafo de modo que T corresponda con la matriz de adyacencia del grafo. Los nodos del grafo representan los productos totales a dividir en grupos. Dada esta representación existirá una arista entre dos nodo i y j si $T[i, j] = 1$ o $T[j, i] = 1$. Dada esta manera nueva forma de ver el problema, pues este puede verse como un problema de grafos.

Puesto que hay que dividir los productos en conjuntos disjuntos, se plantea la posibilidad de realizar $r - 1$ cortes en el grafo para dividirlo en r regiones que estén conectadas entre sí

con el menor número de aristas posibles. Es de importancia comentar que un corte en un grafo corresponde a una partición de los vértices en dos subconjuntos disjuntos, lo que encaja perfectamente con el problema que se plantea resolver.

En la Figura 1, se muestra un grafo en el que se han realizado dos cortes, dividiéndolo en 3 regiones. La línea roja representa un corte que cruza tres aristas, mientras que la verde es un corte que cruza sólo dos.

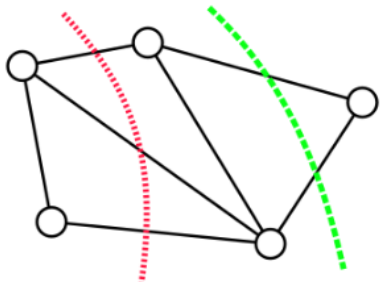

Figura 1: Ejemplo de grafo con dos cortes
`https://en.wikipedia.org/wiki/Minimum_cut`

Este problema puede verse como una generalización del problema de mínimo corte (*min-cut* en inglés). Dicha generalización en función de r, consiste en encontrar $r-1$ cortes que separen el grafo en r conjuntos con la mínima interacción posible entre ellos (*r-min-cut*).

5. Implementación del algoritmo

Siguiendo las indicaciones del enunciado, se optó por realizar una implementación generalizada del algoritmo de Karger[1]. Originalmente, el algoritmo de Karger realiza contracciones seleccionando aristas de manera aleatoria hasta que solamente quedan 2 nodos en el grafo. Esto permite hallar el mínimo corte global del grafo. No obstante, para el problema a resolver, hay que formar un total de r conjuntos. Es por esto que el bucle original del algoritmo de Karger se modificó cambiando el número de iteraciones de $n-2$ a $n-r$, con $r < n$. A continuación, puede verse el pseudocódigo de dicho algoritmo.

```
1:  function KARGER(G = (V, E), r, k)
2:      n ← |V|
3:      C ← |E|
4:      for i ∈ {1 ... k} do
5:          G' ← G
6:          while |V(G')| > r do
7:              e ← Elección uniforme aleatoria de E(G')
8:              G' ← CONTRAER(G', e)          ▷ Contraer la arista e en el grafo G'
9:          if V(G') < C then
10:             C ← V(G')
11:     return C
```

Para implementar el algoritmo de Karger se decidió utilizar C++. En el fichero `grafo.hpp` se

[1] `https://www.cc.gatech.edu/~vigoda/6550/Notes/Lec1.pdf`

encuentra la implementación de la clase grafo y en el fichero `min_cut.cpp`, la del algoritmo de Karger (función `Karger()`).

Para poder realizar las contracciones de la aristas, se cuenta con una lista de las aristas existentes en el grafo. De forma uniforme se selecciona una de estas aristas mediante un número aleatorio i de modo que $i \in \{0, \ldots, numAristas - 1\}$. Una vez se selecciona una arista (x, y), se contrae siempre hacia el nodo x. Puesto que se cuenta con la matriz de adyacencia del grafo, esta no se representó como una matriz de booleanos si no de enteros. Dado que se trataba de una matriz simétrica donde `true` indica que dos nodos están conectados y `false` lo contrario, se cambio la representación por números enteros. Ahora los valores `true` correspondían con 1 y los `false` con 0. Esto permitió que las contracciones consistieran en sumar a la fila x de la matriz la fila y. Con esto se consigue tener la información del número de conexiones del nuevo nodo resultante de la contracción en la fila x.

Una vez hecha la suma, solo había que eliminar los posibles bucles sobre el nuevo nodo, lo que se traducía a hacer $T[x, x] = 0$. Adicionalmente, había que eliminar la conexión entre el nuevo nodo y el nodo eliminado, lo que suponía que $T[x, y] = 0$. Finalmente, solo había que actualizar la información del resto de nodos pues ahora toda conexión de un nodo k con el nodo y debía ser con el nuevo nodo x. Esta actualización de información se podía hacer sumando al elemento $T[k, x]$ el elemento $T[k, y]$ y $T[k, y] = 0$ siempre que $k \neq x \wedge k \neq y$. Con esto, se tenía una matriz de adyacencia con donde las filas y columnas representan el grado de cada vértice. Por último solo hacía falta eliminar la arista seleccionada de la lista de aristas y actualizar el resto de aristas que tenían al nodo x en uno de sus extremos.

6. Análisis del algoritmo

En este apartado se presenta un análisis a priori mostrando real interés en los costes temporales. Si estudiamos el pseudocódigo del Apartado 5, ver que depende de una generación de un número aleatorio. Esto convierte al algoritmo de Karger en un algoritmo probabilista. Esto implica que no tiene por que retornar la solución óptima del problema sino que lo hace con una determinada probabilidad.

En lo que respecta al análisis del tiempo, por lo general si se usa una lista o matriz de adyacencia. El coste de realizar la contracción de dos vértices suele ser del orden de $O(n^2)$. Teniendo en cuenta que esto se repite un total de $n - r$ veces el coste seria de $n^2(n - r) = n^3 - n^2 r$. Puesto que $r < n$ el coste final del algoritmo seria de $O(kn^3)$ donde k es el número de veces que se repite el algoritmo. La razón de que el algoritmo se repita un número k de veces es reducir el error de encontrar el *r-min-cut* óptimo. En el algoritmo original de Karger dicha probabilidad de éxito es la que se ve a continuación:

$$p_n \geq \prod_{i=0}^{n-3} \left(1 - \frac{2}{n-i}\right)$$

Esta expresión extraída de Wikipedia[2] viene dada por la probabilidad de que la arista seleccionada sea la correcta $p_n \geq (1 - \frac{2}{n})p_{n-1}$. La razón de esta cota inferior para la probabilidad es que si se calcula en función del número de nodos el número de casos favorables es 2 y el total en n. El termino P_{n-1} viene por que para que una arista seleccionada sea correcta también tiene que haberlo sido la anteriormente elegida. Por esta razón la probabilidad de éxito de nuestra

[2] `https://en.wikipedia.org/wiki/Karger%27s_algorithm#Success_probability_of_the_contraction_algorithm`

generalización en función de r, quedaría para una contracción de la siguiente manera según el paper original del algoritmo de *Karger* [3].

$$\left[1 - \left(1 - \frac{r-1}{n}\right)\left(1 - \frac{r-1}{n-1}\right)\right] m$$

Por esta razón se puede expresar la probabilidad de que el algoritmo genere el *r-min-cut* óptimo mediante un productorio en función del número total de contracciones realizadas.

$$\prod_{u=r-1}^{n} \left(1 - \frac{r-1}{u}\right)\left(1 - \frac{r-1}{u-1}\right)$$

Aplicando las correspondientes simplificaciones, se sabe que la serie converge a la siguiente expresión la cual representa la probabilidad p_{nr} de que el algoritmo genere la solución óptima.

$$r\binom{n}{r-1}^{-1}\binom{n-1}{r-1}^{-1}$$

Con esta probabilidad podemos establecer que su error $\varepsilon_{nr} = (1 - p_{nr})$. Con esto podemos determinar que la probabilidad de que el algoritmo retorne una corte que no sea el óptimo será de ε_{nr}^{k}. De modo que cuantas mas veces se repita el algoritmo, mas pequeño será su error. Por esta razón se plantea repetir un número de veces $k = \binom{n}{2} \ln n$ de igual forma que en algoritmo original para que la probabilidad fuese mayor a $\frac{1}{n}$, donde n es el número de vértices del grafo.

7. Pruebas Realizadas

Tomando como banco de pruebas el conjunto de entradas creado a partir de las entregas de datos de todos los grupos de practicas. Se pudo hacer un exhaustivo análisis de costes temporales del algoritmo implementado en el Apartado 5. Para dicho análisis se variaron los parámetros r y n con valores de $r \in \{2, 5, 10, 20\}$ y de $n \in \{50, 100, 250, 500, 1000, 1500\}$. Además, se ha probado a ejecutar el algoritmo sobre grafos de diferente densidad, refiriéndonos con ello al cociente del número de aristas del grafo y número de aristas que tendría el grafo completo con ese número de vértices.

El algoritmo se ha probado sin repeticiones para aumentar la probabilidad de acierto ($k = 1$), ya que los tiempos de ejecución simplemente se multiplicarían por el factor k.

Los resultados se muestran en la Figura 2.

En las gráficas se puede apreciar cómo el tiempo de ejecución aumenta conforme aumenta el número de vértices. La densidad del grafo también influye, los tiempos de ejecución aproximadamente se doblan al pasar de grafos con densidad 0,25 a grafos completos.

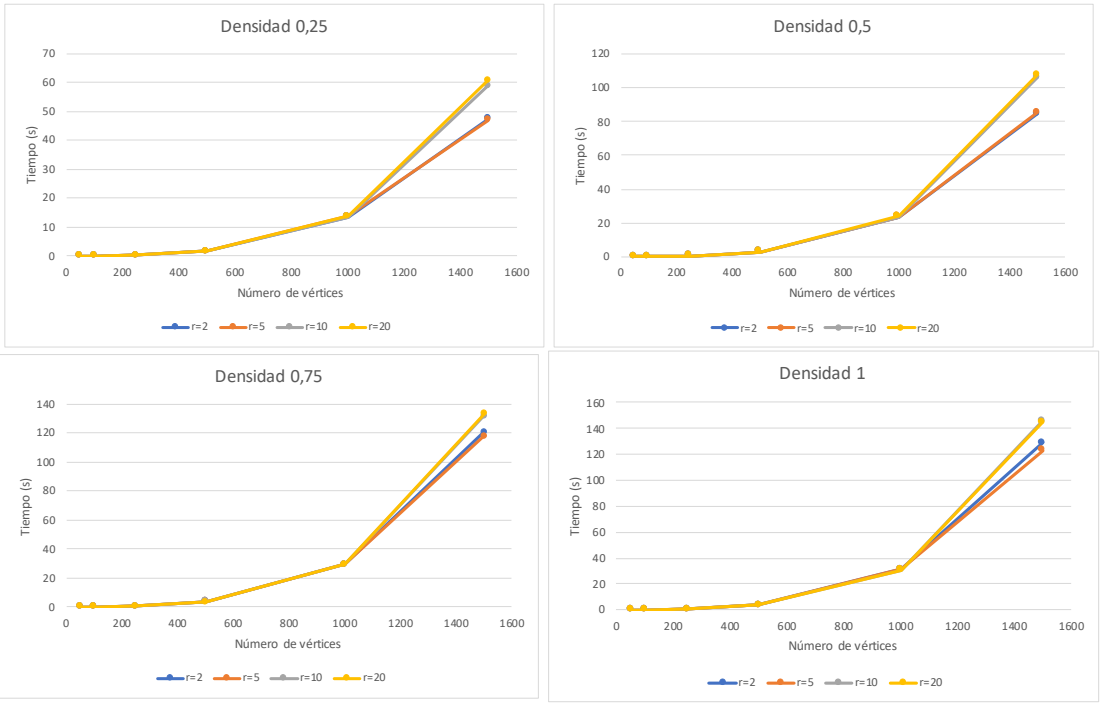

Figura 2: Gráficas con los costes temporales del algoritmo de Karger implementado

8. Algoritmo de Karger Stein

De manera opcional, se implementó una generalización del algoritmo de Karger-Stein[3], que es una mejora del algoritmo original de Karger. Este algoritmo también resuelve *min-cut*, por lo que se volvió a realizar una generalización para que lograr resolver el problema de *r-min-cut*. A continuación puede verse el pseudocódigo del algoritmo desarrollado, cuya implementación se encuentra en la función `KargerStein()` del fichero `min_cut.cpp`:

1: **function** KargerStein($G = (V, E)$, r)
2: **if** $|V| < 6$ **then**
3: **return** Karger(G)
4: **else if** $|V| \neq r$ **then**
5: $t \leftarrow \left\lceil 1 + \frac{|V|}{\sqrt{2}} \right\rceil$
6: $G_1 \leftarrow$ contraer(G, t)
7: $G_2 \leftarrow$ contraer(G, t)
8: **return** mín $\left\{ \begin{matrix} \text{KargerStein}(G_1, \text{máx}\{r, t\}) \\ \text{KargerStein}(G_2, \text{máx}\{r, t\}) \end{matrix} \right\}$
9: **else**
10: **return** Cut(G)

En la Figura 3 se muestran los tiempos medidos utilizando las mismas pruebas que se han utilizado para el algoritmo de Karger, aunque no se ha llegado a ejecutar las de 1500 vértices.

[3] `https://en.wikipedia.org/wiki/Karger%27s_algorithm#KargerStein_algorithm`

El tiempo de cálculo del algoritmo de Karger-Stein es superior al de una sola ejecución del algoritmo de Karger, pero las soluciones que devuelve están más cerca de la óptima.

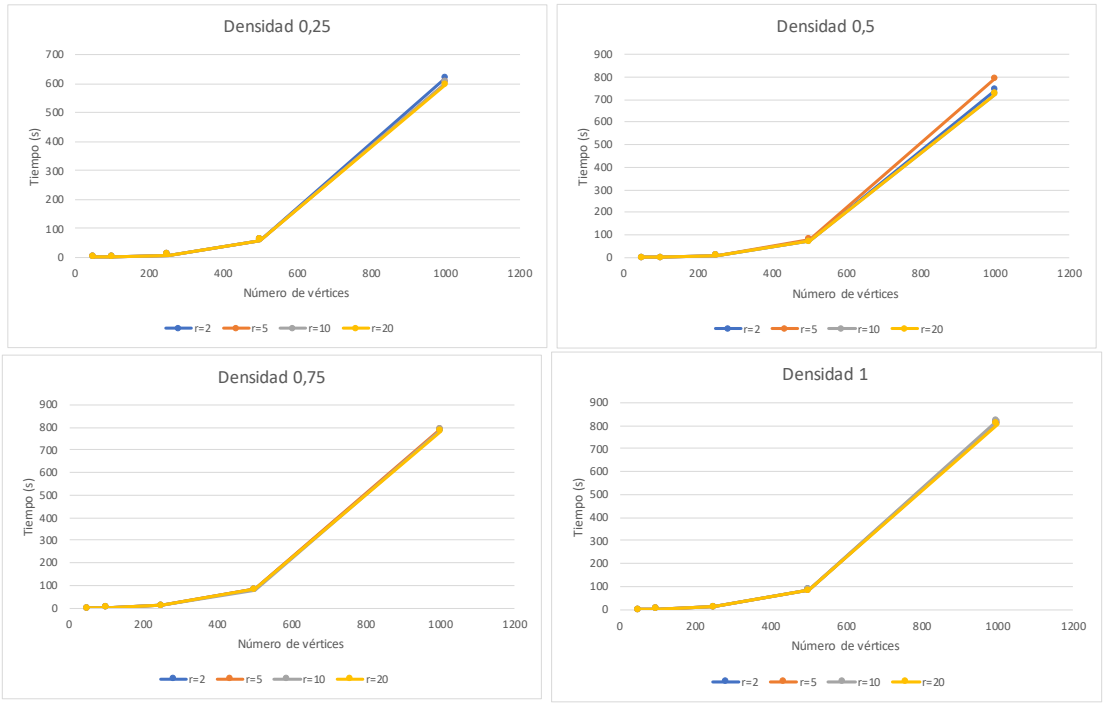

Figura 3: Gráficas con los costes temporales del algoritmo de Karger-Stein implementado

9. Selección de aristas con pesos

Continuado con el penúltimo apartado de los propuestos en el enunciado, se realizó una implementación del algoritmo presentado en el Apartado 5 modificando la selección aleatoria original de las aristas.

Tal y como se exigía en el enunciado ahora la probabilidad de selección de las aristas ya no debía ser igual para todas. Puesto que desde un principio la implementación realizada del grafo se hizo con una matriz de adyacencia de enteros T, se tenía guardado en esta estructura el número de conexiones entre dos nodos.

Dada la lista de aristas sobre las que se hace la selección $\{a_0, a_1, \ldots, a_{m-1}\}$, si denominamos como peso de una arista (x, y) al número de conexiones entre los nodos x e y, podemos crear una lista $\{w_0, w_1, \ldots, w_{m-1}\}$ donde w_i es el peso de la arista a_i. Para poder hacer una selección de la aristas de modo que las que mayor peso tuviesen tuvieran mayor probabilidad de ser elegidas. Los pesos w_i se asignaban de manera que $w_0 = T[x_0, y_0]$ y $w_i = T[x_i, y_i] + w_{i-1}$. Con esto se generaba un número aleatorio $a \in [0, M]$ donde M es el número total de aristas en el grafo en el momento de la selección.

Una vez obtenido dicho número aleatorio se recorría la lista de pesos hasta encontrar uno que cumpliera la condición $w_i \leq a$ y se seleccionaba la arista i para hacer la compresión de forma análoga al presentado en el Apartado 5. La implementación del nuevo método de selección de

aristas puede verse en el método `get_prob_arista()` del fichero `grafo.hpp`.

10. Análisis de selección de aristas con pesos

Finalmente, y de manera opcional, en este apartado se expone un breve análisis, sobre las prestaciones que puede ofrecer el algoritmo descrito en el apartado anterior. Puesto que ahora no todas las aristas tienen la misma probabilidad de ser elegidas, es más probable seleccionar dos productos x e y que se hayan comprado juntos en mayor número de ocasiones. Esto permite realizar agrupaciones de los productos que estén más relacionados entre sí. De este modo se consigue aumentar la probabilidad de que el algoritmo de Karger original genere un grafo con r conjuntos disjuntos de manera óptima. En lo que respecta al coste de ejecución de esta última versión, debería ser. Esto se debe al sobre coste que implica el cálculo de las probabilidades de selección de cada arista, ya que esta operación requiere un recorrido de todas las aristas con coste lineal $O(m)$ donde m es número de aristas. Dicho sobre coste puede verse reflejado en las siguientes gráficas, donde se puede apreciar que el tiempo es muy superior al de la mediciones hechas en el apartado 7.

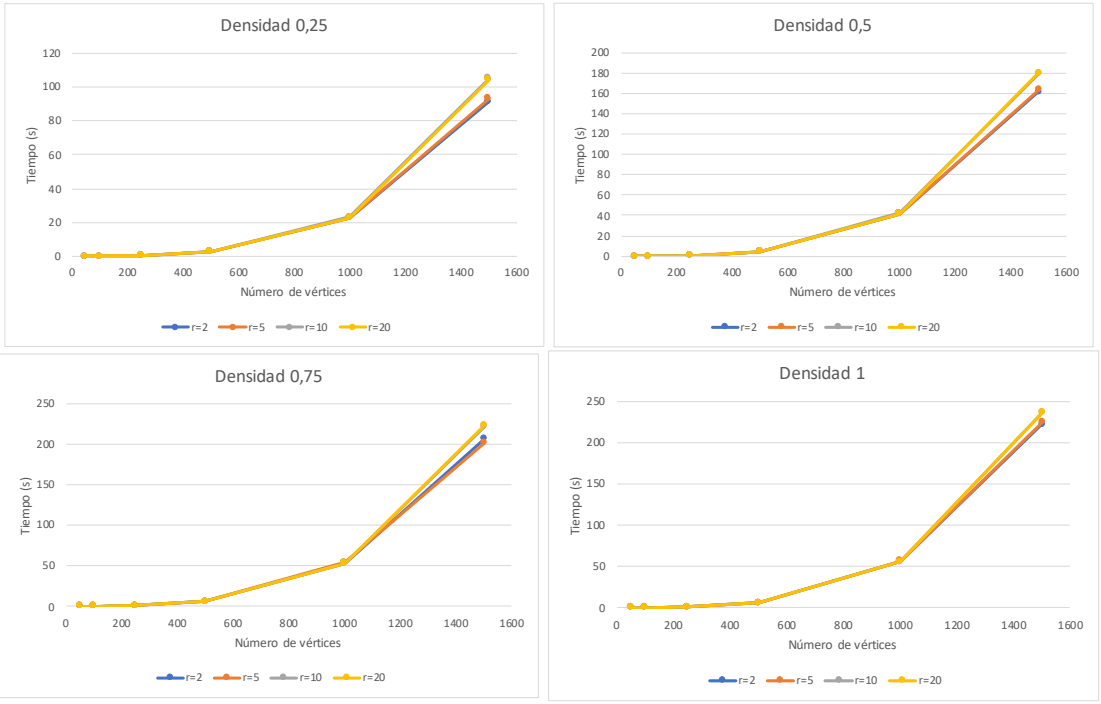

Figura 4: Gráficas con los costes temporales del algoritmo de Karger con pesos implementado

11. Reparto de trabajo

José Daniel Subías Sarrato
Horas invertidas:

- Implementación de la generación de los datos.

- Generación de datos.
- Implementación de los algoritmos.
- Redacción del informe.

Fernando Peña Bes
Horas invertidas:

- Implementación de la generación de los datos.
- Generación de datos.
- Diseño y ejecución de pruebas.
- Redacción del informe.

Referencias

[1] Definición del algoritmo de Karger de Wikipedia. `https://en.wikipedia.org/wiki/Karger%27s_algorithm`.

[2] Análisis de los algoritmos de Karger y Karger-Stein `https://www.cc.gatech.edu/~vigoda/6550/Notes/Lec1.pdf`.

[3] David R. Karger. *Global Min-cuts in $\mathcal{RNC}$, and Other Ramifications os a Simple Min-Cut Algorithm* `http://people.csail.mit.edu/karger/Papers/mincut.ps`