|  |
| --- |
|  |
| **Gestión de Actores y Películas Grafos de conexión II** |
| Estructuras de Datos y Algoritmos |
|  |
|  |
|  |

**Fecha:** 23-XII-2015

**Participantes**

XABIER CASADO

SERGIO ERLANTZ TOBAL

PEIO VALLE

Índice de contenido

[Introducción 3](#_Toc438728284)

[Descripción de las estructuras de datos principales 5](#_Toc438728285)

[Diseño de las clases 6](#_Toc438728286)

[Clase GraphHash 7](#_Toc438728287)

[Clase Pareja 7](#_Toc438728288)

[Clases de la Fase de proyecto 1 7](#_Toc438728289)

[Clases Actor y Película 7](#_Toc438728290)

[Clases ListaActores y ListaPeliculas 8](#_Toc438728291)

[Clases CatalogoActores y CatalogoPeliculas (MAEs) 8](#_Toc438728292)

[Clase LectorFichero 8](#_Toc438728293)

[Diseño e implementación de los métodos principales 9](#_Toc438728294)

[Método gradoRelaciones 9](#_Toc438728295)

[Método calcularGrado 9](#_Toc438728296)

[Método losDeMasCentralidad 10](#_Toc438728297)

[Métodos de la Fase de proyecto 3 10](#_Toc438728298)

[Método crearGrafo 10](#_Toc438728299)

[Método estanConectados y devolverCaminoConectado 10](#_Toc438728300)

[Casos de prueba 11](#_Toc438728301)

[public boolean estanConectados(String a1, String a2) public ArrayList<String> devolverCaminoConectados(String a1, String a2) 11](#_Toc438728302)

[Código 12](#_Toc438728303)

[GraphHash 12](#_Toc438728304)

[Coste de los métodos principales 16](#_Toc438728305)

[Conclusiones 17](#_Toc438728306)

# Introducción

La práctica final de la asignatura **Estructuras de Datos y Algoritmos**, está enfocada en la ampliación de funcionalidad del **grafo de conexión** y por tanto sustentada por las prácticas primera y tercera.

La primera práctica de la asignatura consistía en la creación, en base a un fichero ordenado obtenido de la base de datos de **IMDb**, de un catálogo de **actores** (~ 1283000 elementos), y otro de **películas** en las que participan, sabiendo que un actor puede participar en múltiples películas y, a su vez, que una película está compuesta por múltiples actores, conformando una relación de cardinalidad M:N, tal y como expresa la siguiente figura:

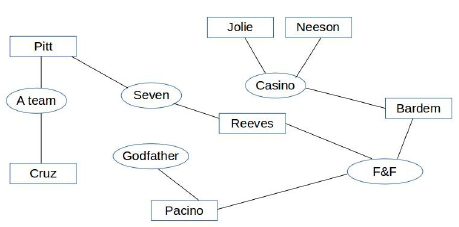


De esta forma se obtuvo un programa capaz de implementar las operaciones básicas de una base de datos, denominadas operaciones CRUD (Create, Read, Update, Delete).

A su vez, la tercera práctica buscaba obtener en base al primer proyecto de la asignatura, un sistema indicador de las **relaciones**, en caso de haberlas, **que unen un** **par determinado de parámetros** (actores y/o películas).

Para poder resolver este problema, previamente era imprescindible la creación de un **grafo**, implementado en base a una *tabla hash*, con una lista de elementos o **nodos** y sus elementos relacionados o **arcos**. El grafo de conexión se realizó a partir de los datos de la lista de actores obtenida en la primera fase de la práctica.

En este punto pasamos de una estructura formada por dos listas, con elementos diferenciables en dos tipos, a una única estructura de datos, en la que todos los elementos son tomados como iguales y que se puede representar a modo de grado, como muestra la siguiente figura:



En este punto entramos en la parte correspondiente a la práctica final, para la cual es necesario conocer antes los siguientes conceptos propios de los grafos:

* *Distancia,* que en un grafo hace referencia al número de nodos mínimo que debe recorrerse para unir dos nodos y por tanto, es la longitud del camino más corto entre éstos.
* *Centralidad*, que se refiere a una medida posible de un nodo, la cual determina su importancia relativa dentro de éste. Para la presente práctica, este valor hace referencia al número de veces que aparece un nodo en el conjunto de todas distancias del grafo. Esta medida se puede calcular de acuerdo al siguiente algoritmo:

Para obtener la centralidad de un vértice *v* en el grafo *G := (V, A)* con *V* número de vértices:

* Para cada par de vertices *(s,t)* obtenemos el camino más corto entre éstos (su distancia).
* Para cada par de vertices *(s,t)* obtenemos la proporción de la distancia que pasa por el vértice *v* en cuestión.
* Calculamos el sumatorio de esta fracción para todos los pares de vértice *(s,t)* del grafo.

Bajo este escenario se desarrolla la cuarta práctica, pretendiendo por tanto obtener la distancia media entre nodos de nuestro grafo, por una parte, y aquellos nodos de mayor centralidad en éste, como segundo punto.

# Descripción de las estructuras de datos principales

Las estructuras de datos principales serán las mismas que ya se utilizaron en la anterior práctica.

La estructura de datos elegida para implementar los grafos ha sido la tabla hash *HashMap,* en la cual vamos almacenando como clave indistintamente bien la tupla nombre completo del actor/actriz como *key* y un *ArrayList* con su lista de películas como *value*, o bien el título de la película como *clave* y la lista de actrices/actores que conforman el reparto de dicha película como *valor*.

El hecho de guardar indiscriminadamente películas y actores nos sirve para implementar eficientemente la función *estanConectados*, y que nos indique las relaciones existentes entre el par de parámetros que recibe.

Por otro lado, para la implementación de colas hemos optado por utilizar la clase de Java *ArrayDeque*, que utilizaremos como variable auxiliar en el método que busca relación entre un par de elementos dado, para mantener los elementos que nos quedaban por examinar.

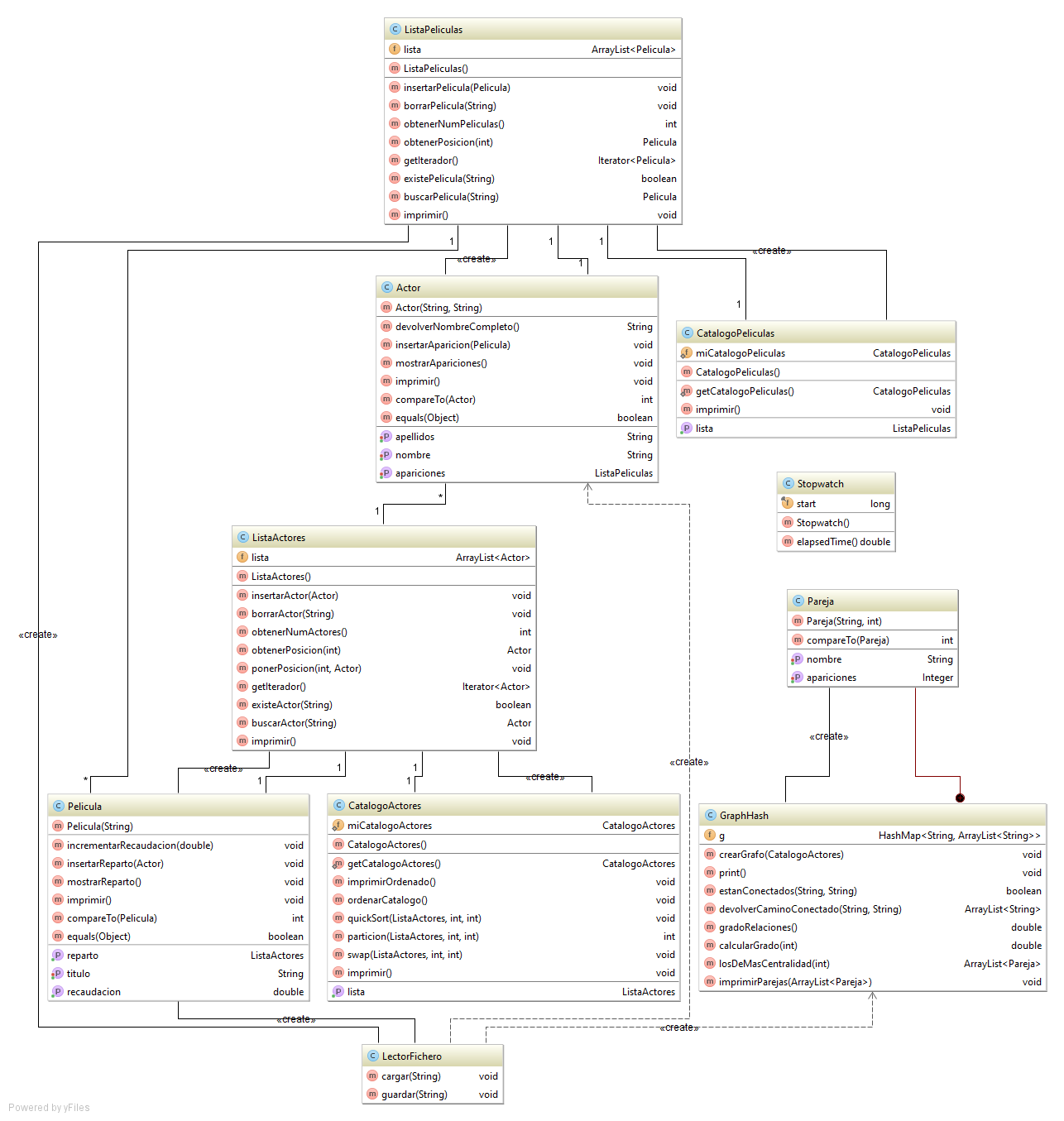
Lo hemos realizado así ya que en Java las colas no se pueden implementar directamente usando *Queue*, por ser ésta una *interfaz*, considerando por tanto que lo mejor era una *Deque*, que nos permite extraer elementos tanto por delante como por detrás, implementando de forma conjunta los algoritmos *FIFO* y *LIFO*.

Como se indica en la documentación de esta clase, es probablemente más rápida que una *lista enlazada* cuando se usa como cola, y más rápida que *Stack* cuando se usa como pila.

El resto de estructuras utilizadas son las ya vistas en el primer proyecto:

* *Object*, que es la clase genérica en base a la cual definimos nuestras clases **Película** y **Actor**.
* *ArrayList* para guardar las listas de actores y películas a tratar.
* *HashMap* en la carga del fichero de datos como lista auxiliar, porque el tiempo en búsqueda es mucho menor ya que buscamos por claves con un coste logarítmico y mediante las propias claves únicas evitamos duplicidad.

# Diseño de las clases



## Clase GraphHash

Es la parte fundamental de esta fase del proyecto. En esta clase ya hay implementados tres métodos de la anterior práctica, los cuales son:

* *crearGrafo*, que crea el grafo *g* (atributo de la clase) mediante un *HashMap*, en el que cada actor esté relacionado con la lista de sus películas y las películas estén relacionadas con la lista de los actores que participan en ella.
* *estanConectados,* que buscará en el grafo que hemos creado si hay camino o no entre los 2 parámetros que le pasamos como entrada.
* *devolverCaminoConectado,* este último método opcional busca la relación entre 2 parámetros dados y devuelve el camino que ha recorrido; por lo tanto, nos especifica cómo están conectados esos parámetros.

Por ejemplo, si los parámetros fueran dos actores, “*Neeson”* y “*Pitt”,* el resultado se mostraría así: *“<Neeson>, <Casino>, <Bardem>, <F&F>, <Reeves>, <Seven>, <Pitt>”*

A parte de estas funcionalidades ya implementadas tenemos que añadir los dos nuevos métodos:

* *gradoRelaciones,* cuyo resultado será un valor que dará el número medio correspondiente a la distancia del camino que une a cualquier par de actores. Para ello tendremos en cuenta la hipótesis llamada **seis grados de separación**, que plantea que cualquiera en la Tierra puede estar conectado a cualquier otra persona del planeta a través de una cadena de conocidos que no tiene más de cinco intermediarios (por tanto, seis arcos).
* *losDeMayorCentralidad*, que obtendrá los actores con mayor valor de centralidad del grafo, de mayor a menor número de apariciones, acorde al algoritmo presentado en la introducción. Este método devuelve un *ArrayList<Pareja>*.

## Clase Pareja

Esta clase se encuentra a su vez dentro de la clase *GraphHash*, conformada por los atributos **nombre** y **apariciones**, sirviéndonos por lo tanto para devolver al usuario la tupla de actores con su centralidad en el grafo, de una forma sencilla y fácilmente imprimible.

## Clases de la Fase de proyecto 1

### Clases Actor y Película

Conforman el *esqueleto* de la aplicación, ya que son las dos estructuras principales que obtendremos de la lectura del fichero anteriormente comentado.

Ambas clases implementan la interfaz **Comparable**, para poder comparar actores por apellido (y por nombre en caso del mismo apellido) y películas por título mediante el método a implementar *compareTo*.

Un actor dispondrá de una lista de películas, llamada apariciones. En contraposición, una película contiene una lista de actores, con nombre reparto.

Destacar que la clase **Actor** guarda de forma separada (en diferentes atributos) el nombre y los apellidos de cada elemento, para una posterior ordenación de estos más eficiente así como una impresión por consola de sus nombres completos más entendible (el método *devolverNombreCompleto* se encarga de devolver de forma concatenada el nombre y apellidos, en ese orden, de cada actor).

Por su lado, la clase **Pelicula** contiene el método *incrementarRecaudacion*, que eleva la recaudación dada una variable del tipo double que sumará a la ya guardada.

### Clases ListaActores y ListaPeliculas

En el siguiente nivel tenemos las dos clases *Lista*, que se encargan de almacenar conjuntos de objetos Actor o Pelicula, respectivamente, en una estructura del tipo *ArrayList.*

Estas clases podrán numerar el total de elementos en la lista, así como realizar las operaciones de inserción, borrado y búsqueda de objetos de su correspondiente tipo.

### Clases CatalogoActores y CatalogoPeliculas (MAEs)

Estas dos clases siguen el patrón **Singleton** y son clases con un atributo del tipo **ListaActores**o **ListaPeliculas**.

### Clase LectorFichero

Por último, comentar que esta clase será la que se ocupe de las tareas de carga y guardado del fichero, utilizando las clases anteriormente comentadas para guardar cada uno de los objetos **Actor** y **Pelicula**.

# Diseño e implementación de los métodos principales

En este punto comentaremos los métodos de la clase *GraphHash*, implementandos tanto en esta práctica como en la tercera fase y de los cuales se puede ver el coste en el apartado correspondiente al **código** del proyecto.

## Método gradoRelaciones

**public double gradoRelaciones ()**

inicializamos a 250 el número de pruebas “nPruebas”

incializamos la constante ERROR ABSOLUTO con valor 10-4

creamos grado de la prueba anterior “g0” y gr. de la actual “g1”

inicializamos g1 a 0

hacer

asignamos a g0 el valor de g1

asignamos a g1 el resultado de **calcularGrado**(nPruebas)

doblamos el valor de nPruebas

mientras la resta |g0 – g1| sea mayor que ERROR ABSOLUTO

## Método calcularGrado

**public double calcularGrado (nPruebas)**

obtenemos el CatalogoActores

para cada valor de i desde 0 hasta que sea igual a nPruebas

a1 y a2 son dos actores al azar del CatalogoActores

si hay camino entre ellos

guardamos en un acumulador el tamaño del camino /2

incrementamos en 1 el número de caminos hallados

fin si

fin para cada

devolvemos el acumulador / número de caminos hallados

## Método losDeMasCentralidad

Para que todos los actores salgan al menos una vez tenemos que realizar 42700 pruebas (*ver apartado* ***Conclusiones***).

**public ArrayList<Pareja> losDeMasCentralidad (n)**

inicializamos la constante NÚM DE ITERACIONES a 427000

obtenemos el CatalogoActores

volcamos todos los nombres de los actores en apariciones con su contador de apariciones a 0

para cada valor de i desde 0 hasta que sea igual a N\_ITERACIONES

a1 y a2 son dos actores al azar del CatalogoActores

si hay camino entre ellos

para cada nodo j del camino

si el nodo j-ésimo es un actor

sumar en apariciones 1 al actor j-ésimo

fin si

fin para cada

fin si

fin para cada

volcamos apariciones en un ArrayList de objetos Pareja

ordenamos por número de apariciones

si el ArrayList es mayor que n

imprimimos los n primeros

devolvemos los n primeros

si no

imprimimos todos

devolvemos todos

fin si

## Métodos de la Fase de proyecto 3

### Método crearGrafo

El siguiente método recibe la lista de actores obtenida del fichero inicial (realizado en la práctica 1) y la vuelca en un grafo implementado sobre un *HashMap*, compuesto éste de todos los actores/actrices del fichero original así como todas las películas de dicho archivo que están vinculadas como mínimo a un actor.

**public void crearGrafo (CatalogoActores)**

para cada actor del CatalogoActores recibido

guardamos el actor en la posición i-ésima en el grafo

para cada película del actor i-ésimo

si la película no está en la lista del actor i-ésimo

guardamos la película j-ésima en el grafo

fin si

guardamos el actor en el reparto de la peli j-ésima

guardamos la peli en la lista pelis del actor i-ésimo

fin para cada

fin para cada

### Método estanConectados y devolverCaminoConectado

El primer método devuelve un valor booleano verdadero o falso, de acuerdo a si dos elementos dados están relacionados o no. De forma análoga, el segundo método devuelve además el camino seguido para llegar del primer elemento al segundo. Sus implementaciones por lo tanto siguen el mismo algoritmo.

**public boolean estanConectados (elemento a1, elemento a2)**

si a1 es elemento de a2 o a2 es elemento de a1 entonces

devolver TRUE

si no

mientras no hayamos encontrado recorrido

y queden elementos por examinar

seleccionamos el primer elemento de la cola

si el primer elemento de la cola es a2

devolver TRUE

si no

para cada valor de lista de elementos del actual

si no hemos examinado aún ese elemento lo

añadimos a la lista de elementos por

examinar y lo marcamos como examinado

fin para cada

fin si

fin mientras

fin si

# Casos de prueba

Como casos de pruebas comentaremos de forma conjunta los métodos de la anterior práctica *estanConectados* y *devolverCaminoConectado*, que actúan de forma análoga, como se comenta en el anterior apartado:

## public boolean estanConectados(String a1, String a2) public ArrayList<String> devolverCaminoConectados(String a1, String a2)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **a** | **a1** | **a2** | **resultado** | **Lista resultante** |
| () | x | y | false | () |
| (x, <y, z>) | x  x | y  a | true  false | (x y)  () |
| (  x, <a, b, c>  y, <a, b>  z, <e, f, g>  ) | x  x  x | a  y  z | true  true  false | (x a)  (x a y)  () |

# Código

En este apartado se incluye el código correspondiente a la clase del proyecto que implementa la estructura de grafo, así como el coste de los métodos implementados en ésta.

## GraphHash

**public void** crearGrafo(CatalogoActores miCatalogoActores) {  
 **g** = **new** HashMap<String, ArrayList<String>>();  
 ListaActores actores = miCatalogoActores.getLista();  
 Actor a;  
 String apellidoActor, tituloPelicula;  
 **for** (**int** i = 0; i < actores.obtenerNumActores(); i++) {  
 a = actores.obtenerPosicion(i);  
 nombreActor = a.devolverNombreCompleto();  
 **g**.put(nombreActor, **new** ArrayList<String>());

**for**(**int** j=0; j<a.getApariciones().obtenerNumPeliculas(); j++){  
 tituloPelicula =

a.getApariciones().obtenerPosicion(j).getTitulo();  
 **if** (!**g**.containsKey(tituloPelicula))  
 **g**.put(tituloPelicula, **new** ArrayList<String>());ArrayList<String> apariciones = **g**.get(nombreActor);  
 apariciones.add(tituloPelicula);ArrayList<String> reparto = **g**.get(tituloPelicula);  
 reparto.add(nombreActor);  
 }  
 }  
}

**public boolean** estanConectados(String a1, String a2) {  
 **boolean** hayCamino = **false**;  
 **if**(!***g***.containsKey(a1) || !***g***.containsKey(a2)) **return** hayCamino;  
 **else** {  
 String actual;  
 Queue<String> porExaminar = **new** ArrayDeque<String>();  
 porExaminar.add(a1);  
 HashSet<String> examinados = **new** HashSet<String>();  
 examinados.add(a1);  
  **while**(!hayCamino && !porExaminar.isEmpty()) {  
 actual = porExaminar.poll();  
 **if**(actual.equals(a2)) hayCamino = **true**;  
 **else** {  
 **for** (String x : ***g***.get(actual)) {  
 **if**(!examinados.contains(x)){

porExaminar.add(x);

examinados.add(x);  
 }  
 }  
 }  
 }  
 **return** hayCamino;  
}

**public** ArrayList<String> devolverCaminoConectado(String a1,String a2){  
 **boolean** hayCamino = **false**;  
 ArrayList<String> dev;  
  
 **if**(!***g***.containsKey(a1) || !***g***.containsKey(a2))

**return new** ArrayList<String>();  
 **else** {  
 String actual;  
 Queue<String> porExaminar = **new** ArrayDeque<String>();  
 HashMap<String,String> examinados = **new** HashMap<>();  
  
 porExaminar.add(a1);  
 examinados.put(a1,**null**);  
  
 **while**(!hayCamino && !porExaminar.isEmpty()) {  
 actual = porExaminar.poll();  
 **if**(actual.equals(a2)) hayCamino = **true**;  
 **else** {  
 **for** (String x : ***g***.get(actual)) {  
 **if**(!examinados.containsKey(x)) {  
 porExaminar.add(x);  
 examinados.put(x,actual);  
 }  
 }  
 }  
 }  
 dev = **new** ArrayList<String>();  
 String actu2 = a2;*//variable para no cargarse la de entrada*  **while** (actu2 != **null**){  
 dev.add(actu2);  
 actu2 = examinados.get(actu2);  
 }  
 }  
 Collections.*reverse*(dev);

**return** dev;  
}

**public void** print() {  
 **int** i = 1;  
  **for** (String s: **g**.keySet()){  
 System.***out***.print**("Element: "** + i++ + **" "** + s + **" --> "**);  
 **for** (String k: **g**.get(s)){  
 System.***out***.print(k + **" ### "**);  
 }  
 System.***out***.println();  
 }  
}

**public** **double** gradoRelaciones() {

**int** nPruebas = 250;

**final** **double** E\_ABS = 0.0001;

**double** g0, g1 = 0;

**do** {

g0 = g1;

g1 = calcularGrado(nPruebas);

nPruebas += nPruebas;

} **while**(Math.*abs*(g0 - g1) > E\_ABS);

**return** g1;

}

**private** **double** calcularGrado(**int** nPruebas) {

ListaActores actores = CatalogoActores.*getCatalogoActores*().getLista();

Random rnd = **new** Random();

String a1, a2;

**double** acum = 0;

**int** numCaminos = 0;

**for** (**int** i = 0; i < nPruebas; i++) {

a1 = actores.obtenerPosicion(rnd.nextInt(actores.obtenerNumActores())).devolverNombreCompleto();

a2 = actores.obtenerPosicion(rnd.nextInt(actores.obtenerNumActores())).devolverNombreCompleto();

ArrayList<String> temp = devolverCaminoConectado(a1, a2);

**if**(temp.size() > 0) {

acum += temp.size() / 2;

numCaminos++;

}

}

**return** acum / numCaminos;

}

**public** ArrayList<Pareja> losDeMasCentralidad(**int** n){

**final** **int** N\_ITERACIONES = 427000;

Random rnd = **new** Random();

String a1, a2, nombre;

ListaActores actores = CatalogoActores.*getCatalogoActores*().getLista();

HashMap<String, Integer> apariciones = **new** HashMap<String, Integer>();

ArrayList<String> camino = **new** ArrayList<String>();

//Rellenar HashMap con actores

**for** (**int** i = 0; i < actores.obtenerNumActores(); i++) {

apariciones.put(actores.obtenerPosicion(i).devolverNombreCompleto(),0);

}

**for** (**int** i = 0; i < N\_ITERACIONES; i++) {

a1 = actores.obtenerPosicion( rnd.nextInt(actores.obtenerNumActores()) ).devolverNombreCompleto();

a2 = actores.obtenerPosicion( rnd.nextInt(actores.obtenerNumActores()) ).devolverNombreCompleto();

camino = devolverCaminoConectado(a1, a2);

**if** (camino.size() > 0) {

**for** (**int** j = 0; j < camino.size(); j++) {

nombre = camino.get(j);

**if** (apariciones.containsKey(nombre))

apariciones.put(nombre, apariciones.get(nombre) + 1);

}

}

}

ArrayList<Pareja> parejas = **new** ArrayList<Pareja>();

**for** (HashMap.Entry<String, Integer> entry : apariciones.entrySet())

parejas.add(**new** Pareja(entry.getKey(), entry.getValue()));

Collections.*sort*(parejas);

**if** (parejas.size() > n) {

ArrayList<Pareja> primerasParejas = **new** ArrayList<Pareja>();

**for** (**int** i = parejas.size(); i > parejas.size() - n; i--) {

primerasParejas.add(parejas.get(i - 1));

}

imprimirParejas(primerasParejas);

**return** primerasParejas;

} **else**{

imprimirParejas(parejas);

**return** parejas;

}

}

**public** **void** imprimirParejas(ArrayList<Pareja> parejas) {

**for** (Pareja pareja : parejas) {

System.***out***.println(pareja.getNombre() + ", apariciones: "

+ pareja.getApariciones());

}

}

**public** **class** Pareja **implements** Comparable<Pareja> {

String nombre;

Integer apariciones;

**public** Pareja(String nombre, **int** apariciones) {

**this**.nombre = nombre;

**this**.apariciones = apariciones;

}

**public** String getNombre() { **return** nombre; }

**public** **void** setNombre(String nombre) { **this**.nombre = nombre; }

**public** Integer getApariciones() { **return** apariciones; }

**public** **void** setApariciones(Integer apariciones) {

**this**.apariciones = apariciones;

}

@Override

**public** **int** compareTo(Pareja o) {

**return** **this**.apariciones.compareTo(o.apariciones);

}

}

}

### Coste de los métodos principales

**gradoRelaciones**: Su coste en el peor de los casos es lineal O(n), ya que en este supuesto tendrá que hacer la prueba para el total de actores del fichero inicial al no haber superado el error absoluto de 10-4, siendo el total de actores n.

**calcularGrado**: Para m número de pruebas su coste será O(2nm+ am), ya que llama al método devolverCamino m veces.

**losDeMasCentralidad**: Tendrá un coste O(2nm+ am \* p), ya que llama al método devolverCamino m veces. Tras esto, vuelca en tiempo lineal p el *Hasmap* en un *ArrayList*.

**crearGrafo:** El coste de este método es O(n\*m) siendo n el número de actores del catálogo y m el número medio de películas que tiene cada actor.

**estanConectados:** El coste de este método es lineal O(n + a) ya que, si bien el sacar de la *ArrayDeque* tiene un coste constante y el añadir a un *HashSet* también es constante, el ir tomando los adyacentes de cada elemento del grafo será lineal para n número de elementos (actores y películas mezclados). Del mismo modo, buscar si un elemento está en el *HashSet* también es constante, aunque en el peor de los casos llegaría a ser lineal. El parámetro a referencia al número total de arcos del grafo, que se puede obtener del producto n\*m, siendo m el tamaño medio del *value* de cada tupla.

**devolverCamino**:Al igual que el método anterior su coste es O(2n +a), siendo n el número de elementos del camino recorrido, ya que invertir el *ArrayList* a devolver tiene un coste lineal, y a el número de arcos del grafo. Este valor se dobla al incluir también el volcado de elementos del *HashMap* al *ArrayList* en el coste.

# Conclusiones

El volcado de fichero a grafo tarda menos de 20 segundos en su totalidad, y nos devuelve el camino entre dos nodos en 1 segundo.

Al finalizar la implementación de esta práctica y realizar las pruebas, hemos obtenido que el grado medio de relaciones es aproximadamente 3.322, en base a las siguientes pruebas:

* Grado actual (250 pruebas): 3.364 (error actual: 3.364)
* Grado actual (500 pruebas): 3.418 (error actual: 0.05400000000000027)
* Grado actual (1000 pruebas): 3.438 (error actual: 0.020000000000000018)
* Grado actual (2000 pruebas): 3.3225 (error actual: 0.11550000000000038)
* Grado actual (4000 pruebas): 3.32275 (error actual: 2.5000000000030553E-4)

Por lo tanto, para calcular la centralidad de todos los nodos del tipo actor contenidos en el grafo, apareciendo cada actor al menos 1 vez, necesitamos realizar el método con 427000 iteraciones.

Los 10 actores con mayor centralidad para nuestra prueba han sido los siguientes:

* Lloyd Kaufman, apariciones: 15
* Noel Gugliemi, apariciones: 15
* Keith (I) David, apariciones: 12
* Xander Berkeley, apariciones: 12
* Tony Devon, apariciones: 12
* Danny Trejo, apariciones: 10
* Michael (I) Madsen, apariciones: 9
* Valentina Cervi, apariciones: 8
* Spencer (I) Garrett, apariciones: 8
* Vincent Regan, apariciones: 8