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

**Fecha:** 23-XI-2015

**Participantes**

XABIER CASADO

SERGIO ERLANTZ TOBAL

PEIO VALLE

Índice de contenido

[Introducción 3](#_Toc436641941)

[Descripción de las estructuras de datos principales 5](#_Toc436641942)

[Diseño de las clases 6](#_Toc436641943)

[Clase GraphHash 7](#_Toc436641944)

[Clases de la Fase de proyecto 1 7](#_Toc436641945)

[Clases Actor y Película 7](#_Toc436641946)

[Clases ListaActores y ListaPeliculas 7](#_Toc436641947)

[Clases CatalogoActores y CatalogoPeliculas (MAEs) 8](#_Toc436641948)

[Clase LectorFichero 8](#_Toc436641949)

[Diseño e implementación de los métodos principales 8](#_Toc436641950)

[Método crearGrafo 8](#_Toc436641951)

[Método estanConectados y devolverCaminoConectado 9](#_Toc436641952)

[Casos de prueba 9](#_Toc436641953)

[public boolean estanConectados (String a1, String a2)   
public ArrayList<String> devolverCaminoConectados (String a1, String a2) 9](#_Toc436641954)

[Código 10](#_Toc436641955)

[GraphHash 10](#_Toc436641956)

[Coste de los métodos principales 12](#_Toc436641957)

[Conclusiones 12](#_Toc436641958)

# Introducción

El trabajo a continuación expuesto está basado en la primera práctica de la asignatura, la cual 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.

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:



La composición del fichero está dispuesta por líneas, correspondiendo cada una de éstas a un actor en la forma “*Apellido, Nombre”* seguido de las películas en las que forma parte del reparto, las cuales irán separadas por 3 almohadillas:

*Affleck, Ben ### Daredevil ### Gigli ### Curb Your Enthusiasm ### Gone Girl ### He's Just Not That Into You ### Changing Lanes ### Smokin' Aces ### To the Wonder ### Reindeer Games ### The Town ### Hollywoodland ### Pearl Harbor ### Surviving Christmas ### Batman vs. Superman ### The Company Men ### Paycheck ### Argo ### Jay and Silent Bob Strike Back ### Live by Night ### Runner Runner ### Boiler Room ### Man About Town*

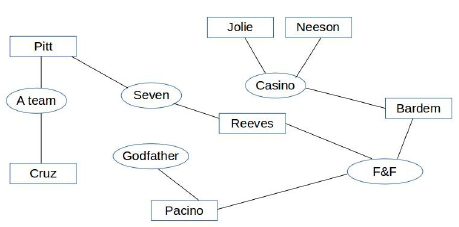
En cuanto a funcionalidad, el objetivo de la aplicación resultante de la primera práctica era la realización de forma eficiente (en lo que a tiempo de ejecución) de las siguientes tareas:

* Cargar los datos desde un fichero
* Búsqueda de un actor/actriz
* Inserción de un nuevo actor/actriz
* Devolver las películas de un actor dado
* Devolver los actores de una película dada
* Incrementar el dinero recaudado por una película en un valor dado
* Borrado de un actor/actriz
* Obtener una lista de actores ordenada (nombre, apellido)
* Guardar la lista en un fichero

De esta forma el programa buscaba implementar las operaciones básicas que realiza una base de datos, llamadas CRUD (Create, Read, Update, Delete).

Centrándonos en la presente práctica, su objetivo final consiste en obtener, en base al primer proyecto de la asignatura, un sistema que indique las **relaciones**, si las hubiera, **entre un** **par determinado de parámetros** (actores y/o películas). El algoritmo que implemente esta operación llamada *estanConectados* ha de ser eficiente, teniendo los componentes del grupo que razonar los motivos de la eficiencia.

Un esbozo de esta funcionalidad se presenta en la siguiente figura:



El resultado del método *estanConectados* será *True* si hay una cadena de relaciones que une al parámetro *a* con el *b* (siendo *a* y *b* bien actor-actor, película-película o actor-película). En este ejemplo concreto, llamar a *estanConectados(“Neeson”, “Pitt”)* devolverá *True*.

Para poder resolver este problema, previamente, deberemos obtener una estructura de datos adecuada, a partir de los datos de la lista de actores obtenida en la primera fase de la práctica, presentándonos varias opciones para ello:

* Tabla hash con una lista de elementos relacionados para cada elemento.
* Grafo de *Algorithms*, de *Sedgewick y Wayne (página 552).*

En nuestro caso, nos decantamos por la tabla hash, por tanto en la presente documentación implementaremos y comentaremos ésta solución.

Por otro lado, se nos pide devolver la cadena de uniones hechas desde el primer parámetro hasta llegar al segundo parámetro, sabiendo así qué conexiones se establecen entre ambos elementos, pudiendo observar, por ejemplo, las relaciones entre “*Neeson”* y “*Pitt”* en el esquema anterior.

En resumidas cuentas, los presentes objetivos nos suponen un primer acercamiento a la programación de grafos en Java.

# Descripción de las estructuras de datos principales

Como bien se comenta en la introducción, hemos optado por la primera solución propuesta para la implementación del grafo, por lo que la estructura de datos elegida ha sido la tabla hash *HashMap,* estructura 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.

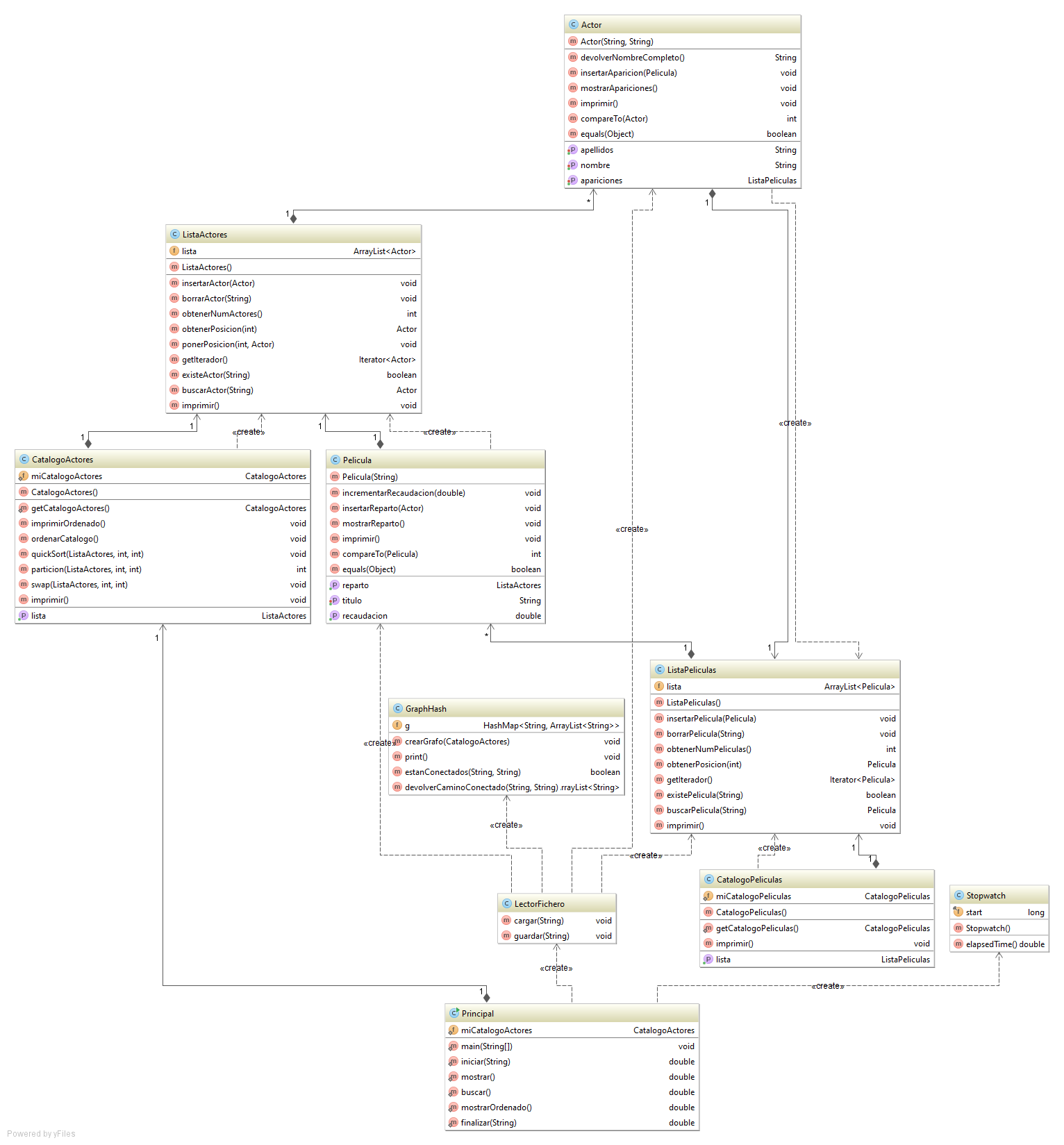
Lo hemos realizado así ya que en Java las colas no se pueden implementar directamente usando *Queue*, por ser ésta una *interfaz*, considerando por tanto que lo mejor era una *Deque*, que nos permite extraer elementos tanto por delante como por detrás, implementando de forma conjunta los algoritmos *FIFO*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

Por lo demás, las 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 tenemos que implementar tres métodos. Dichos métodos serán:

* *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>”*

## 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 implementados en esta tercera fase de la práctica, de los cuales se puede ver el coste en el apartado correspondiente al **código** del proyecto.

## 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 *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();  
 }  
}

### Coste de los métodos principales

**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) 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.

**devolverCamino**:El coste de este método es O(n), siendo n el número de elementos del camino recorrido, ya que invertir el *ArrayList* a devolver tiene un coste lineal.

# Conclusiones

Hemos aprendido a usar las colas y las pilas, en nuestro caso usando un *ArrayDeque*, una implementación de Java para las colas que permite acceder a ellas como si fueran bicolas (*ver* ***Descripción de la estructuras de datos principales****, de la presente documentación*).

Para hacer las pruebas funcionales *estanConectados,* hemos ido seleccionando pares de nombres aleatorios del fichero de actores original, dejando al método buscar las posibles relaciones entre éstos, mediante la estructura de grafo implementada.

El problema surge cuando los elementos no están relacionados, o bien tienen una cadena de relaciones muy larga, ya que entonces el algoritmo desarrollado tiene que recorrer todo el grafo y almacenarlo en memoria, no siendo posible en ordenadores comunes. El grafo, una vez es creado en el método inicial de carga, contiene 1.521.947 elementos que se deben procesar en memoria e ir mirando uno por uno, estableciendo relaciones con los parámetros de entrada; es consecuencia de este proceso que en un momento dado a la JVM se le agota la pila, incluso incrementándola a 4Gb.

Algunas de las relaciones más largas encontradas han sido de entre 7 y 9 elementos:

* [J.C. Harrington, I Kill to Luv You, Christine Fisler, Midnight Show, Lemmy , The Curse of El Charro, Thadd Turner]
* [Asmund G.S. Grinaker, Skal vi danse, Tom A. Haug, Hjem, Mette (I) Holt, CSI: Miami, Adrianne Palicki]
* [Finn Schjoll, Skal vi danse, Tom A. Haug, Hjem, Meg (I) Foster, The Mentalist, Noel Gugliemi, Seven Mummies, Tim Eichorn]

Haciendo pruebas con relaciones cortas y llamando al método *estanConectados* hemos visto que en unos 5 segundos podemos hacer 3 llamadas, y en un minuto haríamos 32 llamadas al método, así que creemos que, viendo el fallo de no poder alojar tantos elementos en la RAM, sería más apropiado usar un Trie, una base de datos, o pasar el grafo a un fichero y hacerlo desde ahí, ya que entonces gastaríamos recursos del HDD en lugar de la RAM, evitando así las excepciones de *outOfMemoryException* o las *GC overhead limit exceded*.