# ALGORITMIA PARA PROBLEMAS DIFÍCILES

# MEMORIA PRÁCTICA 1

*Curso 2019-2020*

Víctor Peñasco Estívalez 741294
Rubén Rodríguez Esteban 737215

# Tabla de contenidos

# Introducción

En este informe se describe el trabajo realizado para la primera práctica de la asignatura de Algoritmia para Problemas Difíciles. En este trabajo se plantea un problema de separación entre dos conjuntos de productos. Se posee la información de si dos conjuntos se han comprado alguna vez juntos o no. El objetivo es encontrar una partición cercana a la óptima, en el caso de que hubiera que dividir la empresa distribuidora en dos, quedando así dos proveedores distintos. Este problema se puede identificar como un problema de mínimo corte, traduciendo los datos de entrada a un grafo, en el que inicialmente cada producto es un vértice y es adyacente con otro si se han comprado alguna vez juntos.

# Diseño e implementación

## Estructuras de datos

La implementación del problema a resolver se ha realizado en lenguaje de programación Java debido a que cuenta con una amplia gama de estructuras de datos ya implementadas en las librerías del lenguaje, las cuales permiten la abstracción a la hora de tener que programar el algoritmo. Además, un lenguaje compilado como Java ofrece más rendimiento frente a otros interpretados como Python, el cual se había tenido en consideración como alternativa.

Como entrada del programa se proporcionan dos ficheros de texto. Uno de ellos es opcional y contiene en cada línea los atributos de un producto. El segundo fichero es obligatorio y contiene una matriz que indica si un producto se ha comprado junto a otro o no alguna vez. Si un producto i nunca se ha comprado junto a otro producto j, el elemento de la matriz [i, j] toma como valor 0. Si se han comprado juntos x veces, el elemento de la matriz [i, j] toma como valor x. Esto supone que el diseño básico del programa ya tiene en cuenta la posibilidad de utilizar **productos con pesos** (matriz de enteros).

Los productos se almacenan en un TAD **Product**, que se compone de un id y una lista de atributos. Si se proporciona el fichero opcional, la lista de atributos se rellena con los atributos correspondientes leídos del fichero, y si no, la lista de atributos del producto queda vacía. El id de los productos se corresponde con el orden de aparición tomando rango de valores desde 0 hasta n-1, donde n es el número de productos.

Antes de comenzar la ejecución del algoritmo, se inicializa un grafo que representa los productos y las compras simultáneas de pares de productos. Cada uno de los vértices del grafo es un tipo de dato **Node**, que en primera instancia representa un único producto, y cada una de las aristas una compra simultánea de un par de productos.

En la inicialización de este grafo, cada producto (instancia de **Product**), se almacena en una tabla hash que utiliza como clave el id de producto y como valor el objeto Product. Esta tabla hash es un atributo del tipo de dato **Node**.

Posteriormente se lee de fichero la matriz de adyacencia. Si un producto i se ha comprado junto a otro producto j, entonces se crea una instancia de dato **Edge** que contiene el nodo

del producto j y el número de veces que se han comprado juntos (degree). En la instancia Node del producto i se guarda una tabla hash que contiene los datos Edge (uno por cada nodo al que son adyacentes), utilizando como clave el id de ese nodo.

El grafo en su conjunto se almacena en una tabla hash, que contiene tipos de dato **Node** representando los vértices, utilizando como clave el id de cada uno de los nodos. La información referente a las artistas está contenida en cada uno de estos tipos de dato tal y como se ha descrito anteriormente.

## Algoritmo de Karger

El algoritmo de Karger[1] es un algoritmo aproximado para resolver el problema de mínimo corte en un grafo. El procedimiento consiste en seleccionar una arista aleatoria del grafo y contraer sus dos vértices en uno solo. Este nuevo vértice resultante preserva las aristas que los dos nodos emparejados tenían antes de la contracción, de manera que el número de aristas resultante sólo se decrementa en una unidad (arista contraída). Este proceso se repite hasta que el número de vértices restantes en el grafo sea 2. Estos dos vértices representan los conjuntos disjuntos (de vértices iniciales) en los que se ha dividido el grafo.

El nodo generado por la contracción de dos tiene como clave (dentro de la tabla hash que representa el grafo) la concatenación de la clave del primer nodo escogido, una barra baja (_) y la clave del segundo nodo, logrando así que la unicidad de los identificadores se preserve en todo momento.

Gracias a la implementación desarrollada, en cada contracción se almacena la información de los dos nodos elegidos, de forma que en cada uno de los dos nodos finales, se encuentra una tabla hash que almacena los productos que pertenecen a ese nodo, o lo que es lo mismo, a ese conjunto.

Karger se trata de un algoritmo aproximado, por lo que no siempre encuentra la solución óptima (hace uso de generadores de números aleatorios para elegir la arista a contraer). Una manera de mejorar las posibilidades de encontrar la solución óptima es mediante la repetición del algoritmo un número preestablecido de veces, y quedarse con los conjuntos cuyo valor de corte sea mínimo de entre todos los obtenidos.

Al utilizar el algoritmo de Karger, la probabilidad de no llegar a la solución óptima repitiendo la ejecución del algoritmo un total de C(n,2)log n veces, donde C(n,2) es el número de subconjuntos de dos elementos que pueden formarse a partir de un conjunto de n elementos, no supera 1/n.

$$\left[1 - \binom{n}{2}^{-1}\right]^{T} \leq \frac{1}{e^{\ln n}} = \frac{1}{n}$$

[1] "Karger's algorithm - Wikipedia." https://en.wikipedia.org/wiki/Karger%27s_algorithm#Contraction_algorithm.

## Algoritmo de Karger-Stein

El algoritmo de Karger-Stein[2] es una mejora del algoritmo de Karger explicado en el apartado anterior. Este algoritmo tiene la mejora de que posibilita hallar con una probabilidad mayor el mínimo corte del grafo. Dicha variante del algoritmo de Karger toma como premisa que existe un 50% de probabilidad de unir dos nodos en el corte mínimo al contraer los n nodos iniciales hasta tener 1 + n / √2 nodos.

En base a lo anteriormente comentado, es suficiente con realizar solamente dos llamadas al algoritmo de Karger original reduciendo el grafo hasta obtener 1 + n / √2 nodos (en lugar de 2) ya que al ser la probabilidad de acierto de un 50%, es altamente probable que una de las dos llamadas obtenga el grafo correcto. Dicho valor de probabilidad se ha obtenido en base a los siguientes cálculos.

Sea n - t = n - ⌈ 1 + n / √2 ⌉ se tiene que:

$$\mathbb{P}[x_0 \cap \ldots \cap x_{n-t}] \geq \frac{t(t-1)}{n(n-1)} = \frac{(\lceil 1+n/\sqrt{2} \rceil)(\lceil 1+n/\sqrt{2} \rceil - 1)}{n(n-1)} \geq \frac{1}{2}$$

Una vez que el número de nodos del grafo es relativamente pequeño (6), se aplica el algoritmo de Karger directamente ya que la probabilidad de que una arista e del conjunto de corte sea seleccionada aumenta conforme se hacen las contracciones del grafo. Este hecho motiva la idea de cambiar a un algoritmo más lento (usar el algoritmo de Karger directamente) después de un cierto número de pasos de contracción.

En definitiva, el algoritmo consiste en realizar llamadas sucesivas al algoritmo de Karger hasta obtener un cierto número de vértices t, quedándose con el mejor grafo obtenido de ambos para posteriormente continuar con el mismo proceso sobre ese grafo hasta que quedan pocos nodos, concretamente 6, momento a partir del cual se aplica el algoritmo de Karger directamente.

Observando el algoritmo, se ha podido determinar que el mínimo corte es encontrado con una probabilidad mayor que 1 / log n. La razón estriba en que la probabilidad de no cometer un error (contraer el mínimo corte) en la primera de las dos llamadas a Karger original es mayor que 1 / 2, multiplicada por la probabilidad de que el algoritmo tenga éxito en la llamada recursiva utilizando este grafo (estos dos eventos son independientes). Por consiguiente, la probabilidad de tener éxito en la llamada recursiva con este grafo es de al menos $\frac{1}{2}P(\frac{n}{\sqrt{2}})$, y por tanto, la probabilidad de no tener éxito es la complementaria, es decir, 1 - $\frac{1}{2}P(\frac{n}{\sqrt{2}})$.

De acuerdo a lo anterior, la probabilidad de no tener éxito en las dos llamadas al algoritmo de karger es de $(1-\frac{1}{2}P(\frac{n}{\sqrt{2}}))^2$.

[2] "Karger's algorithm - Wikipedia." https://en.wikipedia.org/wiki/Karger%27s_algorithm#Karger%E2%80%93Stein_algorithm.

En base a esto, se puede determinar la probabilidad de éxito del algoritmo de Karger-Stein por medio de la siguiente fórmula:

$$P(n) \geq 1 - (1 - \frac{1}{2}P(\frac{n}{\sqrt{(2)}}))^2 = P(\frac{n}{\sqrt{(2)}}) - \frac{1}{4}(P(\frac{n}{\sqrt{(2)}}))^2$$

# Análisis del coste

En esta sección del documento se va a razonar los costes temporales de los algoritmos implementados, que derivan de las diversas decisiones de diseño que se han tomado durante el desarrollo de la práctica.

## Coste temporal del algoritmo de Karger

A continuación se muestra en pseudocódigo el proceso de contracción del grafo:

```
procedure contract(G = (V, E), t):
while |V| > t
    choose e ∈ E uniformly at random
    G ← G/e
return G
```

El coste de seleccionar una arista aleatoria es O( |V| + |E| ), calculado en base a:

1. Convertir las claves de la tabla hash del grafo a array, que tiene un coste O(|V|).
2. Acceder al elemento del array en un índice aleatorio con coste O(1).
3. Crear una lista vacía y añadir un elemento por cada conexión (arista) existente con otros nodos, que tiene como coste O(|E|). De esta manera también se tiene en cuenta las probabilidades directamente proporcionales a los pesos de las aristas, en el caso de que se proporcione como entrada una matriz de enteros.
4. Acceder al elemento de la lista en un índice aleatorio con coste O(1).

El coste de actualizar el grafo una vez seleccionada una arista aleatoria es O(|V|), y se ha calculado en base a las siguientes operaciones:

1. Realizar unión de productos, cuyo coste es O(|V|).
2. Suma de los vértices adyacentes a cada nodo, proceso que tiene un coste O(|V|).
3. Añadir arista desde el resto de nodos con el nuevo vértice si conectaban con alguno de los dos anteriores. Dicha operación tiene coste O(|V|).

Estas dos acciones se realizan O(|V|) veces, de forma que el algoritmo de Karger implementado tiene un coste total de O( |V| · ( |V| + |E| )).

Ejecutar el algoritmo de Karger k veces para aumentar la probabilidad de encontrar la solución óptima requiere calcular el valor del corte cada vez. Calcular el valor del corte tiene coste $O(|V|^2)$. Por tanto, repetir k veces la ejecución tiene coste O( k · |V| · ( |V| + |E| )).

En base a la probabilidad de encontrar la solución óptima mencionada anteriormente, el número de repeticiones necesarias para que el error sea como mucho 1 / n es $O(n^2 \log n)$, quedando así un coste total del algoritmo de aproximadamente $O(n^4 \log n)$.

## Coste temporal del algoritmo de Karger-Stein

Dado que el algoritmo de Karger-Stein ha sido implementado de forma recursiva, se va a proceder a calcular el coste del algoritmo en base a la opción del caso peor. A continuación se va a explicar detenidamente cómo se ha calculado:

```
procedure fastmincut(G = (V, E)):
if |V| ≤ 6:
    return mincut(V)
else:
    t ← ⌈1 + |V|/√2⌉
    G1 ← contract(G, t)
    G2 ← contract(G, t)
    return min {fastmincut(G1), fastmincut(G2)}
```

Una llamada recursiva a Karger-Stein tiene el siguiente coste (se asume que no se trata del caso trivial).

1. Calcular el valor del t, que tiene coste O(1).
2. Clonar (deep copy) el grafo para aplicar el sobre él algoritmo de Karger y evitar que se modifique el original $O(|V|^2)$.
3. Ejecutar una primera vez Karger (hasta t vértices), que tiene coste O(( |V| - t ) · ( |V| + |E| )), donde t toma como valor 1 + |V| / √2.
4. Ejecutar una segunda vez Karger (hasta t vértices), que tiene coste O(( |V| - t ) · ( |V| + |E| )), donde t toma como valor 1 + |V| / √2.
5. Obtener valor de corte para el primer grafo, que tiene coste $O(|V|^2)$.
6. Obtener valor de corte para el segundo grafo, que tiene coste $O(|V|^2)$.
7. Llamada recursiva a Karger-Stein con mejor grafo (mejor valor de corte).

Se realizan O(log n) llamadas recursivas, cada una de ellas con coste O((|V|) · ( |V| + |E| )). En definitiva, el algoritmo de Karger-Stein tiene coste O( |V| · ( |V| + |E| ) · log |V| ).

Repetir la ejecución del algoritmo k veces (con el objetivo de aumentar la probabilidad de encontrar la solución óptima) tiene como coste O( k · |V| · ( |V| + |E| ) · log |V| ).

Se puede observar que el coste de Karger-Stein es mayor que el de Karger. Como contrapartida, las probabilidades de encontrar la solución óptima en Karger-Stein son mayores, por lo que con menos repeticiones se obtendrían resultados similares. Este número de repeticiones para Karger-Stein serían $\log^2 n$.

En definitiva, el coste total para encontrar la solución óptima con probabilidad alta utilizando el algoritmo de Karger-Stein es aproximadamente $O(n^2 \log^3 n)$, siendo órdenes de magnitud menor que el de Karger original.

**Nota:** Para el cálculo del coste se ha omitido el tiempo necesario para ejecutar el caso trivial (cuando el grafo tiene menos de 6 vértices).

# Pruebas

Para comprobar la correcta implementación de los algoritmos, se ha creado un script denominado *ejecutar1.sh* que automatiza la ejecución de los casos de prueba, utilizando tanto Karger como Karger-Stein y usando distintos métodos de generación de números pseudoaleatorios. La verificación de los resultados obtenidos se ha hecho por medio de pequeños casos de prueba para los que se conoce el mínimo corte (ficheros test1, test2 y test3). El fichero test3 también comprueba el funcionamiento del programa con matrices de enteros (productos con pesos).

Para medir el tiempo de ejecución de cada caso de prueba, se ha utilizado el comando *time*. Hay que tener en cuenta que generar salida de texto en cada iteración, hace que los casos de prueba con mayor número de repeticiones aumenten ligeramente su tiempo de ejecución.

Cada una de las pruebas realizadas utiliza un algoritmo distinto con diferente generador de números aleatorios. El número de repeticiones se ha estimado en base al algoritmo usado, en el caso de Karger-Stein multiplicandolo por una constante dado que el número de repeticiones era muy bajo y no supone impacto alguno en el coste.

En la siguiente tabla comparativa se muestran los resultados de la ejecución de los casos de prueba.

| Caso de prueba | Fichero utilizado | Algoritmo | Generador pseudoaleatorio | Repeticiones | Tiempo de ejecución |
|---|---|---|---|---|---|
| 1 | testSet_4 | Karger | rg1 | 6184 | 9.481s |
| 2 | testSet_4 | Karger-Stein | rg1 | 64 | 0.797s |
| 3 | testSet_4 | Karger | rg2 | 6184 | 10.440s |
| 4 | testSet_4 | Karger-Stein | rg2 | 64 | 1.038s |
| 5 | testSet_4 | Karger | rg3 | 6184 | 8.911s |
| 6 | testSet_4 | Karger-Stein | rg3 | 64 | 0.771s |
| 7 | test1 | Karger | rg1 | 26 | 0.117s |
| 8 | test1 | Karger-Stein | rg1 | 8 | 0.111s |
| 9 | test1 | Karger | rg2 | 26 | 0.186s |
| 10 | test1 | Karger-Stein | rg2 | 8 | 0.175s |
| 11 | test1 | Karger | rg3 | 26 | 0.116s |
| 12 | test1 | Karger-Stein | rg3 | 8 | 0.114s |
| 13 | test2 | Karger | rg1 | 26 | 0.114s |
| 14 | test2 | Karger-Stein | rg1 | 8 | 0.116s |
| 15 | test2 | Karger | rg2 | 26 | 0.191s |
| 16 | test2 | Karger-Stein | rg2 | 8 | 0.175s |
| 17 | test2 | Karger | rg3 | 26 | 0.116s |
| 18 | test2 | Karger-Stein | rg3 | 8 | 0.113s |
| 19 | test3 | Karger | rg1 | 26 | 0.117s |
| 20 | test3 | Karger-Stein | rg1 | 8 | 0.102s |
| 21 | test3 | Karger | rg2 | 26 | 0.191s |
| 22 | test3 | Karger-Stein | rg2 | 8 | 0.203s |
| 23 | test3 | Karger | rg3 | 26 | 0.118s |
| 24 | test3 | Karger-Stein | rg3 | 8 | 0.108s |

## Organización del trabajo

| **Tarea realizada** | | **Horas dedicadas (por integrante de equipo)** | |
|---|---|---|---|
| | | Víctor Peñasco | Rubén Rodríguez |
| Comprensión del problema | | 1 | 1 |
| Diseño e implementación | Estructuras de datos | 3 | 3 |
| | Karger | 2 | 2 |
| | Karger-Stein | 2.5 | 2.5 |
| Pruebas | | 4 | 4 |
| Memoria | | 4.5 | 4.5 |
| **Total** | | **17** | **17** |

## Referencias


“Algorithms for the mincut problem - Hongwei Jin”
http://courses.csail.mit.edu/18.337/2016/final_projects/matthew_lindsay/MinCutAlgo-master/presentation&report/Presentation.pdf

"Karger's algorithm - Wikipedia." https://en.wikipedia.org/wiki/Karger%27s_algorithm