Teoría Avanzada de la Computación

Grado en Ingeniería Informática

Curso 2021-2022

Práctica 3

Algoritmos aplicados a Optimización Combinatoria

Introducción

En esta práctica se propone estudiar algunos algoritmos que se aplican a resolver problemas de clase NP-C y relacionados con optimización combinatoria, en concreto el problema de la **k-colorabilidad** en grafos.

Se pide aplicar los conocimientos adquiridos en Magistral, en especial Análisis de Coste Computacional, y también algunas cuestiones de Complejidad Computacional.

El problema seleccionado es la **k-colorabilidad en grafos**, uno de los 22 problemas originales identificados (en su versión de problema de decisión) pertenecientes a la clase *NP-Complete*. Lo podremos relacionar con los problemas de Satisfacibilidad (caso **3-Sat**) y **k-Clique**.

Definiciones

Dado un grafo G de n nodos y a aristas no dirigidas:

Problema k-Colorabilidad y Número Cromático de un grafo

La definición **k-Colorable** implica que exista una asignación de un color entre k posibles a cada uno de los n nodos de forma que no haya dos nodos adyacentes en G que tengan el mismo color. La versión de decisión del problema determina si es posible o no realizar dicha asignación.

El **número cromático** del grafo será el valor máximo de k para G que permita realizar alguna asignación de colores sin conflictos.

Problemas k-Clique y max-Clique

Se define como **k-Clique** aquel subgrafo completo de k nodos que se pueda localizar dentro G. Su variante de decisión consiste en determinar si existe o no una k-Clique.

El problema **max-Clique** consiste en determinar el valor máximo de k para el cuál se pueda hallar una k-Clique en G.

Relación entre Número Cromático y max-Clique

Es posible demostrar que el max-Clique define el valor del Número Cromático. Intuitivamente, si dentro de G existe un (k+1)-Clique, la respuesta a la k-Colorabilidad será: no.

Trabajo a realizar

Abordar los problemas de k-Colorabilidad para grafos aleatorios con k=2 y k=3, su reducción al problema k-Clique y las transiciones de fase P/NP en función de ciertas características de los grafos. Para ello necesitaremos implementar algunos métodos de resolución:

1. Estudiad con detenimiento el enunciado y los Anexos I a III.

   Probad el programa proporcionado para analizar la conectividad de grafos. El problema de determinar el número de subgrafos inconexos (*clústers*) dentro del grafo se puede reducir al problema PATH(p,q) que determina si existe algún camino en el grafo que conecte dos nodos dados p y q. De este problema se sabe que pertenece a la clase P, con diversos algoritmos de resolución con complejidad $O(n^2)$ - $O(n^3)$. Por ejemplo, el algoritmo de Warshall permite calcular el cierre transitivo de un grafo (representado cómo matriz) para determinar si se trata de un grafo completo. En este caso se utiliza una función **`propagar_marcaR`** que no está especialmente optimizada en cuanto a eficiencia. Se basa en la propagación de una marca a través de nodos adyacentes.

   a. Estudiad su complejidad analítica aproximada
   b. Estudiad su complejidad de forma empírica en base a tiempos y/o testigos, usando grafos de diversos tamaños y conectividades..
   c. ¿Podéis determinar en qué circunstancias aparecen las instancias de peor caso?
   d. ¿Véis opciones para mejorar la eficiencia del proceso de contar *clústers*?

2. Problema de k-colorabilidad para k=3.
   a. Implementad un método de búsqueda exhaustiva básico (consultad el Anexo 1, y en particular el apartado sobre métodos de búsqueda).
   b. Determinad hasta qué tamaños de grafos aleatorios es posible realizar la búsqueda de todas las soluciones en un tiempo razonable (no más de 1 hora, por ejemplo).
   c. Calculad el coste computacional, de forma analítica y empírica en base a tiempos y a testigos.
   d. Calculad el número de soluciones existentes para una instancia dada. Se entiende que en el espacio de asignaciones colores-nodos sólo una fracción estará libre de conflictos.
   e. Determinad la influencia de la densidad de aristas en el grafo en los resultados obtenidos en el apartado anterior (para más detalle repasad el concepto de Transición de Fase del Anexo I y los resultados del Anexo II).

f. Implementad un método basado en *Backtracking* para obtener una solución. Calculad el coste computacional del método y hasta qué punto permite mejorar la eficiencia respecto al método en a).
g. Incluid alguna heurística y poda que permita mejorar el rendimiento del método basado en *Backtracking*. Repetid los apartados b, c y e.

3. Problema de k-colorabilidad para k=2.
    a. Utilizad un método de búsqueda basado en *Backtracking* con podas y heurísticas que garanticen la solución en caso de existir.
    b. Determinad hasta qué tamaños de grafos es posible realizar la búsqueda de una solución en un tiempo razonable (no más de 1 hora, por ejemplo).
    c. Calculad el coste computacional, de forma analítica y empírica en base a tiempos y a testigos.

4. Reducción del problema de k-Colorabilidad al problema k-Clique y max-Clique. Determinad si dada una instancia de un grafo de tamaño n y con a aristas, ésta es k-colorable buscando el max-Clique o un (k+1)-Clique (problema de decisión).
    a. Emplead un método basado en Backtracking con heurísticas y podas que garanticen una solución en caso de existir.
    b. Calculad el coste computacional, de forma analítica, y empírica en base a tiempos y a testigos.

5. Realizad un breve estudio teórico relacionando conceptos vistos en magistral.
    a. A qué clase (P, NP, NP-Complete, NP-Hard) pertenecen los problemas de k-coloreado y k-clique, y por qué. Razonadlo en base a las definiciones disponibles.
    b. ¿Qué implicaciones tiene la posibilidad de resolver los problemas asociados a k-coloreado con algoritmos de k-clique y a la inversa?

Debéis incluir siempre los desarrollos, tablas y gráficas que sirvan para apoyar vuestros análisis y conclusiones.

Recomendaciones

Programas:

En web disponéis de numerosos recursos para obtener los algoritmos ya programados. Pero tened en cuenta:

- Es frecuente que tengan errores.
- El hecho de que un programa figure como implementación de cierto algoritmo u heurística, no significa que se le pueda atribuir la complejidad que deriva de su propuesta original.
- Los algoritmos deben ser sencillos. Una heurística muy elaborada también será compleja de analizar.

Es importante que programéis ciertas funciones para facilitar el desarrollo de los experimentos. En el anexo III disponéis de un programa en C que desarrolla algunas de estas funciones, entre otras:

- generación de un grafo con n nodos y a aristas
- supresión de aristas escogidas al azar
- impresión del grafo y su conectividad para verificar que es correcto
- contar conflictos

Es importante que os aseguréis que el código y ficheros de prueba que entregáis puede funcionar correctamente en cualquier ordenador. Eso quiere decir que evitéis usar instalaciones complejas de librerías, etc.

En caso de programas en Python, se recomienda:

- **Entregar el código en Notebooks para Google Colab. Estos también pueden funcionar como Jupyter Notebooks en ordenadores personales. Comprobad que les funcionan correctamente a otros usuarios que no comparten vuestro drive.**
- **Programar algunas funciones críticas en C o C++ para evitar la ineficiencia del lenguaje Python. Podéis consultar un documento de ayuda en AG respecto a cómo combinar Python y C.**

Memoria:

- Incluid los diseños, tablas de pasos y de diferencias, desarrollos de las ecuaciones analíticas, y gráficas. Son necesarias para demostrar vuestros razonamientos respecto a lo que obtenéis como resultado.
- Revisad las escalas de vuestras gráficas y que son coherentes.
- Poner el polinomio directamente, sin mostrar cómo se obtiene no aporta mucho. Puede ser verdad o no. Es tarea vuestra mostrar cómo se llega a él.
- Los resultados obtenidos mediante regresiones automáticas no sirven en nuestro contexto. En nuestro caso necesitamos polinomios y exponenciales con bases y exponentes enteros.

Entrega

Fechas de Entrega

Dispondréis de varias fechas de entrega:

- Entrega 1, viernes 29-04-2022, ejercicio 1, comienzo del 2.
- Entrega 2, viernes 6-05-2022, ejercicios 2 y 3.
- Entrega **Final**, viernes 13-05-2022.

Entrega FINAL:

Debéis entregar un archivo zip de nombre
OPT_grupo_Apellido1_Apellido2_Apellido3.zip, donde

- ApellidoK s el primer apellido de cada alumno, puestos en orden alfabético.
- grupo es el grupo de trabajo, no el grupo reducido.

Con el contenido:

- Un documento de Nombre:
  Memoria_OPT_grupo_Apellido1_Apellido2_Apellido3,pdf
- Imágenes, siempre en formato PNG.
- Hojas de cálculo con datos y gráficos
- Scripts
- Código implementado y ficheros de prueba.
- Notebook de Colab o Jupyter.

Bibliografía

(disponible en AG)

[Cheeseman 1991] P, Cheeseman,, B. Kanefsky, and W. M. Taylor: “Where the really hard problems are”. *Proc. 12 IJCAI* 1991, pp. 331–337.

[Hayes 2003] B. Hayes: “Computing Science: On the Threshold”. *American Scientist*, vol. 91, No. 1, January-February 2003, pp.12-171.

Anexo I

Conceptos de Complejidad Computacional

Tipos de problemas. Recordamos brevemente los tipos que podemos tratar dentro la complejidad computacional:

1. **Problemas de decisión**. Dada una instancia del problema (un grafo arbitrario de tamaño n), determinar si es k-colorable o no. Puede haber procedimientos capaces de determinar si la respuesta es sí o no, pero que no proporcionen una solución concreta (asignación de colores a nodos).
2. **Problemas de búsqueda**. En este caso el procedimiento debería ser capaz de proporcionar una solución de asignación de colores a nodos, en caso de que sea posible.
3. **Problemas de optimización**. En algunos casos de problemas NP-completos puede suceder que no baste con determinar una solución que verifique la cuestión que define el problema.

   Sucede en problemas de grafos con arcos valuados, por ejemplo el Problema del Viajante (*Travelling Salesman Problem,* TSP). En este caso se busca una solución que sea ciclo hamiltoniano y que cumpla un criterio respecto al coste (por ejemplo, longitud) del recorrido:

   a. Caso del problema de decisión: Dado la instancia del problema, ¿existe una solución cuya longitud sea inferior a una cota dada?
   b. Caso del problema de búsqueda: Dada la instancia del problema, ¿cuál es una solución que verifique que su longitud es inferior a una cota dada?
   c. Caso del problema de optimización. No buscamos una solución cualquiera que sea inferior en longitud a una cota dada, buscamos aquella que tenga longitud inferior a todas las demás soluciones posibles.

Métodos Exactos

Garantizan encontrar la solución en caso de existir. No es el caso cuando se usan ciertas heurísticas y podas, o métodos aproximados, métodos aleatorios, metaheurísticas, etc.

Soluciones Óptimas y Subóptimas

En problemas de búsqueda y optimización puede suceder que sean válidas soluciones que no sean las óptimas pero que se aproximen a ellas. Soluciones subóptimas serían:

- Caso TSP: un recorrido que se desvíe del óptimo estimado. Se buscará acotar esta desviación, por ejemplo a un máximo del 10%
- Caso k-Colorabilidad: una asignación de colores que tenga algún conflicto. Se buscará minimizar la cantidad de conflictos.

Este tipo de soluciones no se suelen contemplar dentro de la Teoría de Complejidad Computacional, pero pueden tener interés si a cambio se obtienen en un tiempo razonable y la desviación o los conflictos no son críticos.

Transiciones de Fase P/NP

Una propiedad conocida de los problemas de tipo *NP-Complete* es que muchas de las instancias son fácilmente resolubles, en tiempo polinomial, y otras requieren de tiempos no polinomiales [Cheeseman 1991]. En muchos casos se pueden caracterizar las instancias en base a una parametrización que permite estudiar la complejidad en función de los valores de los parámetros.

En el caso de problemas basados en grafos se puede utilizar la relación entre número de arcos y de nodos, $\alpha = a/n$. Por ejemplo, en el caso del problema max-Clique (encontrar el mayor subgrafo completo dentro de un grafo dado) es sabido que la mayor complejidad de resolución se encuentra para $\alpha = n/2$. Para valores de α que se alejan de n/2 la complejidad de resolución se simplifica, y además de forma bastante brusca. Por ello se denomina transición de fase.

Para problemas de k-colorabilidad también existen transiciones de fase en función del valor de α. Consultad la bibliografía [Hayes 2003] para disponer de más detalles. El Anexo II ilustra las transiciones de fase que se encuentran en grafos de conectividad variable.

Reducción Polinómica entre problemas NP-Completos.

En 1971 Steven Cook identificó el problema de Satisfacibilidad Booleana (SAT) como uno particular que definía una nueva clase de problemas (NP-Completos). En 1972 Richard Karp identificó otros 21 problemas para los que existía una reducción polinómica desde SAT, y que por ello también eran de clase NP.Completa. Entre ellos se encontraban los de k-Colorabilidad y los relacionados con Clique.

Las reducciones polinómicas que nos interesan son:

$$3\text{-}Sat \leq_p k\text{-}Colorability \leq_p max\text{-}Clique$$ para k número cromático.

Es importante que las reducciones transformen un problema en otro en un tiempo polinómico. Podéis consultar el material de magistral y la bibliografía para disponer de más detalles.

Métodos de Búsqueda

1. Enumeración. Búsqueda exhaustiva basada en fuerza bruta sin restricciones. Consiste en:
    a. generar una por una cada una de las posibles soluciones
    b. verificar para cada una de las posibles soluciones si la asignación de colores a nodos cumple o no con la definición de k-colorabilidad.

    Este método es el más simple posible. Recordamos que la asignación de k colores a n nodos se puede representar cómo la asignación de un número de n dígitos en base k, donde cada dígito entre 0,1, …, k representa un color y la posición en el número representa un nodo del grafo.

2. *Backtracking*. Un método algo más elaborado en vez de verificar la solución una vez construida, podría verificarla durante el proceso de construcción. Esta búsqueda debería implementarse mediante backtracking, de forma que si al añadir un nodo con un color se detecta una colisión con sus nodos adyacentes. retroceda a un paso anterior en busca de una combinación alternativa sin conflictos.

    El uso de *Backtracking* permite realizar un análisis recurrente del coste computacional.

    Recordamos también que los algoritmos recursivos se pueden transformar en su equivalente iterativo añadiendo una estructura de datos de tipo pila.

3. Heurísticas. Se puede mejorar el rendimiento añadiendo alguna heurística. Por ejemplo, podemos ordenar los nodos de forma decreciente en base a la cantidad de arcos que parten de él. A la hora de asignar colores conviene comenzar con los nodos con mayor conectividad. Empezar por los nodos de menor conectividad puede crear conflictos que requieran retrocesos de largo alcance para evitarlos. El uso de dichas heurísticas debería acelerar la localización de soluciones, aunque puede suceder que resulten contraproducentes con ciertas instancias. Es importante escoger heurísticas y podas que garanticen encontrar soluciones exactas siempre que existan.

4. Métodos Alternativos. Puede ser una opción emplear métodos *greedy*, de búsqueda local, métodos específicos cómo *survey propagation* o metaheurísticas para encontrar soluciones.

    Pero debemos tener en cuenta que no proporcionan garantías de encontrar soluciones exactas. Podemos recurrir a métricas para determinar la calidad de las soluciones. Por ejemplo, el número de conflictos nos indica que se trata de soluciones subóptimas. Siempre se puede buscar soluciones con el menor número de conflictos, pero eso complica el cálculo del coste computacional.

    Estos métodos suelen introducir parámetros adicionales, no vinculados al tamaño de la instancia, que inciden en la calidad de la solución obtenida y también en el tiempo necesario para su finalización. Esto supone otra complicación considerable para el cálculo del coste computacional.

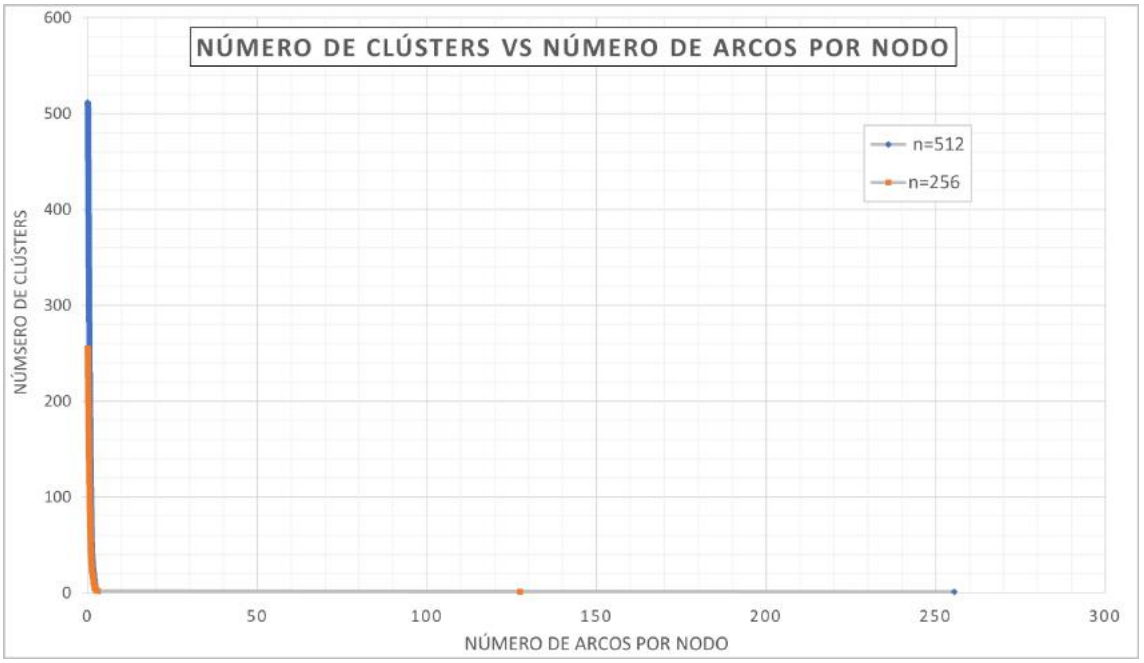
Anexo II

Ejemplo de Transiciones de Fase

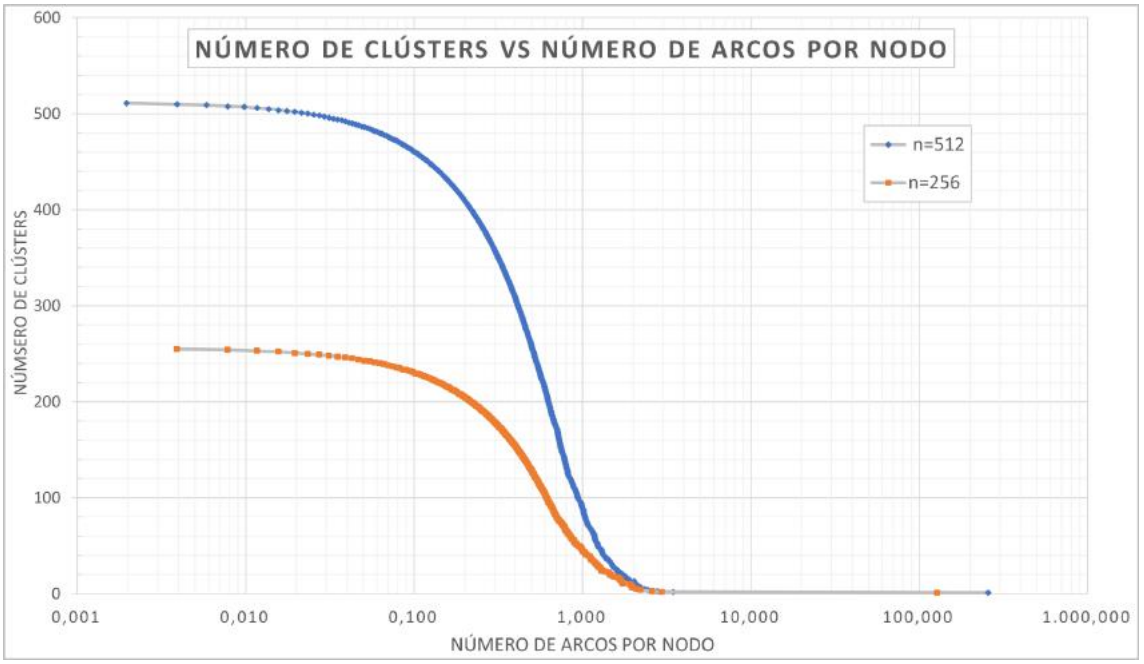
Las transiciones de fase se pueden observar en muchos sistemas complejos físicos, químicos y biológicos y se manifiestan por la aparición repentina de una propiedad al variar algunos de sus parámetros u otras circunstancias. Esto también se puede apreciar en otros dominios incluidos los que estudia la Teoría de la Computación.

Un ejemplo conocido de transición de fase fácilmente observable se da al estudiar la conectividad de grafos, por ejemplo, los que pueden servir para modelar redes sociales. La aparición de subgrafos inconexos ocurre de forma repentina cuando la conectividad de los nodos se reduce a unos pocos arcos (3 o menos).

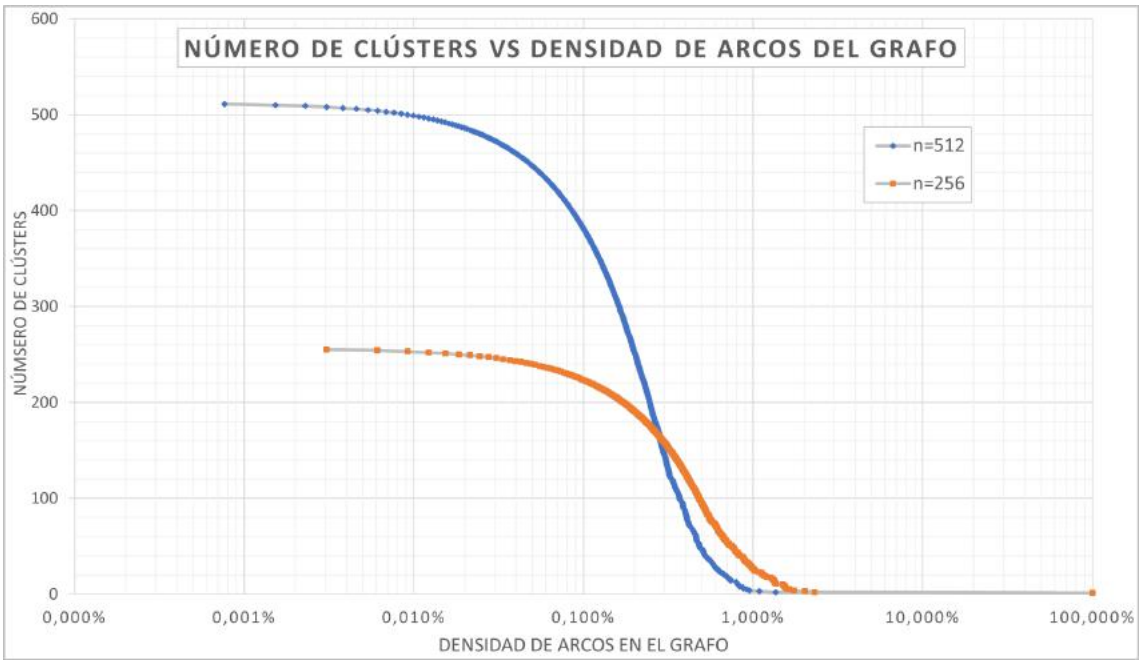
En el experimento realizado partiendo de un grafo completo de n nodos (n=256, n=512), se eliminan progresivamente y al azar las aristas desde el 100% de densidad hasta llegar al 0%. Al mismo tiempo se estudia el número de subgrafos inconexos (*clústers*) que van surgiendo. Se puede apreciar que la aparición de clústers se dispara de forma abrupta.



Usando escala logarítmica en el eje X podemos destacar detalles adicionales. La aparición de clústers comienza al bajar de 2-3 aristas por nodo.



Las medidas de número de clústers y número de aristas por nodo ($\alpha = a/n$) dependen del tamaño del grafo, serán valores más altos cuantos más nodos haya. Para una medida invariante al tamaño escogemos representar en el eje X la densidad de aristas en forma de porcentaje, indicando el 100% un grafo completo.



De Igual forma que aparecen transiciones de fase en la conectividad de los grafos aleatorios, también existen transiciones de fase en la complejidad de problemas relacionados con grafos, en particular en aquellos que pertenecen a la clase *NP-Complete*.

Anexo III

Programa grafos.c utilizado para generar los datos del Anexo II (disponible en AG).

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_NODOS  2000

int *matriz [MAX_NODOS] ;

int clusters [MAX_NODOS] ;

char *mi_malloc (int nbytes) {
    char *p ;
    static long int nb = 0L ;
    static int nv = 0 ;

    p = malloc (nbytes) ;
    if (p == NULL) {
         fprintf (stderr, "Error, no queda memoria disponible para %d bytes mas\n", nbytes) ;
         fprintf (stderr, "Se han reservado %ld bytes en %d llamadas\n", nb, nv) ;
         exit (0) ;
    }

    nb += (long) nbytes ;
    nv++ ;

    return p ;
}
```

```c
/* crea matriz: vector de punteros a vectores en memoria dinámica */
void crear_matriz (int **matriz) {
	int i ;

	for (i = 0 ; i < MAX_NODOS ; i++) {
		matriz [i] = (int *) mi_malloc (sizeof (int)*MAX_NODOS) ;
	}
}

/* 0 indica que no hay arista entre los nodos i-j */
void inicializar_grafo (int **matriz, int nodos) {
	int i ;
	int j ;

	for (i = 0 ; i < nodos ; i++) {
		for (j = 0 ; j < nodos ; j++) {
			matriz [i][j] = 0 ;
		}
	}
}
```

```c
/* crea grafo con n nodos y a arcos: no se controlan los limites */
/* 1 indica que hay arista entre los nodos i-j */
void crear_grafo (int **matriz, int nodos, int arcos) {
	int i ;
	int p ;
	int q ;

	inicializar_grafo (matriz, nodos) ;

	for (i = 0 ; i < arcos ; i++) {
		do {
			p = rand () % nodos ;
			q = rand () % nodos ;
		} while (p == q || matriz [p][q] != 0) ; // evitar diagonal y arcos existentes
		matriz [p][q] = 1 ;
		matriz [q][p] = 1 ;	// arista simétrica
	}
}

int borrar_arco_aleatorio (int **matriz, int nodos, int arcos) {
	int p ;
	int q ;

	if (arcos == 0) {
		return 0 ;
	}
	do {
		p = rand () % nodos ;
		q = rand () % nodos ;
	} while (matriz [p][q] == 0 || p == q) ;  // p==q es redundante

	matriz [p][q] = 0 ;
	matriz [q][p] = 0 ;
	return arcos-1 ;
}
```

```c
/* imprime nodos del grafo en diagonal y en las intersecciones si son adyacentes */
/* imprime antes el número de grafo incoexo (cluster) si está disponible */
void imprimir_grafo (int **matriz, int nodos) {
	int i ;
	int j ;

	for (i = 0 ; i < nodos ; i++) {
		printf ("%4d, c:%3d: ", i, clusters [i]) ;
		for (j = 0 ; j < nodos ; j++) {
			if (i == j) {			// marca diagonal con punto
				printf (".", i) ;
			} else if (matriz [i][j] == 1) {	// si i-j son adyacentes
				printf ("+") ;		// marca interseccion i-j con +
			} else printf (" ") ;
		}
		printf ("\n") ;
	}
}

/* para propagar recursivamente una marca (número de cluster) a los nodos adyacentes del nodo n */
void propagar_marcaR (int **matriz, int nodos, int n, int n_cluster) {
	int j ;

	for (j = 0 ; j < nodos ; j++) {
		if (n != j && matriz [n][j] != 0 && clusters [j] == -1) {
			clusters [j] = n_cluster ;
			propagar_marcaR (matriz, nodos, j, n_cluster) ;
		}
	}
}
```

```c
/* wrapper para propagar una marca (número de cluster) a los nodos adyacentes  */
int contar_clusters (int **matriz, int nodos) {
	int i ;
	int j ;
	int n_cluster ;

	for (i = 0 ; i < nodos ; i++) {
		clusters [i] = -1 ;			// todos los nodos sin marcar para empezar
	}
	n_cluster = 0 ;

	for (i = 0 ; i < nodos ; i++) {
		if (clusters [i] == -1) {	// nodo i no está maecado ==> es nodo de grafo inconexo
			clusters [i] = n_cluster ;  // nuevo cluster y propagar a sus adyacentes
			propagar_marcaR (matriz, nodos, i, n_cluster) ;
			n_cluster++ ;
		}
	}

	return n_cluster ;
}
```

```
void analizar_grafo (int **matriz, int nodos) {
	int arcos ;
	int clust0 ;
	int clust1 ;

	clock_t start ;
	clock_t end ;
	double cpu_time_used ;

	arcos = nodos*(nodos-1)/2 ;

	crear_grafo (matriz, nodos, arcos) ;    // crea grafo completo

	start = clock () ;
	clust0 = 0 ;

	do {
		clust1 = contar_clusters (matriz, nodos) ;
		if (clust1 > clust0) {     // imprime solo si aparece nuevo cluster
			printf ("Nodos\t %4d\t Arcos\t %4d\t Clusters\t %3d\n", nodos, arcos, clust1) ;
			clust0 = clust1 ;
		}                                       // elimina un arco al azar:
		arcos = borrar_arco_aleatorio (matriz, nodos, arcos) ;
	} while (arcos > 0) ;

	clust1 = contar_clusters (matriz, nodos) ;
	if (clust1 > clust0) {     // imprime solo si aparece nuevo cluster
		printf ("Nodos\t %4d\t Arcos\t %4d\t Clusters\t %3d\n", nodos, arcos, clust1) ;
		clust0 = clust1 ;
	}

	end = clock () ;
	cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
	printf ("TIempo %3f \n", cpu_time_used) ;
}
```

```
int main (void)
{
	srand (3) ;

	crear_matriz (matriz) ;

	analizar_grafo (matriz, 4) ;
	analizar_grafo (matriz, 16) ;
	analizar_grafo (matriz, 32) ;
	analizar_grafo (matriz, 64) ;
	analizar_grafo (matriz, 128) ;
	analizar_grafo (matriz, 256) ;
	analizar_grafo (matriz, 512) ;
//	analizar_grafo (matriz, 1024) ;

	mi_malloc (-1) ;
	system ("PAUSE") ;
}
```