

Título: Algorismes de detecció
automàtica de grups de recerca.

Volumen: 1 / 1

Alumno: Eduard Suñen Piñol

Director: Conrado Martínez Parra

Departamento: LSI

Fecha: 20/06/2001

Tabla de contenidos

1	Introducción	5
1.1	Definición del proyecto.....	5
1.2	Motivaciones	5
2	Objetivos	7
3	Planificación	8
4	Material y Métodos.....	9
4.1	Hardware y software	9
4.2	Fuentes de datos	10
4.3	Metodología	12
4.3.1	Modelización.....	12
4.3.1.1	Entrada	12
4.3.1.2	Salida.....	16
4.3.1.3	Calidad de una solución	18
4.3.1.4	Descripción del problema	26
4.3.2	Diseño.....	28
4.3.3	Evaluación de los algoritmos	28
5	Resultados.....	31
5.1	Introducción	31
5.2	El grafo de coautorías.....	31
5.3	Algoritmos.....	36
5.3.1	Tipos de algoritmos y características	36
5.3.2	Algoritmos de primera fase	38
5.3.2.1	Buscar las componentes conexas	38
5.3.2.2	Vecindad	42
5.3.2.3	Partición Forzada	46
5.3.2.4	Partición Natural	51
5.3.2.5	Divide y vencerás.....	54
5.3.2.6	Filtro Bipartición.....	57

5.3.2.7	Algoritmo de Clustering.....	61
5.3.3	Algoritmos de segunda fase	65
5.3.3.1	Optimo Local	71
5.3.3.2	Colocación	73
5.3.3.3	Extremal Optimization.....	75
5.3.3.4	Simulated Annealing.....	78
5.3.4	Algoritmos de tercera fase.....	87
5.3.4.1	Algoritmo Genético.....	87
5.4	Diseño de la aplicación	91
5.4.1	Dominio de la aplicación.....	93
5.4.2	Secuencia de ejecución.....	96
5.4.3	Componentes destacables.....	100
5.4.3.1	Grafo	100
5.4.3.2	Cola de prioridad.....	100
5.4.3.3	Clase PropuestaParticion.....	100
6	Conclusiones	112
7	Discusión	113
7.1	Planes de futuro.....	113
8	Agradecimientos.....	115
9	Bibliografía	116

1 Introducción

1.1 Definición del proyecto

Consiste en el diseño e implementación de algoritmos de búsqueda que muestren grupos de investigación establecidos a partir de su producción bibliográfica.

Bajo la denominación de grupos de investigación incluimos a todo conjunto de investigadores que se dedican a la actividad científica en cualquier ámbito y que trabajan en los mismos proyectos.

Entendemos por producción bibliográfica los documentos publicados principalmente en revistas científicas, o cualquier otro medio, que son el resultado de la actividad de estos grupos. Evidentemente si un grupo de investigación existe como tal y es productivo, los artículos de los proyectos que realizará contarán entre sus autores a todos o parte de los miembros de este grupo. Los algoritmos propuestos en este proyecto deben ser capaces de identificar cuáles son esos grupos a partir de las relaciones de coautoría.

1.2 Motivaciones

La idea de desarrollar un proyecto de estas características se originó en el seno del *Servei d'Estudis de l'Institut Municipal d'Investigació Mèdica (IMIM)*. En este centro, a parte de proyectos de investigación biomédica tal y como su nombre indica, se realizan estudios bibliométricos para caracterizar la producción científica española y más concretamente la originada en Cataluña. Estos estudios manejan datos bibliométricos de autores, centros y artículos clasificados en distintas áreas temáticas. En la mayor parte de los ámbitos de la ciencia los investigadores no trabajan en solitario sino que se agrupan formando un equipo de trabajo para aumentar su productividad. En la realización de estos estudios es de gran interés la identificación de estos equipos de investigación para compararlos de la misma manera que se compara la producción entre autores, entre centros o entre los distintos ámbitos de la ciencia.

Una manera de determinar los grupos es que ellos mismos o los centros en los que están asociados los identifiquen. Pero esta no es una práctica habitual y, cuando se hace, se suelen tratar de grupos falsos, maquillados estratégicamente para que contengan investigadores de

peso de cara a la obtención de financiación. Al hacerlo así no encontramos una colaboración real entre los individuos. Nos interesan los grupos de investigación reales, los formados por aquellas personas que, siendo de distintos centros y de distintos ámbitos científicos han desarrollado proyectos conjuntamente formando lo que aquí hemos definido como grupo de investigación. La manera más sincera de obtener estos grupos es analizar la producción bibliográfica, analizar los rastros de su actividad científica. A través de la coautoría de los artículos publicados los algoritmos que aquí se propondrán deben permitir la definición de los grupos de investigación de una manera automática.

2 Objetivos

Objetivo general:

- Diseñar un sistema automatizado para la búsqueda de grupos de investigación a partir de los artículos que han publicado.

Objetivos específicos:

- Describir formalmente del problema a resolver:
 - Especificar la entrada del problema.
 - Especificar la salida del problema.
 - Definir algoritmo de búsqueda de grupos.
- Especificar una metodología que permita valorar la cualidad de los resultados obtenidos.
- Diseñar diferentes técnicas algorítmicas que resuelvan el problema.
- Diseñar un aplicativo que de soporte a la implementación de todos los algoritmos antes propuestos y de las funciones evaluadoras de calidad.
- Evaluar los algoritmos mediante pruebas experimentales utilizando la metodología propuesta como segundo objetivo.

3 Planificación

<u>Tarea</u>	<u>Días</u>
Asimilación del problema	1
Descripción formal	8
Diseño general aplicación	15
Implementación aplicación	25
Pruebas y mantenimiento aplicación	30
Diseño algoritmos	1 * 11 algoritmos = 11
Implementación algoritmos	1 * 11 algoritmos = 11
Pruebas y mantenimiento algoritmos	2 * 11 algoritmos = 22
Completar memoria	10
TOTAL	130 días

130 días a 7 horas / día = **910 horas.**

910 horas a 1.500 ptas/hora = **1.365.000 Ptas. 8.203,82 euros.**

4 Material y Métodos

Para realizar el proyecto se ha usado tanto material puramente informático (hardware y software) como fuentes de datos bibliométricos. Se detalla a continuación todo el material así como la metodología seguida para el diseño de las aplicaciones algorítmicas.

4.1 Hardware y software

En cuanto a hardware disponemos de un ordenador personal con un procesador Pentium III a 550Mhz, 256Mb de RAM y 8Gb de disco que corre sobre el sistema operativo Windows 2000 Professional. En este ordenador se ha llevado a cabo todo el desarrollo del proyecto y se han ejecutado en él todos los algoritmos. Se ha utilizado un ordenador personal y no un supercomputador para ejecutar algoritmos que manejan grandes cantidades de datos. Esta aclaración sirve como justificación ante la imposibilidad de hacer una sesión de pruebas exhaustivas y como punto a tener en cuenta en las medidas de eficiencia mostradas.

La base de datos que alimenta la aplicación esta en un Dual Pentium II a 400Mhz con 1GB de RAM y 32Gb de disco duro que actúa de servidor.

El diseño de la aplicación se ha desarrollado mediante diagramas UML usando el programa Microsoft Visio 5.0. La implementación del algoritmo se ha realizado en el lenguaje Perl v5.6. Se trata de un lenguaje interpretado que soporta la orientación a objetos siendo compatible como destinatario de un diseño propuesto en UML. Se ha elegido este lenguaje por la facilidad que muestra a la hora de programar estructuras de datos complejas, permitiendo así al programador todo tipo de modificaciones y pruebas con gran rapidez. Para escribir los programas en Perl se ha usado el editor de texto EditPlus v2.01 que permite trabajar con múltiples ficheros a la vez y también nos ayuda el hecho de que entiende la sintaxis de Perl y nos colorea el código.

En la realización de la documentación se han utilizado las herramientas ofimáticas proporcionadas por el paquete Office 2000 de Microsoft.

Los datos que alimentan la aplicación se encuentran en una base de datos relacional Oracle v8.1.6 y se accede a ellos desde el programa en Perl mediante una conexión con la interfaz ODBC (Open Data Base Connection).

4.2 Fuentes de datos

Como se ha dicho en la definición del proyecto debemos encontrar los grupos a partir de las relaciones de coautoría presentes en su producción bibliográfica. Esta producción bibliográfica está almacenada en la base de datos Oracle mencionada en el apartado anterior. Aquí discutiremos el contenido de esta base de datos y como este va a condicionar mucho nuestros algoritmos.

La información que contiene la base de datos bibliográfica proviene de la empresa estadounidense *Institute for Scientific Information (ISI)* de Philadelphia a través de uno de sus productos, el *National Citation Reports (NCR)*, que proporciona toda la producción bibliográfica de un determinado país. En nuestro caso disponemos de la producción científica de Cataluña entre los años 1981 y 1998, ambos incluidos.

Los datos de este producto están llenos de problemas que deben considerarse, tanto en contenido como en estructura.

La unidad básica de información del NCR son los artículos. Cada registro de artículo consta de la información bibliográfica básica, es decir el título, la revista de publicación, con indicación del número, volumen y páginas, la lista completa de autores así como una lista de centros y una lista de áreas temáticas. La relación entre los autores de un artículo y los centros asociados no existe.

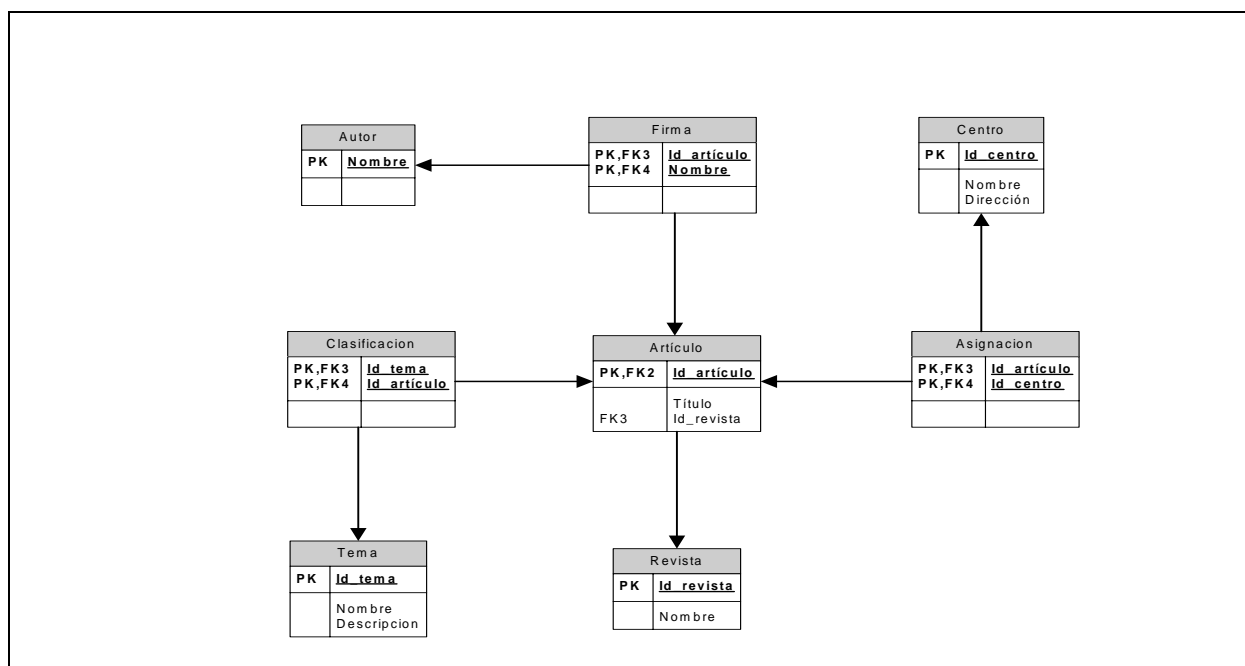


Figura 4.1 Diagrama entidad-relación de las tablas de la base de datos origen.

El problema principal está en la manera en como se identifican los autores en la base de datos. Hay que tener en cuenta que la información bibliográfica es recopilada por los miembros del ISI ‘manualmente’ a partir de las publicaciones científicas en donde aparecen como autores de un artículo una lista de nombres. Esa lista de nombres es la información que nos encontramos en la base de datos. Por lo tanto tenemos que un autor es simple y llanamente un nombre, o más concretamente un apellido más las iniciales del nombre. Esta característica conlleva una serie de problemas que a continuación se detallan.

En primer lugar el hecho de que la recopilación de datos sea manual introduce todo tipo de errores tipográficos en los nombres de los autores. Un autor que se llame Juan José González Pérez puede ser encontrado como González JJ, JoseGonzalez J, GonzalezPerez JJ, Perez JJG o con cualquier letra mal ‘picada’. Esto supondrá que un mismo autor estará varias veces en la base de datos y su producción bibliográfica estará diseminada o dispersa bajo las distintas denominaciones del autor.

Pero el problema principal nos lo encontramos en el caso inverso: dos o más investigadores que utilizan el mismo nombre bibliográfico para firmar los artículos científicos de los que son autores. Tanto porque se llaman de forma similar o por algún error en el transcurso de la captura y procesado de la información nos encontramos en nuestra base de datos gran cantidad de nombres de autores que hacen referencia, en la vida real, a más de un investigador. En un ejemplo como el anterior con un apellido tan común como González y unas iniciales como J pueden referenciar a varios Jorge González, varios Juan González, Joses, Joaquines etc. Esta realidad tiene como consecuencia directa que toda la producción bibliográfica de los diferentes investigadores aparezca ‘como si’ perteneciera a un único individuo y por lo tanto se altere la estructuración en grupos de investigación. Dos autores distintos con un mismo nombre que formen dos grupos de investigación provocarán la unión de estos grupos ya que dicho autor es indistinguible con los datos actuales.

La base de datos que se ha utilizado contiene la información de toda la producción científica, los artículos, publicados en revistas registradas por el ISI a lo largo del periodo 1981-1998 y en los que como mínimo uno de los centros estaba arraigado en Cataluña. Concretamente disponemos de 52.609 artículos, 50.423 nombres de autores distintos y una media aproximada de 5 autores por artículo creando un total de 266.778 firmas. El volumen de datos es considerable sobretodo teniendo en cuenta que la ejecución del algoritmo se realizará sobre un ordenador personal.

4.3 Metodología

Para resolver el problema que nos hemos propuesto hacen falta esclarecer una serie de pasos. Primero debemos identificar y modelizar nuestro problema para que pueda ser resuelto computacionalmente. Luego diseñaremos estrategias algorítmicas que con la modelización de nuestro problema nos irán proporcionando distintas soluciones. Finalmente necesitamos una aplicación que se encargue de todos los trámites de obtención de datos, implementación de los algoritmos, gestión de su ejecución, tratamiento de los resultados, etc.

4.3.1 Modelización

Nuestro problema consiste en encontrar grupos de investigación a partir de su producción bibliográfica.

Para poderlo modelizar debemos saber como serán los datos de entrada y como los datos de salida. Para evaluar las salidas formalizaremos también una metodología para puntuar la calidad de los resultados de los algoritmos.

Vamos a definir con precisión cada uno de los elementos que intervienen en nuestro problema. En primer lugar la entrada, después la salida, la calidad de esta salida y finalmente en qué va a consistir nuestro problema.

4.3.1.1 Entrada

La entrada del problema es la producción bibliográfica concretamente nos fijaremos en la coautoría de los artículos contenidos en nuestra base de datos. Tendremos autores y tendremos relaciones de coautoría entre estos autores.

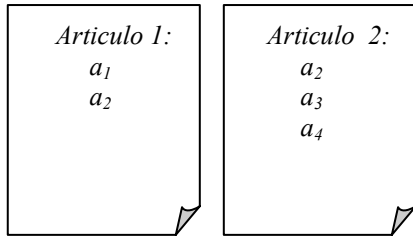
Def 4.1 Sea EN la entrada definida como un par de conjuntos: $EN = \langle A, C \rangle$.

Def 4.2 Sea A el conjunto de autores: $A = \{a_1, a_2, \dots, a_n\}$.

Def 4.3 Sea n su cardinalidad: $n = |A|$.

Def 4.4 Sea C el conjunto de relaciones de coautoría entre todo par de autores que alguna vez han firmado algún artículo juntos: $C = \{(a_x, a_y) \mid a_x \in A \wedge a_y \in A \wedge x \neq y\}$.

Por ejemplo, si tenemos como producción bibliográfica de 2 artículos:



Los conjuntos A y C son:

$$A = \{a_1, a_2, a_3, a_4\}$$

$$C = \{(a_1, a_2), (a_2, a_3), (a_2, a_4), (a_3, a_4)\}$$

Para modelizar la entrada de manera que sea fácilmente manejada por los algoritmos utilizaremos un grafo. Con un grafo podemos representar elementos de un conjunto mediante los vértices y relaciones entre estos elementos mediante las aristas.

Def 4.5 Sea G un grafo definido como una pareja de conjuntos $G = \langle V, E \rangle$

Def 4.6 Sea $V = \{v_1, v_2, \dots, v_n\}$ el conjunto de sus vértices.

Def 4.7 Sea $E = \{(v_x, v_y) \mid v_x \in V \wedge v_y \in V \wedge x \neq y\}$ el conjunto de aristas (edges).

La analogía con la coautoría es directa. El conjunto de autores es representado por los vértices del grafo y las relaciones de coautoría por las aristas del grafo.

Estableceremos una equivalencia entre los conjuntos A y V :

$$\forall_{x=1}^n a_x \in A \wedge v_x \in V \wedge a_x \equiv v_x \quad (4.1)$$

y entre los conjuntos C y E :

$$\forall_{x=1}^n \forall_{y=1}^n a_x \in A \wedge a_y \in A \wedge v_x \in V \wedge v_y \in V \wedge a_x \equiv v_x \wedge a_y \equiv v_y \wedge x \neq y \wedge (v_x, v_y) \in E \wedge (a_x, a_y) \in C \Rightarrow (a_x, a_y) \equiv (v_x, v_y) \quad (4.2)$$

Existen más características que definen los grafos y que debemos tener en cuenta. En primer lugar pueden ser dirigidos o no dirigidos.

Def 4.8 Un grafo dirigido es aquel en el que las relaciones entre un vértice v_1 y un vértice v_2 es distinta a la de v_2 y v_1 : $(v_1, v_2) \neq (v_2, v_1)$. Interesa la dirección de la relación.

Def 4.9 Un grafo no dirigido es aquel en el que la relación entre dos elementos es simétrica y no tiene sentido hablar de direcciones: $(v_1, v_2) = (v_2, v_1)$.

En el caso de las coautorías el grafo es no dirigido porque la relación entre dos autores que han firmado un mismo artículo es simétrica. Otra característica de los grafos es que tanto los vértices como las aristas pueden estar etiquetadas, pueden tener información.

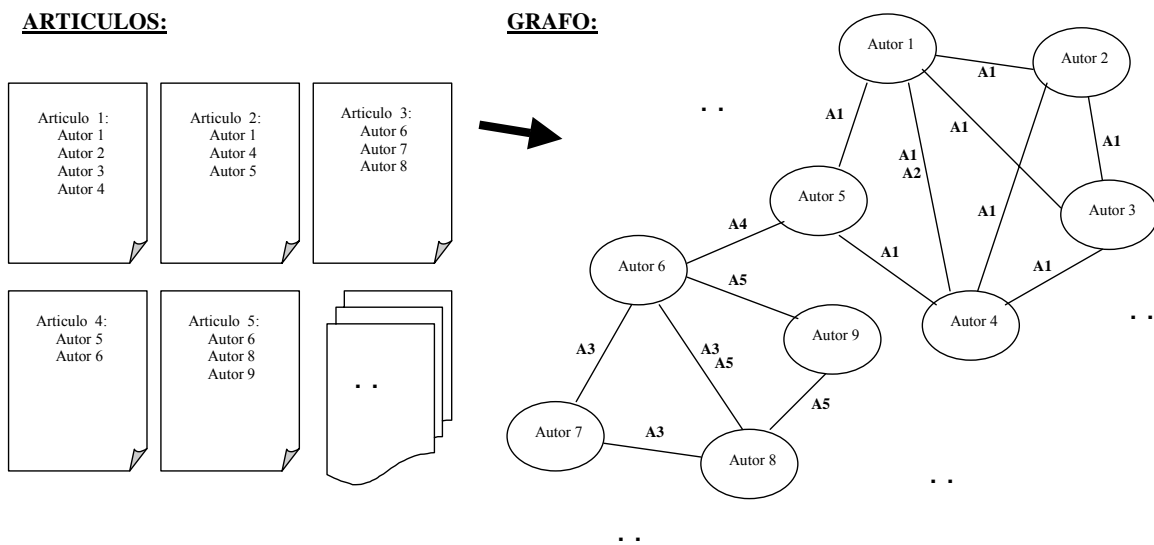


Figura 4.2 Esquema de como pasamos de la información bibliográfica de artículos con sus autores aun grafo de coautorías.

Aprovecharemos este detalle para almacenar en el grafo de coautorías un valor tanto en los vértices como en las aristas. En los vértices almacenaremos un peso que representará el número de artículos en los que ha participado ese autor. En las aristas almacenaremos un peso que representará el número de publicaciones en las que los dos autores que une la arista han participado juntos.

Modelizaremos la coautoría con un grafo no dirigido y etiquetado $G=(V,E)$ con las siguientes características:

Def 4.10 V : los vértices representan los autores.

Def 4.11 E : las aristas representan las relaciones de coautoría entre autores.

Def 4.12 $\text{peso}: V \rightarrow \mathbb{N}$ etiqueta en los vértices que representa la producción total de artículos de un autor.

Def 4.13 $\omega: E \rightarrow \mathbb{N}$ etiqueta en las aristas con un natural que corresponde con el número de artículos en los que dos autores han coincidido. También le llamaremos peso de la arista.

Def 4.14 $n = |V|$ el número de autores que participan en el grafo.

Def 4.15 $m = |E|$ el número aristas.

Def 4.16 $\text{peso_grafo}: G \rightarrow \mathbb{N}$ el peso de todos los vértices del grafo.

$$\text{peso_grafo}(G) = \sum_{v \in V} \text{peso}(v) \quad (4.3)$$

Def 4.17 $\text{peso_total}: G \rightarrow \mathbb{N}$ el peso de todas las aristas del grafo.

$$\text{peso_total}(g) = \sum_{a \in E} \omega(a) \quad (4.4)$$

Def 4.18 Sea $\text{ady}: V \rightarrow V^*$ la lista de los adyacentes de un vértice.

$$\text{ady}(x) = \{y \mid y \in V \wedge (x, y) \in E\} \quad (4.5)$$

Def 4.19 $\text{aridad}: V \rightarrow \mathbb{N}$ el número de aristas que son incidentes en un vértice. También se le llama *grado*.

$$\text{aridad}(x) = \sum_{y \in V \wedge (x, y) \in E} 1 \quad (4.6)$$

Def 4.20 $\text{peso_incidentes} : V \rightarrow \mathbb{N}$ la suma de los pesos de las aristas incidentes a un vértice.

$$\text{peso_incidentes}(x) = \sum_{y \in V} \omega(x, y) \quad (4.7)$$

Def 4.21 Sea $\text{densidad_local} : V \rightarrow [0..1]$ de un vértice la proporción de aristas que en el subgrafo que definen el y todos sus adyacentes.

$$\text{densidad_local}(v) = \frac{\sum_{(x,y) \in E} 1 | (x, v) \in E \vee (y, v) \in E}{\text{aridad}(v) * (\text{aridad}(v) + 1)} \quad (4.8)$$

Existen algunas palabras sinónimas que a partir de este momento pueden ser usadas indistintamente en este documento. Por ejemplo: (vértice, autor, elemento, individuo, miembro) o (arista, relación, coautoría, adyacencia).

4.3.1.2 Salida

Como salida obtendremos los grupos encontrados por los algoritmos. Este conjunto de grupos es lo que llamaremos una partición. Vamos a formalizar cada uno de estos elementos junto con sus propiedades.

Def 4.22 Un grupo $GR := A^+$ es un conjunto de autores que a su vez es subconjunto del conjunto total de autores o de los vértices del grafo.

$$GR = \{a_1, a_2, \dots, a_r\} \wedge GR \subseteq A \wedge GR \subseteq V \wedge r \leq n \quad (4.9)$$

Def 4.23 Sea $\text{card} : GR \rightarrow \square$ el tamaño de un grupo.

$$\text{card}(g) = | g | \quad (4.10)$$

Def 4.24 Una partición $PA := GR^+$ es un conjunto de grupos disjuntos entre ellos.

$$PA = \{gr_1, gr_2, \dots, gr_p\} \wedge \forall_{i=1}^p \bigcap_{i=1}^p gr_i = \phi \wedge \bigcup_{i=1}^p gr_i \subseteq A \quad (4.11)$$

Por facilidad de nuestros algoritmos no permitiremos la múltiple adscripción de un autor a más de un grupo. Además puede darse el caso en que algunos autores no acaben asignados a ningún grupo a causa de los filtros que aplican algunos algoritmos.

De cara a la manipulación de esta información dotaremos de un identificador para cada grupo mediante la función id :

Def 4.25 Sea $id : GR \rightarrow \mathbb{N}$ una función que dado un grupo nos devuelve un natural que lo identifica.

Def 4.26 Sea $GP : \rightarrow \square^+$ el conjunto de identificadores de grupo de una partición.

$$GP = \left\{ id(gr) \mid gr \in PA \wedge \forall_{g \in PA - \{gr\}} id(g) \neq id(gr) \right\} \quad (4.12)$$

Def 4.27 Sea $p = | PA | = | GP |$ como el número de grupos de una partición.

Definimos una función inversa a id :

Def 4.28 Sea $grupo : GP \rightarrow GR$ una función que dado un natural nos devuelve el grupo que este natural identifica.

Utilizando la notación de grupos mediante un natural identificador definiremos una función de asignación tal como sigue:

Def 4.29 Sea $f : A \rightarrow GP \cup \{0\}$ una función de asignación que dado un autor nos devuelve el identificador del grupo al que pertenece o 0 cuando no tiene grupo.

$$f(a) = \begin{cases} 0 & \text{si } \forall_{g \in PA} a \notin g \\ id(gr) & \text{si } a \in gr \wedge \forall_{g \in PA - \{gr\}} a \notin g \end{cases} \quad (4.13)$$

Para acercarnos más a como será finalmente la implementación de una partición damos ahora una definición alternativa utilizando las funciones de asignación a elementos.

Def 4.30 Sea $P = \{f(a_1), f(a_2), \dots, f(a_n)\}$ un conjunto de funciones de asignación a cada uno de los elementos de A que nos dan un mapa de la partición resultante elemento a elemento.

Finalmente daremos una función que nos indicará para un autor no sólo a que grupo pertenece si no a todos los grupos con los que ha coautorado con alguno de sus elementos.

Def 4.31 Sea $VA: A \rightarrow GP^*$ una función que devuelve los identificadores de grupos a los que el autor mantiene una relación de coautoría con alguno de sus integrantes.

$$VA(a) = \left\{ g \mid \exists_{a' \in A - \{a\}} f(a') = g \wedge (a, a') \in E \right\} \quad (4.14)$$

4.3.1.3 Calidad de una solución

Entendemos por solución la salida de nuestro problema que hemos formularizado con detalle en el apartado anterior. Para poder comparar los resultados que nos devuelven distintos algoritmos necesitamos un criterio objetivo que nos puntúe estas soluciones. Es evidente que la mejor evaluación nos la harían personas expertas que conocieran la realidad de los grupos de investigación. Pero nosotros proponemos que el sistema sea automático, no solo en la búsqueda de las soluciones sino también en la ponderación de las mismas. Además, tal y como se verá más adelante, dentro de algunos esquemas algorítmicos que intentan resolver el problema nos encontramos que necesitan saber en todo momento la calidad de la solución que están tratando.

Vamos a definir una función que medirá la calidad de una solución propuesta, de una partición, usando los términos antes planteados. Formalizaremos por partes cada uno de los componentes que intervienen en la función de calidad. Tendremos en cuenta en todo momento que evaluamos una partición de salida como la definida en **Def 4.30** con las funciones de asignación elemento a elemento y que nos referenciaremos continuamente al grafo de coautorías que define la entrada.

Mediremos primero lo que llamaremos afinidad de un autor a un grupo. Debe entenderse la afinidad como la proximidad de este elemento al grupo en términos de coautoría.

Def 4.32 Sea $afinidad : A \times GP \rightarrow [0..1]$ una función que dado un autor y un id_grupo nos devuelve la afinidad que tiene este autor a este grupo mediante las coautorías entendida como la suma de las coautorías de ese autor con miembros de ese grupo partido por el total de coautorías de este autor. En el grafo sería el sumatorio del peso de las aristas que van del elemento al resto del grupo partido por el peso de todas las aristas incidentes al autor.

$$afinidad(a, g) = \frac{\sum_{x \in A} \omega(a, x) \wedge f(x) = g}{\sum_{x \in A} \omega(a, x)} \quad (4.15)$$

En el caso de la cualidad nos interesará sólo la afinidad de un autor a su grupo. Siguiendo este patrón podemos definir la afinidad de un grupo:

Def 4.33 Llamamos $afinidad_grupo : GP \rightarrow [0..1]$ la afinidad media de un grupo y se corresponde con la afinidad media de sus autores a este grupo.

$$afinidad_grupo(g) = \frac{\sum_{a \in A} afinidad(a, g) \wedge f(a) = g}{card(g)} \quad (4.16)$$

Finalmente vamos al que realmente nos interesa que es el de la afinidad media de la partición que, no es la media de las afinidades de los grupos sino la media de las afinidades de todos los elementos de la partición a su grupo.

Def 4.34 La $afinidad_particion : P \rightarrow [0..1]$ es una función que calcula la media de afinidad de cada elemento a su grupo asignado.

$$afinidad_particion = \frac{\sum_{a \in A} afinidad(a, f(a))}{n} \quad (4.17)$$

En algunos algoritmos necesitamos calcular que aporta cada elemento al cálculo final de ese factor. Serán las funciones que llamaremos $ap_$ (de aportación) y indicarán que aportaría un elemento al factor final si estuviera asignado a un determinado grupo con el que comparte alguna coautoría.

En el caso preciso de la afinidad calcularemos qué aporta cada elemento de afinidad de la siguiente manera:

Def 4.35 Sea $ap_afinidad : A \times GP \rightarrow [0..1]$ una función que calcula la aportación al sumatorio global de afinidad empleada por el elemento si estuviera asignado al grupo indicado.

$$ap_afinidad(a, g) = \frac{afinidad(a, g)}{n} \quad (4.18)$$

Otro término parecido a la afinidad es el que denominaremos intrarrelaciones. Aquí ponderaremos las aristas que unen autores de un grupo propio con relación al total de aristas del grafo. Se trata de un factor medido a nivel de partición.

Def 4.36 Sea $intrarrelaciones_particion : \rightarrow [0..1]$ una función que mide la proporción que ocupa la suma de los pesos de las aristas que van entre autores de un mismo grupo (aristas internas) sobre la suma de los pesos de todas las aristas del grafo (todas las aristas).

$$intrarrelaciones_particion = \frac{\sum_{x \in A \wedge y \in A} \omega(x, y) \wedge f(x) = f(y)}{\sum_{x \in A \wedge y \in A} \omega(x, y)} \quad (4.19)$$

Si tenemos la proporción de peso de las aristas internas podemos obtener la proporción de peso de las aristas externas entendidas como interrelaciones.

Def 4.37 Sea $interrelaciones_particion : \rightarrow [0..1]$ una función que mide la proporción que ocupa la suma de los pesos de las aristas que van entre autores de distintos grupos (aristas externas) sobre la suma de pesos de todas las aristas del grafo.

$$interrelaciones_particion = \frac{\sum_{x \in A \wedge y \in A} \omega(x, y) \wedge f(x) \neq f(y)}{\sum_{x \in A \wedge y \in A} \omega(x, y)} \quad (4.20)$$

Evidentemente estos dos últimos factores son complementarios:

$$intrarrelaciones_particion = 1 - interrelaciones_particion \quad (4.21)$$

Podemos refinar las intrarrelaciones a nivel de elemento y a nivel de grupo. En el primer caso nos referimos a la aportación de que haría un elemento a la función de intrarrelaciones_partición si estuviera asignado a ese elemento.

Def 4.38 Sea $ap_intrarrelaciones : A \times GP \rightarrow [0..1]$ la función que calcula la suma de pesos de las aristas que unen en autor con los miembros del grupo en relación a la suma total de las aristas del grafo.

$$ap_intrarrelaciones(a, g) = \frac{\sum_{x \in A} \omega(a, x) \wedge f(x) = g}{\sum_{x \in A \wedge y \in A} \omega(x, y)} \quad (4.22)$$

Def 4.39 Entendemos por $intrarrelaciones_grupo : GP \rightarrow [0..1]$ la suma del peso de las aristas entre elementos del grupo en relación al total de aristas del grafo.

$$intrarrelaciones_grupo(g) = \frac{\sum_{x \in A \wedge y \in A} \omega(x, y) \wedge f(x) = f(y) = g \wedge x \neq y}{\sum_{x \in A \wedge y \in A} \omega(x, y)} \quad (4.23)$$

Tanto la afinidad como la intrarrelación son factores que miran por un lado que los autores estén asignados al grupo que les pertoque y si hay muchas aristas que cruzan estos grupos. Uno puede adivinar que si la partición resultante fuera un solo grupo conteniendo todos los autores la afinidad de estos es total y todas las aristas son internas, no hay interrelaciones. Si un algoritmo intenta maximizar estos parámetros simplemente no tiene que partir los datos de entrada y conseguirá su objetivo. Falta algún tipo de indicador que mire que los grupos que han salido tengan sentido, que sean compactos. Existen varias alternativas en este sentido.

Podemos analizar los grupos encontrados a través de los subgrafos del grafo de entrada que estos grupos representan.

Def 4.40 Sea $G' = \langle V', E' \rangle$ un subgrafo del grafo $G = \langle V, E \rangle$ si G' cumple las definiciones Def 4.5, Def 4.6 y Def 4.7 y $V' \subseteq V \wedge E' \subseteq E$.

Def 4.41 Entenderemos por $G^{g \in GP} = \langle V, E \rangle$ como un subgrafo (**Def 4.40**) del grafo de coautorías $G = \langle V, E \rangle$ tal que:

$$\forall_{x \in V', y \in V'} f(x) = g \wedge f(y) = g \wedge (x, y) \in E \quad (4.24)$$

A partir de ahora cuando definamos la *densidad* y la *aridad_media* lo haremos refiriéndonos a el subgrafo que representa al grupo.

Un factor que nos puede servir es la densidad de este subgrafo entendida como el número de aristas que contiene frente al máximo de aristas que puede tener. Recordemos que el máximo de aristas de un grafo es:

$$\max_aristas = \frac{n \cdot (n-1)}{2} \quad (4.25)$$

Def 4.42 Llamamos *densidad* $G \rightarrow [0..1]$ al número de aristas que hay en el grafo partido por el máximo que puede haber.

$$densidad(\langle V, E \rangle) = \frac{m}{\max_aristas} = \frac{2m}{n \cdot (n-1)} \quad (4.26)$$

El problema que nos lleva la densidad es que el denominador crece cuadráticamente a medida que los grafos contienen más elementos. Cuando un grafo es de un tamaño considerable o bien tiene una gran cantidad de aristas la densidad decrece drásticamente. Este parámetro forzaría a que los grupos sean lo más pequeños posible o, si realmente hay un grupo grande, que demuestre su razón de ser con una gran cantidad de aristas. Pero aun así es muy sensible a estos grupos grandes. Imaginemos que en la realidad existe un grupo de, pongamos, 50 investigadores. Para que la densidad llegue a puntuar bien todos los integrantes deberán haber publicado con casi todos los 49 restantes. A poco que algunos no lo hayan hecho la densidad bajará.

Otro parámetro parecido es la *aridad_media*. Hacemos una media de las aristas que reciben los elementos.

Def 4.43 Sea *aridad_media* $G \rightarrow \square$ una función que mide la media de aristas que son incidentes a cada vértice del grafo.

$$aridad_media(\langle V, E \rangle) = \frac{\sum_{v \in V} aridad(v)}{n} = \frac{2m}{n} \quad (4.27)$$

Aquí ocurre un caso contrario al de la densidad. A medida que se vayan añadiendo elementos al grafo, estos suelen llevar más aristas con lo que el valor crece drásticamente. Aquí se puntuarán mucho aquellos grupos grandes que tengan una cantidad de aristas relativamente grande. Para ilustrar los diferentes comportamientos de estos dos factores tenemos la tabla Tabla 4.1.

n	m	<i>aridad media</i>	<i>densidad</i>
3	3	2	1
4	4	2	0,66
5	5	2	0,5
10	10	2	0,22
50	50	2	0.041
100	100	2	0.02

Tabla 4.1 Comparación de los factores *aridad_media* y *densidad* con grafos en los que las aristas forman un círculo entre todos los vértices. Notar que la *aridad media* es 2 siempre.

Para afinar mejor la compactibilidad de un grafo utilizaremos una función que va a ser una media entre la *densidad* y la *aridad media*. Mientras que en la *densidad* el denominador tiene el tamaño del grafo elevado al cuadrado castigando demasiado la falta de aristas en los grafos grandes y la *aridad media* tiene el tamaño en el denominador elevado a 1 permitiendo que los grafos grandes sean vacíos intentaremos buscar una función parecida en la que el denominador tenga un exponente entre 1 y 2.

Otra característica que nos hemos pasado por alto en la definición de las dos funciones anteriores es que contamos las aristas sin tener en cuenta que llevan asociado un peso. No es lo mismo contar una arista entre dos autores con una sola coautoría que otra arista con 50. De todas maneras hay que ir con mucho cuidado ya que tampoco es 50 veces más importante una arista con ese peso que otra con peso 1. Queremos que puntúen más las aristas pesadas pero no queremos que lo hagan directamente proporcional a su peso. Para lograr este equilibrio sumaremos los pesos pero previamente los someteremos a un exponente < 1 que rebajará el efecto que comentamos.

De todas estas conclusiones hemos sacado una nueva función que permitirá medir lo que hemos llamado *cohesión* de un grupo.

Def 4.44 Sea $cohesion_grupo : GP \rightarrow \mathbb{R}$ una función que mide la suma de los pesos de las aristas internas al grupo elevadas a un cierto exponente con valor entre 0 y 1 partido por el tamaño de ese grupo elevado a otro exponente con valor entre 1 y 2

$$cohesion_grupo(g) = \frac{\sum_{x \in A, y \in A} \omega(x, y)^\alpha \wedge f(x) = f(y) = g}{card(grupo(g))^\beta} \quad (4.28)$$

dónde $0 \leq \alpha \leq 1$ y $1 \leq \beta \leq 2$.

Esta última ecuación hace referencia a un grupo y si deseamos una función de calidad de toda una partición deberemos calcular una media entre todos los grupos.

Def 4.45 Sea $cohesion_particion : P \rightarrow \mathbb{R}$ una función es una media de las cohesiones de los grupos que forman la partición ponderada según el tamaño de cada grupo.

$$cohesion_particion = \frac{\sum_{g \in GP} \frac{cohesion_grupo(g)}{card(grupo(g))}}{n} \quad (4.29)$$

Definiremos finalmente la función de aportación a la cohesión detallada por autor y grupo afin.

Def 4.46 Sea $inc : A \times GP \rightarrow \{0,1\}$ una función que devuelve 1 si el autor no pertenece al grupo y 0 si pertenece.

$$inc(a, g) = \begin{cases} 0 & \text{si } f(a) = g \\ 1 & \text{si } f(a) \neq g \end{cases} \quad (4.30)$$

Def 4.47 Sea $ap_cohesion : A \times GP \rightarrow \mathbb{R}$ una función que calcula la aportación de una hipotética asignación del elemento al grupo a la función de cohesión general. Calculamos la suma de aristas que unen el autor con el grupo en cuestión, elevadas al exponente correspondiente, partido por el tamaño del grupo (+1 si el elemento no está en el grupo, simulando su pertenencia) elevado al otro exponente. Todo esto partido por el número de elementos de todo el grafo.

$$ap_cohesion(a, g) = \frac{\sum_{x \in A} \omega(a, x)^\alpha \wedge f(x) = g}{(card(grupo(g)) + inc(a, g))^\beta} \quad (4.31)$$

dónde $0 \leq \alpha \leq 1$ y $1 \leq \beta \leq 2$.

Ya tenemos los ingredientes para hacer una función de calidad de una solución propuesta. Utilizaremos la afinidad media de la partición, las intrarrelaciones y la relación de cohesión de toda la partición. Debemos tener en cuenta que mientras las funciones de afinidad e interrelaciones nos devuelven resultados acotados ente 0 y 1 la cohesión no tiene límite superior. Aun así, si se eligen bien los parámetros que definen los exponentes, podemos garantizar que en la mayoría de veces las cohesiones no superarán la unidad. Existe también

la posibilidad de limitar la puntuación de la cohesión a 1 interpretando que todo aquel valor de cohesión superior a 1 es suficientemente alto y no nos interesa saber cuánto más bueno es.

Para combinar los tres factores podemos hacer una media aritmética, una media ponderada o, tal y como haremos al final una media ponderada con exponentes. Hacer una media aritmética da poco juego ya que no se pueden manipular parámetros que nos ayuden a dar más importancia a un factor que a otro. Haciendo la media aritmética con pesos en cada factor solucionamos este problema pero puede pasar que: aunque un factor, pongamos por caso la cohesión, sea muy bajo o nulo, puede estar compensado porque otro factor esté muy alto (esto ocurre en el ya mencionado caso en el que todos los autores pertenecen a un mismo grupo). Nos interesa hacer una media entre todos los factores, que tenga pesos para jugar a dar más o menos importancia a cada factor y, también queremos que todos los factores tengan una cierta consistencia de manera que si uno baja excesivamente toda la función de calidad lo haga independientemente de lo que hagan el resto de factores, por esto hemos propuesto la siguiente función de calidad:

Def 4.48 Sea $calidad_particion : P \rightarrow \mathbb{R}$ una función que mide la calidad de la partición utilizando los factores definidos en **Def 4.34**, **Def 4.37** y **Def 4.45**.

$$calidad_particion = \left(afinidad_particion^\alpha \cdot intrarrelaciones_particion^\beta \cdot cohesion_particion^\gamma \right)^{\frac{1}{\alpha+\beta+\gamma}} \quad (4.32)$$

dónde α, β y γ son exponentes parametrizables.

Utilizaremos esta función para comparar los resultados de distintas técnicas algorítmicas de una manera objetiva y también la utilizarán algunos algoritmos para guiarse a la hora de encontrar soluciones.

En la implementación de los algoritmos utilizaremos esta función de calidad multiplicada por 100 para que sea más fácil de leer.

De una manera similar a como lo hemos hecho con cada factor por separado calcularemos la aportación de las parejas autor grupo a la función de calidad.

Def 4.49 Sea $ap_calidad : A \times GP \rightarrow \mathbb{R}$ una función que calcula la aportación de la hipotética asignación del autor al grupo a la función de calidad.

$$ap_calidad(a, g) = \left(ap_afinidad(a, g)^\alpha \cdot ap_intrarrelaciones(a, g)^\beta \cdot ap_cohesion(a, g)^\gamma \right)^{\frac{1}{\alpha+\beta+\gamma}} \quad (4.33)$$

dónde α, β y γ son exponentes parametrizables.

4.3.1.4 Descripción del problema

Nuestra preocupación en este proyecto es encontrar los grupos de investigación pero como hemos formularizado la entrada a un formato de grafo vamos a convertir el problema en un problema de partición de grafos.

Existen en la literatura computacional discusiones sobre el problema del particionado de grafos (*graph partitioning*) pero nuestro caso tiene circunstancias especiales. En la mayoría de problemas de particionado de grafos se establecen unos parámetros a priori que ajustan el problema. Tales parámetros son un número fijo de grupos a encontrar o un tamaño fijo de cada grupo encontrado. Nosotros tenemos que particionar el grafo sin conocer el número de grupos que saldrán ni su tamaño.

Cualquier problema de *graph partitioning* intenta encontrar la solución óptima, o una de ellas. En el particionado de grafos sin parámetros encontrar la solución óptima va a ser prácticamente imposible. En los primeros el problema de encontrar esta solución óptima se convierte en un problema NP-duro. Por lo tanto encontrar la mejor partición de un grafo sin tener en cuenta tamaños fijados tiene que pertenecer a la clase de problemas que se resuelven en tiempo exponencial. Tal y como hemos visto en las fuentes de datos manejaremos tales cantidades de datos que necesitamos garantizar que el problema lo resolverán algoritmos en tiempo polinómico y, que sean lo más eficientes posibles.

Esto nos hace descartar la idea de plantearnos el problema como una maximización pura de la función de calidad antes propuesta. En primer caso por lo inútil que sería en función del coste y del volumen de datos manejados. En segundo lugar porque la función de calidad no es más que una propuesta que intenta simular una valoración subjetiva hecha por un experto conocedor de los grupos de investigación.

Estamos en un caso de aproximación a la mejor solución. Todos y cada uno de los algoritmos aquí propuestos aproximarán la solución óptima por distintos caminos. Unos llegarán a funciones de calidad sensiblemente altas y otros se quedarán a medio camino. De

todas maneras sería interesante definir qué es un algoritmo para nosotros en el problema. No definimos que significa la palabra algoritmo sino que debe hacer lo que hemos llamado algoritmo para que sea considerado así.

Def 4.50 Un algoritmo $ALG : G \rightarrow \langle P, \square \rangle$ es un procedimiento que dada una entrada en el formato de grafo de coautorías (de Def 4.5 a Def 4.20) devuelve una partición en el formato P definido en Def 4.30 con un valor de calidad calculado definido en **Def 4.48**.

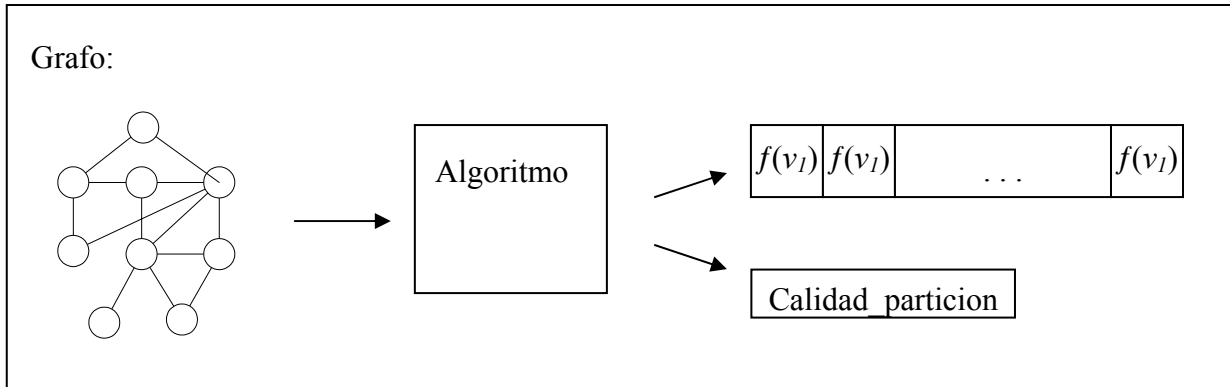


Figura 4.3 Esquema que ilustra las componentes de entrada y salida de un algoritmo.

Siempre hemos hablado de algoritmos en plural y es que ante la imposibilidad de encontrar la solución perfecta utilizaremos más de un esquema para elegir el que mejor nos haya funcionado. Al recibir una entrada la vamos a procesar por todos los algoritmos que sea posible ejecutar y elegiremos el que nos ha generado la solución con mayor factor de calidad.

Def 4.51 Sea $AE = \{al_1, al_2, \dots, al_l\}$ el conjunto de algoritmos existentes.

Def 4.52 Sea $AP : G \rightarrow AL^*$ el conjunto de algoritmos que se pueden utilizar siguiendo unas restricciones impuestas por el propio algoritmo en función del grafo de entrada.

$$AP \subseteq AE \tag{4.34}$$

Def 4.53 Sea $S : G \times ALG \rightarrow \langle P, \square \rangle$ una función que dado un grafo de entrada y un algoritmo devuelve la partición que el algoritmo ha generado al ejecutarse y su calidad. Como precondition tenemos que $ALG \in AP(G)$.

Def 4.54 Llamamos $MS : G \rightarrow \langle P, \square, ALG \rangle$ como una función que dado un grafo devuelve la mejor solución encontrada para partir el grafo G en forma de partición, su calidad y qué algoritmo la ha calculado.

$$MS(g) = \left\{ \langle p, c, a \rangle \mid p \in P \wedge c \in \square \wedge a \in AP(g) \wedge \forall_{a' \in AP(g) - \{a\}} S(g, a') = \langle p', c' \rangle \wedge c' \leq c \right\} \quad (4.35)$$

4.3.2 Diseño

Para el diseño de la aplicación que contiene los algoritmos se ha utilizado una metodología basada en el diagramaje en UML (Unified Modeling Language).

Toda la aplicación se ha construido con tecnología de programación orientada a objetos. Primero se han definido el dominio de la aplicación: las clases de objetos que formarán parte del programa. Para hacerlo se ha usado el diagrama de clases o diagrama estático.

Para ilustrar como interactúan los objetos del dominio se ha utilizado el diagrama de secuencia o diagrama dinámico. En el podemos ver como los diferentes objetos del dominio se invocan las operaciones para interactuar.

No ha hecho falta el diseño de las interfaces de usuario porque estas no existen. El programa en principio se invoca como un comando en la línea de comandos del sistema operativo. Es muy posible que en un futuro este programa formará parte de un sistema de información bibliométrico que incluirá un sistema de consultas a la base de datos fuente y un apartado en el que se podrán ejecutar estos algoritmos. Pero esto queda fuera del alcance de este proyecto.

4.3.3 Evaluación de los algoritmos

Por las limitaciones de maquinaria ejecutaremos los algoritmos sobre un subconjunto de la base de datos. Utilizaremos principalmente la producción bibliográfica catalana del año 1992 con filtraje. En esta ejecución se crea una componente inicial de 373 vértices que los algoritmos tienen que partir. Comprobaremos como la parten todos los algoritmos de primera fase y como a partir del resultado del algoritmo DivideYVenceras lo mejoran los algoritmos de segunda fase.

Para evaluar los algoritmos utilizaremos en primer lugar la función de calidad de las soluciones generadas tal como se ha especificado en el apartado 4.3.1.3.

Mediremos su coste mediante la notación asintótica $O(f_x)$ [4] dándonos como resultado el número de pasos que ejecuta en función de una entrada de longitud x . Esta f_x simboliza el máximo número de pasos que realizará el algoritmo para resolver una entrada de longitud x . Como tamaño de entrada siempre consideraremos que es n el número de autores a agrupar o en número de nodos del grafo de coautorías. Vemos a continuación ejemplos de cómo va a ser esta función:

- $O(1)$: Coste constante. Se realizan cálculos inmediatos independientemente del tamaño de la entrada. Es el más pequeño.
- $O(n)$: Coste lineal. El número de pasos es directamente proporcional al tamaño de la entrada.
- $O(a)$: El número de adyacentes. Cuando el algoritmo hace un bucle sobre el número de adyacentes de un vértice dado. Si el grafo es muy denso el número de adyacentes se acerca a n pero en la mayoría de los casos es mucho menor. Por eso usamos este y no el lineal. $O(a) \leq O(n)$.
- $O(r)$: El número de grupos. Cuando debemos recorrer los grupos adyacentes a un autor. Suelen ser menos que el total de adyacencias a . $O(r) \leq O(a) \leq O(n)$.
- $O(\log x)$: Coste logarítmico. Siendo x cualquiera de los valores (n, a, r, \dots) indica que el número de pasos va a ser en proporción logarítmica al valor indicado. Se le llama también cuasi constante.
- $O(x \log x)$: Coste cuasilineal.
- $O(x^2)$: Coste cuadrático. Indica que el número de pasos va a ser el cuadrado del valor indicado.

En general podemos decir que esos costes pertenecen a la clase de problemas con complejidad polinómica ya que podemos representar su coste asintótico mediante una función que equivale a un polinomio de un cierto grado (en los ejemplos hemos llegado a un máximo

grado 2, pero podríamos subir de grado sin salirnos de la clase). Se dice que el problema pertenece a la clase P.

Existen otro tipo de problemas que aun siendo computacionalmente resolubles necesitan un tiempo de ejecución que se mide mediante una función exponencial. Su coste es $exp(\text{polinomio})$. Es evidente que este tipo de costes son prohibitivos para resolver problemas en la práctica.

Finalmente encontramos una serie de problemas que se denominan NP (NonPolinomic) de los que se sabe que si se encuentra solución al problema esta es devuelta en tiempo polinómico pero se es incapaz, de momento, de encontrar una solución que nos de siempre una respuesta en un tiempo polinómico.

Cuando hemos definido nuestro problema lo hemos clasificado en la clase NP. Pero luego hemos redefinido el problema quitando la idea de maximización, de encontrar una solución óptima a cambio de buscar sencillamente una aproximación. Al reducir la complejidad de nuestro problema los algoritmos que lo van a solucionar se encuentran todos en el conjunto P con resolución en tiempo polinómico.

De cada algoritmo al dar su pseudocódigo analizaremos en detalle su coste asintótico mediante la función $O()$.

También daremos detalle del coste real en segundos de las pruebas que hemos hecho, sobretodo a nivel comparativo entre técnicas distintas.

5 Resultados

5.1 Introducción

En este apartado vamos a ver cual a sido el resultado de la elaboración de este proyecto tal y como ha sido especificado en los anteriormente.

Primero analizaremos la entrada de nuestro problema entendida como un grafo de coautorías. Veremos qué características tiene este grafo.

Primero vamos a especificar como son los algoritmos que se han propuesto en este proyecto. Esta debería ser la parte central de todo el proyecto atendiéndonos a su título.

Después veremos a grandes pinceladas como es el diseño de la aplicación que soporta los algoritmos. Cuales son las clases de objetos que intervienen y cual es el orden de invocación para que al final podamos ejecutar los algoritmos y guardar los resultados.

5.2 El grafo de coautorías

Una vez procesada la información bibliográfica contenida en nuestras bases de datos a un formato grafo obtenemos un grafo con las siguientes características.

Característica	Valor
Número de vértices	50328
Número de aristas	267506
Número de puentes	1596
Número componentes	7990
Diámetro aprox.	14
Suma pesos aristas	495865
Máximo peso aristas	197
Densidad	0.0211228642566662
Cohesión	0.0050383346762242

Tabla 5.1 Características del grafo de coautorías de toda la base de datos.

Seguidamente se le ha aplicado un proceso de filtraje consistente en:

- Eliminar los vértices con un peso inferior a 3: se quitan todos los autores que no han escrito más que 1 o 2 artículos. Se pretende eliminar a los becarios y los autores de muy poco peso.
- Eliminar las aristas con peso inferior a 2: se quitan todas aquellas relaciones de coautoría de menos de 2 artículos. Para considerar que dos autores están relacionados deben haber publicado juntos un mínimo de 2 artículos.
- Eliminar los vértices con aridad inferior a 2: se quitan todos los autores que han publicado sólo con otro autor y con nadie mas. Para considerar que un autor entra a formar parte del estudio necesita haberse relacionado con más de un autor diferente.
- Eliminar todas las aristas puente: todas aquellas aristas que al ser eliminadas desconectan el grafo.

Hay que tener en cuenta que este filtraje se realiza al grafo virgen, se hace todo de golpe y que puede producir efectos como: al grafo resultante aun habiéndole filtrado las aristas puente aun le quedan. Esto ocurre porque al filtrar por varios criterios a la vez un filtraje provoca que se repitan los casos que ha filtrado otro.

Una vez terminado el filtraje obtenemos un grafo con las siguientes características:

Característica	Valor
Número de vértices	14294
Número de aristas	60946
Número de puentes	1235
Número componentes	1321
Diámetro aprox.	14
Suma pesos aristas	277117
Máximo peso aristas	197
Densidad	0.0596620307384664
Cohesión	0.0130173455924225

Tabla 5.2 Características del grafo resultante de filtrar el grafo de coautorías de la base de datos.

Vamos a comparar con más detalle las características de ambos grafos:

Peso de los vértices:

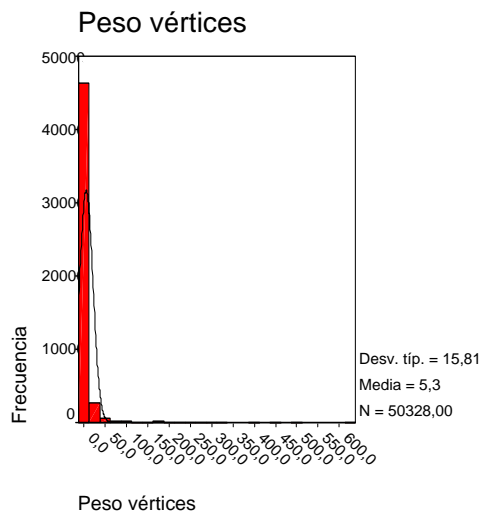
Grafo original

N	50328
Media	5,2910
Error típ. de la media	7,049E-02
Mediana	1,0000
Moda	1,00
Desv. típ.	15,8146
Varianza	250,1003
Asimetría	9,736
Error típ. de asimetría	,011
Curtosis	161,841
Error típ. de curtosis	,022
Rango	623,00
Mínimo	1,00
Máximo	624,00

Grafo filtrado

N	14294
Media	13,3108
Error típ. de la media	,1923
Mediana	6,0000
Moda	3,00
Desv. típ.	22,9931
Varianza	528,6849
Asimetría	7,206
Error típ. de asimetría	,020
Curtosis	97,290
Error típ. de curtosis	,041
Rango	621,00
Mínimo	3,00
Máximo	624,00

Histograma



Histograma

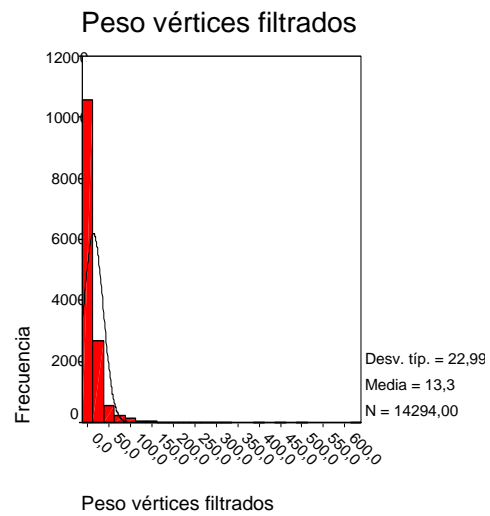


Figura 5.1 Comparación de los pesos de los vértices entre el grafo original y el filtrado.

Peso de las aristas:

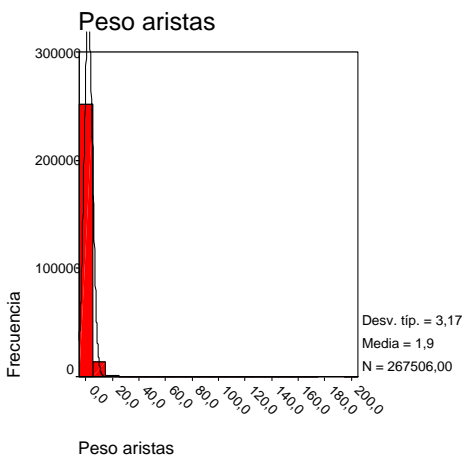
Grafo original

N	267506
Media	1,8537
Error típ. de la media	6,139E-03
Mediana	1,0000
Moda	1,00
Desv. típ.	3,1750
Varianza	10,0805
Asimetría	13,992
Error típ. de asimetría	,005
Curtosis	372,913
Error típ. de curtosis	,009
Rango	196,00
Mínimo	1,00
Máximo	197,00

Grafo filtrado

N	60946
Media	4,5469
Error típ. de la media	2,385E-02
Mediana	3,0000
Moda	2,00
Desv. típ.	5,8872
Varianza	34,6593
Asimetría	8,175
Error típ. de asimetría	,010
Curtosis	120,325
Error típ. de curtosis	,020
Rango	195,00
Mínimo	2,00
Máximo	197,00

Histograma



Histograma

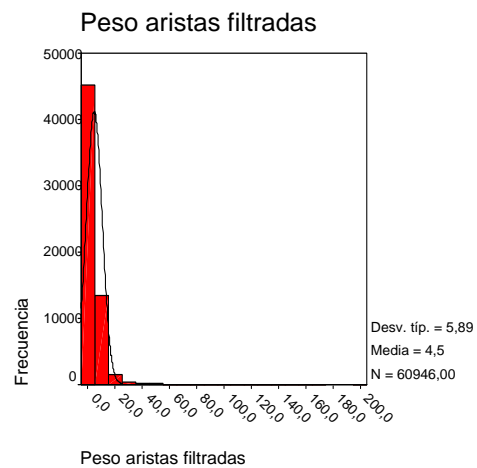


Figura 5.2 Comparación de los pesos de las aristas entre el grafo original y el filtrado.

Aridad de los vértices:

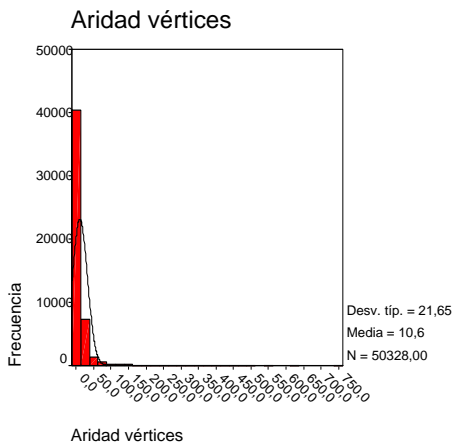
Grafo original

N	50328
Media	10,6305
Error típ. de la media	9,650E-02
Mediana	5,0000
Moda	,00
Desv. típ.	21,6487
Varianza	468,6645
Asimetría	8,378
Error típ. de asimetría	,011
Curtosis	124,209
Error típ. de curtosis	,022
Rango	747,00
Mínimo	,00
Máximo	747,00
Suma	535012,00

Grafo filtrado

N	14294
Media	8,5275
Error típ. de la media	,1108
Mediana	5,0000
Moda	2,00
Desv. típ.	13,2444
Varianza	175,4143
Asimetría	5,773
Error típ. de asimetría	,020
Curtosis	62,203
Error típ. de curtosis	,041
Rango	322,00
Mínimo	,00
Máximo	322,00
Suma	121892,00

Histograma



Histograma

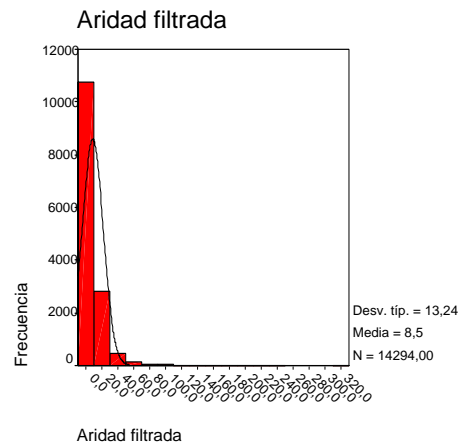


Figura 5.3 Comparación entre las aridades de los vértices de los dos grafos.

5.3 Algoritmos

Los que veremos a continuación son los algoritmos que se proponen en este proyecto para resolver el problema de la búsqueda de grupos de investigación que hemos convertido en un problema de partición de grafos.

Definiremos primero los distintos tipos de algoritmos que hay y en qué se diferencian. Después pasaremos a discutir algoritmo por algoritmo diciendo en primer lugar la idea general en que se basa, daremos un pseudocódigo que lo implementa, propondremos variantes y analizaremos los resultados.

5.3.1 Tipos de algoritmos y características

Según la clasificación que sigue este documento existen tres tipos de algoritmos que se diferencian entre sí por el momento del proceso en el que se ejecutan.

Tenemos primero los algoritmos denominados de ‘primera fase’. Como se puede adivinar son los primeros que se ejecutan. reciben un grafo y generan una partición de este. Digamos que son los algoritmos normales y corrientes tal y como se han definido en Def 4.50.

Los algoritmos de segunda fase son los que se ejecutan inmediatamente después de un algoritmo típico o de primera fase. Parten de los resultados que estos han generado e intentan mejorarlos. No son algoritmos que construyan soluciones a partir del grafo sino que alteran iterativamente soluciones existentes con la intención de optimizarlas.

Finalmente tenemos el algoritmo genético que también podríamos llamarlo de tercera fase porque se ejecuta después de los de segunda fase. Este algoritmo recoge todas las soluciones generadas por los algoritmos anteriores y las recombina generando nuevas soluciones. Parecido a los algoritmos de segunda fase la intención no es partir el grafo sino navegar por las soluciones posibles encontrando la óptima.

podríamos clasificar los algoritmos siguiendo otros criterios. Por ejemplo existen algoritmos que empezando por una agrupación que consiste en un solo grupo con todos los elementos lo parten por las zonas más débiles llegando a un punto en equilibrio. Son algoritmos que particionan. Existen otros algoritmos que empiezan por una partición donde

cada uno de los elementos está en un grupo aparte. Poco a poco se encargan de ir juntando aquellos elementos que están más cercanos. Se trata de algoritmos que fusionan. Todos los algoritmos de segunda fase alteran las soluciones y no se puede considerar que parten ni que fusionan, simplemente mueven elementos de lugar haciendo un grupo más pequeño para que otro sea más grande. Por otro lado el algoritmo genético lo que hace es recombinar. El resultado de una recombinación no se puede considerar tampoco ni una fusión ni una partición, es una mezcla. Aun así, por las pruebas que se han hecho, puede considerarse el movimiento de elementos en la solución como una fusión porque la mayoría de ellos son hacia un grupo mayor. Por otra parte la recombinación suele dar particiones con grupos más pequeños por lo tanto tiende mas a ser una partición que una fusión.

Básicamente estas son las características que van a categorizar los algoritmos:

	particionadores	fusionadores	alteradores	recombinadores
1ª fase	Partición natural, Partición forzada, Filtro bipartición	Vecindad, Clustering, Divide y venceras		
2ª fase			Optimo local, Colocación, Extremal Optimization, Simulated Annealing	
3ª fase				Algoritmo genético.

Tabla 5.3 Los distintos algoritmos clasificados según la fase en que se ejecutan y la filosofía que utilizan.

Otra de las características que habrá que tener en cuenta de cada algoritmo es si se puede repetir en una secuencia de ejecuciones sobre particiones hijas. Si se para atención al diseño de la aplicación el programa veremos que este ataca la solución partiendo un grafo y a cada grupo resultante se le vuelve aplicar el mismo tratamiento hasta que se considera que es un grupo definitivo y no merece ser partido. Existen algoritmos que una vez aplicados no se pueden volver a utilizar a ningún grupo resultante ni a ningún grupo resultante de las particiones de estos grupos resultantes. Esto ocurre porque las condiciones que le han llevado a generar ese grupo se repiten en el subgrupo y no será capaz de partirlo dando como resultado el mismo grupo. En la explicación de cada algoritmo haremos hincapié en este de detalle y se captará el sentido que tiene.

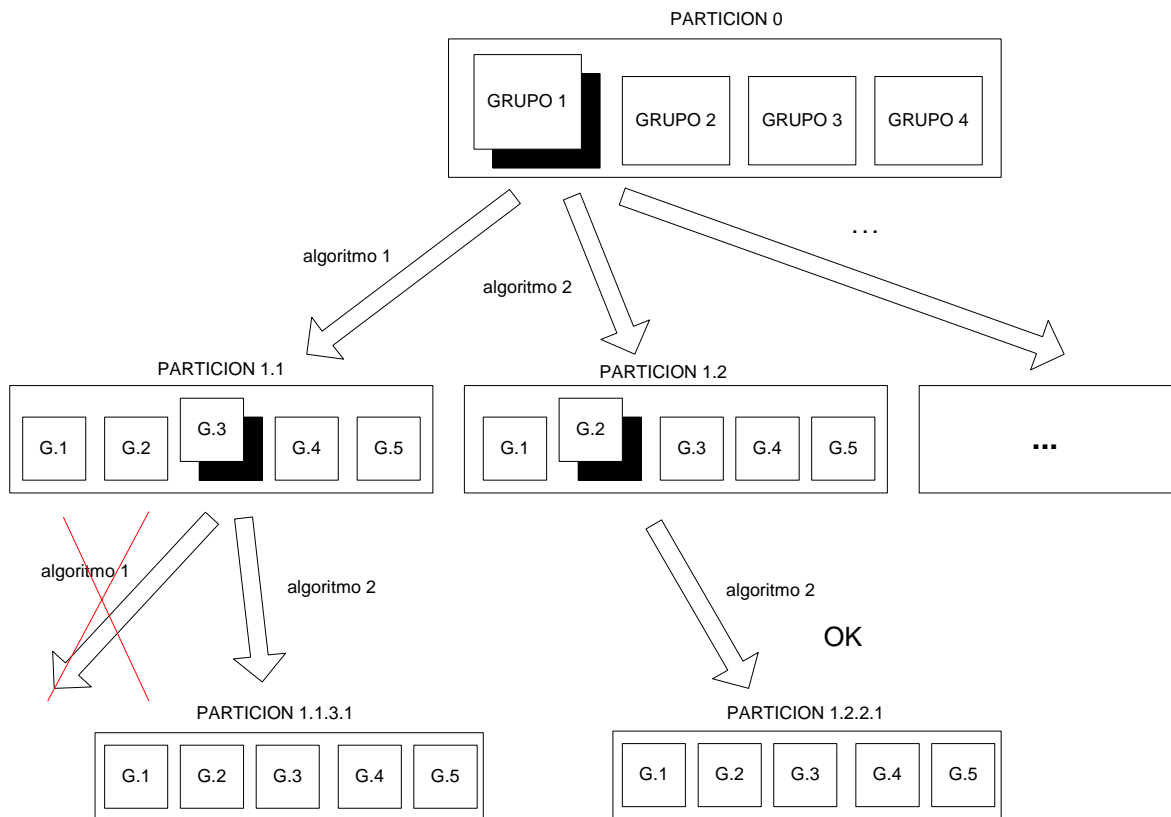


Figura 5.4 Esquema de ejecución de los algoritmos en el que vemos el caso en que se intentan repetir los algoritmos 1 y 2. En el caso de la izquierda no dejamos repetir el algoritmo 1 porque tenemos indicado que no admite repetición en el curso de una partición recursiva. En algoritmo 2 en cambio permitimos que se pueda ejecutar varias veces.

5.3.2 Algoritmos de primera fase

Llamamos algoritmos de primera fase aquellos que atacan en primera instancia el problema. Reciben un grafo de coautorías como entrada y nos devuelven una partición de este grafo en grupos de investigadores.

5.3.2.1 Buscar las componentes conexas

5.3.2.1.1 Idea

La primera técnica que a uno se le puede ocurrir para partir un grafo es hacerlo por las diferentes componentes conexas que lo componen.

Def 5.1 Una componente conexas es un grupo de vértices que están todos conectados entre si.

Def 5.2 Que dos vértices estén conectados significa que existe un conjunto de aristas consecutivas (un camino) que los une.

$$\text{conectados}(x, y) = (x, y) \in E \vee \exists \{v_1, v_2, \dots, v_c\} \subseteq V \left| \begin{array}{l} (x, v_1) \in E \wedge \forall_{i=1}^{c-1} (v_i, v_{i+1}) \in E \wedge (v_c, y) \in E \end{array} \right. \quad (5.1)$$

Una componente conexa debe contener todos los vértices posibles conectados entre ellos. Entre dos componentes de un mismo grafo no pueden haber vértices comunes.

A partir de esta definición, se puede intentar encontrar grupos de vértices (autores) que estén desconexos entre si, es decir, que no exista ningún camino de aristas (coautorías) que los pueda comunicar. Si esto ocurre significa que dos autores que estén en distintas componentes conexas nunca podrán formar un grupo porque no existe ninguna relación ni directa ni indirecta de coautoría que los una.

5.3.2.1.2 Pseudocódigo

Para implementar esta búsqueda se utiliza un algoritmo que realiza un recorrido en profundidad sobre el grafo [4]. Empezando por un vértice cualquiera va marcando a él y a sus adyacentes a una misma componente conexa. Este mismo proceso se realiza para cada uno de estos adyacentes de manera recursiva. Al final se tienen todos los vértices conectados entre sí asignados a una misma componente conexa. Si todavía existen vértices sin componente asignada se elige uno de ellos y se vuelve a realizar el algoritmo con una componente nueva hasta que todo vértice tenga una de asignada.

```

procedimiento BuscarComponentesRecursivo
  entrada vertice, componente
inicio
  Visto[vertice]:=componente;
  para cada adyacente de Adyacentes(vertice) hacer
    si Visto[adyacente]=0 entonces
      BuscarComponentesRecursivo(vertice,componente);
    fsi
  fmientras
fin

procedimiento BuscarComponentes
inicio
  componente:=0;
  para cada vertice de Vertices hacer
    si Visto[vertice]=0 entonces
      BuscarComponentesRecursivo(vertice,componente++);
    fsi
  fmientras
fin

```

5.3.2.1.3 Resultados

Esta primera idea resulta sencilla de entender y también de codificar, pero no da resultados efectivos ya que, con pruebas que se han ido realizando con la base de datos disponible, no era capaz de dividir el grupo de forma significativa. Más del 90% de los autores estaban en la misma componente conexa, ver Tabla 5.4 y Tabla 5.4. Vemos que el grupo más grande se lleva 40686 autores y que hay

Las razones por las que esta metodología falla son varias. En primer lugar tenemos la poca entereza de la información de la base de datos al identificar los autores por el apellido y las iniciales del nombre. Esto aporta que varios autores que pueden pertenecer a distintos grupos con nombre parecido tengan la misma firma, sean indistinguibles y, por lo tanto, el algoritmo pase a unir los grupos. Una posible solución a este problema es la que propuso Zulueta [ZCB] que consistía en, de alguna manera, juntar el nombre del autor con el centro de trabajo, pero ella misma dedujo la precariedad de tal método debido a la posible movilidad de los autores. En segundo lugar por la propia ‘flaqueza’ del algoritmo al juntar dos grupos definidos por el mero hecho de que uno o algunos de sus componentes haya publicado alguna vez con algún componente del otro grupo. Además, en tercer lugar hay que destacar que solo echando una mirada a los datos nos damos cuenta que la mayoría de autores ha publicado con muchos otros y que, por tanto, existe una gran conectividad entre toda la comunidad científica que hace difícil la búsqueda de estos grupos e invalida este tipo de técnicas demasiado sencillas

N	7990
Media	6,2989
Error típ. de la media	5,0920
Mediana	1,0000
Moda	1,00
Desv. típ.	455,1555
Varianza	207166,5410
Asimetría	89,386
Error típ. de asimetría	,027
Curtosis	7989,920
Error típ. de curtosis	,055
Rango	40685,00
Mínimo	1,00
Máximo	40686,00
Suma	50328,00

Tabla 5.4 Tabla con los datos estadísticos de los tamaños de las componentes conexas del grafo original.

Valor	Frecuencia	Porcentaje	Porcentaje acumulado
1,00	7376	92,3	92,3
2,00	257	3,2	95,5
3,00	131	1,6	97,2
4,00	80	1,0	98,2
5,00	54	,7	98,8
6,00	31	,4	99,2
7,00	16	,2	99,4
8,00	17	,2	99,6
9,00	10	,1	99,8
10,00	4	,1	99,8
11,00	3	,0	99,9
12,00	2	,0	99,9
15,00	1	,0	99,9
16,00	3	,0	99,9
18,00	1	,0	99,9
19,00	1	,0	100,0
21,00	1	,0	100,0
27,00	1	,0	100,0
40686,00	1	,0	100,0
Total	7990	100,0	

Tabla 5.5 Tabla de frecuencias de tamaños de las componentes conexas del grafo original.

De todas maneras la búsqueda de componentes conexas resulta esencial para el desarrollo de la aplicación ya que se convierte en el primer paso para la aplicación de otras técnicas que necesitan la condición de conectividad para funcionar. Después de la lectura de datos y la representación de los mismos mediante un grafo se divide este grafo en varios grafos, uno por cada componente conexa y se procede a trabajar individualmente sobre cada uno de ellos. De otro modo no tendría sentido seguir trabajando con más algoritmos con un grafo desconexo.

5.3.2.2 Vecindad

5.3.2.2.1 Idea

La idea del algoritmo es agrupar cada autor con aquel autor más cercano, con su vecino. Solo nos falta definir con claridad qué entendemos por vecindad, qué autor es el vecino de otro.

Una primera aproximación consiste en interpretar el vecino de un autor como aquel con el cual ha publicado más artículos. Nótese que esta relación no es simétrica ya que, mientras un autor A ha publicado más artículos con B que con cualquier otro, nada impide que B haya publicado más artículos con otro autor C, diferente de A.

El criterio utilizado no es del todo conciso porque definimos solo un vecino por cada autor, aquel que ha publicado más con ese autor, cuando en realidad pueden haber más de uno y, de hecho, suelen haberlos. Una posible solución a este problema es tratar no solo un vecino inmediato, sino más de uno. Esto es muy fácil de probar en el algoritmo ya que solo basta con especificar el parámetro *MasDeUnVecino* a SI y ajustar el parámetro *PorcentajeVecindad*. Este último parámetro indica el % de artículos de los que ha publicado el autor con su vecino más cercano que tendrá que cumplir otro autor para considerarlo un vecino. Nótese que un 100% quiere decir que se relacionarán aquellos autores con máximo número de coautorías respecto al autor tratado, por lo tanto la temida situación de empate queda resuelta.

Debemos refinar la vecindad de un autor utilizando más factores además de las firmas en común con los demás autores. Intentaremos asociar un autor con aquel vecino más cercano en coautorías y a la vez más importante. Los factores a tener en cuenta son:

- *Número de coautorías*: el utilizado hasta ahora.
- *Peso*: número de artículos que ha publicado el vecino.
- *Aridad*: número de autores que han publicado con el vecino.
- *Densidad local*: densidad local (Def 4.21) del vértice que representa el autor vecino. Si la densidad local es alta significa que el vecino está muy relacionado y debe ser jefe de grupo.

Haremos una suma ponderada de estos factores para sacar el factor vecindad final.

Def 5.3 Llamaremos *vecindad* : $V \times V \rightarrow \mathbb{R}$ de un autor candidato a un autor referencia como la suma ponderada de los factores antes expuestos.

$$vecindad(x, y) = \omega(x, y) \cdot \alpha + peso(x) \cdot \beta + aridad(x) \cdot \delta + densidad_local(x) \cdot \gamma \quad (5.2)$$

dónde α, β, δ y γ son pesos parametrizables.

Nótese que la vecindad no es simétrica:

$$vecindad(x, y) \neq vecindad(y, x) \quad (5.3)$$

Def 5.4 Sea *vecino* : $A \rightarrow A \cup \emptyset$ de un autor aquel autor adyacente que presenta mayor vecindad.

$$vecino(x) = y \mid y \in A \wedge \forall_{y' \in ady(x) - \{y\}} vecindad(y', x) \leq vecindad(y, x) \quad (5.4)$$

Una vez calculadas las vecindades de los autores relacionados con el que estamos tratando se decide como vecino aquel con la mayor de ellas. En caso de empate se mirará factor por factor, en el orden que especifica el parámetro *OrdenFactoresDesempate*, para ir desempatando. Así elegimos el vecino que nos parece más cercano o más adecuado para agruparlo con nuestro autor.

5.3.2.2.2 Parámetros

- *PesosFactoresVecindad*: corresponden a α, β, δ y γ e indican el peso que tienen en la suma de factores la coautoría, la producción del vecino, su aridad y su densidad local respectivamente.
- *MasDeUnVecino*: indica si a cada autor se le puede asociar más de un vecino. En dicho caso se utilizará el siguiente parámetro para ver que autores serán los vecinos.
- *PorcentajeVecindad*: indica el % de artículos de los que ha publicado el autor con su vecino más cercano que tendrá que cumplir otro autor para considerarlo un vecino.
- *OrdenFactoresDesempate*: en caso de empate indica en qué orden se irán mirando factor por factor para desempatar el vecino. Sólo tiene sentido si *MasDeUnVecino* vale NO.

5.3.2.2.3 Pseudocódigo

Mostramos el pseudocódigo del algoritmo de vecindad sin tener en cuenta la casuística de todas las opciones que se pueden dar mediante la configuración. Sólo nos interesa dar una idea de cómo funciona.

```
procedimiento Vecindad
  entrada grafo
inicio
  Creamos un nuevo grafo.
  grafo2=Nuevo Grafo;

  Añadimos todos los vértices.
  para cada vertice desde 0 hasta n hacer
    AnadirVertice(vertice, Peso[vertice]);
    Marca[vertice]=$NO;
  fpara

  Añadimos todas las aristas necesarias.
  para cada vertice desde 0 hasta n hacer
    si Marca[vertice]=NO entonces
      vecino:=NINGUNO;vecindad_max:=0;
      para cada adyacente de vertice hacer
        si Vecindad(adyacente,vertice)>vecindad_max entonces
          vecino:=adyacente;
          vecindad_max:= Vecindad(adyacente,vertice);
        fsi
      fpara
      AnadirArista(vertice, vecino, $\omega$ (vertice,vecino));
      Marca[vertice]:=SI;
  fpara
retorna Crear partición(grafo2);
fin
```

5.3.2.2.4 Resultados

Los resultados del algoritmo pueden variar mucho debido a la gran cantidad de parámetros que permite configurar.

Agrupando por el máximo número de coautorías nos encontramos con que más del 90% de los elementos se quedan en un mismo grupo. Como conclusión de estos resultados podemos decir que el grafo de coautorías parece estar bastante conectado y que existen muchas coautorías de igual peso. La mayor parte de ellas, más de la mitad, son de peso 1, provocadas por asociaciones entre autores que han coincidido sólo en un artículo. También se puede considerar que estas relaciones de autores por coincidencia en un artículo no son significativas para determinar los grupos y se pueden eliminar en el proceso de filtraje.

Si se agrupan también los casos de empate con la opción MasDeUnVecino aún es peor, se forma un grupo con la mayoría de autores. Por lo tanto se descarta como opción válida, como mínimo por lo que hace a los datos que disponemos.

La suma ponderada de factores parece ser el camino a seguir. Aquí se tienen en cuenta más detalles en el vecindaje y por tanto los emparejamientos son más ajustados.

En las pruebas realizadas sobre la base de datos en el año 1992 nos devuelve el siguiente resultado:

Grupos	Tam.Medio	Tam.Maximo	Tiempo	Calidad
48	7.77	53	2.013	83.59

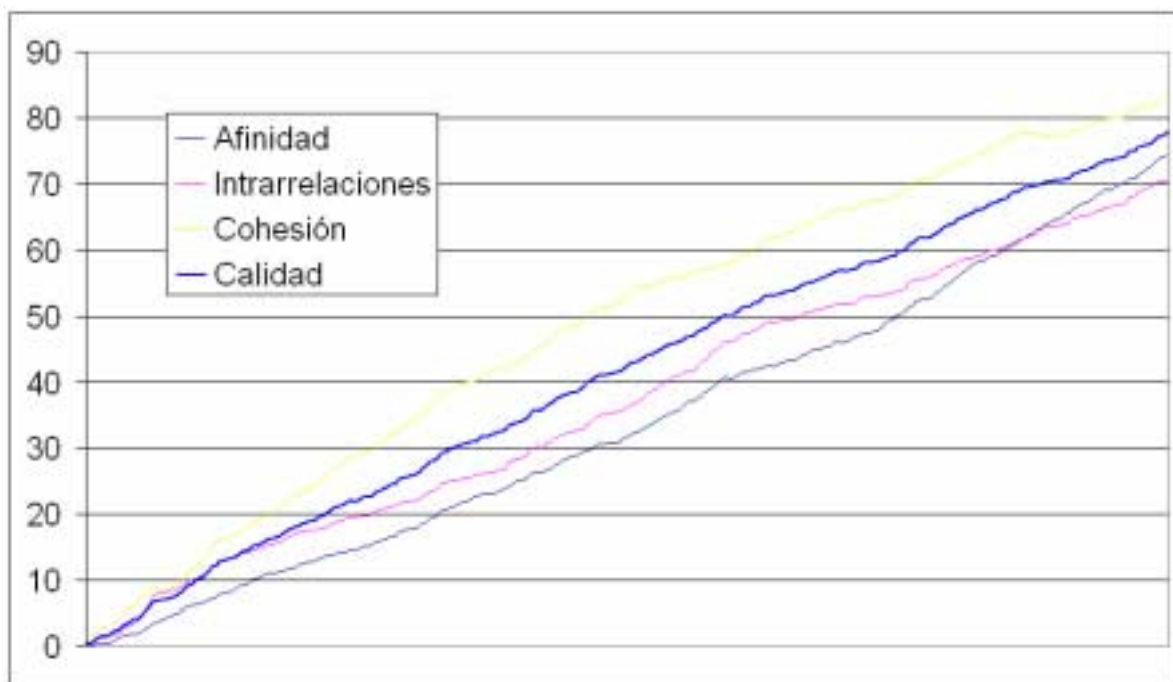


Figura 5.5 Evolución de los parámetros de calidad.

En la gráfica se puede ver como en un principio con todos los vértices separados los parámetros de calidad son 0. A medida que vamos uniendo los autores van subiendo todos.

5.3.2.3 Partición Forzada

5.3.2.3.1 Idea

A diferencia del algoritmo anterior este no va agrupando autores sino que parte de un único grupo con todos los autores posibles, un grafo conexo, y lo va partiendo en grupos más pequeños. Se trata de un algoritmo particionador y no fusionador según la clasificación de la Tabla 5.3.

La gracia del algoritmo está en encontrar los puntos por donde partir el grupo de autores. Para ello utiliza una representación alternativa del grafo mediante un árbol de expansión mínima [4]. Este árbol no es más que un grafo inducido del grafo original que contiene todos los vértices y el mínimo número de aristas con el mínimo peso total posible que los mantenga conectados. Para encontrar este árbol se ha utilizado el algoritmo de Kruskal [4].

Hay que tener en cuenta que la distancia entre vértices en nuestro grafo es inversamente proporcional al peso de las aristas ya que estas representan el número de veces que dos autores han publicado juntos: cuanto más alto sea, más cercanos, cuanto más bajo, más distantes. Por lo tanto creamos más que un árbol de expansión minimal, un árbol de expansión maximal.

Nuestro árbol de expansión maximal se puede interpretar como el esqueleto del grafo, un resumen en forma de árbol, sin ciclos, que mantiene los vértices conectados por los puntos de conexión más fuertes, aquellos en los que los dos vértices están más próximos. Al ser un árbol cada arista es puente y si la quitamos partiremos el árbol en dos componentes; si quitamos 2 tendremos 3 componentes, si quitamos n tendremos $n+1$ componentes. Cada arista del árbol contiene el coste o la distancia entre los vértices a los que une. Interpretaremos este peso como la distancia entre los dos conjuntos que parte. Nuestro algoritmo partirá el grafo por aquellas aristas del árbol de expansión minimal que estén más distantes (o mejor dicho menos juntas), minimizando el impacto de los cortes.

Otra característica de este algoritmo es que hay que decidir el número de cortes que se desean hacer, a diferencia de la agrupación por vecindad en donde el número de grupos salía de un modo natural. Para decidir este valor, al que llamaremos k , se utiliza el parámetro de la configuración *MaxAutoresGrupo* que, a nivel indicativo, marca el máximo número de

elementos que debería tener un grupo correcto. Cuando se procede a partir un conjunto se ajusta el número de particiones como si tuviera que partir el conjunto con subconjuntos de *MaxAutoresGrupo* elementos:

$$k = \frac{n}{MaxAutoresGrupo} \quad (5.5)$$

Se puede deducir de esta fórmula que nunca se intentará partir un conjunto con menos de *MaxAutoresGrupo* elementos ya que se considerará de tamaño suficientemente pequeño. Aunque lo ideal sería que el algoritmo partiera el conjunto en k partes iguales, la verdad es que no lo hace y, como ha pasado anteriormente, la mayor parte de los autores van a parar a un mismo grupo. Por esta razón se ha permitido que el algoritmo se aplique de manera recursiva sobre aquellos grupos mayoritarios que no han quedado suficientemente partidos. Hay un parámetro en la configuración *AplicarRestriccionTamañoMaximo* que junto con *TamañoMaximo* indican si se recurrirá a partir los grupos resultantes que superen el tamaño máximo especificado. Esto se hará recursivamente hasta que no quede ningún grupo que supere el tamaño especificado.

Las primeras pruebas que se han realizado con este algoritmo han demostrado que al intentar partir el conjunto por el punto de conexión más débil lo hacía con aristas más bien periféricas en el árbol que sólo separaban unos pocos autores del grupo principal. Si aplicábamos la opción de seguir partiendo el grupo principal recursivamente seguía pasando lo mismo: el grupo más que partirse significativamente seguía puliendo las relaciones débiles circundantes hasta que no superaba el máximo especificado. El resultado final era un grupo principal muy grande, o como mínimo hasta el lindar permitido, y un sinnúmero de grupitos satélite, muchos de ellos con uno o dos elementos.

De esta primera experiencia se deduce que para que una partición tenga sentido debe hacerse con aristas que conecten dos grupos significativos, dicho de otra manera, la arista a partir debe estar bastante centrada en el árbol de expansión minimal. Se procedió entonces a calcular la centralidad de cada arista mediante un algoritmo de elaboración propia.

Def 5.5 Sea *centralidad* : $A \rightarrow \square$ de una arista que pertenece a un árbol como el número de vértices de la componente más pequeña de las dos que conecta.

Una arista que sea hoja tendrá una centralidad de 1, ya que desconecta un vértice de todo el grafo. Una arista que esté justo en el medio del grafo tendrá una centralidad de $\frac{n}{2}$ porque partirá el grafo justo por la mitad. Ahora aparte de tener en cuenta la fragilidad de la relación a eliminar también se tiene en cuenta que esta esté lo más centrada posible.

Sea *excentricidad* : $A \rightarrow \mathbb{R}$ de una arista el contrario de *centralidad*.

$$excentricidad(e) = n - centralidad(e) \quad (5.6)$$

El algoritmo calcula para cada arista del árbol de expansión un valor que representa la consistencia de esa arista, lo costoso que debe ser eliminarla. En este cálculo intervienen tanto el peso de la arista como su centralidad, o mejor dicho su excentricidad. Cuanto más peso tenga y más excéntrica sea más difícil es que se elija para ser eliminada. Para hacer este cálculo utiliza los valores de la configuración: *FactorExcentricidad*, *ExpExcentricidad*, *FactorPeso* y *ExpPeso* para calcular la siguiente fórmula para cada arista:

$$consistencia(e) = \left(FactorPeso \cdot \frac{\log \omega(e)}{\log max_peso} \right)^{ExpPeso} \cdot \left(FactorExcentricidad \cdot \frac{\log excentricidad(e)}{\log max_excentricidad} \right)^{ExpExcentricidad} \quad (5.7)$$

Este cálculo se realiza por todas las aristas y se ordenan mediante un algoritmo Counting Sort [4] en tiempo lineal. Se utilizan exponentes para exagerar las diferencias de consistencia, ya que de lo contrario el algoritmo seguía con el empeño de eliminar solo las aristas circundantes y no partía el grupo de forma significativa. Los factores sirven para acotar los valores y que el resultado no se dispare. Un detalle importante de implementación es que estos valores de consistencia no tendrían que ser demasiado altos. En la configuración por defecto los factores son 10 y los exponentes 2 y 3 respectivamente, con lo cual el valor máximo de consistencia es 100000. Nunca debería pasarse de este orden de valores, ya que de lo contrario el algoritmo de ordenación Counting Sort no funcionaría correctamente. Una vez calculados los valores y ordenados se cogen los k más pequeños y se eliminan las aristas en el árbol partiendo el conjunto. Haciéndolo de este modo existe el peligro de que se elija una arista que esté muy al centro pero que tenga un peso considerable. En este caso el algoritmo partiría la relación entre dos autores que por su grado de cocitación deberían estar juntos. Para

evitar este tipo de situaciones existe un parámetro *MaximoPesoCorte* que especifica el máximo peso permitido en las aristas por donde cortar el grafo.

Utilizando este tipo de cálculos el algoritmo empieza a funcionar mejor. Aunque se forman grupos de tamaño considerable, el conjunto logra partirse, que era nuestro objetivo. Además hay que recordar que existe la posibilidad de aplicar recursivamente el algoritmo con los grupos que han quedado demasiado grandes. Esta opción recursiva no siempre funciona bien porque precisamente las partes que pretende partir son los grupos más difíciles y no logra su objetivo. Por esta razón se ha creado el parámetro de la configuración *MaxLlamadas* que indica el número máximo de llamadas recursivas que se permitirán para partir los grupos grandes. Una vez alcanzado este tope de llamadas se dejará el grupo tal y como está. Si no se quiere limitar el número de llamadas basta con ponerlo a -1 .

5.3.2.3.2 Parámetros

- *MaxAutoresGrupo*: a nivel indicativo, marca el máximo número de elementos que debería tener un grupo correcto.
- *AplicarRestriccionTamañoMaximo*: indica si se recurrirá a partir los grupos resultantes que superen el tamaño máximo especificado.
- *TamañoMaximo*: especifica el tamaño máximo de los grupos a partir del cual deberán ser partidos otra vez.
- *MaxLlamadas*: indica el número máximo de llamadas recursivas que se permitirán para partir los grupos que superen el tamaño máximo.
- *FactorExcentricidad*, *ExpExcentricidad*, *FactorPeso* y *ExpPeso*: factores y exponentes que intervienen en la fórmula de cálculo de consistencia de una arista.
- *MaximoPesoCorte*: especifica el máximo peso permitido en las aristas por donde cortar el grafo.

5.3.2.3.3 Pseudocódigo

El código es:

procedimiento ParticionForzada

entrada grafo

inicio

```

v_aristas:=CrearVectorAristas(grafo);            $O(m)$ 
Ordenar(v_aristas);                              $O(m)$ 
    ejecutamos el algoritmo de Kruskal que nos devuelve el árbol de expansión maximal
arbol=Kruskal(grafo,v_aristas);                  $O(m \log m)$ 

```

```

k=n/TamanoIdeal;
Creamos un vector de aristas con la consistencia y lo ordenamos crecientemente.
v_aristas:=VectorConsistenciaAristas(arbol);
Ordenar(v_aristas);

eliminadas:=0;arista:=0;
mientras eliminadas<k hacer O(k)
    si  $\omega(v\_aristas[arista]) \leq \text{MaximoPesoCorte}$  entonces
        EliminarArista(arbol, arista);
        eliminadas++;
    fsi
    arista++;
fmientras
Creamos el subgrafo correspondiente a quedarse con las aristas que esten en un mismo grupo del árbol.
grafo2:=CrearGrafoPartido(arbol,grafo); O(m)
retorna CrearParticion(grafo2);
fin

```

El coste del algoritmo es básicamente el coste de calcular el árbol de expansión minimal mediante el algoritmo de Kruskal [4] de $O(m \cdot \log m)$. Los vectores de aristas dan unos costes $O(m)$ tanto en la creación como en la ordenación mediante Counting Sort [4]. La creación del vector de consistencia de aristas también es lineal ya que el algoritmo que calcula la centralidad de las aristas está basado en un recorrido en profundidad $O(m)$.

5.3.2.3.4 Resultados

Grupos	Tam.Medio	Tam.Máximo	Tiempo	Calidad
19	18.65	73	1.534	66,86

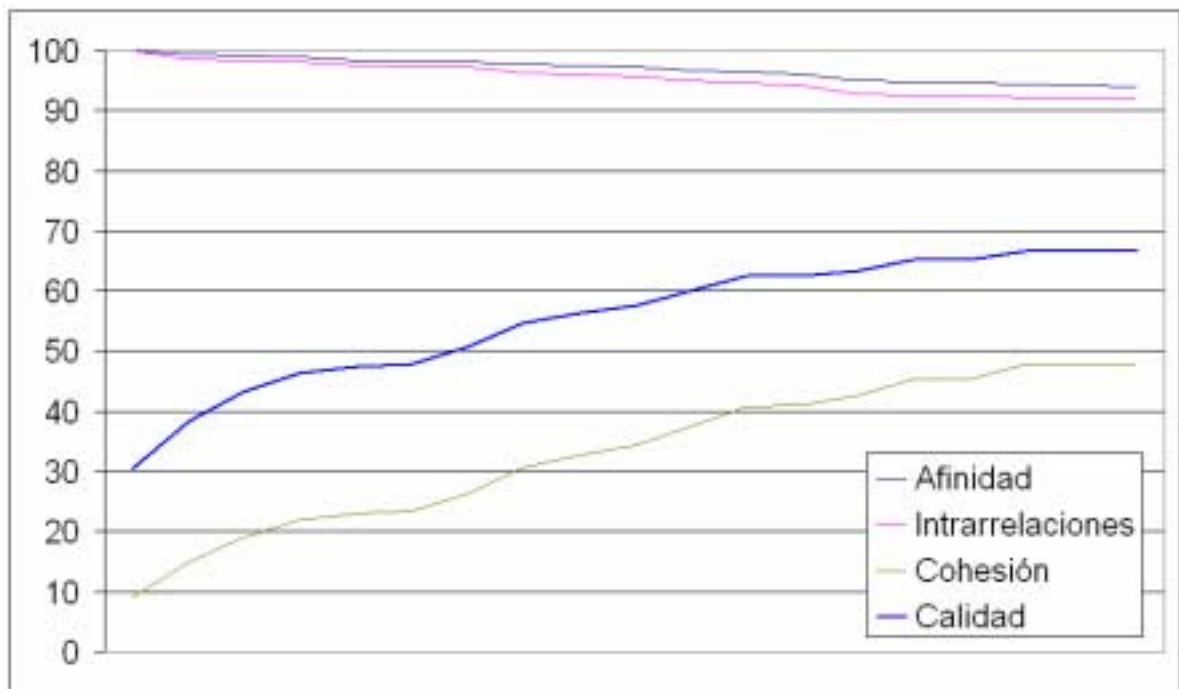


Figura 5.6

En la gráfica se puede observar como al iniciar la partición con un solo grupo el valor de cohesión es muy bajo y, en cambio los valores de afinidad e intrarrelaciones son máximos porque todos los autores están dentro del grupo. A medida que el algoritmo va partiendo el conjunto los valores de afinidad y intrarrelaciones van bajando pero el de cohesión sube considerablemente lo que hace subir a la calidad.

5.3.2.4 Partición Natural

5.3.2.4.1 Idea

Se trata de una variante del algoritmo anterior en la cual no se utiliza una partición forzada a un número determinado de cortes sino que se cortan las aristas que lo merecen de un modo natural. El algoritmo simplemente recorre todas las aristas del árbol de expansión y aquellas que no cumplan una determinada condición de permanencia son eliminadas. Tal y como pasaba anteriormente cada eliminación crea un nuevo grupo. La condición de permanencia de una arista viene determinada mediante los parámetros *MaximoCentralidad* y *MinimoPeso*. Toda aquella arista que no los cumpla será quitada del árbol generando un corte.

Esta versión del algoritmo parte el conjunto por puntos en los que seguro que merece la pena, en este sentido parece más seguro que la versión de Partición Forzada. Quizá no sea del

todo efectivo porque en las particiones que hemos probado quedan algunos grupos mayoritarios y no se puede repetir el algoritmo porque no tiene sentido volver a aplicar un filtraje sobre un conjunto de aristas que ya han pasado ese filtraje, no va a eliminar ninguna arista.

Un refinamiento que se ha hecho para determinar la condición de eliminación de las aristas ha sido no sólo el incumplimiento de unos límites de los parámetros de *centralidad* y de *peso* sino que le hemos añadido un *CoficienteCentralidadPeso*:

$$\text{CoficienteCentralidadPeso}(e) = \frac{\text{centralidad}(e)}{\omega(e)} \quad (5.8)$$

y exigiremos que las aristas del árbol que quieran permanecer en el no superen un máximo configurable *CoficienteCentralidadPesoMáximo*.

5.3.2.4.2 Parámetros

- *MaximoCentralidad*: máxima centralidad mostrada por una arista para que no pueda ser eliminada. Todas las aristas que superen este valor son candidatas a ser eliminadas.
- *MinimoPeso*: mínimo peso que garantiza que una arista no sea eliminada. Cualquier arista que baje de este peso es candidata a ser cortada.
- *CoficienteCentralidadPesaMáximo*: valor máximo del cociente entre centralidad y peso que va a aguantar una arista para no ser eliminada. De las aristas candidatas a ser eliminadas todas aquellas que superen este coeficiente van a ser eliminadas definitivamente.

5.3.2.4.3 Pseudocódigo

El código es parecido al de su hermano *ParticiónForzada*:

procedimiento ParticionNatural

entrada grafo

inicio

```
v_aristas:=CrearVectorAristas(grafo);           O(m)
Ordenar(v_aristas);                             O(m)
ejecutamos el algoritmo de Kruskal que nos devuelve el árbol de expansión maximal
arbol=Kruskal(grafo,v_aristas);                 O(m log m)
```

k=n/TamanoIdeal;

Creamos un vector de aristas con la consistencia y lo ordenamos crecientemente.

```
v_aristas:=VectorConsistenciaAristas(arbol);
```

```
Ordenar(v_aristas);
```

```
eliminadas:=0;arista:=0;
```

```

para cada arista de Aristas(arbol) hacer  $O(m)$ 
  si centralidad(arista)  $\geq$  MaximoCentralidad Y  $\omega$ (arista)  $\leq$  MinimoPeso entonces
    si centralidad(arista)/ $\omega$ (arista)  $\geq$  CoeficienteCentralidadPesoMaximo) entonces
      EliminarArista(arbol, arista);
    fsi
  fsi
fmientras
  Creamos el subgrafo correspondiente a quedarse con las aristas que esten en un mismo grupo del árbol.
  grafo2:=CrearGrafoPartido(arbol,grafo);  $O(m)$ 
retorna CrearParticion(grafo2);
fin

```

El coste del algoritmo es básicamente el coste de calcular el árbol de expansión minimal mediante el algoritmo de Kruskal [4] de $O(m \cdot \log m)$ de la misma manera que la Partición Forzada.

5.3.2.4.4 Resultados

Grupos	Tam.Medio	Tam.Máximo	Tiempo	Calidad
18	19,63	69	1,094	67,15

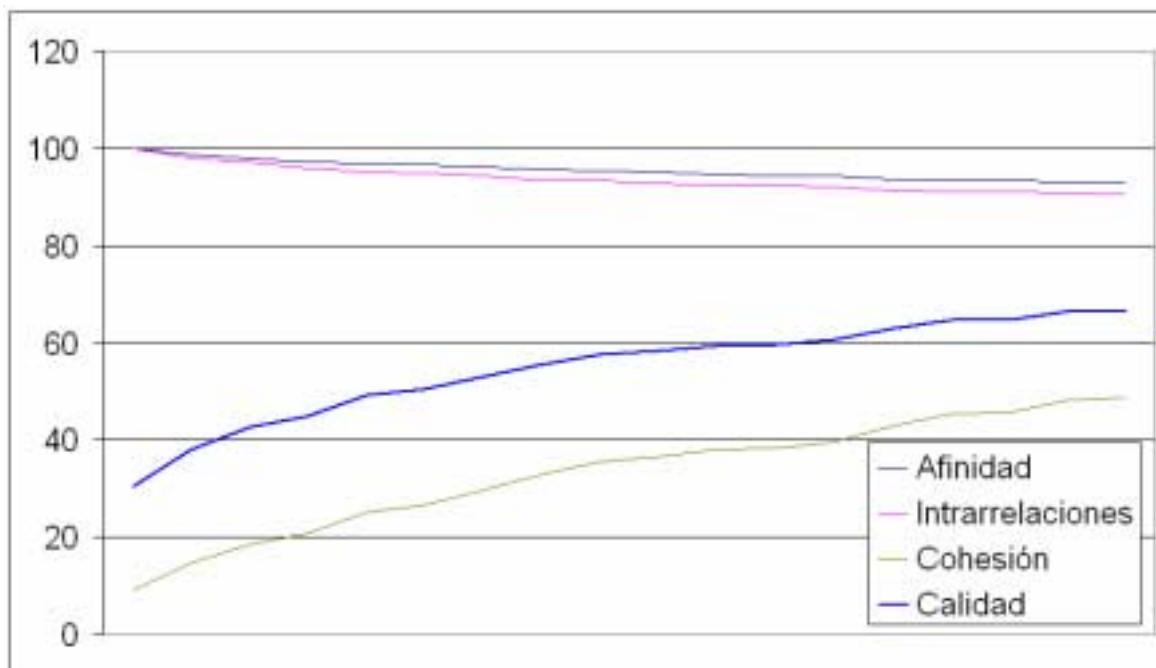


Figura 5.7

El comportamiento de la gráfica es el mismo que en la partición forzada.

5.3.2.5 Divide y vencerás

5.3.2.5.1 Idea

Este método sigue el esquema algorítmico de divide and conquer o divide y vencerás. La idea consiste en dividir el problema en varios subproblemas más pequeños, aplicar una solución básica a cada uno de ellos y proceder a juntar esas soluciones. Este último paso es el más importante de esta metodología. Centrándonos en nuestro problema tenemos que dividir el espacio de búsqueda de los autores en partes más pequeñas, formar en esas los grupos que sean necesarios y juntar las partes fusionando aquellos grupos que se consideren suficientemente próximos.

Partimos con la característica de que los autores están enumerados en el grafo principal lo que nos concierne la posibilidad de dividir el espacio de búsqueda en intervalos de esta enumeración. Recursivamente se puede ir partiendo el vector que contiene los vértices por la mitad hasta llegar a un tamaño suficientemente pequeño. Ahora tenemos un conjunto muy reducido donde encontrar los grupos de autores. Simplemente asociamos esos autores que tengan una arista en el grafo con el algoritmo de componentes conexas y formamos así los grupitos iniciales. A la vuelta al punto donde se hizo cada partición se analizan las asociaciones entre los grupos de cada parte del vector a unir. Si se considera que hay suficientes aristas que unen dos grupos de distintas particiones se procede a hacer dicha unión.

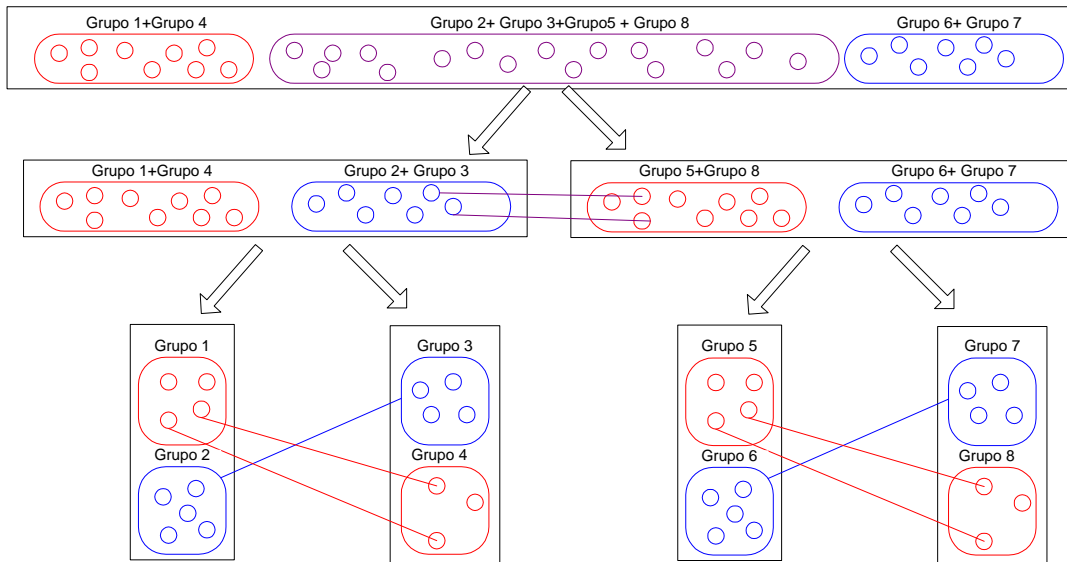


Figura 5.8 Esquema visual del algoritmo Divide dónde se puede observar como se ha dividido el espacio de búsqueda para luego rejuntrar los grupos afines.

El algoritmo parte el conjunto hasta que el su tamaño sea igual o más pequeño que *MinimoDivisión*. De aquí saca los grupos con la búsqueda de componentes conexas. Para juntar dos grupos de distintas partes se realiza el siguiente cálculo: por un lado se suman los pesos de las aristas que tienen los dos grupos, por otro se suman los pesos de todas las aristas que unen ambos grupos (equivalente a las interrelaciones si lo comparamos con la metodología de cálculo de la función de calidad) y si esta suma es igual a un cierto porcentaje de la primera significa que la unión de los grupos es suficientemente fuerte y se juntan.

Definimos unos límites de interrelaciones entre grupos de distintas particiones con los parámetros *PorcentajeFusionMinimo* y *PorcentajeFusionMaximo*. En las particiones más pequeñas nos interesa ser más estrictos a la hora de admitir la unión de dos grupos porque al tratar con pocos autores, cuando unos pocos se asocian en seguida se unen los grupos. En cambio, cuando se tratan muchos autores y los grupos son más grandes cuesta más que se unan. Por esta razón se dan dos porcentajes de fusión, un máximo que servirá para los casos de unión de los grupos más pequeños y el porcentaje mínimo que sirve para hacer la unión final de los grupos de las dos partes del vector. Este porcentaje se va modificando proporcionalmente en los casos intermedios.

5.3.2.5.2 Pseudocódigo

El pseudocódigo de este algoritmo tiene su complejidad y, además, esconde muchos detalles que deben de tratarse en la implementación que hacen de este uno de los algoritmos más rebuscados.

procedimiento DivideYVenceras2

entrada grafo, desde, hasta, peso

inicio

elementos:=hasta-desde+1;

si elementos<=MinimoElementosDivision)

Caso base:recojemos el grafo que resulta de juntar todo el trozo que tratamos.

subgrafo1=SubgrafoRango(grafo,desde,hasta);

sino

Llamadas recursivas que buscan los grupos en cada parte.

subgrafo1=DivideYVenceras2(grafo,desde,(desde+hasta)/2,peso+IncrementoPesoFusion);

subgrafo2=DivideYVenceras2(grafo,(desde+hasta)/2+1,hasta,peso+IncrementoPesoFusion);

Tenemos un metagrafo cuyos vértices son los subgrafos de ambas particiones.

metagrafo=Nuevo Grafo;

Tenemos un grafo con los vértices de ambos lados y sólo con las aristas que cruzan de un lado al otro.

bipartido=Nuevo Grafo;

Rellenamos estos dos grafos recorriendo los vértices y las aristas de cada lado.

RecorrerVerticesSubgrafo(subgrafo1,metagrafo,bipartido);

RecorrerVerticesSubgrafo(subgrafo2, metagrafo,bipartido);

RecorrerAristasGrafo(grafo,subgrafo1,subgrafo2,desde,hasta,metagrafo,bipartido);

Vamos a filtrar las aristas del metagrafo que no cumplan los porcentajes mínimos.

FiltrarMetagrafo(metagrafo, ,peso);

Calculamos las metacomponentes entre grafos

CalcularComponentes(metagrafo);

Juntamos los grafos:

subgrafo1->AbsorverGrafo(subgrafo2);

FusionarGrupos(subgrafo1,desde,hasta,metagrafo,bipartido);

fsi

Actualizando componentes del subgrafo a devolver

CalcularComponentes(subgrafo1);

retorna subgrafo1;

fin

procedimiento DivideYVenceras

entrada grafo

inicio

grafo2=DivideYVenceras2(grafo,0,n-1,PorcentajeFusionMinimo);

retorna CrearParticion(grafo2);

fin

Para evaluar el coste de este algoritmo debemos utilizar el *master theorem* definido en [4] que sirve para analizar los costes de algoritmos recursivos. Tenemos un espacio de búsqueda que se divide en 2 partes cada vez de tamaños $n/2$ ($a=2$ y $b=2$). En cada llamada recursiva recorreremos de distintas maneras los diferentes subgrafos generados por cada subproblema $O(n) \leq O(m) \leq O(n^2)$. Como el exponente que define el coste de cada llamada

recursiva va entre 1 y 2 (tendiendo sobretodo al 1) se puede considerar igual que $\log_b a \log_2 2$ que es 1: seguimos el segundo caso del master theorem y afirmamos que el algoritmo tiene coste $O(m \log m)$.

5.3.2.5.3 Resultados

Grupos	Tam.Medio	Tam.Máximo	Tiempo	Calidad
167	2,22	24	1,852	54,75

5.3.2.6 Filtro Bipartición

5.3.2.6.1 Idea

Como el propio nombre indica esta metodología propone hacer un filtraje del grafo inicial. De esta criba deben salir diferentes componentes conexas que representarán los grupos de investigación.

El elemento a filtrar son las aristas. El factor sobre el que se filtrarán las aristas es lo que denominaremos coeficiente de bipartición.

Sea $biparticion: E \rightarrow [0..1]$ de una arista un valor que nos indica la proporción de aristas adyacentes a los vértices que componen la arista tratada que no se relacionan con ningún vértice que a la vez sea adyacente al otro vértice componente de la arista (las aristas que no forman triángulos con la propia arista).

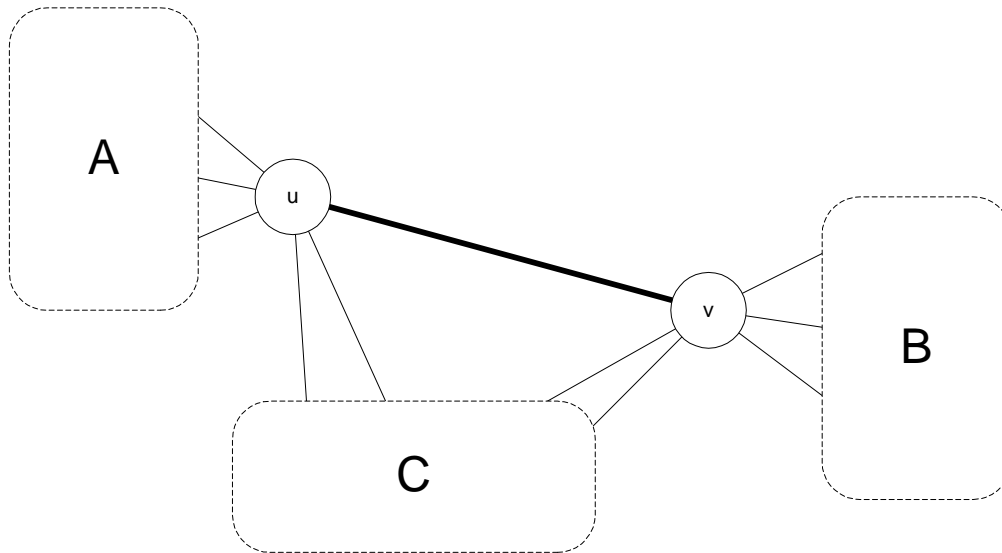


Figura 5.9 Representación de las distintas modalidades de aristas adyacentes a los vértices de la arista que se analiza. Los adyacentes a los conjuntos A y B favorecen a la bipartición porque no forman triángulos entre los extremos de la arista. Los adyacentes a C son los que forman los triángulos con u y con v.

Basándonos en la distribución de adyacencias mostrada por la Figura 5.9 calcularemos este coeficiente según:

$$biparticion(u, v) = \frac{\sum_{x \in A} \omega(u, x) + \sum_{x \in B} \omega(u, x)}{\sum_{x \in A} \omega(u, x) + \sum_{x \in B} \omega(u, x) + \sum_{x \in C} \omega(u, x)} \quad (5.9)$$

Una vez calculado el coeficiente para cada arista filtraremos aquellas aristas que tengan un valor más grande que un máximo especificado por el parámetro *MaximoBipartición*. Del grafo resultante se calcularán las componentes conexas y esas serán los grupos resultantes de este algoritmo.

5.3.2.6.2 Parámetros

- *BiparticionMaxima*: Coeficiente máximo a partir del cual se filtrarán las aristas.

5.3.2.6.3 Pseudocódigo

Se trata de hacer un nuevo grafo copia del de entrada filtrando aquellas aristas con coeficiente de bipartición demasiado alto.

procedimiento FiltroBiparticion

entrada grafo

inicio

CalcularCoeficienteBiparticion(grafo);

grafo2=Nuevo Grafo;

para cada vertice **de** grafo **hacer**

grafo2->Insertar(vertice);

fpara

para cada arista **de** grafo **hacer**

si Biparticion(arista)≤BiparticionMaxima **entonces**

grafo2->InsertarArista(arista);

fsi

fpara

retorna CrearParticion(grafo2);

fin

5.3.2.6.4 Resultados

Grupos	Tam.Medio	Tam.Máximo	Tiempo	Calidad
44	8,48	103	2,432	72,32

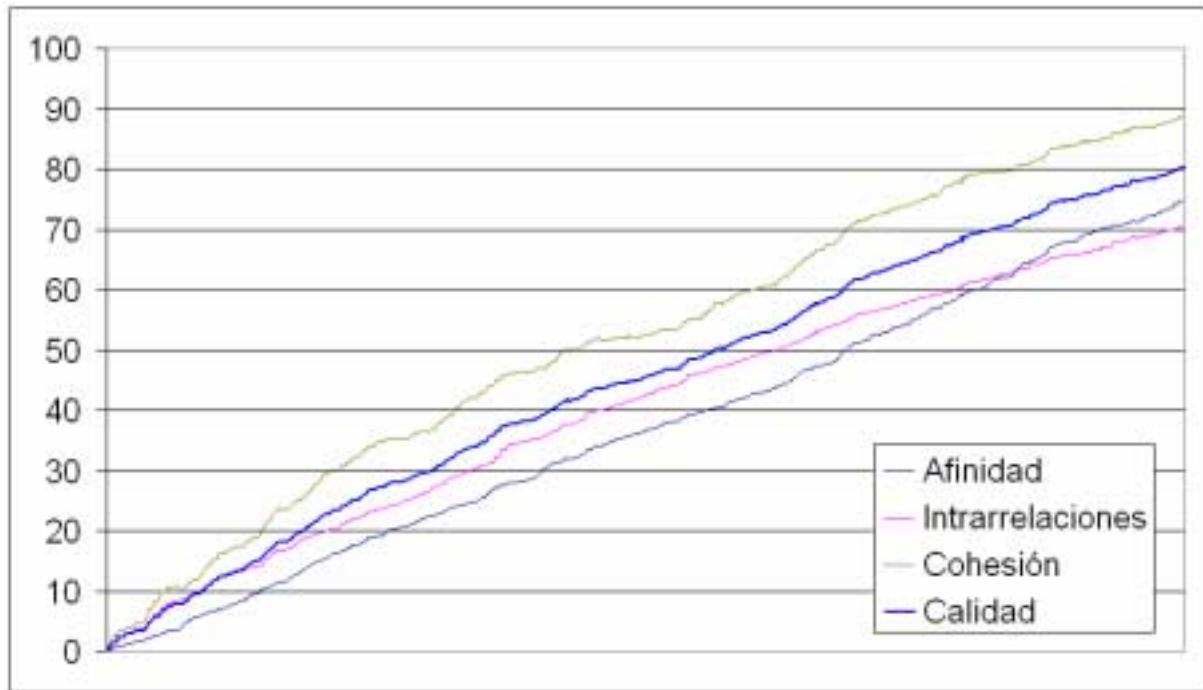


Figura 5.10

5.3.2.7 Algoritmo de Clustering

5.3.2.7.1 Idea

Si nuestro problema consiste en encontrar agrupaciones de elementos perdidos en una colección de datos encontramos en la literatura científica gran cantidad de trabajos que discuten estos temas. Se trata de técnicas estadísticas de análisis de conglomerados o como se conoce en inglés *Clustering Analysis*.

Nos fijaremos en una serie de algoritmos descritos por Kaufman y Rousseeuw en su trabajo *Finding Groups in data* [5]. En este trabajo se proponen distintos algoritmos de clustering que pretenden encontrar agrupaciones perfectas de elementos. Se hace hincapié en lo de perfectas porque se trata de algoritmos que minimizan funciones de coste de soluciones, no en aproximaciones como nosotros hemos tratado hasta ahora.

En el análisis de conglomerados disponemos de n elementos que deben de ser agrupados (de momento nos vale la entrada de cualquiera de nuestros problemas). Debemos disponer de una matriz $n \times n$ donde guardaremos la disimilaridad que hay entre todo par de elementos y definiremos:

Def 5.6 Sea $d : A \times A \rightarrow \mathbb{R}$ una función que devuelve la disimilaridad entre un par de elementos del conjunto.

Dividimos los algoritmos de clustering en dos variedades:

Clustering particionador

Se les pasa un parámetro k de entrada y dividen el conjunto de elementos en k grupos minimizando el cruce de disimilaridades entre elementos de distintos grupos

Clustering jerárquico

Empiezan con una partición con todos los elementos en un mismo grupo di van realizando particiones del grupo más poco consistente de una forma iterativa hasta llegar al punto que cada elemento forma un grupo por si solo. También existe el caso contrario: empezamos con los elementos separados siendo cada uno un grupo y los vamos fusionando eligiendo cada vez la pareja más afín hasta llegar a que todos pertenezcan al mismo grupo.

Podríamos utilizar la primera clase de algoritmos en los que se especifica a priori el número de grupos a encontrar de un modo similar al que hacíamos en la ParticionForzada, calculando cuantos grupos deberían salir. No se ha entrado a probar este tipo de técnicas de momento porque tienen un coste muy grande. Es verdad que nos devuelven una partición ‘perfecta’, pero el hecho de que tengamos que decir a priori el número de grupos anula la perfección. Nosotros no sabemos cuantos grupos vamos a encontrar.

Cuando no sabemos el total de grupos va bien utilizar los métodos jerárquicos. Aun así la dificultad es grande, porque estos métodos proponen n soluciones con 1,2, ... , n grupos en cada solución, con lo que al final también debemos elegir indirectamente el número de grupos.

El algoritmo de clustering que aquí propondremos es de la clase jerárquica y el método es fusionador: empieza con todos los elementos separados y acaba con todos juntos. Está basado en el algoritmo AGNES (Agglomerative Nestings) [5]. Para que el método sea efectivo debemos responder a dos cuestiones clave:

- *¿Cómo calculamos la disimilaridad?*
- *¿Cuándo dejamos de fusionar?*

¿Cómo calculamos la disimilaridad?

Nuestra entrada del problema está codificada mediante un grafo. Podemos aproximar la disimilaridad que hay entre un par de autores como la distancia que tienen en el grafo de coautorías. La distancia entre dos vértices de un grafo viene dada por la suma de los costes del camino más corto que los une y se calcula con el algoritmo de Dijkstra [4].

Como ya se ha comentado en otros apartados nuestro grafo es especial porque el peso de las aristas no significa coste sino más bien beneficio. Si queremos ejecutar los algoritmos de caminos mínimos para calcular las distancias debemos convertir el peso de las aristas, el número de coautorías, a un valor que represente el coste:

Sea $coste: E \rightarrow \mathbb{R}$ una función que evalúa el coste de una arista dividiendo el peso máximo de todo el grafo por el peso de la arista.

$$coste(e) = \frac{max_peso}{\omega(e)} \quad (5.10)$$

Así la arista de mayor peso tendrá coste 1 y las demás tendrán coste ≥ 1 .

Con esta inversión de los pesos de las aristas ya podemos usar el algoritmo convencional de Dijkstra para que nos devuelva la distancia de un vértice a todos los demás. Si aplicamos Dijkstra a cada vértice obtenemos la distancia entre todo par de vértices, tenemos ya la matriz de disimilaridad.

Def 5.7 Sea $d : A \times A \rightarrow \mathbb{R}$ la disimilaridad entre un par de vértices calculada mediante el algoritmo de Dijkstra.

A medida que vamos fusionando elementos cada grupo pasa a ser considerado como un elemento unitario, por lo tanto se puede considerar la disimilaridad entre él y los demás grupos como si fueran vértices. Al fusionar dos grupos debemos recalcular la disimilaridad entre el nuevo grupo y los demás de la siguiente manera:

Def 5.8 Sea C el cluster resultante de la fusión de los clusters A y B:

$$|C| = |A| + |B| \quad (5.11)$$

$$\forall_{D \neq A \neq B} : d(C, D) = \frac{d(A, D) \cdot |A| + d(B, D) \cdot |B|}{|A| + |B|} \quad (5.12)$$

¿Cuándo dejamos de fusionar?

El algoritmo AGNES va fusionando hasta que todos los vértices están en un mismo grupo. Nos interesa parar en un cierto momento para devolver una partición con sentido.

Ejecutamos el algoritmo hasta el final, hasta que se juntan todos los elementos. Guardamos las progresivas fusiones realizadas por el algoritmo en un árbol binario. En este árbol cada nodo representa una fusión de los clusters que se han generado con los subárboles hijos. Guardaremos en el coste que ha supuesto la fusión que no es otro que la disimilaridad entre los grupos que hemos unido. Guardaremos también en cada nodo un elemento que significará el peso del cluster.

Def 5.9 Definimos como $peso_cluster : GP \rightarrow \mathbb{R}$ como una función que lleva la media del coste de las fusiones que han generado el cluster:

$$\text{peso_cluster}(A) = \text{coste_fusion}(A, X) \wedge |A| = 1 \wedge X : \text{cluster mas afin} \quad (5.13)$$

$$\text{peso_cluster}(\{A \cup B\}) = \frac{\text{peso_cluster}(A) \cdot |A| + \text{peso_cluster}(B) \cdot |B|}{|A| + |B|} \quad (5.14)$$

Al final recorreremos el árbol descendientemente empezando por la última fusión que ha juntado toda la población. En cada nodo comparamos el coste de la fusión con el peso de los clusters unidos. Si el coste de la fusión es demasiado grande comparado con el poco peso que tienen los grupos se considera que la unión ha sido forzada y por tanto no se ejecuta. Aquí nos paramos y devolvemos la configuración encontrada.

5.3.2.7.2 Pseudocódigo

```

1.  procedimiento AGNES
2.    entrada grafo
3.    inicio
4.    d[n*n]:=CargarDisimilaridades(grafo); Ejecuta el algoritmo de Dijkstra para cada vértice.
5.    árbol:=Nuevo árbol; Creamos el árbol de fusiones.
6.    Añadimos cada elemento como un árbol con una hoja.
7.    para cada objeto hacer
8.      árbol->InsertarHoja(objeto, Coste =0, Peso=0, Card =1)
9.    fpara
10.   NumClusters:=n;
11.   mientras NumClusters>1 hacer
12.     (objeto1,objeto2):=DisimilaridadMinima(); Devuelve el par de clusters mas cercanos
13.     (objeto,coste_fusion):=Fusionar(objeto1,objeto2); Realiza la fusión y devuelve el coste de esta.
14.     peso1:=ActualizarPeso(objeto1); Actualiza en el peso de los dos hijos fusionado con el coste de la
15.     fusiones.
16.     peso2:=ActualizarPeso(objeto2);
17.     árbol-
18.       >Plantar(objeto1,objeto2,Coste=coste_fusion,Card=|objeto1|+|objeto2|,Peso=(peso1*|objeto1|
19.       |+ peso2*|objeto2|/(|objeto1|+|objeto2|)));
20.     NumClusters--;
21.   fmientras
22.   retorna RecorrerArbol(árbol);
23. fin
24. funcion RecorrerArbol
25.   entrada árbol
26.   inicio
27.   si árbol->Raiz->EsHoja entonces
28.     retorna raiz;
29.   fsi
30.   (objeto1,objeto2,coste,tamaño,peso):=árbol->Raiz;
31.   si coste ≥ peso[objeto1] entonces Fusión válida, seguimos explorando
32.     particion1:=RecorrerArbol(árbol->HijoIzquierdo);
33.   else
34.     particion1:=ListarHojas; Fusión no válida. Nos quedamos aquí. Todo el hijo izquierdo es un grupo,
35.     listamos sus hojas para crearlo.
36.   fsi
37.   si coste ≥ peso[objeto2] entonces Fusión válida, seguimos explorando
38.     particion2:=RecorrerArbol(arbol->HijoDerecho);
39.   else
40.     particion2:=ListarHojas; Fusión no válida. Nos quedamos aquí. Todo el hijo derecho es un grupo, listamos
41.     sus hojas para crearlo.
42.   fsi
43.   retorna Fusionar(particion1,particion2);
44. fin

```

El coste del algoritmo se lo come literalmente la carga de las disimilaridades. Para hacerlo debemos ejecutar Dijkstra con coste $O(n^2)$ [4] un total de n veces dando un coste final cúbico $O(n^3)$!!

Tenemos un parámetro en este algoritmo que impide que se ejecute con particiones de más de *MaxElementos*, normalmente fijado entre 100 y 200.

5.3.2.7.3 Resultados

Grupos	Tam.Medio	Tam.Maximo	Tiempo	Calidad
7	46,625	244	80,14	50,9

5.3.3 Algoritmos de segunda fase

Son un conjunto de algoritmos que se ejecutan inmediatamente después de un algoritmo típico o de ‘primera fase’. Parten de los resultados que estos han generado e intentan mejorarlos. No son algoritmos que construyan soluciones a partir del grafo sino que alteran iterativamente soluciones existentes con la intención de optimizarlas.

Partimos de un resultado generado por un algoritmo convencional, una partición en la que cada elemento es asignado a un grupo. Podríamos decir que tenemos una propuesta de partición y que nuestro algoritmo se encargará de convertirla en definitiva, mejorándola si puede. A esta propuesta se le van haciendo pequeñas modificaciones, cambiar la asignación de un autor a otro grupo, que alteran su estado y se va evaluando la calidad del mismo. Al final de la ejecución nos quedamos con el estado de la partición con la máxima calidad.

Se trata de algoritmos que navegan por el espacio de soluciones del problema, en nuestro caso la asignación correcta de autores a grupos, buscando la solución de mayor calidad. Este espacio de soluciones suele ser de tamaños inabastables para recorrerlos exhaustivamente, por lo tanto el algoritmo ha de tener alguna técnica astuta que le guíe hasta la mejor solución. Aun así no se garantiza encontrar dicha solución óptima, simplemente se intenta encontrar a una de calidad parecida.

Cuando nos movemos por el espacio de soluciones avanzamos de la solución actual a una solución ‘vecina’, que no sea muy distinta a la actual. Repetimos el proceso hasta que consideremos que ya hemos recorrido suficiente espacio.

Este tipo de algoritmos se conocen como algoritmos *hill climbing* subidores de montañas. Consideremos el espacio de soluciones a un problema como un paisaje en el que tenemos distintas montañas. Cada punto de ese paisaje son las distintas soluciones y la altura en que nos encontremos representa la calidad de esa solución. Moverse por el espacio de soluciones es como andar por este espacio. Así cada monte representa un óptimo local, una solución que es mejor que todas sus vecinas. A esta solución localmente óptima se puede llegar de distintas maneras pero se cumple que ninguna solución vecina la mejora. Análogamente cuando estamos en un la cima de una montaña avancemos en la dirección que avancemos siempre bajaremos. Haber llegado a una solución que es superior a todas sus vecinas no significa que hayamos encontrado la solución óptima. Puede ser que si siguiéramos avanzando, aunque durante un rato bajaría la calidad de la solución encontrásemos otro camino que mejoraría aún mas la solución antes encontrada. Es como llegar andando a la cima de una montaña y bajarla para poder subir a otra aun mayor. Aunque comparemos la búsqueda como el hecho de andar por un paisaje de montes, faltaría añadir que evidentemente no tenemos la capacidad de ver a lo lejos dónde está la montaña más alta. Es como si fuéramos ciegos y lo único que podemos utilizar es un bastón que nos permite explorar el espacio que nos rodea (el vecindaje) eligiendo el punto que mas sube.

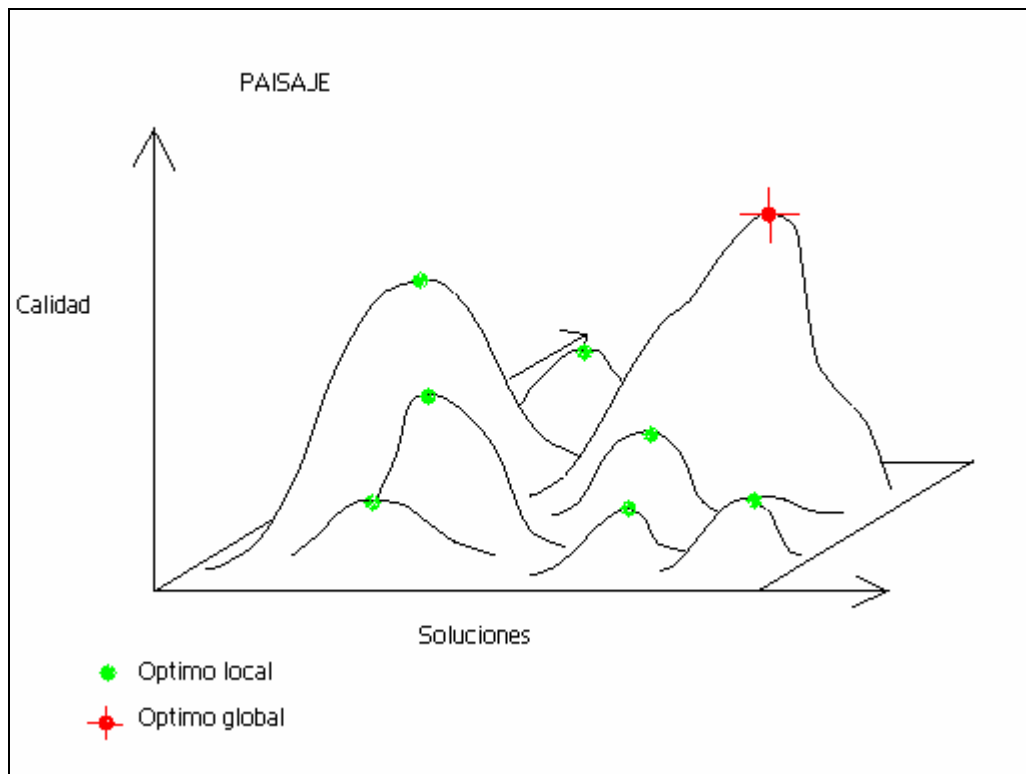


Figura 5.11 Paisaje de soluciones. Cada punto del plano representa una solución y su altura es la calidad. Como vemos existen varios puntos que son superiores a los que tienen alrededor, son óptimos locales, y sólo uno de ellos es el más alto, el óptimo global.

Para utilizar este tipo de técnicas hay que responder a estas preguntas que se nos plantean:

- ¿Qué son soluciones vecinas a la actual?
- ¿Cuándo consideramos acabada la búsqueda?
- ¿Cómo son básicamente los algoritmos?

¿Qué son soluciones vecinas a la actual?

La transición que hay para pasar de una partición a otra en el espacio de soluciones es cambiar de grupo a un elemento. Por lo tanto una solución vecina a la actual será toda aquella que tenga un elemento cambiado de sitio.

Def 5.10 Sea $PV : P \times P \rightarrow \text{booleano}$ una función que determina si las dos soluciones pasadas son vecinas. Entendiendo como una solución vecina aquella en la que se ha alterado la asignación de un elemento.

$$PV(\{f(a_1), \dots, f(a_i), \dots, f(a_n)\}, \{f(a_1), \dots, f'(a_i), \dots, f(a_n)\}) = (f(a_i) \neq f'(a_i)) \quad (5.15)$$

Def 5.11 Definiremos una transición o un movimiento $M : P \times A \times GP \rightarrow P$ como la modificación de la asignación de elemento indicado a el nuevo grupo.

$$M(\{f(a_1), \dots, f(b), \dots, f(a_n)\}, b, g) = \{f(a_1), \dots, g, \dots, f(a_n)\} \wedge f(b) \neq g \quad (5.16)$$

Def 5.12 Entendemos $MP : P \rightarrow M^+$ como el conjunto de movimientos posibles de una partición a todas aquellas particiones que son vecinas a P.

$$MP(P) = \{P' \mid PV(P, P')\} \quad (5.17)$$

En este caso el número de posibles movimientos a realizar se puede medir teniendo en cuenta que se puede mover cada uno de los n elementos a un grupo que no sea el suyo.

$$\text{movimientos_posibles} = n \cdot (r - 1) < n^2 \quad (5.18)$$

Para facilitar y acelerar los algoritmos hemos elegido un criterio que reduce este espacio de vecindad. Los movimientos de cualquiera de los n elementos sólo se podrán hacer a un grupo, a aquel que le es más propicio a mejorar la calidad total de la partición. Ahora es cuando utilizaremos esas funciones de aportación de calidad que habíamos definido en el apartado 4.3.1.3.

Def 5.13 Sea $\text{max_calidad} : A \rightarrow GP$ una función que dado un autor nos devuelve aquel grupo alternativo, al que no está asignado, que nos ofrece una mejor aportación de calidad.

$$\text{max_calidad}(a) = g \mid \forall_{g' \in VA(a) - \{g\}} \text{ap_calidad}(a, g') \leq \text{ap_calidad}(a, g) \quad (5.19)$$

Puede ser que un elemento no tenga alternativa y no sea candidato a moverse.

Mediante esta función escogeremos para un elemento el grupo alternativo con más calidad, con más certeza que al moverlo allí la función de calidad crecerá más que moverlo a otra de sus alternativas. Tenemos una nueva función de movimiento en la que no se especifica el grupo destino puesto que es implícitamente un movimiento a la alternativa más buena.

Def 5.14 Sea $MV : A \rightarrow P$ una nueva función de movimiento en el cual sólo se especifica el autor que quiere moverse ya que como destino se utilizará la función max calidad definida anteriormente en **Def 5.13**.

$$MV(a) = M(a, \text{max_calidad}(a)) \quad (5.20)$$

Restringiremos también el hecho de que se repitan movimientos. Para eso tenemos que guardarnos los movimientos hechos en un conjunto que irá creciendo conforme vaya avanzando el algoritmo.

Def 5.15 Sea $MH : P \rightarrow MV^*$ un conjunto en dónde guardaremos los movimientos hechos.

Establecidos todos los criterios redefiniremos el conjunto de movimientos posibles que serán las soluciones vecinas a la actual:

$$MP = \{MV(a) \mid a \in A \wedge MV(a) \notin MH\} \quad (5.21)$$

¿Cuándo consideramos acabada la búsqueda?

Esta cuestión es personalizada para cada algoritmo y será respuesta en las explicaciones de cada técnica utilizada. El máximo de movimientos que se pueden realizar son $n \cdot (r - 1)$ (sabiendo que no se pueden repetir) y, si tenemos en cuenta la posibilidad de que un elemento pueda crear por si solo un nuevo grupo, el conjunto de soluciones posibles sube hasta 2^n . Es evidente que ninguna técnica utilizada recorrerá exhaustivamente todas las soluciones. Algunas técnicas realizan movimientos hasta que no se detecta ninguna mejora, otras dejan que las cosas empeoren durante un rato para que luego mejore aproximándose al óptimo. Cada una de ellas utilizan distintos criterios para elegir un movimiento dentro del conjunto MP .

¿Cómo son básicamente los algoritmos?

Debemos considerar que para la implementación de estos algoritmos se hace un uso abusivo de las operaciones que ofrece el módulo PropuestaPartición que permiten mantener la partición o solución actual y su puntuación de calidad y permite recalcular incrementalmente estas funciones de manera rápida. Además mantiene información continua de los

movimientos que son recomendables hacer. Este módulo es la base de implementación de los algoritmos de segunda fase.

Cada algoritmo consistirá básicamente en un bucle que irá moviendo los elementos de partición hasta que se considera buena. Para ello hay que responder a estas preguntas:

- *¿Qué elemento elegimos para mover en cada paso?*
- *¿A qué grupo lo movemos?*
- *¿Cuándo acabamos la búsqueda?*
- *¿Qué resultado devolvemos?*

¿Qué elemento elegimos para mover en cada paso?

Utilizaremos la propiedad *ap_calidad* que nos ofrece un autor a un cierto grupo para saber cual es el elemento a mover en cada paso. En la estructura de datos PropuestaPartición guardaremos todos los elementos que tienen un grupo alternativo a donde ir en una cola de prioridad ordenada según el movimiento que nos aporta más calidad. Otro criterio es utilizar otra cola en la que el orden no es el movimiento mas bueno sino el elemento que está peor situado, el más débil, aquel en el que la aportación de calidad entre él y su grupo es menor.

¿A qué grupo lo movemos?

Para elegir el grupo al cuál moveremos los elementos estableceremos como criterio único moverlo al grupo alternativo con más aportación de calidad, elegiremos *max_calidad(a)*, siendo *a* el autor a mover. Para cada elemento guardaremos en una cola de prioridad Alternativas aquellos grupos *a* con los que mantiene coautoría ordenados por aportación de calidad. No admitiremos movimientos repetidos: si un elemento *a* en algún momento del algoritmo se ha desplazado al grupo *g*, *a* podrá moverse a cualquier grupo pero nunca podrá volver a *g*. Recordar que el módulo PropuestaPartición nos proporciona una estructura de datos MovimientosHechos donde se van guardando cada movimiento que hacemos, es la implementación del conjunto *MH*. De esta manera evitamos entrar en un bucle infinito de movimientos repetidos.

¿Cuándo acabamos la búsqueda?

La condición de final de búsqueda es tratada de maneras distintas según el algoritmo. Los hay que fijan un número de pasos al principio mientras que otros realizan movimientos hasta que consideran que el resultado es estable.

¿Qué resultado devolvemos?

Al final de la ejecución siempre se devuelve la mejor partición encontrada. Existe la posibilidad de que el algoritmo vaya haciendo movimientos que empeoren la calidad de la solución. Esto no supondrá ningún problema porque el módulo PropuestaPartición se guardará continuamente aquella solución de mejor calidad aunque esta sea la solución de partida del algoritmo.

Def 5.16 Definimos $MR : \rightarrow P$ MejorResultado como una función que nos devuelve el mejor resultado acumulado durante la ejecución del algoritmo de segunda fase.

$$MR = p \mid \forall_{p' \in MH - \{p\}} calidad(p') \leq calidad(p) \quad (5.22)$$

5.3.3.1 Optimo Local

5.3.3.1.1 Idea

La filosofía básica de este algoritmo consiste en mover los elementos hasta que la solución deja de mejorar. Subiremos hasta encontrar el primer monte.

¿Cuándo acabamos la búsqueda?: cuando la solución siguiente tenga una puntuación inferior a la actual.

La manera de elegir a los elementos a mover puede variar, concretamente distinguimos 3 variantes:

- Elegir elemento con mejor alternativa según la cola AlternativasPartición.
- Elegir elemento peor colocado con la cola PeorElemento.
- Probar de mover todos los elementos y elegir aquel que provoca un partición con mejor calidad.

Las dos primeras alternativas utilizan las colas de prioridad con la función *ap_calidad* Def 4.49 como heurísticas para guiarse a través del vecindaje. La última opción en cambio explora todo el vecindaje para moverse a aquella solución vecina dónde más va a subir la calidad.

Es evidente que esta tercera opción tiene un coste mucho mayor que las dos primeras aunque es la única que nos encuentra estrictamente el óptimo local. Las otras alternativas utilizan las colas de prioridad como estimadoras de cual va a ser el mejor movimiento a realizar.

5.3.3.1.2 Parámetros

- *Modalidad*: indica si el algoritmo utilizará como guía la heurística proporcionada por la cola de prioridad de mejor aportación de calidad o bien explorará continuamente toda la vecindad.

5.3.3.1.3 Pseudocódigo

procedimiento OptimoLocal

inicio

mejora:=SI;

mientras mejora **hacer**

Según el método de elección

o (elemento,grupo,calidad):=MejorAlternativa; $O(1)$

o (elemento,grupo,calidad):=PeorElemento; $O(1)$

calidad_siguiete:=CalcularMovimiento(elemento,grupo); $O(a)$

o (elemento,grupo,calidad_siguiete):=MejorMovimiento; $O(n \cdot a)$

si calidad_siguiete \geq Calidad **entonces**

MoverElemento(elemento,grupo); $O(a \cdot \log n)$ o $O(a \cdot \log r)$

sino

mejora:=NO;

fsi

fmientras

fin

Utilizando las colas de prioridad para la elección del elemento a mover el coste de cada iteración es de $O(a \cdot \log n)$, el propio de mover elemento y efectuar todas las actualizaciones. Si utilizamos la versión de probar todos los movimientos el coste de cada iteración es de $O(a \cdot n)$ teniendo en cuenta también que nos ahorramos un poquito con MoverElemento al no actualizar las colas grandes.

5.3.3.1.4 Resultados

Grupos	Tam.Medio	Tam.Máximo	Tiempo	Calidad	Mejora
71	5,25	24	30,063	90,637	36.1128

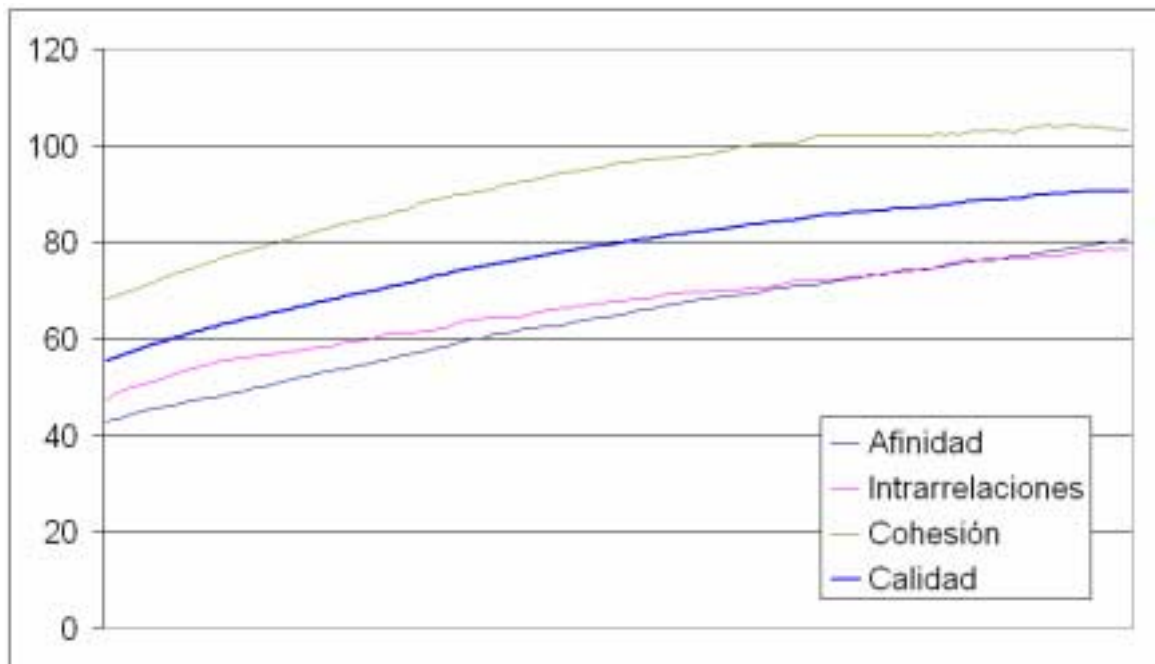


Figura 5.12

5.3.3.2 Colocación

5.3.3.2.1 Idea

Esta metodología pretende colocar cada autor en el grupo que más le corresponde. Repasaremos autor por autor y todo aquel que tenga un grupo alternativa mejor lo asignaremos a este. Haremos un empleo exclusivo de la heurística proporcionada por la función de aportación de calidad. Utilizaremos la cola de prioridad de AlternativasPartición en donde recordemos se guardan los elementos junto con un factor que indica la proporción de calidad del elemento a su mejor alternativa según la calidad a la propia:

$$\frac{ap_calidad(max_calidad(a))}{ap_calidad(f(a))}$$

Nótese que si este valor es > 1 significa que el elemento es mas afín al grupo alternativo que al propio y por lo tanto es candidato a ser movido. El algoritmo va recogiendo el elemento cabeza de esta cola y lo va moviendo hasta que nos encontramos con uno que tiene una calidad proporcional ≤ 1 . Si pasa esto significa que el elemento que tiene el coeficiente $calidad_alternativa/calidad_actual$ mas grande no supera 1 y, por tanto, todos los elementos están en un grupo donde son, como mínimo tan afines que a sus alternativas. Podemos decir entonces que todos los elementos están ‘bien’ colocados.

5.3.3.2 Pseudocódigo

procedimiento Colocacion

inicio

mejora:=SI;

mientras mejora **hacer**

(elemento,grupo,calidad_proporcional):=MejorAlternativa; $O(1)$

si calidad_proporcional > 1 **entonces**

MoverElemento(elemento,grupo); $O(a \log n)$

sino

mejora:=NO;

fsi

fmientras

fin

El coste de cada iteración en este caso es simplemente el coste de mover el elemento mal colocado al grupo que le corresponde: $O(a \log n)$.

5.3.3.2.3 Resultados

Grupo	Tam.Medio	Tam.Máximo	Tiempo	Calidad	Mejora
94	3,97	26	0,330	71,86	17.34

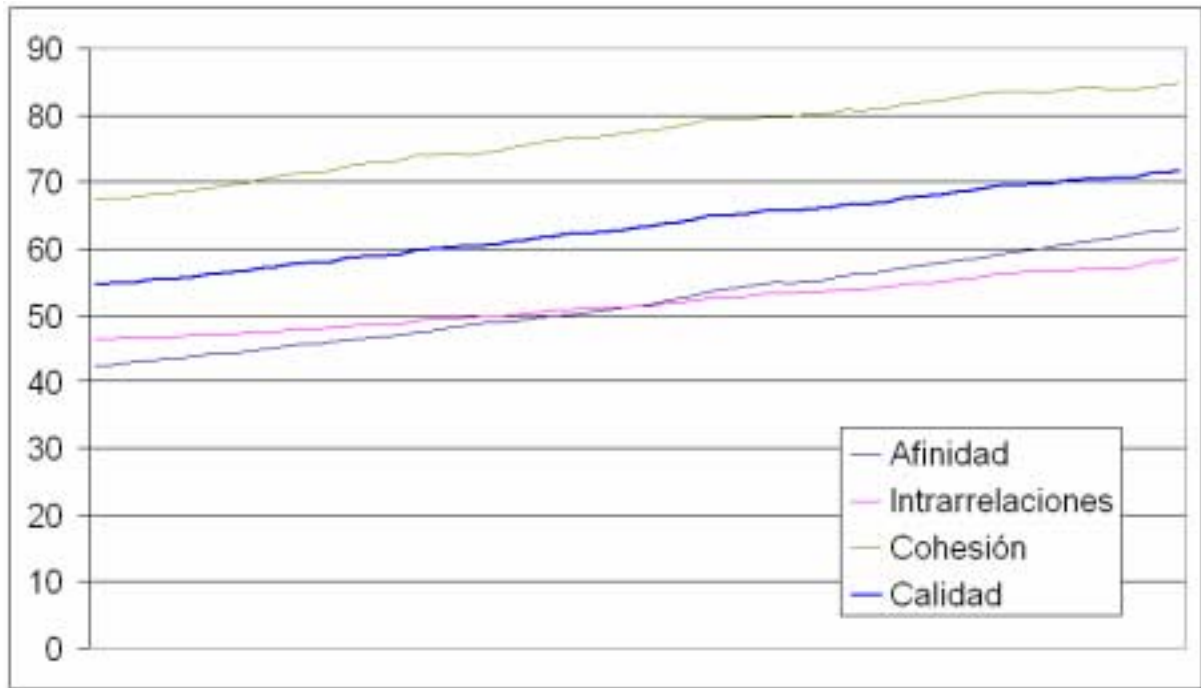


Figura 5.13

5.3.3.3 Extremal Optimization

5.3.3.3.1 Idea

Este algoritmo utiliza un esquema propuesto por Stefan Boettcher y Allon G. Percus en el artículo [3] que sigue normas parecidas a las de la evolución natural de las especies en la naturaleza. El criterio que se usa es simplemente hacer una selección natural de la población sustituyendo a los elementos débiles por otros más fuertes en busca de ir evolucionando. Aplicado al algoritmo que nos toca simplemente seleccionamos al elemento más débil (selección natural) y le reasignamos el grupo (sustitución).

En este esquema no sólo necesitamos una función de calidad general de toda la partición, sino que necesitamos una función que nos determine la calidad de cada elemento su estado de forma ‘fitness’.

Def 5.17 Determinamos como $fortaleza : A \rightarrow \square$ de un autor la aportación de calidad que tiene con el grupo que está asignado.

$$fortaleza(a) = ap_calidad(a, f(a)) \quad (5.23)$$

Utilizaremos la cola de prioridad PeorElemento como la estructura que nos mantendrá continuamente informados sobre el elemento que tiene menos calidad a su grupo, el elemento considerado más débil del conjunto. El criterio de elección del elemento a mover en este caso lo dictamina única y exclusivamente la cola de prioridad de PeorElemento.

Dónde hay variantes es en la selección del grupo al que se moverá el elemento. Según los esquemas básicos del Extremal Optimization propuestos por Boettcher y Percus se debe sustituir el elemento débil por otro cualquiera, aplicado a nuestro problema, asignar al peor elemento otro grupo cualquiera, aleatorio y seguir adelante.

Otra variante es mantener los criterios del resto de algoritmos y mover el elemento al grupo alternativo más afín. La primera opción, más indeterminista, sirve para moverse más por el espacio de búsqueda y evitar así los máximos locales, mientras que la segunda tiende a mejorar más cada vez aunque la búsqueda puede caer en óptimos locales.

El punto más difuso de este algoritmo está en decidir cuándo se considera acabada la búsqueda. Hemos optado en una primera instancia en establecer a priori un número fijo de movimientos en función del total de elementos de la partición.

Se ha establecido que se harán un total de

$$num_movimientos_EO = n \cdot \frac{NumVecesPor100Elementos}{100} \quad (5.24)$$

movimientos para equiparar un coste temporal semejante a las otras técnicas.

5.3.3.3.2 Parámetros

- *NumVecesPor100Elementos*: indica el número de iteraciones del algoritmo por cada 100 autores a agrupar.
- *MovimientoAleatorio*: indica si una vez elegido el autor a mover se elegirá un destino aleatorio o bien el alternativo más afín.

5.3.3.3 Pseudocódigo

procedimiento ExtremalOptimization

```

inicio
  num_iteraciones:=NumElementos · NumVecesPor100Elementos / 100;

  para i:=0 hasta num_iteraciones hacer
    (elemento,grupo,calidad_proporcional):=PeorElemento;       $O(1)$ 
    MoverElementoAleatorio(elemento);                           $O(a \cdot \log n)$ 
  o
    MoverElemento(elemento,grupo);                              $O(a \cdot \log n)$ 
  fpara
fin

```

Al tener limitados el número de iteraciones podemos calcular el coste final del algoritmo. Tenemos $O\left(n \cdot \frac{\alpha}{100}\right)$ iteraciones que equivalen a $O(n)$ asintóticamente, con $O(a \cdot \log n)$ para cada una de ellas. Esto hace un total de $O(n \cdot a \cdot \log n)$. El producto de $n \cdot a$ significa que para cada elemento de los n recorreremos sus a adyacentes lo que equivale a recorrerse el total de aristas del grafo dos veces, una para cada sentido: $2m$, asintóticamente m . El coste se puede expresar como $O(m \cdot \log n) \leq O(n^2 \cdot \log n)$.

5.3.3.3.4 Resultados

Grupos	Tam.Medio	Tam.Máximo	Tiempo	Calidad	Mejora
125	2,98	24	0,439	63.24	8.72

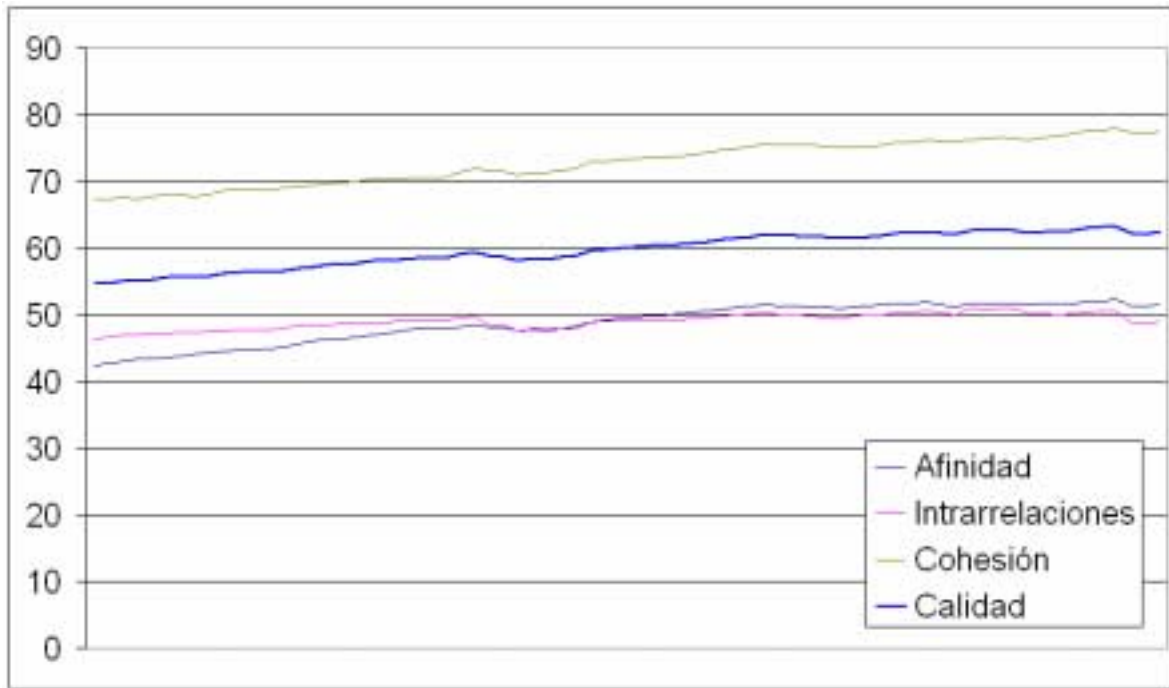


Figura 5.14

5.3.3.4 Simulated Annealing

5.3.3.4.1 Idea

Con el pretexto de salirnos de las soluciones localmente óptimas generadas por los algoritmos deterministas presentados al principio nos encontramos ahora con dos alternativas más probabilísticas. Por un lado el ya tratado Extremal Optimization en la variante de ‘sustitución aleatoria’ que no nos da los resultados deseados y, por otro lado, con esta nueva técnica denominada Simulated Annealing que a continuación vamos a detallar.

El Simulated Annealing está inspirado en la cristalización de los minerales en la naturaleza. En un principio, en una fase de altas temperaturas, se permiten realizar muchos cambios de estructura pero, a medida que va pasando el tiempo, el cristal va enfriándose y las modificaciones se reducen en cantidad hasta llegar a un punto de ‘congelación’ a partir del cual el cristal se considera ya formado.

En nuestro algoritmo la analogía se traduce empezando con la partición inicial y un cierto valor de temperatura. A medida que vamos avanzando esta temperatura puede subir o bajar en función de si la solución por la que pasamos mejora o empeora: si mejoramos

aumentamos la temperatura, si empeoramos la bajamos. Cuando la temperatura baja por debajo de un mínimo entonces terminamos la búsqueda.

Para implementar esta técnica es necesario responder a las siguientes cuestiones:

- *¿Cuál va a ser la temperatura inicial?*
- *¿Cómo se irá modificando la temperatura?*
- *¿Cuál va a ser la temperatura mínima?*
- *¿Cómo va a afectar la temperatura a la elección de una solución?*

Empezaremos respondiendo vagamente esta última cuestión entrando en más detalle posteriormente. Si tenemos una temperatura elevada tenemos más posibilidades de aceptar movimientos que empeoran la solución actual, dada la calidad de la solución tenemos crédito para poderla empeorar. Si en cambio tenemos temperaturas bajas la probabilidad de aceptar una solución peor es muy baja y casi sólo aceptamos soluciones buenas, dado a que llevamos unos cuantos fracasos en el intento de mejorar las soluciones se nos va acabando el crédito. Cuando hacemos un movimiento, si este es para mejorar lo aceptamos incondicionalmente y subimos la temperatura, si empeora lo aceptamos según una probabilidad que dependerá de la temperatura según hemos descrito, si no es aceptado elegimos otra solución.

¿Cuál va a ser la temperatura inicial?

En cuanto a la temperatura inicial existen dos posibilidades: una es que sea un parámetro del usuario; otra es coger como temperatura la calidad de la partición inicial. Si elegimos esta segunda opción tenemos que fijar la condición de que la temperatura puede tomar valores de 0 a 100 de manera análoga a la función de calidad. La razón por la que es bueno utilizar la calidad de una partición como temperatura inicial es que: si la partición es mala, de calidad baja, los primeros cambios que se le van a hacer deberían mejorar la calidad para salir del bache en que se encuentra, por lo tanto forzamos a que sólo mejore con una temperatura mala.

Además si la solución es mala porque el algoritmo anterior se ha equivocado la probabilidad de que los movimientos a realizar la mejoren y suban la temperatura es alta, no es necesario que seamos permisivos. Por el contrario si nos encontramos con una buena solución probablemente ya sea cercana a la óptima y intentar mejorarla es muy difícil. Por eso le damos mucha temperatura para que sea susceptible a explorar más el espacio de soluciones aunque vaya empeorando.

Imaginemos la analogía con el paseo por un paisaje lleno de montañas. Si empezamos el paseo desde un punto elevado (una solución de buena calidad) para intentar llegar a otro más alto debemos darle la posibilidad de que baje, ya que o bien estamos en la ladera de la montaña más alta o iremos a parar a un óptimo local. Si por el contrario la solución es mala y nos a dejado en un altiplano, por así decirlo, debemos intentar que empiece a subir alguna montaña si quiere acceder a la más alta, no queremos que pierda el tiempo bajando.

¿Cómo se irá modificando la temperatura?

La temperatura se va modificando a lo largo del algoritmo siguiendo la estrategia de que mejorar soluciones implica subir la temperatura y empeorar soluciones implica bajar temperatura. ¿Pero en qué grado? Depende de muchos factores. Para empezar de la diferencia de calidad de la nueva solución: a más diferencia positiva o negativa más sube o más baja la temperatura. Estamos hablando de la diferencia que hay entre la calidad de la solución actual y la de la solución propuesta por el movimiento a hacer. En el caso del descenso de temperatura por empeoramiento también podemos tener en cuenta no sólo la diferencia con la solución anterior sino la diferencia con la mejor de las soluciones para ver si aunque respecto a la solución anterior empeoramos poco nos encontramos a tanta diferencia de la mejor hasta el momento que no permitimos bajar más. Finalmente también tendremos en cuenta, tanto subiendo como bajando la temperatura, el número de veces seguidas que vamos en la misma dirección. Cuantas más veces subimos o bajamos más inercia cogemos y más se exagera el cambio. Si mejoramos continuamente significa que la mejora era clara y que debemos subir la temperatura para permitir poder bajar luego. Y viceversa, si bajamos continuamente debemos frenar la caída bajando la temperatura e impidiendo más soluciones peores. Tendremos en cuenta este factor con unos contadores de subidas y de bajadas seguidas.

Siendo:

Def 5.18 Sea $t : \rightarrow [0..100]$ la temperatura del sistema.

Def 5.19 Llamaremos $p : \rightarrow P$ la partición o solución en curso y $p' : \rightarrow P$ la nueva solución propuesta, entonces sea $diferencia : \rightarrow \square$ la diferencia de calidad entre la solución nueva y la actual.

$$diferencia = |ap_calidad(p') - ap_calidad(p)| \quad (5.25)$$

siendo siempre la *diferencia* en valor absoluto.

Def 5.20 Sea $diferencia_mejor : \rightarrow \square$ la diferencia de calidad entre la solución nueva y la mejor acumulada.

$$diferencia_mejor = ap_calidad(MR) - ap_calidad(p') \quad (5.26)$$

Def 5.21 Sea $B : \rightarrow \square$ un parámetro de bonificación de subida de temperatura.

Si estamos subiendo la calidad y, además estamos muy cerca o directamente mejorando la mejor de las soluciones se incrementa la temperatura. Nótese que el factor intermedio $(1 + \dots)^\beta$ si fijamos que $\beta \geq 1$ siempre bonificará y nunca perjudicará. El sentido de este factor es promover más aquellas mejoras que además están mejorando la mejor solución encontrada. Si la solución es la mejor $diferencia_mejor=0$ el incremento se multiplicará por $(1+B)^\beta$.

Def 5.22 Sean $iteraciones_subiendo, iteraciones_bajando : \rightarrow \square$ el número de iteraciones seguidas de subida y de bajada.

Hay varias formas de actualizar estos valores, una de ellas es contar exclusivamente los movimientos seguidos en una dirección con lo que al cambiar de sentido se ponen a cero. Otra manera es mantener un poco de inercia. Cada vez que subimos la temperatura aumentamos en 1 las $iteraciones_subiendo$ y decrementamos en 1 las de bajando. Al revés en caso de bajada. Esta segunda opción es más aconsejable.

Def 5.23 Sean $\alpha, \beta, \gamma : \rightarrow \square$ exponentes que permiten al usuario ponderar más o menos los factores antes expuestos. Son distintos los de subida que los de bajada.

Las fórmulas propuestas como modificadoras de la temperatura son las siguientes:

De subida:

$$t = t \cdot \frac{100 + diferencia^\alpha \cdot \left(1 + \min \left(B - \frac{diferencia_mejor}{100}, 0 \right) \right)^\beta \cdot iteraciones_subiendo^\gamma}{100} \quad (5.27)$$

De bajada:

$$t = t \cdot \frac{100 - diferencia^\alpha \cdot diferencia_mejor^\beta \cdot iteraciones_bajando^\gamma}{100} \quad (5.28)$$

Nótese también del hecho de que la temperatura se decrementa proporcionalmente a ella misma ($t := t \cdot \dots$) lo que significa que a altas temperaturas las variaciones serán mayores que a bajas temperaturas. También hay que recalcar que la temperatura nunca excederá su valor máximo (normalmente 100) ya que de lo contrario un ritmo continuamente creciente de temperatura la podría dejar a unos valores que luego costarían muchas iteraciones de abandonar.

¿Cuál va a ser la temperatura mínima?

La temperatura mínima es especificada por el usuario. Tiene que ser superior a 0 ya que el método empleado para disminuir la temperatura esta se decrementa proporcionalmente a ella misma y tiende a 0 pero no llega nunca.

¿Cómo va a afectar la temperatura a la elección de una solución?

El hecho de que haya el factor temperatura en nuestro algoritmo tiene sentido a la hora de determinar que hacer con aquellas soluciones que se proponen que no mejoran la actual. Ya hemos advertido que las soluciones que suponen una mejora son aceptadas siempre, pero las que no, son aceptadas con una cierta probabilidad. Esta probabilidad depende, entre otros factores, de la temperatura actual del sistema. Así, a más temperatura más probabilidades de aceptar cambios negativos y a más baja temperatura menos.

La función de probabilidad no sólo depende de la temperatura actual sino que también depende del grado de empeoramiento de la solución que deseamos adoptar, o sea de la diferencia, usando los mismos términos que en el procedimiento de enfriamiento. Esta diferencia, pero, se tiene que relativizar en función del tamaño de la partición. No es lo mismo cambiar una autor de grupo en una partición de 10000 elementos que en una de 100, el efecto sobre la función de calidad no será el mismo. Por lo tanto en la ecuación que determina la probabilidad hay que tener en cuenta el número de elementos de la partición.

Def 5.24 Sea $t_max : \rightarrow \square$ la temperatura máxima permitida.

Def 5.25 Sea $D : \rightarrow \square$ un factor que representa la diferencia máxima recomendable.

Def 5.26 Sean $\alpha, \beta : \rightarrow \square$ exponentes que permiten al usuario ponderar más o menos los factores de temperatura y diferencia respectivamente.

Tenemos esta propuesta como fórmula que determina la probabilidad:

$$\rho = \frac{t^\alpha}{t_max^\alpha} \cdot \frac{D^\beta}{(diferencia \cdot n)^\beta} \quad (5.29)$$

En la fórmula nos encontramos con un producto de dos factores: por un lado la temperatura y por el otro la diferencia de calidad con la solución actual.

La temperatura afecta directamente a la probabilidad: a más temperatura más probabilidades de aceptarla. Para ponderarla en un intervalo [0..1] la dividimos por su valor máximo, ambos bajo el exponente α .

En el apartado de la diferencia la fórmula se complica un poco. La diferencia debe penalizar la probabilidad, por lo tanto aparece en el denominador. Esta multiplicada por n para compensar el hecho ya comentado de que cuando más grande es el número de elementos menor debe ser la diferencia por mover uno. En el numerador encontramos un factor que representa el valor de diferencia máximo previsto que sirve para ajustar la probabilidad.

5.3.3.4.2 Parámetros

- *Modalidad*: indica si el S.A utilizará como guía la heurística proporcionada por la cola de prioridad de mejor aportación de calidad o bien explorará continuamente toda la vecindad.
- *t_max*: límite máximo que puede alcanzar la temperatura. No dejamos que suba más de este valor.
- *t_min*: límite inferior que puede alcanzar la temperatura. Si bajamos de este límite se acaba el algoritmo.
- *B*: parámetro de bonificación de subida de temperatura.
- *ExponenteDecrecimientoDiferencia*: exponente que lleva la diferencia de cualidades en la función de decremento de temperatura. Corresponde al símbolo α .

- *ExponenteDecrecimientoMejorOpcion*: exponente que lleva la diferencia de cualidades hacia la mejor opción en la función de decremento de temperatura. Corresponde al símbolo β .
- *ExponenteDecrecimientoIteraciones*: exponente que lleva el numero de iteraciones seguidas decrementando de temperatura. Corresponde al símbolo γ .
- *ExponenteCrecimientoDiferencia*: exponente que lleva la diferencia de cualidades en la función de incremento de temperatura. Corresponde al símbolo α .
- *ExponenteCrecimientoMejorOpcion*: exponente que lleva la diferencia de cualidades hacia la mejor opción en la función de incremento de temperatura. Corresponde al símbolo β .
- *ExponenteCrecimientoIteraciones*: exponente que lleva el numero de iteraciones seguidas incrementando de temperatura. Corresponde al símbolo γ .
- *D*: factor que representa la diferencia máxima estimada entre las cualidades de dos soluciones separadas por un movimiento.

5.3.3.4.3 Pseudocódigo

Ahora que ya hemos discutido sobre las cuestiones básicas para el diseño del algoritmo basado en el Simulated Annealing pasamos a construir el esquema.

En realidad tenemos dos variantes del mismo algoritmo. Análogamente como ocurría con el óptimo local al buscar el siguiente movimiento o bien utilizamos la heurística de la cola de prioridad, la manera más rápida, o bien exploramos todo el vecindaje eligiendo el movimiento más productivo.

Sea cual sea el método de elección del siguiente movimiento, si este mejora la calidad lo aceptamos y actualizamos la temperatura según la fórmula propuesta. Si el movimiento no mejora calculamos la probabilidad de aceptarlo, en caso de aceptarlo realizamos el movimiento. Si el movimiento no es aceptado pedimos por el siguiente.

Cada vez que un movimiento propuesto empeora la calidad, lo aceptemos o no, decrementamos la temperatura. Cuando la temperatura baja de los límites propuestos acabamos la ejecución.

El esquema es el siguiente:

procedimiento SimulatedAnnealing

inicio

```

t:=Calidad;
iteraciones_subiendo:=0;
iteraciones_bajando:=0;
mientras t > t_minima ∧ TieneAlternativas hacer
    calidad:=Calidad;
    (elemento,grupo,calidad_proporcional):=SiguieteAlternativa;           O(1)
o (elemento,grupo,cohesion_proporcional):=MejorMovimiento;           O(n·a)
si no MovimientosHechos[elemento] → grupo entonces
    calidad_movimiento:=CalcularMovimiento(elemento,grupo)           O(a)
    diferencia:=calidad_movimiento - calidad;
    diferencia_mejor:=calidad_movimiento – CalidadMejorSolucion;
si diferencia < 0 entonces
    probabilidad:= {cáculo de la probabilidad};
    si random(1) ≤ probabilidad entonces
        MoverElemento(elemento,grupo);           O(a·log n)
    fsi
        iteraciones_bajando:= iteraciones_bajando +1;
        iteraciones_subiendo:=maximo(iteraciones_subiendo -1, 0);
        t:= {cáculo de decremento de temperatura};
    sino
        MoverElemento(elemento,grupo);           O(a·log n)
        iteraciones_subiendo:= iteraciones_subiendo +1;
        iteraciones_bajando:=maximo(iteraciones_bajando -1, 0);
        t:= {cáculo de incremento de temperatura};
        t:=minimo(t, t_maximo);
    fsi
fsi
fpara
fin

```

Los costes variarán según la variante utilizada:

Para la variante que usa la cola de prioridad como heurística:

El coste de cada iteración depende de si se realiza el movimiento o no. En caso de que se realice el coste es de $O(a \cdot \log n)$ si sólo se realiza el cálculo de calidad simulando el movimiento el coste solo es de $O(a)$.

Para la variante que explora todo el vecindaje:

También el coste de cada iteración depende de si se realiza o no el movimiento. Al realizarse significa que deberemos explorar otra vez el nuevo vecindaje generado escogiendo

el mejor $O(n \cdot a)$. Si el movimiento no se hace simplemente pasamos al siguiente movimiento siguiendo un orden arbitrario y simulamos el coste del mismo $O(a)$.

5.3.3.4.4 Resultados

Grupos	Tam.Medio	Tam.Maximo	Tiempo	Calidad	Mejora
72	5,1	24	1,802	88,86	34,34

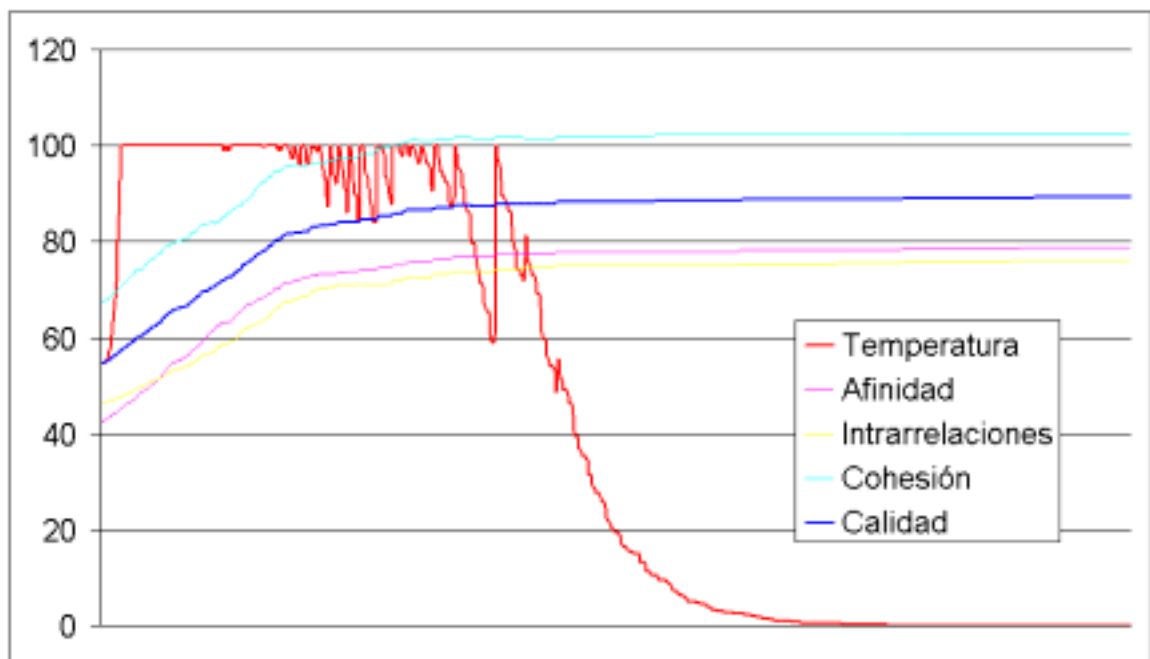
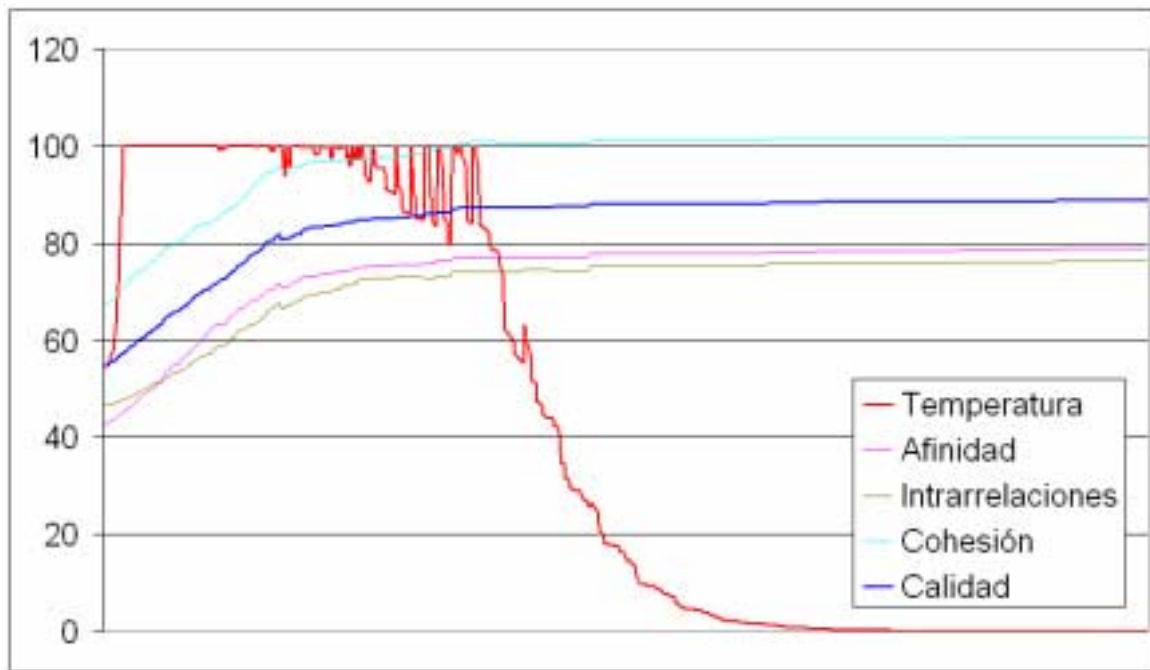


Figura 5.15 Evolución de los parámetros de calidad y la temperatura en un S.A.

5.3.4 Algoritmos de tercera fase

Si a la primera clase de algoritmos que atacaban directamente el grafo les hemos denominado algoritmos de primera fase, a los siguientes que modificaban las soluciones obtenidas con el propósito de mejorarlas les hemos apodado de segunda fase, a esta nueva técnica algorítmica por su emplazamiento en el proceso de resolución del problema la clasificamos dentro de los algoritmos de tercera fase.

En realidad sólo hay un algoritmo en este apartado que es el denominado Algoritmo Genético. Esta propuesta basada en técnicas similares a la evolución de las especies mediante la combinación de material genético, se denomina de tercera fase porque se ejecutan después de las otras técnicas. A partir de las soluciones generadas por los algoritmos convencionales y mejoradas por las técnicas de segunda fase los algoritmos genéticos recombinarán estas soluciones para ver si encuentran alguna mejor.

5.3.4.1 Algoritmo Genético

5.3.4.1.1 Idea

La connotación de genético nos revela el ámbito que ha inspirado este esquema. En los seres vivos, las características como especie y como individuos independientes están descritas mediante cadenas de información en forma de ácido desoxirribonucleico (ADN) que se empaquetan en cromosomas dentro de cada una de sus células. Esta gran cantidad de información es muy parecida dentro de los individuos de una misma especie, pero aun así conserva las características suficientes para hacer cada individuo distinto de los demás. Cuando aparece un nuevo individuo su ADN es el resultado de combinar los ADN's de dos individuos de la especie. De esta manera las partes comunes que le categorizan como especie se mantienen y los detalles que le singularizan son una mezcla de los de sus progenitores. Esta es la manera que tiene la naturaleza de ir probando todas las posibilidades en el objetivo de ir evolucionando la especie. La selección natural, la supervivencia ante las adversidades y la feroz competencia irán revelando cuales han sido las recombinaciones mejores.

Esta sencilla filosofía se puede aplicar para resolver algunos problemas computacionales como el que ahora nos ocupa. Mediante técnicas ya descritas hemos encontrado varias soluciones al problema de la agrupación. Los algoritmos de segunda fase intentaban hacer progresar estas técnicas poco a poco para que fueran mejorando, pero en

realidad no hacían grandes cambios. Con los algoritmos genéticos se pretende combinar dos soluciones a un mismo problema que son distintas creando otra que sea una mezcla de las anteriores. Seguidamente hay que hacer evolucionar esta solución para que mejore mediante las técnicas que conocemos. Al mezclar dos soluciones nos salimos radicalmente de los problemas de los máximos locales que teníamos en los algoritmos de segunda fase, ya que no hacemos pequeños cambios sino que la solución resultado acaba siendo bastante distinta a las que la han generado. Este proceso se va realizando iterativamente hasta que se considera oportuno.

Estableciendo analogías entre el proceso de evolución genético natural y la computacional tenemos que empezar diciendo que cuando hablamos de especie nos referimos a un problema a resolver, cuando hablamos de población de una especie lo hacemos de las distintas soluciones que tenemos para ese problema. Si en una especie su información está almacenada en el código ADN en nuestro problema la información de la solución es un vector de asignaciones de elementos a grupos. Cuando recombinamos dos elementos para formar otro crearemos una solución cuyo vector de asignaciones sea una mezcla de los vectores de asignaciones de las soluciones que la han generado.

Def 5.27 Declaramos $N \rightarrow \square$ como el número de soluciones que han generado las dos primeras fases algorítmicas y que ahora debemos mezclar.

El esquema general de un algoritmo genético sigue los siguientes pasos:

1. Empezamos con una población de N soluciones a nuestro problema. Cada solución es también llamada cromosoma y lleva implícita un valor que determina su calidad.
2. Repetimos los siguientes pasos hasta obtener N soluciones nuevas:
 - Seleccionamos un par de cromosomas candidatos a formar solución hija.
 - Recombinamos los cromosomas y creamos una nueva solución.
 - Hacemos madurar esta solución.
3. Seleccionamos N soluciones y eliminamos las otras.
4. Volvemos al paso 2 hasta considerarlo necesario.

Este proceso se realiza un número predeterminado de veces. Cada vuelta del bucle se denomina generación nueva. Tal y como se ve en el esquema algunos de los pasos han quedado un poco ambiguos ya que, no hay que olvidar, que este esquema de algoritmo genético es muy general y hace falta concretar algunos aspectos. Para poder implementar una solución tipo algoritmo genético hay que responder a las siguientes preguntas:

- *¿Cómo seleccionamos las parejas de cromosomas a cruzar?*
- *¿Cómo hacemos la recombinación de cromosomas?*
- *¿Cómo se hace madurar una solución?*
- *¿Qué criterio se elige para hacer la selección y eliminación de cromosomas?*
- *¿Cuántas veces hay que repetir el proceso?*

¿ Cómo seleccionamos las parejas de cromosomas a cruzar?

Por un lado nos interesa que se emparejen aquellos cromosomas de más calidad, pero también nos interesa que haya variedad y que cuantos más cromosomas distintos se utilicen más rico será el espacio de soluciones creado. Con el propósito de satisfacer estas dos máximas se ha elaborado un sistema de emparejamiento como sigue. Cada individuo intentará encontrar una pareja. Nada impide que dos cromosomas elijan una misma pareja. De esta manera garantizamos, por un lado, que todos los individuos participarán como mínimo en un cruce y tendrán un descendiente y, por otro lado, que haya exactamente N descendientes. El criterio que utiliza un individuo para elegir pareja es aleatorio, pero teniendo más probabilidades de escoger aquellos cromosomas con más calidad. Se satisface así que haya más descendencia de aquellas soluciones más buenas.

¿Cómo hacemos la recombinación de cromosomas?

Básicamente recombinar cromosomas significa coger parte de una solución y parte de la otra y crear una nueva. Nuestras soluciones son compuestas por un vector de asignaciones, por lo tanto coger parte de una solución y parte de otra no resulta nada difícil. Los vectores, en principio, van de $[0..n-1]$ y existe una correspondencia biyectiva entre cada elemento. Hemos dicho ‘en principio’ porque no es exactamente así. Las soluciones provienen de distintos algoritmos con distintas técnicas y, si está activado el flag de filtraje de grupos pequeños (de menos de 4 elementos por ejemplo), puede ser que se pierdan elementos en la solución. Entonces nos encontramos con que las soluciones no son exactamente simétricas y no existe una correspondencia directa entre cada posición de los vectores de asignaciones.

Pero no hay problema ya que lo que sí comparten ambas soluciones es que parten del mismo grafo y, mediante la ayuda del módulo de propuesta de partición podemos traducir el elemento a vértice y viceversa con rapidez. Relegaremos la resolución de este problema de incompatibilidades de soluciones a la implementación estricta del algoritmo y supondremos que las dos soluciones son equivalentes en tamaño y en la relación posición \leftrightarrow elemento:

$$S_1 = \begin{array}{|c|c|c|c|} \hline f(0) & f(1) & & f(N-1) \\ \hline 0 & 1 & & N-1 \\ \hline \end{array} \quad \text{dónde } f: A \rightarrow [1..r_1]$$

$$S_2 = \begin{array}{|c|c|c|c|} \hline f'(0) & f'(1) & & f'(N-1) \\ \hline 0 & 1 & & N-1 \\ \hline \end{array} \quad \text{dónde } f': A \rightarrow [1..r_2]$$

Mezclar las soluciones significa crear un nuevo vector de $[0..N-1]$ donde parte seguirá la función f y parte f' . Pero a primera vista se puede detectar que ambas funciones van a parar a un mismo conjunto de números $[1..\min\{r_1, r_2\}]$ que no tienen nada que ver. Para evitar estas colisiones redefiniremos f' como f'' desplazando las imágenes en r_1 siendo $f'' = f' + r_1$ obteniendo un rango de valores $[1..r_1 + r_2]$.

Tenemos distintos criterios para elegir cada elemento a qué función de asignación le hacemos caso. En [6] se propone un sistema sencillo de emparejamiento que consiste en partir las soluciones por una misma posición y quedarnos con la primera mitad de una y la segunda mitad de la otra. El punto donde cortar puede ser absolutamente aleatorio, aleatorio en función de la calidad de cada cromosoma o directamente proporcional estas calidades. Si el cromosoma 1 y el 2 son de idéntica calidad cortamos por la mitad, si uno es el doble de calidad que el otro cortamos un tercio de manera que la cantidad de elementos seleccionados sea equivalente a la calidad.

Otra manera equivalente de hacer la repartición es ir elemento a elemento independientemente. Para cada elemento calculamos la probabilidad de elegir a una asignación o otra en función, otra vez, de la calidad de ambas propuestas.

¿Cómo se hace madurar una solución?

Una vez cruzadas las soluciones tenemos una solución un poco aleatoria y mal organizada porque se han partido los grupos. Los algoritmos de segunda fase se encargarán de recolocar aquellos elementos mal puestos al grupo que les corresponde. Llamamos este proceso como la obtención de la madurez de la solución.

¿Qué criterio se elige para hacer la selección y eliminación de cromosomas?

Entre los múltiples criterios disponibles se ha elegido el más sencillo de todos: se eliminan las soluciones viejas y nos quedamos con las que acabamos de engendrar.

¿Cuántas veces hay que repetir el proceso?

Hay que repetirlo un número determinado de veces que será especificado por el usuario. Cada solución que participa en todo el desarrollo del algoritmo genético es candidata a ser la final. Por eso nos guardamos siempre la solución que ha alcanzado la mayor calidad de todas sea en la generación que sea, incluso en la inicial.

5.3.4.1.2 Parámetros

- *NumGeneraciones*: indica el número de generaciones, de mezclas de todos los elementos, deben hacerse.

5.3.4.1.3 Pseudocódigo

procedimiento AlgoritmoGenetico

inicio

para cada generación **desde** 0 **hasta** *NumGeneraciones* **hacer**

 descendencia es vector [0..*N*-1];

para cada cromosoma **desde** 0 **hasta** *N*-1 **hacer**

 (cromosoma1,cromosoma2)=ElegirPareja(cromosoma);

 descendencia[cromosoma]=Mezclar(cromosoma1,cromosoma2);

fpara

 Seleccionar(descendencia);

fpara

fin

5.3.4.1.4 Resultados

5.4 Diseño de la aplicación

Como hemos especificado en la metodología se ha diseñado un aplicativo utilizando técnicas de diagramaje en UML (Unified Modeling Language). No vamos a mostrar todos los detalles del siempre tedioso proceso de diseño de una aplicación porque extendería innecesariamente el documento y nos apartaríamos de la tendencia que debe marcar este proyecto que no es otra que una discusión algorítmica ante un problema determinado y no el informe de un diseño de aplicación en toda su magnitud.

Mostraremos un diagrama de clases en el que se puede ver representados los objetos que intervienen en el programa para poder ejecutar el algoritmo.

Una parte muy importante a destacar es describir el comportamiento dinámico de los algoritmos. Explicaremos mediante diagramas de secuencia como la aplicación va controlando la ejecución de los algoritmos.

5.4.1 Dominio de la aplicación

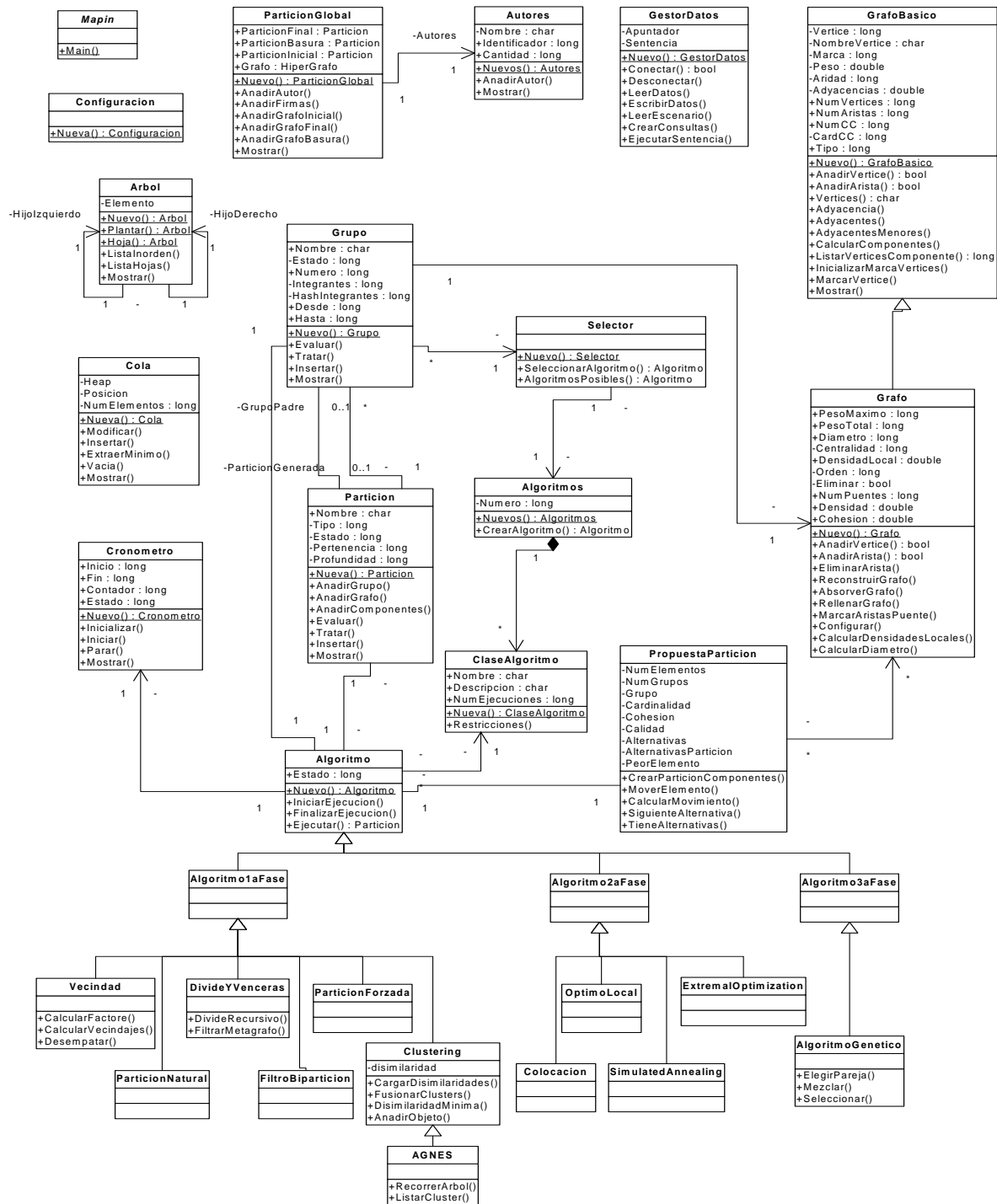


Figura 5.16 Diagrama de clases de todo el dominio de la aplicación

En el diagrama de clases Figura 5.16 vemos todos los objetos necesarios para implementar una aplicación que soporte los algoritmos.

La clase *Mapin* es la principal del sistema y es por donde se empieza a ejecutar todo.

Tenemos una *Configuración* donde se guardan los parámetros que ha definido el usuario para adaptar los algoritmos (ver cada algoritmo su apartado de parámetros).

Las clases *Cola* y *Árbol* implementan una cola de prioridad y un árbol binario respectivamente.

La clase *Cronometro* implementa un contador de tiempo para medir la eficiencia de los algoritmos.

Las clases *GrafoBasico* y *Grafo* son las que implementan los grafos de coautoría. Se han dividido en dos porque mientras la primera implementa un grafo mediante listas de adyacencia la segunda además extiende las propiedades de un grafo básico añadiéndole funciones requeridas por los algoritmos. A veces nos puede interesar tener un grafo para manejar una estructura determinada pero sin la necesidad de que se guarde información añadida de factores de densidad, cohesión, densidad local, etc... utilizados por los algoritmos.

En el centro del diagrama encontramos las clases que contienen los objetos básicos de la aplicación que son: los grupos, las particiones o conjuntos de grupos y los algoritmos.

Los grupos son representados mediante la clase *Grupo* donde se guarda la lista de sus integrantes y una referencia a un grafo que representa las relaciones de coautoría entre los miembros del grupo.

La clase *Partición* incluye resultados de las ejecuciones de los algoritmos. Básicamente se trata de agrupaciones de objetos de la clase *Grupo*. Existe una instancia de la clase *Partición* que contendrá la partición final o resultado final de toda la aplicación.

La clase *Algoritmo* contiene objetos que representan instancias de ejecución de un algoritmo de los especificados. Cada objeto *Algoritmo* representa una ejecución real del algoritmo y tiene un *Grupo* que es la fuente a partir y una *ParticionParticion* que es el resultado de ese algoritmo.

En la clase *ClaseAlgoritmo* tenemos una instancia por cada tipo diferente de algoritmo que tengamos. En este caso habrá 11 instancias de esta clase correspondientes a los 11 algoritmos distintos que se han desarrollado.

La clase *Algoritmos* no es más que la lista de objetos *ClaseAlgoritmo* que encontramos en la aplicación.

La clase *Selector* tiene la responsabilidad de conocer todos los algoritmos que hay implementados, lo hace con una referencia a *Algoritmos* y debe ser capaz de seleccionar los algoritmos que se podrán ejecutar para partir un determinado grupo. De momento lo único que hace *Selector* es seleccionar la lista de algoritmos posibles a ejecutar para un grupo en concreto. Hay una instancia de *Selector* para cada instancia de *Grupo*. En un futuro a la clase *Selector* se le podrán aplicar técnicas más sofisticadas que sean capaces de, dado un grupo con todas sus características elegir el mejor algoritmo a priori para partirlo. Así se evitará tener que probar todos los algoritmos.

La clase *PropuestaParticion* contiene una versión provisional de una partición de un conjunto de autores en grupos de investigación. Esta clase es manipulada directamente por los algoritmos y permite mucha flexibilidad a la hora de realizar cambios de asignaciones de autores a grupos. Mantiene información continuada de la calidad de la partición. Para ver más detalles consultar [ref]. La diferencia con la *Partición* normal es que esta última es ya definitiva e inalterable y en cambio la *PropuestaPartición* está diseñada específicamente para operar sobre ella.

La clase *ParticiónGlobal* contiene la información de la entrada de datos. Contiene el grafo de coautorías original sin filtrar y la correspondiente lista de autores implementada en la clase *Autores*. Contiene también una referencia a una instancia de la clase *Partición* dónde se irán metiendo los grupos definitivos y otra donde se meterán los grupos descartados.

La clase *GestorDatos* implementa la interficie de la aplicación con la base de datos. Esta se encarga de conectarse a la base de datos, lanzar la consulta y devolver los resultados a *ParticiónGlobal* para crear la entrada del problema. Se conecta a la base de datos mediante una interfaz ODBC (Open Data Base Connection) y realiza las consultas mediante sentencias SQL. Análogamente se encargará de, una vez finalizada la aplicación coger los datos de la partición resultante y guardarlos en la base de datos. El proceso de lectura y el de escritura sólo tienen lugar al principio y al final de la búsqueda de grupos y no suponen ningún coste significativo en comparación con el resto del proceso.

Finalmente en la parte inferior nos encontramos con la jerarquía de clases que implementan los algoritmos. Nótese que todas acaban heredando de una misma clase *Algoritmo* y que se subdividen en tres categorías según la fase de ejecución de los algoritmos.

5.4.2 Secuencia de ejecución

A continuación veremos una serie de diagramas de secuencia que ilustran el comportamiento dinámico del sistema.

Primero veremos como se realiza la captura de datos:

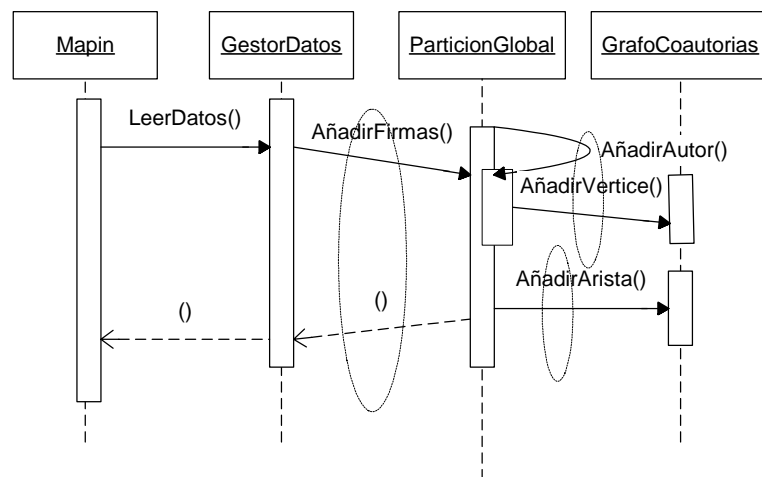


Figura 5.17 Diagrama de secuencia de la carga de datos.

En la Figura 5.17 vemos qué objetos intervienen en la captura de datos. En primer lugar la orden de capturar los datos viene del módulo principal *Mapin*. Este se comunica mediante el *GestorDatos* que es el responsable de todo el proceso de captura de datos. Este irá leyendo artículos de la base de datos iterativamente. Cuando se encuentre con un artículo recogerá todas las firmas que encuentre y se lo enviará todo junto a *ParticionGlobal* que es el módulo que conoce los datos globales de coautorías. Este por cada autor del paquete de firmas recibido lo añadirá como vértice al grafo de coautorías. Luego para cada par de autores creará una arista que corresponderá con la coautoría.

Seguidamente veremos como se arranca todo el proceso de búsqueda de grupos:

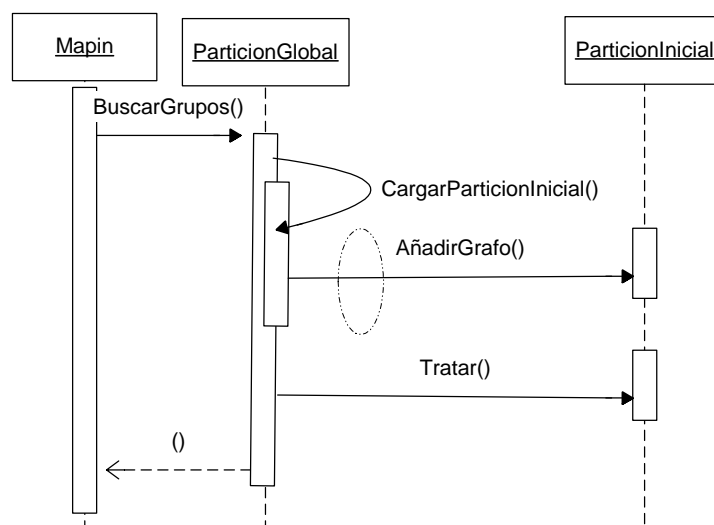


Figura 5.18 Diagrama de secuencia del inicio de ejecución del proceso de búsqueda de grupos.

Para arrancar la búsqueda de grupos el programa principal *Mapin* lanza la orden sobre *ParticionGlobal*. Este se dispone a cargar la partición inicial, proceso que consiste en filtrar el grafo si hace falta y ejecutar el algoritmo de componentes conexas. Para cada componente se crea un subgrafo y a partir de este se van creando grupos que se van añadiendo a esta partición inicial a través de la llamada *AñadirGrafo*. Una vez tenemos configurada la partición inicial empezamos el proceso con una simple llamada a *Tratar ParticionInicial*.

La llamada *Tratar* sobre una partición provoca el efecto siguiente:

Tratar una partición significa tratar cada uno de sus grupos. *Tratar* un grupo significa en primer término que este grupo se debe evaluar. En la evaluación se miran características como el tamaño de este grupo, la densidad del grafo que lo representa, etc... Al final de la evaluación se categoriza el grupo:

Si el grupo es definitivo porque tiene un tamaño correcto y una densidad y cohesión aceptables lo pasamos a *ParticiónGlobal* para que se lo guarde en la partición final dónde están todos los grupos definitivos.

Si el grupo es eliminable lo guardamos en la partición basura, siempre a través del módulo *ParticiónGlobal*, dónde se guardan aquellos grupos que por tamaño insuficiente no se consideran como tales.

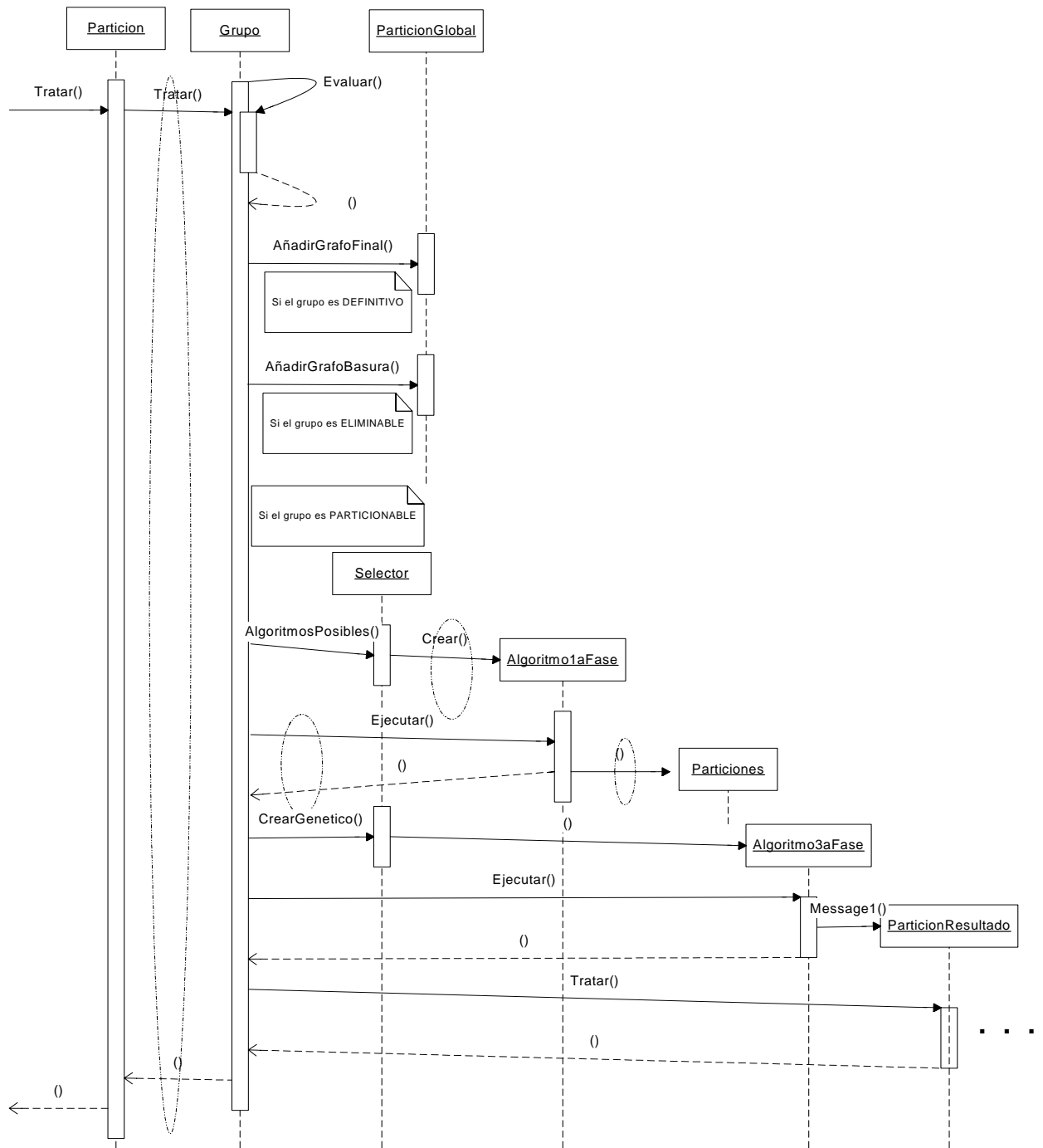


Figura 5.19 Diagrama de secuencia del tratamiento que se le hace a una partición.

Si el grupo es particionable sobretodo porque es muy grande y su grafo poco denso nos disponemos a partir-lo.

Primero llamamos a la clase *Selector* para que nos devuelva una lista con los algoritmos de primera fase que nos es posible ejecutar. Si no tenemos algoritmos disponibles porque ninguno pasa las restricciones reasignamos el grupos como definitivo y lo insertamos en la partición final. Si tenemos algoritmos ejecutamos uno por uno.

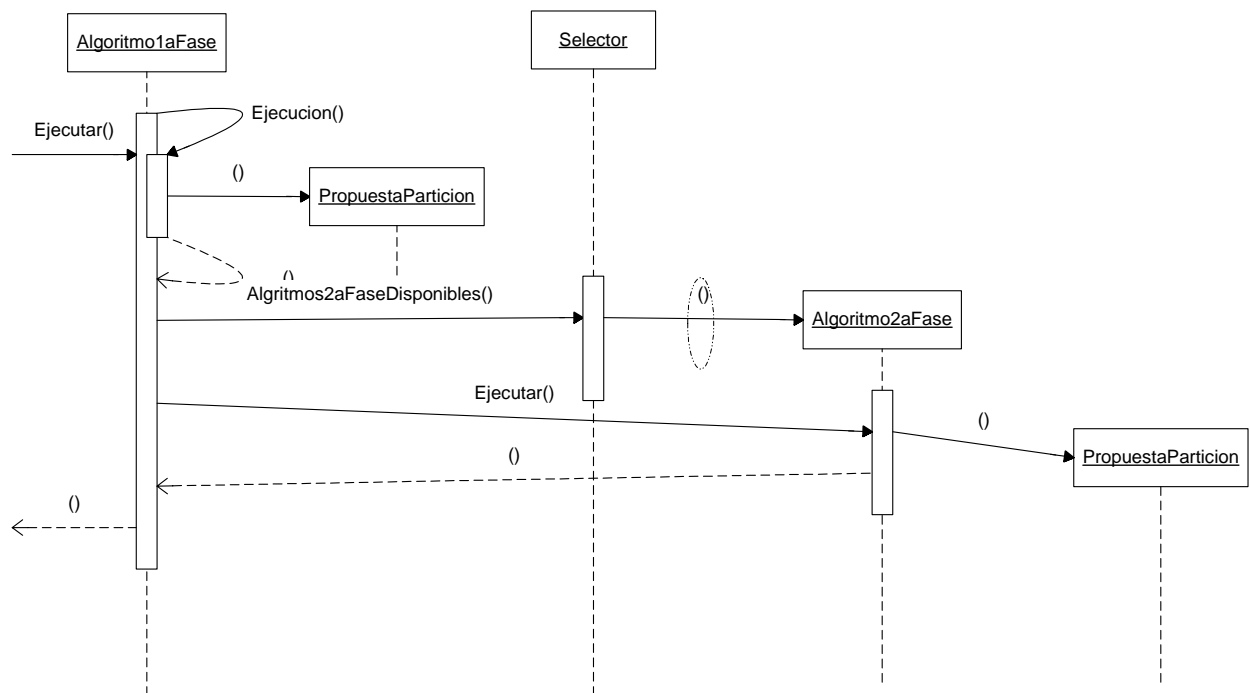


Figura 5.20 Diagrama de secuencia de la ejecución de un algoritmo de primera fase.

La ejecución de un algoritmo de primera fase sigue el esquema de la Figura 5.20. Empezamos ejecutando el algoritmo en si y creando una *PropuestaParticion*. Llamamos a la clase *Selector* para que nos devuelva los algoritmos de segunda fase que están disponibles para ejecutar. Los ejecutamos uno por uno y recogemos las *PropuestaParticion* que nos devuelvan. Este será el resultado de este apartado.

Una vez hemos ejecutado todos los algoritmos de primera fase junto con todas las variaciones de segunda fase recogemos todo el conjunto de *PropuestaPartición* y lo pasamos a un algoritmo de tercera fase, básicamente al genético. Este recombinará las soluciones y nos devolverá la que mayor calidad tenga.

El esquema se repite hasta que no quedan grupos particionables.

5.4.3 Componentes destacables

En este apartado destacaremos las características de algunas de las estructuras utilizadas en la aplicación. Mas concretamente detallaremos con mucha exactitud la implementación de la clase *PropuestaPartición* por su importancia sobretodo para los algoritmos de segunda fase y para que sirva de ejemplo de cómo se han planteado las estructuras de datos en este proyecto. Detallar con tanta exactitud el resto de estructuras sería una tarea muy pesada por lo no mucho importante que resultaría.

5.4.3.1 Grafo

Para implementar el grafo se ha utilizado la implementación propuesta en [4] mediante listas de adyacencia.

5.4.3.2 Cola de prioridad

5.4.3.3 Clase PropuestaParticion

Para ayudar a todas las técnicas algorítmicas se ha implementado una clase *PropuestaPartición* que alberga la partición actual y facilita los movimientos de los elementos entre grupos recalculando las funciones de calidad.

Este módulo debe proporcionar las siguientes operaciones abstractas:

- *Crear*: \rightarrow partición, dado un conjunto de asignaciones de autores a grupos crea la estructura.
- *Mover*: elemento, grupo \rightarrow partición, mueve el elemento al nuevo grupo y actualiza las estructuras. Hay que tener en cuenta que si el elemento a mover formaba un grupo con él solo este grupo desaparece.
- *AñadirGrupo*: grupo \rightarrow partición, añade un grupo vacío a la estructura.
- *MovimientoHecho?* elemento, grupo \rightarrow bool, responde si se ha hecho ya este movimiento.
- *DeshacerMovimiento*: \rightarrow partición, deshace el último movimiento realizado.
- *CalidadPartición*: \rightarrow [0..100] devuelve la calidad de la partición.

- *Calidad*: elemento, grupo \rightarrow [0..100], devuelve la calidad relativa del elemento al grupo.
- *CalcularMovimiento*: elemento, grupo \rightarrow [0..100], evalúa la calidad de la partición una vez realizado el movimiento del elemento al grupo destino pero sin hacer este movimiento.
- *RecuperarMejorSolucion*: \rightarrow partición, se restaura la mejor solución encontrada hasta el momento.
- *Grupo*: elemento \rightarrow grupo, dado un elemento que responda a que grupo pertenece.
- *Afinidad*: elemento, grupo \rightarrow [0..1], devuelve la afinidad del grupo al elemento tal y como hemos definido anteriormente.
- *AfinidadGrupo*: grupo \rightarrow [0..1], devuelve la afinidad media de los elementos del grupo.
- *AfinidadParticion*: \rightarrow [0..1], devuelve la afinidad media de todos los elementos respecto a su grupo asignado.
- *Cohesión*: grupo \rightarrow [0..1], devuelve el grado de cohesión del grupo.
- *CohesiónPartición*: \rightarrow [0..1], devuelve la media de cohesión de todos los grupos.
- *CohesiónElemento*: elemento, grupo \rightarrow [0..1], devuelve la aportación de cohesión del elemento al grupo.
- *Interrelaciones*: \rightarrow [0..1], devuelve la proporción de peso de las aristas que cruzan grupos.
- *Cardinalidad*: grupo \rightarrow N, dado un grupo cuantos elementos contiene.
- *Alternativas*: elemento \rightarrow {(grupo,[0..100])}, devuelve una lista de grupos a los que el elemento es adyacente y su calidad. No se incluye el propio grupo en la lista.
- *MejorAlternativaElemento*: elemento \rightarrow grupo, [0..100], devuelve la alternativa con el valor de calidad más alto.
- *MejorAlternativa*: \rightarrow elemento, grupo, [0..100], devuelve el elemento que tiene la mejor alternativa de todos junto con cuál es el grupo alternativo y su calidad a él.
- *MejoresAlternativas*: \rightarrow {(elemento, grupo, [0..100])}, devuelve los elementos que tienen la mejor alternativa (en caso de empate devuelve más de 1) de todos junto con cuál es el grupo alternativo y su calidad a ellos.
- *PeorElemento*: \rightarrow elemento, grupo, [0..100], devuelve el elemento con peor afinidad al grupo propio junto con su mejor grupo alternativo y su calidad a éste último.

- *SiguienteAlternativa*: \rightarrow elemento, grupo, $[0..100]$, siguiendo un orden establecido va recorriendo el conjunto de soluciones vecinas y nos devuelve la que toca.
- *TieneAlternativas*: \rightarrow bool, indica si aún quedan alternativas por visitar en el conjunto de soluciones.

Para satisfacer estas operaciones el módulo tiene las siguientes estructuras de datos:

- *Grupo*: Vector $[0..n-1]$ con el identificador de grupo de cada uno de los n elementos. Damos servicio a la operación Grupo en coste $O(1)$. Coste espacial: $O(n)$.
- *Afinidad*: Vector $[0..n-1]$ con una tabla de hash en la que cada posición en la que se guardarán las parejas (grupo, afinidad) con los grupos y la afinidad del elemento a estos. Damos servicio a la operación Afinidad en coste $O(1)$. Coste espacial: $O(n \cdot r)$.
- *Alternativas*: Vector $[0..n-1]$ con una cola de prioridad con todos aquellos grupos que son alternativos al elemento junto con su afinidad. En la cabeza de la cola nos encontramos con el grupo alternativo con mayor grado de calidad. Damos servicio a las operaciones Alternativas con coste $O(r)$ de recorrer toda la cola y MejorAlternativaElemento con coste $O(1)$ de consultar la cabeza de la cola. Coste espacial: $O(n \cdot r)$. Cardinalidad: Tabla de hash indexada por identificador de grupo que contiene el número de elementos de cada grupo. Damos servicio a la operación Cardinalidad en coste $O(1)$. Coste espacial: $O(r)$.
- *CohesiónGrupo*: Tabla de hash con el grado de cohesión de cada grupo. Damos servicio a la operación Cohesión en coste $O(1)$. Coste espacial: $O(r)$.
- *AfinidadGrupo*: Tabla de hash con la afinidad media del grupo. Damos servicio a la operación AfinidadGrupo en coste $O(1)$. Coste espacial: $O(r)$.
- *MovimientosHechos*: Vector $[0..n-1]$ con los movimientos realizados por cada elemento. Los movimientos son representados mediante una tabla de hash con los grupos destino. Damos servicio a la operación MovimientoHecho? en coste $O(1)$. Coste espacial: $O(n \cdot r)$.
- *AlternativasPartición*: Cola de prioridad con los elementos que tienen alternativas. En la cola encontramos las parejas elemento, calidad. Dónde aquí calidad es la calidad relativa de la alternativa al grupo propio $calidad(elemento, grupo_alternativo) / calidad(elemento, g(elemento))$. Si la calidad > 1 significa que la alternativa es más buena y que el elemento está supuestamente mal colocado. Si es < 1 la calidad del grupo propio es mayor y por lo tanto está en el lugar correcto. La cola sigue el orden

decreciente de la calidad proporcional, por lo tanto en la cabeza nos encontramos con el elemento que proporcionalmente está más desubicado. Damos servicio junto con la estructura MovimientosHechos a las operaciones: MejorAlternativa con coste $O(1)$, MejoresAlternativas con coste $O(n)$ en el peor de los casos cuando todos los elementos tienen una calidad proporcional idéntica, y TieneAlternativas con coste $O(1)$ que consiste en mirar si la cola está o no vacía. Coste espacial: $O(n)$.

- *AlternativaActual*: Puntero sobre la cola de prioridad AlternativasPartición para mantener la alternativa actual que está siendo tratada con la operación SiguieneAlternativa. Damos servicio a la operación SiguieteAlternativa en coste $O(1)$. Coste espacial: $O(1)$.
- *PeorElemento*: Cola de prioridad con los elementos que tienen alternativas. Al igual que con AlternativasParticion encontramos las parejas elemento,calidad. Pero en este caso la calidad es la calidad del elemento a su grupo principal y sigue un orden creciente, de manera que en la cabeza de la cola encontramos el elemento que está más débilmente colocado. Coste espacial: $O(n)$.
- *Vértice*: Vector $[0..n-1]$ con la correspondencia entre id _ elemento y id_vertice del grafo de coautorías que representa a la partición. Coste espacial: $O(n)$.
- *Elemento*: Tabla de hash indexada por id_vertice y que almacena el correspondiente id_elemento. Es la estructura inversa a Vértice. Coste espacial: $O(n)$.
- *GrupoMejorSolucion*: Vector $[0..n-1]$ con la asignación de grupos para cada elemento correspondiente a la mejor solución encontrada hasta el momento. Coste espacial: $O(n)$.
- *Versión*: Indica la versión de la partición. Se pasa de una versión a otra cada vez que se encuentra una solución que es mejor que la anterior. Coste espacial: $O(1)$.
- *MovimientosMejorSolucion*: Tabla de hash indexada por elemento donde guardaremos los grupos a los que se ha movido en las dos últimas versiones de la partición. El contenido de cada elemento de la tabla de hash es un vector de 5 posiciones. En la primera tenemos un puntero $[0,1]$ que indica si la última versión es la representada por las 2 siguientes posiciones ($=0$) o bien por las 2 últimas ($=1$). Las siguientes posiciones son dos parejas: versión del movimiento, grupo al que se ha movido. Cada vez que se realiza un movimiento se actualiza esta estructura de manera que, si el movimiento es en una nueva versión dentro del elemento incrementamos el puntero módulo 2 y realizamos la actualización de la pareja. Representa una cola con 2

posiciones donde al llenarse se quita el primero que ha entrado. De esta manera tenemos siempre para cada elemento su posición según la última versión y según la penúltima. La versión óptima o bien es la última, con lo cual no hace falta recuperarla, o bien es la penúltima. Al recuperar la penúltima versión pueden pasar 2 cosas: que el elemento haya sido actualizado en versiones anteriores a esta: cogemos la última versión de este elemento; o bien que el elemento haya sido actualizado en la última versión, posterior a la óptima, con lo cual hay que pasar de este cambio y recoger la penúltima actualización si existe. Da servicio a la operación *RecuperarMejorSolución*. Coste espacial: $O(n)$.

- *NumElementos*: Total de elementos en la estructura. Coste espacial: $O(1)$.
- *NumGrupos*: Total de grupos en la estructura \leq *NumElementos*. Coste espacial: $O(1)$.
- *AfinidadParticion*: Afinidad elementos \rightarrow grupo propio media de toda la partición. Coste $O(1)$.
- *Cohesion*: Vector $[0..n-1]$ de tablas de hash con las cohesiones parciales de cada elemento a su grupo adyacente. Coste $O(n \cdot r)$.
- *CohesionParticion*: Cohesión media de los grupos de la partición. Coste espacial: $O(1)$.
- *Intrarrelaciones*: Vector $[0..n-1]$ de tablas de hash con las intrarrelaciones parciales de cada elemento a su grupo adyacente. Coste $O(n \cdot r)$.
- *IntrarrelacionesPartición*: Proporción de peso de las aristas que cruzan grupos. Coste espacial: $O(1)$.
- *Calidad*: Vector $[0..n-1]$ de tablas de hash con las calidades parciales de cada elemento a su grupo adyacente. Coste $O(n \cdot r)$.
- *CalidadParticion*: Medición de la calidad de la partición actual. Coste espacial: $O(1)$.
- *CalidadMejorSolucion*: Medición de la calidad de mejor partición encontrada. Coste: $O(1)$.
- *Grafo*: Puntero a una estructura grafo de coautorías que representan la partición. Coste: $O(1)$.
- *UltimoMovimiento*: Si se ha hecho, contiene el elemento y el grupo destino del último movimiento realizado en la estructura. Coste espacial: $O(1)$.

Asimptóticamente hablando el coste espacial de la estructura sube a $O(nr)$ por las estructuras afinidad, cohesión, intrarrelaciones, calidad, colas de alternativas de cada elemento y por los movimientos hechos para cada elemento. Esto significa que la estructura

ocupa un espacio considerable aunque se justifica por que tiene que estar preparada para mantener una serie de cálculos de una manera dinámica (lo que exige rapidez).

También disponemos de las siguientes subrutinas:

- *CrearParticionComponentes*: Dado un grafo modelo, que no es más que un grafo inducido del grafo de coautorías, calcula las componentes conexas y crea un partición utilizando estas componentes como referencias para los grupos. Tiene que calcular las componentes conexas del grafo modelo con coste $O(m')$ dónde m' son el número de aristas del grafo modelo que siempre serán menos que las aristas del grafo de la partición. Creará un vector de asignaciones que será la entrada de la subrutina *CrearPartición* en coste $O(n) \leq O(m')$. Coste $O(m')$.
- *CrearParticiónAleatoria*: Dados el número de elementos de la partición y el número de grupos deseado crea un vector de `num_elementos` con una asignación aleatoria de cada elemento a uno de los `num_grupos` deseados. Al igual que con la función anterior este vector de asignaciones se pasa como argumento a la función *Crear partición*. Coste $O(\text{num_elementos})=O(n)$.
- *Crear partición*: Dado un vector `[0..n-1]` de asignaciones a grupos inicializa todo el módulo según el siguiente algoritmo:

```

1. procedimiento Crear partición
2.   entrada: modelo es Vector [0..n-1]
3. inicio
4.   NumElementos:=n;
5.   para cada elemento de modelo hacer           O(n) iteraciones
6.     Actualizaremos las asignaciones de la partición:
7.     Grupo[elemento]:= modelo[elemento];
8.     GrupoMejorSolucion[elemento]:= Grupo[elemento];
9.     si Grupo[elemento] es nuevo entonces
10.      Contamos un grupo nuevo e inicializamos sus características
11.      NumGrupos:= NumGrupos+1;
12.      añadir hash Cardinalidad (Grupo[elemento],0);
13.      añadir hash CohesionGrupo (Grupo[elemento],0);
14.      añadir hash AfinidadGrupo (Grupo[elemento],0);
15.     fsi
16.     Incrementamos el número de elementos
17.     incrementar hash Cardinalidad(Grupo[elemento],1);
18.     Inicializamos las características del elemento.
19.     Afinidad[elemento]:=Nuevo Hash;
20.     Cohesion[elemento]:=Nuevo Hash;
21.     Intrarrelaciones[elemento]:=Nuevo Hash;
22.     Calidad[elemento]:=Nuevo Hash;
23.     Alternativas[elemento]:=Nueva Cola;
24.     MovimientosHechos[elemento]:=Nuevo Hash;
25. fpara
26.   sea grafo el grafo de la partición.

```

```

27. Recorreremos todas las aristas del grafo para actualizar las afinidades, las intrarrelaciones y el grado de
28. cohesión de cada grupo.
29. peso_grafo:=suma del peso de todas las aristas del grafo.
30. para cada elemento de modelo hacer  $O(n)$  iteraciones
31.   pesos_elemento:=suma de los pesos de todas las aristas adyacentes a elemento.
32.   para cada adyacente de elemento en el grafo hacer  $O(m/n)$  iteraciones
33.     peso:=peso adyacencia entre elemento y adyacente en el grafo.
34.     pesos_adyacente:=suma de los pesos de todas las aristas adyacentes a adyacente.
35.     Actualizamos incrementalmente las afinidades de los dos vertices al grupo adyacente.
36.     incrementar hash Afinidad[elemento] (Grupo[adyacente], peso/pesos_elemento);
37.     incrementar hash Afinidad[adyacente] (Grupo[elemento], peso/pesos_adyacente);
38.     incrementar hash Cohesion[elemento] (Grupo[adyacente], pesoa/Cardinalidad→Grupo[adyacente]b);
39.     incrementar hash Cohesion[adyacente] (Grupo[elemento], pesoa/Cardinalidad→Grupo[elemento]b);
40.     incrementar hash Intrarrelaciones[elemento] (Grupo[adyacente], peso/peso_grafo);
41.     incrementar hash Intrarrelaciones[adyacente] (Grupo[elemento], peso/peso_grafo);
42.     modificar hash Calidad[elemento](Grupo[adyacente], CalcularCalidadElemento(elemento,
Grupo[adyacente]));
43.     modificar hash Calidad[adyacente](Grupo[elemento], CalcularCalidadElemento(adyacente,
Grupo[elemento]));
44.     si Grupo[elemento]= Grupo[adyacente] entonces
45.       Si estamos en una arista interna en el grupo actualizamos la cohesión del grupo y las afinidades
46.       medias
47.       incrementar hash CohesionGrupo (Grupo[elemento], pesoa/Cardinalidad→Grupo[elemento]b);
48.       CohesionParticion+:= (pesoa/Cardinalidad→ Grupo[elemento]b)· Cardinalidad→
Grupo[elemento]/NumElementos;
49.       incrementar hash AfinidadGrupo (Grupo[adyacente],
50.         (peso/pesos_elemento + peso/pesos_adyacente)/ Cardinalidad→Grupo[elemento]);
51.       AfinidadParticion+:= (peso/pesos_elemento + peso/pesos_adyacente)/ NumElementos;
52.       Intrarrelaciones:= Intrarrelaciones+ peso/peso_grafo;
53.     fsi
54.   fpara
55. fpara

56. Una vez calculadas definitivamente las afinidades vamos a inicializar las colas con las alternativas.
57. sea elementos Lista de pares vacia; contendrá cada elemento con su afinidad proporcional a la mejor
58. alternativa.
59. sea elementos_locales Lista de pares vacia; contendrá cada elemento con su afinidad al grupo propio
60. para cada elemento de modelo hacer  $O(n)$  iteraciones
61.   sea grupos_alternativos Lista de pares vacia;
62.   para cada grupo de grupos_afines a elemento hacer  $O(r)$  iteraciones
63.     si grupo=Grupo[elemento] entonces si es el grupo propio lo guardamos en elementos
64.       (aquí sólo entraremos una vez en todo el para_cada)
65.       añadir elementos_locales (elemento,Calidad[elemento]→grupo);
66.     sino si no es que es una alternativa
67.       añadir grupos_alternativos (grupo, Calidad[elemento]→grupo);
68.     fsi
69.   fpara
70.   si grupos_alternativos no es vacio entonces
71.     Convertir vector grupos_alternativos en cola_prioridad Alternativas[elemento];  $O(r)$ 
72.     (mejor_grupo,calidad) := consultar cabeza de Alternativas[elemento] ;
73.     calidad_relativa:=calidad/Calidad[elemento]→Grupo[elemento];
74.     añadir elementos (elemento,calidad_relativa);
75.   fsi
76. fpara  $O(n·r) \leq O(m)$ , sumando todos los grupos afines a todos los elementos en el peor de los casos será
77. equivalente al total de aristas del grafo.
78. Convertir Lista elementos en cola_prioridad AlternativasParticion;  $O(n)$ 
79. Convertir Lista elementos_locales en cola_prioridad PeorElemento;  $O(n)$ 
80. CalcularCalidad;
81. fin

```

El módulo debe estar ya creado con todas las estructuras vacías.

El coste de esta operación es equivalente a $O(m)$. Analizando todos los pasos: en el bucle de las líneas 5 a 25 se hacen n iteraciones y a cada paso se actualizan vectores y hashes

en un coste constante: coste total del bucle $O(n)$. De la línea 60 a la 76 encontramos otro bucle que recorre los n elementos, dentro del cual para cada elemento se recorren los grupos adyacentes, en el peor de los casos todos r . El coste de este par de bucles anidados es de $O(n \cdot r)$ que se puede limitar a $O(m)$ ya que en el peor de los casos, cuando cada elemento forma un grupo, el número de adyacentes de cada uno de los elementos será el número de aristas del grafo. Una vez recorrido el bucle de los grupos afines tenemos una lista (o un vector) con r grupos, como máximo, que deberemos convertir a cola de prioridad. Tal operación tiene un coste lineal tal y como se demuestra en [4], con lo que volvemos a $O(n \cdot r)$ equivalente a $O(m)$. El bucle de la 60 a la 76 queda saldado con un coste final de $O(m)$. Salidos de este bucle tenemos que convertir 2 listas de n elementos a colas de prioridad con un coste computacional de $O(n)$ para cada una. Los costes de la operación son $O(m)$ para cada bucle y $O(n)$ para cada cola de prioridad. Como $O(m) \geq O(n)$ nos quedamos con $O(m)$.

- *MoverElemento*: Dado un elemento a mover y un grupo destino modifica el grupo propio del elemento al grupo destino especificado. Sigue el siguiente algoritmo. Sea a el número de elementos adyacentes al elemento a mover:

```

82. procedimiento MoverElemento
83.   entrada: elemento, grupo_destino
84. inicio
85.   grupo_origen:=Grupo[elemento];
86.   Vamos a realizar el movimiento:
87.   Grupo[elemento]:=grupo_destino;
88.   decrementar hash Cardinalidad(grupo_origen,1);
89.   incrementar hash Cardinalidad(grupo_destino,1);
90.   Vamos a modificar las afinidades medias de los grupos origen y destino.
91.   si Cardinalidad→grupo_origen > 0 entonces
92.     modificar hash AfinidadGrupo(grupo_origen, (AfinidadGrupo→grupo_origen*(Cardinalidad→grupo_origen+1)-
Afinidad[elemento]→grupo_origen))/ Cardinalidad→grupo_origen;
93.   sino
94.     modificar hash AfinidadGrupo(grupo_origen,0);
95.   fsi
96.   modificar hash AfinidadGrupo(grupo_destino, (AfinidadGrupo→grupo_destino*(Cardinalidad→grupo_destino-
1)+Afinidad[elemento]→grupo_destino))/ Cardinalidad→grupo_origen;
97.   Así como la afinidad media de toda la partición.
98.   AfinidadParticion- := Afinidad[elemento]→grupo_origen/n;
99.   AfinidadParticion+:= Afinidad[elemento]→grupo_origen/n;
100.  Actualizaremos las afinidades de los adyacentes al vértice movido.
101.  para cada adyacente de elemento en el grafo hacer O(a) iteraciones
102.    peso:=peso adyacencia entre elemento y adyacente en el grafo.
103.    pesos_adyacente:=suma de los pesos de todas las aristas adyacentes a adyacente.
104.    si Grupo[adyacente]=grupo_origen entonces
105.      La arista iba hacia el origen, por lo tanto pasará a ser intergrupo.
106.      Interrelaciones:= Interrelaciones- peso/suma_pesos_grafo;
107.      La afinidad media de la partición y del grupo origen se decrementaran al perderse una arista del grupo
origen.
108.      decrementar hash AfinidadGrupo (grupo_origen,
109.      (peso/pesos_adyacente)/ Cardinalidad→ grupo_origen);
110.      AfinidadParticion:= AfinidadParticion-(peso/pesos_adyacente)/ NumElementos;
111.      Guardaremos el peso quitado potenciado para el cálculo de la cohesión
112.      peso_origen:=pesoa;
113.    fsi

```

114. **si** Grupo[adyacente]=grupo_destino **entonces**
115. *La arista iba hacia el destino, por lo tanto pasará a ser intragrupo.*
116. Interrelaciones:= Interrelaciones+ peso/suma_pesos_grafo;
117. *La afinidad media de la partición y del grupo origen se incrementaran al ganarse una arista del grupo destino.*
118. incrementar hash AfinidadGrupo (grupo_destino,
119. (peso/pesos_adyacente)/ Cardinalidad→ grupo_destino);
120. AfinidadParticion:= AfinidadParticion +(peso/pesos_adyacente)/ NumElementos;
121. *Guardaremos el peso añadido potenciado para el cálculo de la cohesión*
122. peso_destino:=pesoa;
123. **fsi**
124. *La adyacencia pasa del origen al destino, por lo tanto la proporción de afinidad con el grupo origen pasa al grupo destino. Al igual que la aportación de cohesión.*
125. decrementar hash Afinidad[adyacente] (grupo_origen, peso/peso_adyacente);
126. incrementar hash Afinidad[adyacente] (grupo_destino, peso/peso_adyacente);
127. decrementar hash Intrarrelaciones[adyacente] (grupo_origen, peso/peso_grafo);
128. incrementar hash Intrarrelaciones[adyacente] (grupo_destino, peso/peso_grafo);
129. modificar hash Cohesion[adyacente](grupo_origen, (CohesionElemento[adyacente] → grupo_origen · (Cardinalidad→ grupo_origen +1) b - pesoa) / Cardinalidad→ grupo_origenb);
130. modificar hash Cohesion[adyacente](grupo_destino, (CohesionElemento[adyacente] → grupo_destino · (Cardinalidad→ grupo_destino -1) b + pesoa) / Cardinalidad→ grupo_destinob);
131. modificar hash Calidad[elemento](Grupo[adyacente], CalcularCalidadElemento(elemento, Grupo[adyacente]));
132. modificar hash Calidad[adyacente](Grupo[elemento], CalcularCalidadElemento(adyacente, Grupo[elemento]));
133. **fpara**
134. *El grupo destino pasa de ser una alternativa a ser el grupo propio y el origen pasa de ser propio a alternativa. O(log n). Esta actualización pasa a las colas generales de AlternativaPartición y de MejorCohesión.*
135. ExtraerAlternativa(elemento,grupo_destino);
136. InsertarAlternativa(elemento,grupo_origen);
137. Actualizamos el elemento en la cola de PeorElemento. O(log n).
138. Modificar en cola PeorElemento (elemento, Afinidad[elemento] → grupo_destino);
139. *Ahora, calculadas las afinidades nos dispondremos a actualizar las colas de alternativas de los adyacentes.*
140. **para cada** adyacente de elemento en el grafo **hacer** O(a) iteraciones
141. *Por si se ha modificado la afinidad propia del adyacente a su grupo actualizamos esta a la cola de PeoresElementos. O(log n).*
142. Modificar en cola PeorElemento (adyacente, Afinidad[adyacente] → Grupo[adyacente]);
143. *Si el adyacente tenía al grupo origen como alternativa y este aun existirá después del movimiento trasladamos la nueva afinidad a la cola de alternativas, si no lo eliminamos de la alternativa. O(log n).*
144. **si** (Cardinalidad→ grupo_origen > 0) $\dot{\cup}$ (grupo_origen $\dot{\cup}$ Grupo[adyacente]) **entonces**
145. InsertarAlternativa(adyacente,grupo_origen); O(log n).
146. **sino**
147. ExtraerAlternativa(adyacente,grupo_origen); O(log n).
148. **fsi**
149. *Si el adyacente tenía al destino como alternativa trasladamos la modificación a la cola. O(log n).*
150. **si** grupo_destino $\dot{\cup}$ Grupo[adyacente] **entonces**
151. InsertarAlternativa(adyacente,grupo_destino); O(log n).
152. **fsi**
153. **fpara**
154. *Vamos ahora a ajustar la cohesión. Primero nos guardamos la antigua:*
155. cohesion_origen_antigua:= CohesionGrupo→grupo_origen;
156. cohesion_destino_antigua:= CohesionGrupo→grupo_destino;
157. *Procedemos a hacer la modificación de la cohesión incrementalmente, teniendo en cuenta que si el grupo origen desaparece no se calcula la cohesión y ya puede ser eliminado.*
158. **si** (Cardinalidad→ grupo_origen > 0) **entonces**
159. modificar hash CohesionGrupo (grupo_origen, (CohesionGrupo→grupo_origen·(Cardinalidad→ grupo_origen+1)b- peso_origen)/ Cardinalidad→ grupo_origen b);
160. **sino**
161. EliminarGrupo(grupo_origen);
162. **fsi**
163. modificar hash CohesionGrupo (grupo_origen, (CohesionGrupo →grupo_destino·(Cardinalidad→ grupo_destino-1)b- peso_destino)/ Cardinalidad→ grupo_destino b);
164. *Modificamos la cohesión global de la partición según las diferencias con la antigua.*
165. CohesionParticion+:= (CohesionGrupo→grupo_origen·Cardinalidad→ grupo_origen - cohesion_origen_antigua·(Cardinalidad→ grupo_origen +1) + CohesionGrupo→grupo_destino·Cardinalidad→ grupo_destino - cohesion_destino_antigua·Cardinalidad→ grupo_destino)/ NumElementos;
166. *Recalculamos la calidad de la partición.*
167. CalcularCalidad;
168. *Si es necesario actualizamos la mejor solución. O(1).*

169. ActualizarMejorSolucion;
 170. *Guardamos el movimiento por si hay que deshacerlo.*
 171. UltimoMovimiento:=(elemento,grupo_origen);
 172. *Lo marcamos como movimiento hecho.*
 173. Insertar hash MovimientosHechos[elemento] (grupo_destino,SI);
 174.**fin**

El coste de este algoritmo pretende aproximarse la más posible a $O(a)$ dónde a es el número de adyacentes en el grafo del elemento a mover. Si lo analizamos con detenimiento veremos que contiene dos bucles que recorren los adyacentes del elemento en a iteraciones en las líneas 101-133 y 140-143. Pero en el segundo *para_cada* se actualizan las colas de prioridad de la estructura, operación que no es constante y que nos fastidia en el intento de conseguir un $O(a)$. La modificación de la cola de peores elementos que tiene n elementos se realiza en tiempo $O(\log n)$. Las colas con las alternativas tienen tantos elementos como grupos afines al elemento menos el propio, por lo tanto su modificación cuesta $O(\log \text{num_grupos_alternativos}) \leq O(\log a) \leq O(\log r) \leq O(\log n)$ y, además esta modificación se propaga a la cola de mejores alternativas de toda la partición que contiene n elementos, por lo tanto hay que añadirle un coste de $O(\log n)$. Puede que en algún algoritmo no nos sea necesario la actualización de la cola de mejores alternativas ni la de peores elementos. En tal caso el coste del segundo bucle se mantendría en $O(\log r)$. Si nos interesa mantener las colas generales sube a $O(a \cdot \log n)$. Quedando como coste final de la subrutina $O(a \cdot \log n)$ con la utilización de colas globales y $O(a \cdot \log r)$ sin la utilización de dichas colas.

- *CalcularMovimiento*: Simula un movimiento del elemento pasado devolviendo la calidad de la partición resultante. El movimiento no se realiza y al final de la ejecución la partición permanece en el estado del inicio. Básicamente realiza los mismos cálculos que con la operación *MoverElemento*, pero sólo se actualizan los indicadores generales de toda la partición. No se actualizan las colas de prioridad, por lo tanto no se realiza el segundo bucle de la operación anterior lo que mantiene el coste de esta operación en $O(a)$.
- *MoverElementoAleatorio*: Mueve el elemento indicado a un grupo alternativo elegido al azar. Coste: el mismo que *Mover elemento* $O(a \cdot \log n)$.
- *DeshacerMovimiento*: Mueve el elemento al grupo indicados por *UltimoMovimiento*, si es que este había sido inicializado. Coste: $O(a \cdot \log n)$.
- *MejorMovimiento*: Recorre todas las alternativas de movimientos de la cola *Alternativas Partición* calculando cada uno de los movimientos y devolviendo el que más calidad haya generado. Devuelve la terna (elemento, grupo, calidad). Coste: n veces *CalcularMovimiento*: $O(n \cdot a)$.

- *ActualizarMejorSolucion*: Dado un movimiento acabado de realizar (elemento, grupo) se lo guarda en la estructura MovimientosMejorSolucion para poder recuperarla después.

procedimiento ActualizarMejorSolucion

entrada: elemento, grupo_destino

inicio

si no existe MovimientosMejorSolucion → elemento **entonces**

añadir hash MovimientosMejorSolucion (elemento, [0,Version,grupo,0,0]);

sino

vector:= MovimientosMejorSolucion → elemento

puntero:=vector [0];

ultima_version:=vector [puntero*2 + 1];

Si vamos a actualizar un movimiento de una nueva versión para el elemento avanzamos en la cola y actualizamos esta versión.

si ultima_version <> Versión **entonces**

puntero:=(puntero+1) mod 2;

vector [puntero*2 + 1] := Versión;

fsi

Hacemos la actualización.

vector [puntero*2 + 2] := grupo_destino;

modificar hash MovimientosMejorSolucion (elemento,vector);

fsi

Si estamos en la máxima calidad actualizamos el máximo y cambiamos de versión.

si Calidad > CalidadMejorSolucion **entonces**

Versión:=Version+1;

CalidadMejorSolucion:=Calidad;

fsi

fin

El coste del algoritmo es constante $O(1)$ tal y como pretendíamos.

- *RecuperarMejorSolución*: Actualiza la partición a la mejor solución encontrada. Para ello hace uso de los movimientos guardados en la estructura MovimientosMejorSolucion según el algoritmo:

procedimiento RecuperarMejorSolucion

inicio

Si la calidad de la solución actual es la mejor no hace falta recuperarla, ya la tenemos.

si Calidad < CalidadMejorSolucion **entonces**

Recorreremos la tabla de hash con los movimientos. $O(n)$ iteraciones.

para cada elemento de claves hash MovimientosMejorSolucion **hacer**

vector:= MovimientosMejorSolucion → elemento;

puntero:=vector [0];

Si el último movimiento es de la versión actual no nos interesa porque por la condición anterior seguro que no es el óptimo, nos disponemos a analizar el otro movimiento.

si vector[puntero*2+1] = Versión **entonces**

puntero:=(puntero+1) mod 2;

fsi

Si el movimiento existe lo realizamos

si vector[puntero*2+2] <> 0 **entonces**

GrupoMejorSolucion[elemento]:=vector[puntero*2+2];

fsi

fpara

Finalmente a partir del vector de asignaciones GrupoMejorSolucion rehacemos la partición. $O(m)$

Crear partición(GrupoMejorSolucion);

fsi

fin

Por culpa de rehacer la partición al final de esta subrutina recalculando todos los parámetros el coste de la operación sube a $O(m)$.

- *TieneAlternativas*: Mira la cola de prioridad de alternativas y si no está vacía devuelve SI, de lo contrario devuelve NO. Coste: $O(1)$.
- *MejorAlternativa*: Consulta de la cola de AlternativasPartición el elemento más propenso a moverse y de la cola Alternativas de este elemento nos devuelve el grupo al que se moverá. Devuelve la terna (elemento, grupo, calidad nueva). Coste $O(1)$, 2 consultas a las cabezas de cola.
- *MejoresAlternativas*: En caso de empate en la cola de alternativas partición devuelve todos los elementos del empate con el mismo formato que MejorAlternativa pero ahora con una lista. Teniendo en cuenta que todos los elementos pueden estar empatados en el peor de los casos el coste de esta operación es de $O(n)$.
- *SiguienteAlternativa*: Va recorriendo linealmente la cola de AlternativasPartición y actualizando el puntero AlternativaActual. Solo da un avance en cada llamada. Devuelve la alternativa actual en el mismo formato que MejorAlternativa y avanza el puntero. Cuando hacemos un movimiento AlternativaActual vuelve al inicio. Coste: $O(1)$.
- *PeorElemento*: Consulta la cola de prioridad de PeorElemento con el elemento con peor afinidad local y nos devuelve el grupo alternativo mejor y su afinidad a este. Devuelve la terna (elemento, grupo, afinidad nueva). Coste $O(1)$, 2 consultas a las cabezas de cola.

6 Conclusiones

En este trabajo se ha abordado el problema de la búsqueda de grupos de investigación a través de su producción bibliográfica desde una óptica puramente algorítmica. No se han interpretado los resultados bibliométricamente sino que se han discutido metodologías para sacar automáticamente los grupos.

Se ha definido formalmente el problema especificando la naturaleza de los datos de entrada y el formato de los de salida. Concretamente se ha propuesto representar la producción bibliográfica mediante un grafo de coautorías. Además hemos analizado los inconvenientes que encontramos con las fuentes de datos disponibles.

Se ha especificado detalladamente una metodología para la evaluación de la calidad de una solución a nuestro problema. Esta especificación es clara y concisa y se ha detallado exhaustivamente un módulo que implementa gran parte de esta metodología.

Se han propuesto un total de 11 algoritmos distintos clasificados en tres grupos dependiendo de en qué fase se ejecutan. Estos algoritmos han sido diseñados e implementados. El análisis de la efectividad de estos algoritmos se encuentra aun a un nivel experimental. Sólo se han podido comparar utilizando la metodología automática aquí propuesta pero hace falta una evaluación de un experto para ratificar que tanto los algoritmos como la propia metodología son válidos.

Finalmente también se ha diseñado e implementado una aplicación que es capaz de cargar la información bibliográfica al sistema y controlar la ejecución de los algoritmos.

7 Discusión

Queda muy claro que la solución para el problema de la búsqueda de grupos de investigación pasa indiscutiblemente por la representación a través del grafo de coautorías. Cualquier algoritmo que se proponga encontrar los grupos de investigación debería tener en cuenta la valiosa información que este tipo de estructura nos aporta.

Las técnicas utilizadas para tratar este grafo pueden ser muy variadas. Aunque aquí se han propuesto un buen puñado de ideas a cualquiera se le pueden ocurrir infinitas más con infinitas variantes pero seguro que en todas ellas detrás encontraremos las coautorías representadas por el grafo.

Si el grafo de coautorías es idóneo para la representación de estas relaciones no lo son las fuentes de datos que lo alimentan. La fragilidad que nos lleva identificar los autores por su nombre es excesiva y la gran cantidad de errores detectada sólo en la producción bibliográfica de Cataluña es decepcionante. Imaginemos la cantidad de errores que podremos encontrar mezclando los autores de toda España o del Mundo.

Si ya de por sí la comunidad científica catalana está fuertemente conectada por ser Cataluña un país ‘no muy grande’, los nombres multiautorados juntan aun más el conjunto. Esto lleva a que los algoritmos tengan que ser muy restrictivos para poder particionar toda esta masa de autores y se lleven grupos buenos por el camino. Esta dificultad evidente hace que muchas de las ideas de métodos que a uno se le acuden luego en la realidad no fructifiquen o bien que se tenga que ser muy retocados para que lleguen a hacer algo.

7.1 Planes de futuro

El proyecto de búsqueda automatizada de grupos no está acabado ni mucho menos. Se ha cerrado en parte para que pueda ser presentado como Proyecto de Fin de Carrera al considerarse que el volumen e interés del trabajo realizado ya son suficientes.

El primer objetivo es probar mucho más las técnicas algorítmicas propuestas. Descartar las que definitivamente no dan resultados efectivos. Proponer más técnicas, implementarlas y añadir a la secuencia de diseño, implementación, prueba y evaluación.

Hay que mejorar la aplicación para que sea más eficiente a la hora de ejecutar los algoritmos de manera que no tenga que probar todas las técnicas sino que pueda predecir cual va a ser el mejor algoritmo para partir unos datos determinados.

Hasta ahora se ha ejecutado la aplicación sobre los datos disponibles de Cataluña y se ha hecho a modo de pruebas. Algunos resultados se han hecho públicos a ciertos investigadores y han tenido una aceptación sensiblemente positiva. Cuando el sistema esté más desarrollado y probado se utilizará para hacer informes bibliométricos pudiendo fijar los grupos de investigación como elementos nuevos. Existen planes de utilizar estos algoritmos sobre toda la información bibliográfica de España para contribuir a la realización de un mapa científico español en el cual podrá aparecer información a nivel de grupo de investigación, nivel hasta ahora no disponible.

8 Agradecimientos

A Conrado Martínez por haber aceptado la dirección de este proyecto y mostrado mucho interés en él.

Al IMIM por confiarme este proyecto. A su director Jordi Camí por creer en mí. A los compañeros Joan Marc Carbó y Lluís Coma porque sin ellos el proyecto no habría sido posible.

A Marta Álvarez por haberme metido en este enrollo.

A mi familia y amigos por haberme apoyado y haberme dado consejos útiles.

A los profesores de la FIB por haberme aportado los conocimientos necesarios para hacer este proyecto.

A todos ellos gracias.

9 Bibliografía

- [1] Aarts E, Lenstra JK. Local search in combinatorial optimization. John Wiley & Sons, 1997.
- [2] Boettcher S, Percus AG. Extremal optimization for graph partitioning. Phys Rev Lett (in press), cond-mat/0104214.
- [3] Boettcher S, Percus AG. Optimization with extremal dynamics. Phys Rev Lett (in press), cond-mat/0010337.
- [4] Cormen TH, Leiserson CE, Rivest RL. Introduction to algorithms. MIT Press, 1990.
- [5] Kaufman L, Rousseeuw PJ. Finding Groups in Data. John Wiley & Sons, New York, 1990.
- [6] Melanie M. An introduction to Genetic Algorithms. MIT Press, 1996.
- [7] Michalewicz Z, Fogel DB. How to solve it: modern heuristics. Springer, 2000.
- [8] Zulueta MA, Cabrero A, Bordons M. Identificación y estudio de grupos de investigación a través de indicadores bibliométricos. Rev Esp Doc Cient 1999; 23(3):333-347