RESOLUCIÓN DEL PROBLEMA DEL CARTERO CHINO (CPP) MEDIANTE LA APLICACIÓN DE UN PROCESO DE BRANCH AND BOUND.

Miquel MONSERRAT PONS

PROYECTO FINAL DE CARRERA

Director: Ramón COMPANYS PASCUAL

2 de junio de 1997

MEMORIA

El problema del cartero chino (Chinesse Postman Problem en inglés) es un problema clásico de la Investigación Operativa que consiste en, a partir de un grafo no orientado, con unos valores positivos asociados a cada arista, hallar un ciclo de coste mínimo que pase al menos una vez por cada arista. Si el grafo es euleriano, el problema se limita a hallar un camino que recorra el grafo. En caso de que no sea euleriano es necesario añadir algunas aristas para convertir el grafo en euleriano.

Los métodos existentes para la resolución de este problema están basados en algoritmos específicos para hallar un acoplamiento perfecto óptimo del subgrafo completo formado por los nodos de grado impar.

En este proyecto se proponen, diseñan e implementan tanto un procedimiento de tipo *branch and bound* para resolver dicho acoplamiento y convertir el grafo en euleriano como un algoritmo para hallar caminos en grafos eulerianos.

Memoria

Índice

Memoria 2
Índice 2
1. Introducción 5
2. Fundamentos de teoría de grafos 6
2.1. Definición de grafo 6
2.2. Grafos orientados 6
2.1.1. Arco 6
2.1.2. Arcos adyacentes 7
2.1.3. Vértices adyacentes 7
2.1.4. Arco incidente a un vértice 7
2.1.5. Semigrado de un vértice 7
2.1.6. Camino 7
2.1.7. Caminos simples y compuestos 7
2.1.8. Camino elemental 7
2.1.9. Circuito 7
2.1.10. Circuito elemental 8
2.1.11. Longitud de un camino 8
2.1.12. Bucle 8
2.1.13. Grafo fuertemente conexo 8
2.3. Grafos no orientados o simétricos 8
2.3.1. Arista 8
2.3.2. Grado de un vértice 8
2.3.3. Cadena simple 9
2.3.4. Cadena elemental 9
2.3.5. Ciclo 9
2.4. Grafos mixtos 9
2.5. Tipos de grafos 9
2.5.1. Grafo subyacente 9
2.5.2. Grafo parcial 9
2.5.3. Multigrafo 9
2.5.4. Grafo completo 9
3. Acoplamientos 10
3.1. Definiciones 10
3.2. Acoplamiento de coste mayor 11
3.3. Obtención de un acoplamiento de coste mayor 11
3.4. Algoritmo de Edmonds & Johnson para la obtención de un acoplamiento de coste mayor 14
3.1. Demostración del algoritmo de Edmonds & Johnson 17

4. Almacenamiento de grafos 19
4.1. Almacenamiento matricial 19
4.2. Lista de adyacencia 19
5. Grafos eulerianos 21
5.1. Caracterización de circuitos eulerianos 21
5.2. Obtención de un circuito euleriano 22
5.2.1. Regla de Fleury 22
5.2.2. Algoritmo ciclo euleriano 23
5.2.3. Comentario sobre el algoritmo de E&J (1973) 24
6. Problemas combinatorios 25
6.1. Concepto de problema combinatorio 25
6.2. Dificultades de resolución 25
6.3. Métodos de resolución 26
6.4. Ejemplos de problemas combinatorios 27
7. Procesos de branch and bound 28
7.1. Exposición intuitiva 28
7.2. Formalización del proceso 30
7.3. Definición del algoritmo 31
7.4. Procedimientos heurísticos a partir de algoritmos branch and bound 32
8. Problema del Cartero Chino (CPP) 33
8.1. Introducción y aplicaciones 33
8.2. Formalización del problema 33
8.2.1. Ejemplo 33
8.3. Método general de resolución 35
8.3.1. Ejemplo 36
9. Obtención del acoplamiento 38
9.1. Método clásico 38
9.2. Aplicación de un algoritmo de branch and bound 38
9.2.1. Elementos previos 38
9.2.2. Caracterización de los nodos 39
9.2.3. Procedimiento de separación 39
9.2.4. Procedimiento de acotación 39
9.2.5. Elección del nodo a explorar 40
9.2.6. Ejemplo 40
10. Implementación del algoritmo 48
10.1. Estructura del algoritmo 48
10.2. Bloque principal 49
10.3. Estructuras de datos 50
10.3.1. Implementación del árbol 50
10.3.1.1. Estructura de datos del árbol 50
10.3.1.2. Estructura de los elementos del árbol 51
10.3.1.3. Operaciones sobre el árbol 52

10.3.2. Implementación del almacenamiento del multigrafo 52
10.3.2.1. Estructura de datos que almacena el multigrafo 53
10.3.2.2. Estructura de los elementos 53
10.3.2.3. Operaciones sobre el módulo de almacenamiento del multigrafo 53
10.3.2.4. Almacenamiento en matrices 54
10.3.3. Almacenamiento de una matriz simétrica 55
10.3.3.1. Estructura de datos de la matriz simétrica reducida 55
10.3.3.2. Operaciones disponibles 55
10.4. Implementación del algoritmo de caminos eulerianos 55
10.5. Implementación del algoritmo de Fulkerson de caminos mínimos 56
10.5.1. Introducción 56
10.5.2. Algoritmo 56
10.6. Almacenamiento de grafos en ficheros 57
10.7. Hardware utilizado 58
10.8. Modelos de compilación 58
11. Comportamiento del algoritmo 59
11.1. Criterios de extracción 59
11.2. Dimensiones de los problemas 60
11.3. Predicción de tiempos de ejecución 60
12. Obtención de un procedimiento heurístico 62
13. Conclusiones 64
14. Bibliografía 66
15. Presupuesto 67
ANEXO 1: listado de programas 68
MAIN.C 68
ARBOL.C 84
FULKER.C 91
GRAFO.C 93
EULERIAN.C 98
SIMETR.C 100

1. Introducción

Este proyecto consiste en solucionar el problema llamado del Cartero Chino variando la parte principal del proceso de resolución, consistente en hallar un acoplamiento en un grafo simétrico. Todos los autores indican algún algoritmo clásico de acoplamientos, mientras que aquí se propone un enfoque distinto, consistente en resolver el acoplamiento como un problema combinatorio, esto es aplicando un proceso de branch and bound.

Las partes principales del proyecto son las siguientes:

- Introducción de los conceptos que se van a utilizar.
- Presentación del problema.
- Estudio de los algoritmos usados anteriormente.
- Presentación e implementación del método propuesto.
- Estudio de casos reales.
- Exposición de conclusiones.

En los anexos se incluyen los listados de los programas implementados, así como los datos de los problemas estudiados.

2. Fundamentos de teoría de grafos

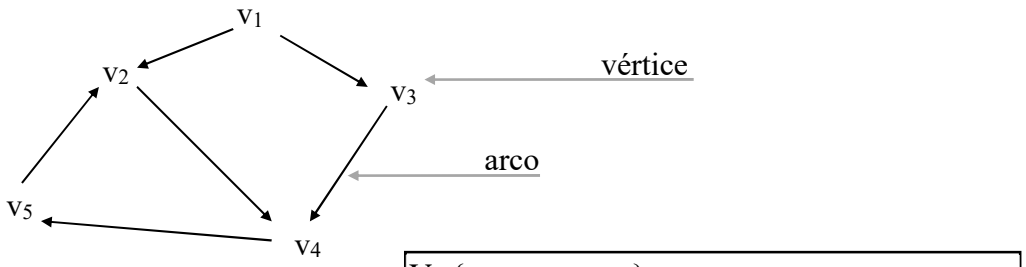
En este capítulo se pretende introducir la nomenclatura más usual en la teoría de grafos que será usada a lo largo del proyecto. Un punto importante, tanto en la teoría de grafos como en este proyecto, es el apartado de acoplamientos; pero debido tanto a su ya referida importancia, como a que su complejidad la aleja de los fundamentos a que este capítulo se refiere, se le dedica un tema aparte con posterioridad.

2.1. Definición de grafo

Un *grafo G(V,A)* está formado por un conjunto de puntos $V=\{v_1,v_2,...,v_k,...\}$, finito o no, pero distintos y numerables (que llamaremos vértices) y por un conjunto de relaciones $A=\{a_1,a_2,...,a_q,...\}$, de forma que cada relación (que llamaremos arco o arista en función de si el grafo es o no orientado) une dos puntos de V.

2.2. Grafos orientados

Denominaremos *grafo orientado* al grafo en el cual se pueden determinar extremos inicial y final en las todas las relaciones de A.



Grafo orientado
G(V,A)

$V=\{v_1,v_2,v_3,v_4,v_5\}$

$A=\{(v_1,v_2),(v_1,v_3),(v_2,v_4),(v_3,v_4),(v_4,v_5),(v_5,v_2)$

2.1.1. Arco

La relaciones orientadas que forman parte de A se denominan *arcos*.

2.1.2. Arcos adyacentes

Diremos que dos arcos son *adyacentes* si:

- Son distintos entre sí
- Tienen un extremo común

2.1.3. Vértices adyacentes

Diremos que dos vértices son *adyacentes* si:

- Son distintos
- Existe un arco que va de uno a otro

2.1.4. Arco incidente a un vértice

Se denomina que un arco a_{ij} es *incidente* a un vértice v_i *hacia el exterior* (o emergente) si v_i es el extremo inicial de a_{ij} y si además el extremo final v_j es distinto de v_i.
Se denomina que un arco a_{ij} es *incidente hacia el interior* si v_i es el extremo final y el inicial es distinto de v_i.

2.1.5. Semigrado de un vértice

Semigrado exterior de un vértice v_k es igual al número de arcos incidentes hacia el exterior de v_k.
Semigrado interior de v_k es igual al número de arcos incidentes hacia el interior de v_k.

2.1.6. Camino

Denominamos *camino* a una secuencia de arcos, tales que el extremo final de cada arco coincide con el extremo inicial del siguiente.

2.1.7. Caminos simples y compuestos

Un *camino simple* es un camino que no utiliza dos veces el mismo arco. En caso contrario se denomina *camino compuesto*.

2.1.8. Camino elemental

Denominamos *camino elemental* al camino que no utiliza dos veces el mismo vértice.

2.1.9. Circuito

Un *circuito* es un camino finito en el cual el vértice inicial coincide con el vértice final.

2.1.10.Circuito elemental

Un *circuito elemental* es aquel circuito en el que se pasa una sola vez por cada vértice a excepción del inicial y que a la vez es final.

2.1.11.Longitud de un camino

Llamamos *longitud de un camino* al número de arcos de la secuencia.

2.1.12.Bucle

Podemos definir un *bucle* como un circuito de longitud igual a 1.

2.1.13.Grafo fuertemente conexo

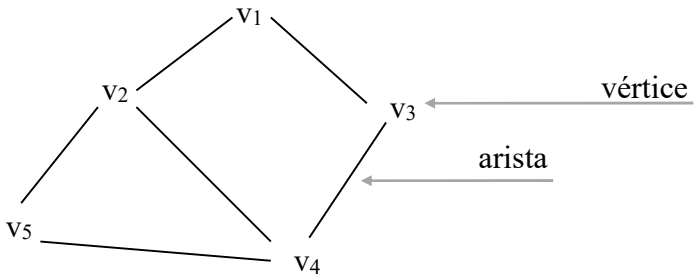
Un *grafo fuertemente conexo* es aquel en que para todo par de vértices existe, por lo menos, un camino entre ellos.

2.3. Grafos no orientados o simétricos

Denominamos *grafos no orientados o simétricos* a aquellos en que no podemos establecer los extremos inicial y final en ninguna de las relaciones de A.

2.3.1. Arista

Las relaciones no orientadas que forman parte del conjunto A se denominan *aristas*.



Grafo no orientado o simétrico

2.3.2. Grado de un vértice

El *grado* de un vértice en un grafo simétrico es el número de aristas incidentes en el vértice.

Llamamos *cadena* a una secuencia de aristas tal que cada arista está unida por uno de sus extremos a la arista anterior y por otro a la posterior (si existen).

2.3.3. Cadena simple

Una cadena es *cadena simple* si todos las aristas se utilizan una sola vez.

2.3.4. Cadena elemental

Si todo vértice que se encuentra en la cadena, se encuentra solo una vez a lo largo de la misma, la llamamos *cadena elemental*.

2.3.5. Ciclo

Un *ciclo* es una cadena que parte de un vértice y llega al mismo.

2.4. Grafos mixtos

Denominamos *grafos mixtos* a aquellos que tienen arcos y aristas.

2.5. Tipos de grafos

2.5.1. Grafo subyacente

El grafo subyacente de un grafo dirigido G(V,A) es el grafo que se obtiene si se prescinde de la orientación de sus arcos.

2.5.2. Grafo parcial

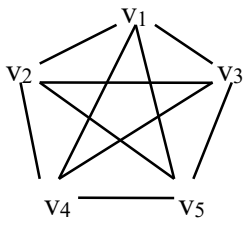
$G_P(V_P,A_P)$ se llama *grafo parcial* de G(V,A) si $A_P \subseteq A$ y $V_P \subseteq V$

2.5.3. Multigrafo

Un *multigrafo* es un grafo donde pueden haber aristas o arcos repetidos.

2.5.4. Grafo completo

Un grafo sin lazos es *completo* si tiene el mayor número posible de aristas (o arcos).
El grafo completo y simétrico de t vértices se representa por K_t.



K_5

3. Acoplamientos

3.1. Definiciones

Un *acoplamiento* M (*matching* en inglés) en un grafo G(V,A) simétrico viene dado por todo subconjunto de aristas de A en el cual no hay dos que tengan un vértice común.
Diremos que un acoplamiento M en un grafo G es *máximo* si $|M| \geq |N|$ para todo otro acoplamiento N de G.
Un vértice v_i es llamado *libre* si no se encuentra en ninguna de las aristas de M, en caso contrario se le suele llamar *saturado*.
Un *camino* es *alternante* si está formado por aristas que, alternativamente, se encuentran en M y en (A - M).
Un *camino* es *incremental* (en relación a M), si es alternante y tiene en los dos extremos sendos vértices libres.
Un *árbol alternante* es un árbol con las siguientes características:
- tiene un vértice libre llamado raíz
- todos los caminos que comienzan en la raíz son alternantes
- todos los caminos que empiezan en la raíz y son maximales tienen un número par de aristas.

Una *flor* en un árbol alternante es un camino incremental que comienza y acaba en un vértice etiquetado exterior. Para formar el circuito siempre se forma una arista de G que no se encuentra en el árbol, por lo que, el número de aristas de una flor es siempre impar.
Un *árbol húngaro* es un árbol alternante T que no se puede extender más. Cada árbol húngaro nos indicará un subgrafo de G en el cual el acoplamiento actual que tenemos formado en el árbol alternante, quedará invariable sea cual sea el acoplamiento máximo de G.

3.2. Acoplamiento de coste mayor

Para todo grafo simétrico G(V,A) podemos definir una *función de coste*:

$$C : A \rightarrow \mathbf{R}^+$$

$$a_i \rightarrow c(a_i)$$

Este número se llama generalmente coste o peso de la arista y puede depender de la distancia o del tiempo que hay entre los vértices usando una cierta métrica, etc...
Un acoplamiento M diremos que es de *coste mayor*, respeto a un grafo G, si maximiza la suma de los costes de las aristas que lo forman, es decir, si para todo acoplamiento M' de G se verifica que:

$$\sum_{a_i \in M} c(a_i) \geq \sum_{a_j \in M'} c(a_j)$$

Dado un acoplamiento M, un *M-camino incremental de coste* es un camino alternante P en el cual la suma de los costes de las aristas de P que no se encuentran en M supera la suma de los costes de las aristas que sí están; más formalmente:

$$\sum_{a_i \notin M} c(a_i) > \sum_{a_j \in M} c(a_i)$$

3.3. Obtención de un acoplamiento de coste mayor

El método descrito a continuación es de los autores Edmonds y Johnson, y está basado en el algoritmo para determinar un acoplamiento máximo que podemos hallar, por ejemplo, en (Bas 93).
Dado un M-camino incremental de coste, diremos que es:

- fuerte si contiene más aristas fuera de M que en M
- neutro si contiene tantas aristas fuera de M como en M
- débil si contiene menos aristas fuera de M que en M

Para seguir el algoritmo, primero en necesario plantearlo en términos de programación lineal, formular el dual y usar algunas propiedades básicas.

Sea c(i,j) el *coste de la arista* (v_i,v_j).

En el planteamiento del problema tendremos que la variable binaria x(i,j) tomará valor 1 si la arista (v_i,v_j) pertenece a M y valor 0 en caso contrario.

Así, el problema de hallar el acoplamiento de coste mayor puede ser formulado como:

$$\max \sum_{v_i\ ,\ v_j} x(i,j)\ c(i,j)$$

donde:

$$\sum_{j} x(i,j) + x(j,i) \leq 1 \qquad \forall\ v_i \in V$$

$$\sum_{(v_i,v_j) \in R_m} x(i,j) \leq r_m \qquad \forall\ m = 1,2,...,z$$

$$x(i,j) \geq 0\ \forall\ (v_i\ ,\ v_j)$$

siendo W_m todo conjunto de $2r_m+1$ vértices de G(V,A) (suponemos que hay z) y R_m el subgrafo inducido por W_m. Es inmediato darse cuenta que ningún acoplamiento M puede contener más de r_m aristas de R_m.

Para formular el problema dual asociado sean y_j la variable dual asociada a la restricción por el vértice v_j en el problema primal, y z_m, la variable dual asociada a la restricción por el subgrafo R_m en el problema primal.

Así, el problema dual correspondiente vendrá dado por:

$$\min \sum_{j} y_i + \sum_{m} r_m\ z_m$$

donde:

$$y_i + y_j + \sum_{m\ /(v_i,v_j) \in R_m} z_m \geq c(i,j) \qquad \forall\ (v_i,v_j)$$

$$y_{i \geq} 0 \quad \forall\ v_j$$

$$z_m \geq 0 \quad \forall\ m = 1,2,...,z$$

En esta pareja primal-dual las condiciones de complementariedad nos dicen que:

$$x(i,j) > 0 \quad \Rightarrow \quad y_i + y_j + \sum_{m \,/\, (v_i,v_j) \in R_m} z_m = c(i,j) \qquad \forall\, (v_i,v_j) \qquad (3.1)$$

$$y_i > 0 \quad \Rightarrow \quad \sum_{j} x(i,j) + x(j,i) = 1 \qquad \forall\, v_i \in V \qquad (3.2)$$

$$z_m > 0 \quad \Rightarrow \quad \sum_{(v_i,v_j) \in R_m} x(i,j) = r_m \qquad \forall\, m=1,2,...,z \qquad (3.3)$$

El algoritmo comienza con M = Ø, es decir $x(i,j) = 0 \quad \forall \quad (v_i,v_j)$, asignando valores factibles a las variables duales y_i y z_m tales que se verifiquen las condiciones de complementariedad 3.1. y 3.3. En cada iteración del método, M o las variables duales (o los dos a la vez) varían de forma que las restricciones del primal y del dual se continúan verificando así las condiciones 3.1 y 3.3. Pero además se consigue que, como mínimo, una de las variables duales y_i pase a satisfacer la condición 3.2. Cuando todas las variables duales verifiquen 3.2., por el teorema de complementariedad (forma simétrica), tendremos que el acoplamiento M resultante será de coste mayor.

La condición 3.2. pide que si la variable del dual $y_i > 0$ entonces v_i ha de formar parte de M; por tanto, los únicos vértices que no cumplirán la condición 3.2. serán los vértices libres que tengan variable dual asociada positiva.

Cuando el algoritmo tiene un vértice libre v_t con $y_t > 0$ construye un árbol alternante donde v_t es el vértice raíz. El proceso de ampliación del árbol se para:

a) cuando se convierte en alternante

b) cuando se forma una flor

c) cuando se encuentra un árbol húngaro

En el caso a), el M-camino incremental formado es fuerte. Intercambiamos el papel de las aristas que se hallan en M con las que no están, tenemos que v_t ya no es libre, y el nuevo acoplamiento tiene un coste superior al anterior. Ahora v_t ya satisface la condición 3.2. En el caso b), se comprime la flor resultante para formar un pseudo-vértice y el algoritmo continúa. Finalmente, en el caso c) se modifica el valor de las variables duales asegurando que se mantengan las restricciones del primal, del dual y las condiciones de complementariedad 3.1. y 3.3., de manera que se pueda añadir una nueva arista al árbol alternante. Para acabar, todos los pseudo-vértices se tendrán que

expandir nuevamente, es por esto que el algoritmo genera una secuencia de grafos G_0, G_1,... hasta el resultado final.

Teorema de complementariedad (forma simétrica)

Sean dos soluciones factibles **x** e **y** para los problemas primal y dual, respectivamente, en la forma simétrica. Entonces, **x** e **y** son óptimas $\Leftrightarrow \forall i,j$:

· $x_i > 0 \Rightarrow \mathbf{y}a_i = c_i$

· $\mathbf{y}a_i < 0 \Rightarrow x_i = 0$

· $y_j > 0 \Rightarrow a^j\mathbf{x} = b_j$

· $a^j\mathbf{x} > b_j \Rightarrow y_j = 0$

3.4. Algoritmo de Edmonds & Johnson para la obtención de un acoplamiento de coste mayor

1. Inicialización

   $M_0 := \emptyset$

   $z_m := 0$ (m=1,2,...,z)

   asignar un valor arbitrario a las variables duales y_i

   tal que $y_i + y_j \geq c(i,j)$ para toda pareja (v_i,v_j)

   $k := 0$

   $G_k\,(V_k,A_k) := G(V,A)$

2. Consideración de un vértice libre

   sea $v_t \in G_k$ un vértice libre con $y_k > 0$

   si v_t no existe entonces ir a 6

   si no sea A' el conjunto de aristas $(v_i,v_j) \in G_k$ tales que:

$$y_i + y_j + \sum_{(v_i,v_j) \in R_m} z_m = c(i,j)$$

   formad un árbol alternante de raíz v_t y que use solo aristas de A'

   si se detecta un camino incremental, entonces ir a 3

   si se detecta una flor, entonces ir a 4

   si se detecta un árbol húngaro, entonces ir a 5

3. Camino incremental

intercambiar en M_k las aristas del camino hallado; ahora v_t ya no es libre

volver a 2

4. Flor

$k := k+1$

sea C_k la flor, comprimirla para formar un pseudo-vértice v_k

sea $G_k(V_k,A_k)$ el nuevo grafo formado

sea M_k el acoplamiento en G_k que tiene todas las aristas de M_{k-1} que se encuentran en G_k

Volver a 2 para extender el árbol alternante de raíz v_t dentro de G_k

5. Árbol húngaro

sean:

$d_1 := \min\{y_i + y_j - c(i,j) \ \ (i,j)\}$

donde $v_i \in V_0$ es un vértice exterior y $v_j \in V_0$ está sin etiquetar

$d_2 := ½ \min\{y_i + y_j - c(i,j) \ \forall \ (i,j)\}$

donde v_i y v_j son vértices exteriores de V_0 y no se hallan dentro del mismo pseudo-vértice

$d_3 := ½ \min\{z_m\}$

donde se evalúan todos los conjuntos W_m que se han comprimido en un pseudo-vértice v_k etiquetado exterior

$d_4 := \min\{y_i\}$

donde se evalúan todos los vi $\in V_0$ etiquetados exteriores

$d := \min\{d_1,d_2,d_3,d_4\}$

modificar el valor de las variables duales de la forma siguiente:

restar d unidades a las y_i correspondiente a vértices exteriores

sumar d unidades a las y_i correspondientes a vértices interiores

sumar 2d unidades a las z_m correspondientes a pseudo-vértices exteriores a G_k

restar 2d unidades a las z_m correspondientes a pseudo-vértices interiores a G_k

si $d = d_1$ entonces

la arista (v_i,v_j) que determinó v_1 se incorpora a A'

esta arista crea una flor dentro del árbol alternante

volver a 2 y continuar extendiendo el árbol de raíz v_t

si $d = d_2$ entonces

la arista (v_i,v_j) que determinó d_2 se incorpora a A'

esta arista crea ahora una flor dentro del árbol alternante

volver a 2 y continuar extendiendo el árbol de raíz v_t

si $d = d_3$ entonces

expandir el pseudo-vértice que corresponde a la variable del dual $z_i = 0$

$k := k+1$

sea $G_k(V_k,A_k)$ el grafo resultante

sea M_k el acoplamiento que contiene todas las aristas de M_{k-1} juntamente con las r_i aristas libres de W_i

volver a 2, continuar extendiendo el árbol alternante de raíz v_t

si $d = d_4$ entonces

alguna variable del dual $y_i = 0$

el camino en el árbol alternante que va de la raíz v_t hasta v_i es un M_k-camino incremental de coste neutro

el vértice v_t queda ahora enlazado en el acoplamiento y v_i queda libre, tal como ha de ser, ya que $y_i = 0$

volver a 2

6. Expansión de los pseudo-vértices

expandir cada pseudo-vértice en orden inverso al que ha sido comprimido y formar un acoplamiento de coste mayor en el circuito de cardinalidad impar que resulta

el acoplamiento final es uno de coste mayor para $G_0 = G$

parar

3.1. Demostración del algoritmo de Edmonds & Johnson

Debido a que en cada instante del proceso tenemos un acoplamiento, inicialmente un acoplamiento sin ninguna arista, en la demostración de que este algoritmo funciona correctamente tan solo hemos de comprobar que el último formado es efectivamente un acoplamiento de coste mayor. En función del planteamiento del programa lineal realizado, esto es equivalente a ver que el valor final de las variables del problema dual, $y_i \ \forall \ v_i \in V$ y z_m (m=1,2,...,z), dan una solución factible para el dual y que verifican las restricciones 3.1, 3.2. y 3.3.

En el paso 1 del método el valor inicial para las variables duales es siempre no negativo y en el paso 5 el valor de d asegura que ninguna pasará a tomar valor negativo. En consecuencia, durante todo el proceso, las variables duales se mantienen no negativas.

Para verificar que las restricciones del dual se verifican tenemos que:

- inicialmente se verifican
- la arista (v_i,v_j) tiene que hallarse:
  a) en un pseudo-vértice
  b) en A' y fuera de los pseudo-vértices
  c) fuera de A' y fuera de los pseudo-vértices

Si se da el caso:

- a), un cambio en la variable dual no afecta a la parte izquierda de la restricción ya que tanto y_i como y_j aumentan en d unidades, mientras que la z_m relativa al pseudo-vértice que contiene (v_i,v_j) disminuye en 2d unidades.
- b), un cambio incrementa el valor de la variable dual del vértice interior y disminuye el valor de la variable dual del vértice exterior, así, la parte izquierda continua como estaba.
- c), tenemos que:

$$y_i + y_j + \sum_{m \ / \ (v_i,v_j) \in R_m} z_m > c(i,j) \qquad (3.4)$$

Ahora bien, si v_i y v_j están sin etiqueta, o bien tienen etiquetas y son diferentes, entonces, el cambio preserva la desigualdad de 3.4.; si uno está etiquetado interior y el otro está sin etiqueta o si los dos están etiquetados interiores, entonces, el cambio incrementa la parte izquierda de la constricción del dual. Si uno está etiquetado exterior y el otro está sin etiqueta, entonces, usando d_1, el valor de la variable dual no invertirá la desigualdad de 3.4. Si los dos vértices son exteriores, entonces, usando d_2, el cambio en las variables duales no invertirá tampoco la desigualdad de 4.5. Así pues, en todos los cambios para las variables duales las restricciones del dual se siguen cumpliendo.

Pasemos ahora a las tres condiciones de complementariedad primal-dual.

Respecto a 4.2. distinguiremos dos casos referentes al comienzo del paso 6 del método:

a) la arista (v_i,v_j) del acoplamiento no está en un pseudo-vértice

b) la arista (v_i,v_j) del acoplamiento está en un pseudo-vértice

En el caso:

- a) (v_i,v_j) A' y la condición 3.1. se verifica
- b) después de hacer el cambio dual, la restricción del dual se convierte en una igualdad; por tanto la condición 3.1. se verifica después de los cambios

Por lo que se refiere a la condición 3.2., seguro que es cumplida por todos los vértices cuando el algoritmo termina, si no fuera así, el paso 2 se tendría que repetir para cada v_j que no la verificara.

Finalmente, para comprobar que la condición 3.3. también se verifica, notemos que z_m es positiva solo si se comprimen los vértices de W_m para formar un pseudo-vértice. En el paso 6 cada pseudo-vértice es expandido y se forma el acoplamiento de coste mayor en el circuito resultante, por tanto la condición 3.3. se verifica.

4. Almacenamiento de grafos

Para poder trabajar con grafos en un ordenador, necesitamos representarlos usando alguna de las estructuras de información disponibles. Hay dos aproximaciones fundamentales en función de si tomamos un estructura fija o una dinámica.

4.1. Almacenamiento matricial

Consiste en describir el grafo por medio de una matriz L de $N \times N$ componentes (siendo N el número de nudos del grafo), en la que la componente l_{ij} es la longitud del arco que une el vértice v_i con el v_j. Si no existe tal arco, el valor de la matriz es infinito.

En el caso de matriz simétrica es suficiente almacenar la mitad superior o inferior de la diagonal principal.

Este método presenta los inconvenientes de que puede llegar a ocupar mucha más memoria de la realmente necesaria si el grafo es "poco denso" ("sparse" en inglés), además de la imposibilidad de representar sencillamente multigrafos.

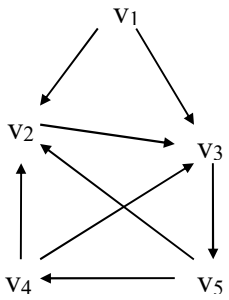
4.2. Lista de adyacencia

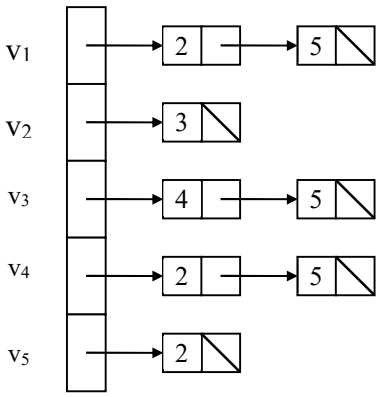
Se trata de tener, para cada vértice v_i del grafo, una lista encadenada mediante apuntadores en la cual habrá, si G es dirigido, todos los vértices sucesores de v_i; mientras que si es simétrico, todos los vértices adyacentes a v_i. Esto comporta también la necesidad de una tabla con la dirección del principio de la lista para cada vértice.

De esta manera podemos representar todo el grafo con más flexibilidad, y con la ventaja de que solo reserva el espacio de memoria necesario para almacenar las aristas y los vértices existentes en el grafo.

También cabe destacar la facilidad que esta opción proporciona para almacenar multigrafos.

Ejemplo de listas de adyacencia.





5. Grafos eulerianos

Llamamos *grafo euleriano* al grafo que contiene algún circuito euleriano.

5.1. Caracterización de circuitos eulerianos

Sea G(V,A) un multigrafo simétrico y conexo, un *circuito euleriano* es aquel que pasa por cada arista una vez y solo una. Dicho de otra forma, un *circuito euleriano* es un circuito simple que usa todas las aristas.

A continuación se expone el siguiente teorema por dos razones fundamentales: la fácil caracterización que proporciona de circuito euleriano y la introducción a la metodología de búsqueda de circuitos eulerianos a la que nos lleva la segunda parte de la demostración.

Teorema 4.1.

El multigrafo simétrico y conexo G(V,A) tiene algún circuito euleriano $\Leftrightarrow$ el número de vértices de grado impar en G es 0.

Demostración 4.1.

($\Rightarrow$) Cualquier circuito euleriano ha de tomar una arista para entrar y una para salir, en cada una de las visitas que haga a un vértice. Por tanto, si G es euleriano, necesita que sus vértices tengan grado par.

($\Leftarrow$) Comencemos a formar un camino simple desde cualquier vértice inicial v hasta que el camino vuelve a v y hemos formado un circuito simple C_0. Si C_0 ha usado todas las aristas, hemos acabado; en caso contrario habrá algún v_i de C_0 que es vértice terminal de dos o más aristas no usadas en C_0. Eliminamos de G las aristas de C_0 y trazamos otro circuito simple C_1 a partir de v_i como antes. Si el circuito $(C_0 \cup C_1)$ contiene todo A, hemos acabado; sino buscamos un v_j de C_0 o de C_1 que tenga dos o más aristas no usadas y repetimos el proceso. Finalmente obtendremos el circuito euleriano como unión de k circuitos $C_0, C_1,..., C_{k-1}$, simples y disjuntos de aristas.

5.2. Obtención de un circuito euleriano

Una vez comprobado que G tiene circuito euleriano, determinarlo es sencillo usando algún procedimiento basado en la demostración del teorema 4.1.

5.2.1. Regla de Fleury

Consiste en comenzar por un vértice arbitrario w, e ir construyendo un camino simple a la vez que se borran las aristas por las cuales pasamos. La única previsión que debemos hacer es no pasar nunca por una arista que una vez borrada deje el grafo resultante disconexo.

En el procedimiento que sigue, sean:

- v^* el vértice actual
- A'={aristas ya recorridas}
- L una lista que contiene los vértices ordenados en la misma secuencia en que han sido recorridos.
- $A(v_i)$ la lista de vértices adyacentes al vértice v_i, dentro del grafo G - A'
- w el primer vértice visitado en el recorrido.

Fleury:

L := w
v^* := w
A' := Ø
mientras |A(w)| > 0 hacer
    si |A(v^*)| > 1 entonces
        buscar un v de A(v^*) tal que (v^*,v) no sea puente de G-A'
    finsi
    sino
        sea v el vértice de A(v^*)
        A(v^*) := A(v^*) - {v}
        A(v) := A(v) - {v^*}
        A' := A' ∪ {(v^*,v)}
        v^* := v

Añadir v^* al final de L

finsino

fin

L es el resultado

5.2.2. Algoritmo ciclo euleriano

La dificultad del algoritmo de Fleury, que es básicamente el que aparece en la bibliografía de teoría de grafos (como en (Bas 95)), reside en la implementación de la acción que comprueba que la arista a coger no deje el grafo resultante inconexo. Es por ese motivo que se ha optado por implementar otro algoritmo basado más directamente en la demostración del teorema 4.1.

inicio

mientras no exista ciclo hacer

si no hemos empezado entonces

elegir primer vértice

finsi

sino

elegir un vértice del ciclo en construcción con aristas incidentes no usadas

finsino

empezar un nuevo recorrido a partir del vértice elegido

mientras el vértice actual tenga aristas incidentes no usadas hacer

ir a un vértice adyacente

guardar la arista recorrida

establecer la arista recorrida como usada

cambiar el vértice actual

finmientras

guardar el último tramo del circuito

concatenar el último tramo y el ciclo en construcción

finmientras

fin

5.2.3. Comentario sobre el algoritmo de E&J (1973)

Cabe señalar que el algoritmo de Edmonds & Johnson (1973) para la obtención de circuitos eulerianos que aparece en alguna bibliografía no funciona de forma correcta para todos los casos, es por eso que aquí se ha optado por no incluirlo.

6. Problemas combinatorios

Aquí se pretende establecer el concepto, y los problemas y métodos generales de resolución de los problemas conocidos como combinatorios. Dentro de los métodos de resolución cabe destacar el método llamado de branch and bound, al que por su importancia en este proyecto se le dedica un tema aparte.

6.1. Concepto de problema combinatorio

Los *problemas combinatorios* son problemas de optimización que tienen en común una serie de características como son:

- Tienen un número finito de soluciones.
- Para encontrar la solución óptima no podemos recurrir a los procedimientos clásicos basados en el cálculo infinitesimal, ya que al modelizarlos nos vemos obligados a utilizar variables discretas.

Dado un conjunto finito de objetos E y una función (función objetivo) que asocia a cada objeto del elemento de un conjunto ordenado C (costes, pesos, distancias,...) se trata de hallar en elemento de E al que corresponda un elemento de C óptimo.

$$[OPT] \quad z = f(X) \qquad \text{con} \qquad X \in E$$

Con frecuencia los elementos de E se definen por los valores que toman las variables de un conjunto finito $(x_1,...,x_n)$ cada una de las cuales puede tomar un número finito de valores.

6.2. Dificultades de resolución

A primera vista, la resolución de un problema de optimización combinatoria no presenta dificultades, puesto que bastaría con enumerar los objetos (cosa siempre posible puesto que su número es finito) y calcular para cada uno de ellos el valor de la función objetivo. La dificultad del problema está en que el número de objetos crece factorialmente, por lo que para los problemas de una cierta dimensión, la enumeración resulta un método inaplicable.

6.3. Métodos de resolución

Los métodos de resolución se pueden clasificar básicamente en heurísticos y exactos.
Los heurísticos son aquellos que no pueden garantizar la obtención de una solución óptima, por lo que interesa es tener una idea de si es o no mucho peor que la solución óptima. Para esto debemos determinar una cota, como más ajustada mejor, del valor óptimo de la función objetivo. La diferencia entre la cota y el valor de la función objetivo para la solución hallada por el algoritmo nos da una cota del error que se comete. Si esta cota es pequeña, la solución es buena; sino, no podemos asegurarlo.
La mayoría de algoritmos heurísticos tienen el siguiente esquema:

- Determinar una solución posible.
- Intentar mejorarla.

Para el primer paso, es corriente usar un algoritmo de tipo "greedy", que se caracterizan por hacer elecciones sin tener en cuenta las repercusiones posteriores de dichas elecciones, por eso algunos autores los califican de "miopes". A pesar de esto, para muchos problemas, un algoritmo de este tipo puede proporcionar soluciones aceptables.
Un método general para la mejora de la solución (obtenida a partir de un algoritmo greedy o por otras vías), suele consistir en definir un movimiento, es decir, una regla que permite obtener soluciones a partir de la solución de referencia. Las soluciones obtenidas de un movimiento a partir de una solución dada se suelen denominar vecinas de la primera, y al conjunto de vecinas, vecindario.
Los algoritmos de mejora pueden clasificarse en tradicionales y actuales. Los tradicionales son los de optimización local, y los métodos más recientes o actuales son dos, el recocido simulado y la búsqueda tabú, que evitan que el proceso desemboque en un óptimo local.
Dentro de los métodos exactos podemos destacar la metodología llamada de branch and bound, es decir, de separación y acotación. En esta metodología se basan muchos métodos exactos de resolución de problemas combinatorios. Debido a la importancia que dicha metodología tiene para este proyecto, se le dedica un capítulo separado a continuación.

6.4. Ejemplos de problemas combinatorios

De entre los problemas combinatorios que tienen más importancia dentro del campo de trabajo de la organización industrial, podemos destacar los siguientes:

Problema del *árbol parcial mínimo* (MST), problemas de *flujos* (transporte, afectación, acoplamiento, flujo máximo,...), problema del *cartero chino* (CPP), problema del *viajante de comercio* (TSP), problema general de *itinerarios* (GRP), problema de *afectación generalizado* (GAP), problema de *scheduling*, problema de *localización*, problema de *recubrimiento, partición y empaquetado*, problema de *coloreado de los nudos de un grafo* o problema de *la mochila* (KP).

7. Procesos de branch and bound

Los procesos de *branch and bound* (separación y acotación o exploración dirigida en castellano) son una técnica general de optimización para problemas combinatorios. Dentro de la definición de proceso de branch and bound caben muchos algoritmos con ciertas características comunes.

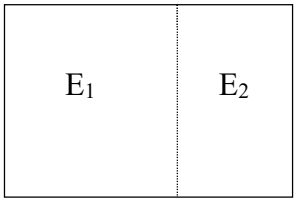
7.1. Exposición intuitiva

Supongamos que una función real f está definida para los elementos de un conjunto finito E, y deseamos determinar el mínimo de dicha función. Esto es:

$$[\text{MIN}]\ f(x) \qquad x \in E$$

Efectuemos una partición del conjunto E:

$$E_1 \cup E_2 = E \qquad \text{y} \qquad E_1 \cap E_2 = \emptyset$$



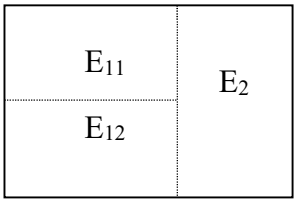
y determinemos k_1 y k_2 cotas inferiores de los valores de f en E_1 y E_2 respectivamente:

$k_1 \# f(x) \qquad \forall x \in E_1$

$k_2 \# f(x) \qquad \forall x \in E_2$

Supongamos que $k_1 < k_2$. Salvo otra información, parece más prometedor de contener el mínimo, el subconjunto E_1 que el subconjunto E_2. Seguimos explorando en E_1, lo que significa una nueva partición:

$$E_{11} \cup E_{12} = E_1 \qquad \text{y} \qquad E_{11} \cap E_{12} = \varnothing$$



y unas nuevas cotas:

$k_{11} \# f(x) \qquad \forall x \in E_{11}$

$k_{12} \# f(x) \qquad \forall x \in E_{12}$

Ahora el conjunto inicial E ha quedado descompuesto en tres subconjuntos: E_{11}, E_{12} y E_2. De los tres, aquel que posea menor cota será, en principio, el que ofrece más garantías de contener el mínimo y que seguiremos explorando.

Resumiendo, disponemos de dos procedimientos:

Procedimiento de *separación* (branch): permite una partición en un subconjunto dado, de acuerdo a ciertas características del mismo, generalmente de carácter combinatorio. Además dicho procedimiento deberá detectarnos las partes o subconjuntos que contienen un solo elemento (y que por tanto no pueden sufrir una partición propia).

Procedimiento de *acotación* (bound): permite determinar una cota de los valores de la función f en un subconjunto dado. Si el subconjunto se compone de un solo elemento la cota coincide con el valor de la función para dicho elemento.

Supongamos que aplicamos sistemáticamente ambos procedimientos, eligiendo como subconjunto a explorar o separar aquel que posee la menor cota de todos los subconjuntos disponibles. Debido a la finitud de E, en un momento determinado la menor cota corresponderá a un conjunto formado por un solo elemento. Dicho conjunto no puede ser separado y por tanto el algoritmo se parará. El óptimo es el elemento contenido en el conjunto y el valor del mínimo es dicha cota.

Evidentemente para un mismo problema podrán definirse distintos procedimientos de separación y acotación que cumplan las propiedades, puesto que de hecho puede hablarse de una familia de métodos más que de un método. Unos problemas serán más eficaces que otros, conduciendo al óptimo con menos pasos. En general el procedimiento de separación estará adaptado a la estructura de E y a la forma de f (por ejemplo permitiendo la eliminación rápida de subconjuntos malos). El procedimiento de acotación debería proporcionar una cota muy ajustada (cercana a coincidente con el mínimo del subconjunto) con poco trabajo.

Cabe señalar que lo que define que se ha alcanzado el óptimo es la posición instantánea del árbol en un instante y no la forma como se ha llegado a dicha posición, es decir el orden en que se han obtenido los vértices. Esto implica que la exploración a partir del vértice de menor cota es una buena estrategia de exploración, pero pueden existir otras, cuya importancia venga dada por la necesidad de hallar soluciones rápidamente (la de seguir siempre por la mejor cota de la última separación efectuada hasta llegar a vértices

finales, por ejemplo) o bien por entrar en juego aspectos ligados con el tamaño del árbol (o el tamaño de la información a conservar en cada momento para tener una buena representación del árbol).

En los branch and bound es importante conocer cuanto antes una solución posible cuanto antes, ya que ello permite eliminar ramas de la arborescencia, la cual tiende a expandirse si el algoritmo necesita muchas iteraciones para hallar la primera solución posible; es interesante, por lo tanto, hallar rápidamente una solución posible, bien sea con el propio algoritmo o con un algoritmo previo de tipo heurístico.

7.2. Formalización del proceso

Una formalización (para el caso de maximización) sería:

Sea el problema P: [MAX] $z = f(X)$ $X \in E$

Sea $E_R \supseteq E$; el problema R(P): [MAX] $z_R = f(X)$ $X \in E_R$

se denomina relajación de P y tiene las siguientes propiedades:

Para todo ejemplar de P:

1) $z_R \geq z^*$

2) Si X^* es óptimo en R(P) y $X^* \in E$, X^* es óptimo en P.

Una formalización del algoritmo de branch and bound para el problema P en el caso de minimización es:

```
inicio
        árbol = {E}
        valor_mejor := 4
        mientras árbol no vacío hacer
                sacar partición e del árbol
                si cota(e) < valor_mejor entonces
                        mientras queden hijos de e hacer
                                generar un nuevo hijo e' de e
                                acotar el nuevo hijo e'
                                si cota(e') < valor_mejor entonces
                                        si e' es terminal entonces
```

```
                                valor_mejor := cota(e')
                                mejor := e'
                                matar a los que no superarán a mejor
                            sino
                                añadir e' al árbol
                            finsi
                        finsi
                    finmientras
                finsi
            finmientras
        fin
```

El procedimiento termina en un número fijo de iteraciones si la arborescencia generada por el procedimiento de separación es finita.

7.3. Definición del algoritmo

Para definir un algoritmo, hay que concretar:

- El procedimiento de elección de la partición e del árbol. Los criterios pueden ser diversos, pero en general se tiene en cuenta:
  - · Valor de la cota del elemento.
  - · Posición del vértice en la arborescencia o carácter más o menos completo de la solución parcial que define el vértice.
  - · Accesibilidad de la información que define el vértice (soporte físico en que se encuentra).
  - · Disponibilidad de información para efectuar cálculos sobre el vértice.
- Procedimiento de separación. Si un vértice está definido por el valor que adoptan algunas variables es normal partir el vértice en los subconjuntos que quedan definidos al dar los valores que puede adoptar una de las variables libres.
- Relajación, es decir, procedimiento de acotación. Puede haber más de uno, tanto para el mismo vértice, como para los diversos vértices de la arborescencia.

7.4. Procedimientos heurísticos a partir de algoritmos branch and bound

Se puede convertir el procedimiento exacto y finito del branch and bound en un procedimiento heurístico prescindiendo de algunas de las condiciones que definen el algoritmo, como pueden ser:

- eliminar los vértices de peor cota entre los sucesores del vértice que se separa.
- eliminar los peores cuando no se dispone de más memoria
- fijar un tiempo límite para la ejecución del algoritmo y retener la mejor solución posible obtenida.

Un algoritmo de branch and bound puede requerir un tiempo de ejecución y una memoria considerables. Típicamente los algoritmos bien diseñados se comportan de modo que la primera solución posible que encuentran es de buena calidad y encuentran el óptimo relativamente pronto (algunos autores establecen un tiempo normal no superior al 25% del tiempo de total). Esto implica que la mayor parte del tiempo de ejecución del algoritmo se invierte en comprobar que se ha obtenido el óptimo. Este comportamiento hace que las heurísticas a partir de branch and bound sean generalmente muy eficientes, que generalmente hallen la solución óptima, aunque no podamos asegurarlo.

8. Problema del Cartero Chino (CPP)

8.1. Introducción y aplicaciones

Imaginemos el caso de un cartero que ha de realizar su trabajo en un determinado bloque de calles. Una cuestión que se puede plantear antes de empezar a repartir el correo es, que ruta debería seguir para poder recorrer todas las calles (pasando como mínimo una vez por cada uno) de manera que la distancia total recorrida sea mínima. Este es el llamado *problema del cartero chino* (*Chinesse Postman Problem (CPP)*, en inglés). El porqué del carácter oriental de este cartero se halla en el hecho de que el primero en considerar el problema fue el matemático chino Kwan Mei-Ko.

Las aplicaciones del CPP son variadas y diversas; quizás las más inmediatas puedan ser, aparte de la ya mencionada del reparto del correo, el recorrido de una máquina quitanieves, el recorrido de un camión de recolecta de desperdicios urbanos, o la ruta de una brigada de parques y jardines.

8.2. Formalización del problema

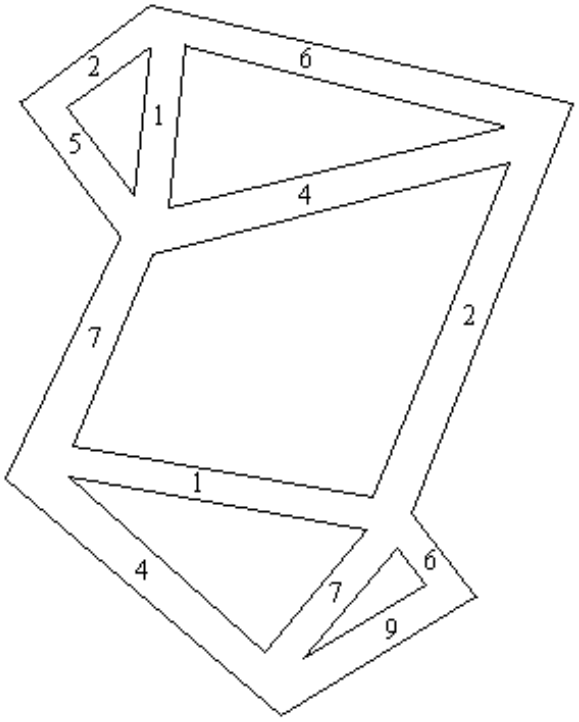
En términos de grafos, el planteamiento hecho por el cartero es el siguiente: dado un multigrafo G(V,A) simétrico, con coste $c_{ij} > 0$ asociado a cada arista (v_i,v_j), se pide hallar un ciclo (no necesariamente simple) que pase por cada arista y minimice su coste total.

Evidentemente, si el grafo G contiene un ciclo euleriano, este será la solución del problema; por tanto nos centraremos en los grafos que no son eulerianos.

8.2.1. Ejemplo

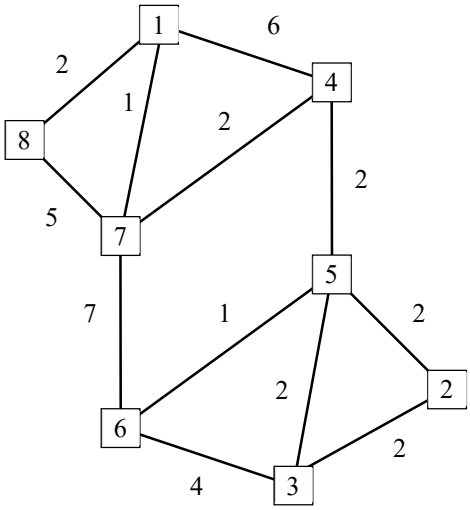
En lo que sigue de capítulo 8, así como en los puntos que sea necesario del capítulo 9, la explicación será ejemplificada con el siguiente ejemplar del problema. Es un caso sencillo que puede resolverse a simple vista, pero que es útil para introducir el problema del CPP.

Imaginemos una brigada municipal de parques y jardines que debe podar los árboles de las calles representadas por el siguiente plano:



Donde los costes, en este caso, no son directamente las distancias sino alguna función de éstas, como podría ser el tiempo o el combustible.

Este ejemplo se formalizaría de la siguiente manera:



Donde la numeración de los vértices es totalmente arbitraria y no condiciona en absoluto la resolución del problema.

Según el Teorema 4.1., G(V,A) no contiene ningún circuito euleriano, ya que tiene algún vértice (concretamente los vértices 1,3,4,6 de grado impar).

8.3. Método general de resolución

La idea central del procedimiento es conseguir que G(V,A) pase a tener circuito euleriano mediante la duplicación de algunas aristas. Naturalmente, la duplicación de aristas lleva como objetivo que *grado*(v_i) sea par para todo v_i de V, y esto tendrá que hacerse de forma que el coste adicional añadido sea el mínimo posible.

Si C_{ij} representa el camino de coste mínimo de v_i a v_j en G i $c(C_{ij})$ el coste, entonces el método siguiente formaliza el procedimiento antes apuntado.

Método de Edmonds:

1. Dado G(V,A), hacer una partición de $V=\{v_1,v_2,...,v_n\}$ en dos subconjuntos:

   $V_P := \{v_i \in V$ tal que grado(v_i) es par$\}$

   $V_I := \{v_i \in V$ tal que grado(v_i) es impar$\}$

2. Obtener $c(C_{ij})$ $\quad \forall\ v_i \in V, \forall\ v_j \in V, \quad v_i \neq v_j$.

3. Formar el grafo completo K_I sobre V_I con un coste asociado

   $c_{ij} := c(C_{ij})$ a cada (v_i,v_j).

4. Determinar un acoplamiento perfecto óptimo M^* en K_I.

5. Duplicar en G las aristas de los caminos C_{ij} correspondientes a los costes de las aristas de M^*.

6. El multigrafo resultante tiene circuito euleriano, ésta es la solución.

8.3.1. Ejemplo

Siguiendo con el ejemplar anterior, los puntos del método de Edmonds se concretarían de la siguiente forma:

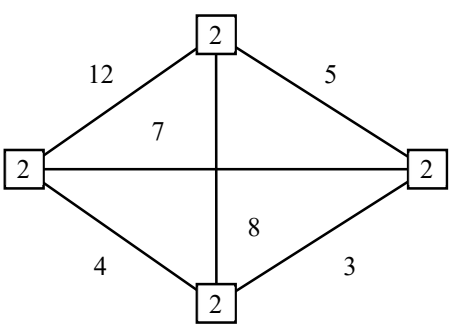
1. $V_P := \{2,5,7,8\}$

   $V_I := \{1,3,4,6\}$

2.

$c(C_{ij})$	**1**	**2**	**3**	**4**	**5**	**6**
1	0	13	12	5	7	8
2	13	0	9	8	6	7
3	12	9	0	7	5	4
4	5	8	7	0	2	3
5	7	6	5	2	0	1
6	8	7	4	3	1	0

3.



K_I

4. En este caso sencillo, el acoplamiento se puede hallar por simple enumeración y evaluación de las posibles soluciones. Esto solo es posible en casos de dimensiones tan reducidas como el presente ejemplo.

(1,3) , (4,6) → 12 + 3 = 15

(1,4) , (3,6) → 5 + 4 = 9

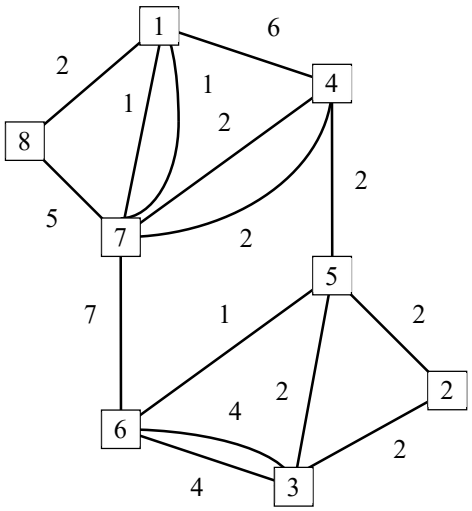
(1,6) , (3,4) → 8 + 7 = 15

Aquí se ve como $M^* = \{(1,4) , (3,6)\}$

5. A la arista de M^* (1,4) le corresponde un camino $C_{14} = (1,7) (7,4)$

   A la arista de M^* (3,6) le corresponde un camino $C_{36} = (3,6)$

   El multigrafo resultante es:



6. De acuerdo con el teorema 4.1, el multigrafo resultante es euleriano, ya que todos los vértices que lo forman tienen grado par.

Un posible circuito euleriano (de los varios posibles que existen) se puede obtener con alguno de los algoritmos del capítulo 5.

Uno podría ser:

1 - 4 - 7 - 6 - 3 - 2 - 5 - 3 - 6 - 5 - 4 - 7 - 8 - 1 - 7 - 1

9. Obtención del acoplamiento

9.1. Método clásico

Un algoritmo para encontrar un acoplamiento perfecto óptimo en K_I se puede deducir directamente del algoritmo para determinar un acoplamiento de coste mayor. Efectivamente, basta con sustituir en K_I cada coste c_{ij} por $(W - c_{ij})$, siendo W una constante que supera el coste de cada una de las aristas de K_I. A continuación se debe determinar un acoplamiento de coste mayor en el grafo resultante.

9.2. Aplicación de un algoritmo de branch and bound

En la resolución del problema del cartero chino, la gran dificultad reside en el algoritmo de resolución del acoplamiento, necesario para duplicar aristas con un coste mínimo. Para este paso del método general de resolución, en este proyecto se propone un método no tan específico y conceptualmente más sencillo, basado en un algoritmo de branch and bound.

Este branch and bound consiste básicamente en, a partir de la matriz de caminos mínimos entre todos los pares de vértices de grado impar, hallar que grupo de aristas debemos duplicar para:

- cumplir la condición de acoplamiento, es decir que cada vértice de grado impar solo forme parte de una arista duplicada; de manera que al final quede un grafo con todos los vértices de grado par y por tanto euleriano.
- cumplir la condición de que la suma de los valores asociados a las aristas duplicadas sea el mínimo posible.

9.2.1. Elementos previos

Antes de entrar en el branch and bound propiamente dicho, se han definido una serie de elementos auxiliares, como son:

- matriz de caminos mínimos entre todos los pares de vértices de grado impar en el grafo original (m0).

- lista de vértices susceptibles de ser usados en el acoplamiento, ordenados de menor a mayor y que es actualizada de acuerdo con la compatibilidad con cada vértice a explorar (l0).

9.2.2. Caracterización de los nodos

Cada nodo del branch and bound quedará definido por las aristas tomadas y rechazadas, es decir, los elementos del subconjunto son los acoplamientos que pueden formarse a partir de las aristas tomadas y las no rechazadas.

9.2.3. Procedimiento de separación

El procedimiento de separación consistirá en considerar la primera arista de la lista de aristas actualizada (l0) para el nodo a explorar y separar éste en dos nuevos nodos, uno en que se toma la arista y otro en que se rechaza. La partición es por tanto binaria.

9.2.4. Procedimiento de acotación

El procedimiento de acotación consistirá en completar el valor de las aristas tomadas con las no rechazadas, compatibles con las tomadas hasta el momento, pero con la relajación de no tener en cuenta las posibles incompatibilidades entre sí.

A partir de la matriz m0 (de dimensión imp × imp) de caminos mínimos entre todos los pares de vértices de grado impar, en la que han sido eliminados los elementos correspondientes a aristas inutilizables por la definición de acoplamiento, de acuerdo con las aristas cogidas o rechazadas por la partición actual, podemos definir:

$$c_1 = \Phi\left(\frac{\sum_i \min_j(m0(i,j))}{imp/2}\right)$$

donde Φ indica el mínimo entero superior

$$c_2 = \sum_{\forall C_{i,j} \in A} c(C_{i,j})$$

donde A es el conjunto formado por todas las aristas de K_I cogidas por el vértice al cual se calcula la cota.

De esta forma nos queda que la cota en cuestión es:

$$cota = c_1 + c_2$$

Donde la primera parte es una valoración de lo máximo a que podemos aspirar con las aristas que nos quedan por coger, mientras que la segunda parte es simplemente la suma de las aristas ya cogidas.

9.2.5. Elección del nodo a explorar

Al elegir la partición del árbol se hace en función de dos posibles criterios, la menor cota del nodo o el número de aristas cogidas en un nodo. La explicación de uno u otro criterio se efectuará en el capítulo de comportamiento del algoritmo, donde se discutirá cual es más conveniente en cada momento de la ejecución del mismo.

9.2.6. Ejemplo

Los elementos previos antes del B&B serían:

m0	**1**	**3**	**4**	**6**
1	-	12	5	8
3	12	-	7	4
4	5	7	-	3
6	8	4	3	-

l0	(4,6)	(3,6)	(1,4)	(3,4)	(1,6)	(1,3)

El nodo raíz se formaría de la siguiente forma:

nº0	*1*	*3*	*4*	*6*	*min*
1	-	12	5	8	**5**
3	12	-	7	4	**4**
4	5	7	-	3	**3**
6	8	4	3	-	**3**

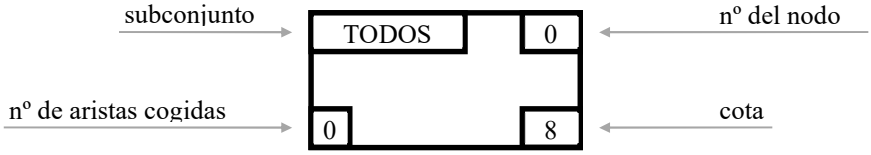
$\sum$ = **15**

c_1 (0) = Φ (15 / (4/2)) = 8

c_2 (0) = 0

cota (0) = $c_1 + c_2$ = 8 + 0 = 8

con lo que el nodo raíz nos queda:



El siguiente paso es explorar un nodo, en este caso solo podemos explorar el nodo raíz. El proceso de separación consiste aquí en elegir la primera arista usable de la lista l0 → (4,6) y separar el nodo a explorar en dos, uno cogiendo y otro rechazando la arista elegida.

El subconjunto de coger (4,6), quedaría:

nº1	*1*	*3*	*4*	*6*	*min*
1	-	12	-	-	**12**
3	12	-	-	-	**12**
4	-	-	-	-	**-**
6	-	-	-	-	**-**

$\sum$ = **24**

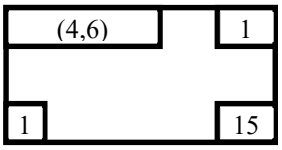
Las filas y columnas de vértices de aristas cogidas tienen valor nulo, ya que dichos vértices ya no pueden participar en otra arista.

c_1 (1) = Φ (24 / (4/2)) = 12

c_2 (1) = c(C_{34}) = 3

cota (1) = $c_1 + c_2 = 12 + 3 = 15$

con lo que el nodo 1 nos queda:



El subconjunto de dejar (4,6), quedaría:

nº2	***1***	***3***	***4***	***6***	***min***
1	-	12	5	8	**5**
3	12	-	7	4	**4**
4	5	7	-	-	**5**
6	8	4	-	-	**4**

$\sum$ = **18**

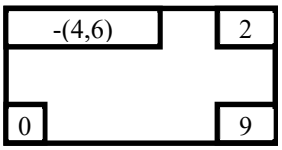
En este caso, sabemos que en el futuro no volveremos a usar l a arista rechazada, por la que pasa a tener valores nulos.

c_1 (2) = Φ (18 / (4/2)) = 9

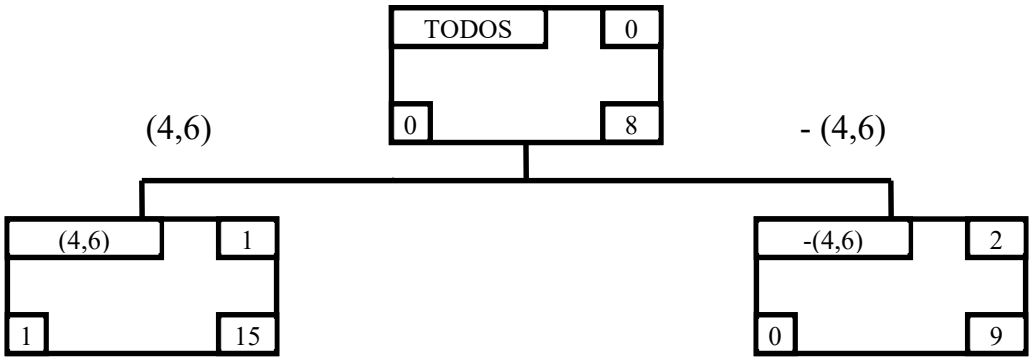
c_2 (2) = 0

cota (2) = $c_1 + c_2 = 9 + 0 = 9$

con lo que el nodo 2 nos queda:



El árbol construido hasta ahora sería:



En este punto, tenemos dos posibles nodos a explorar, supongamos que elegimos el de menor cota, es decir el nodo 2.

La lista l0 nos quedaría:

l0	-	(3,6)	(1,4)	(3,4)	(1,6)	(1,3)

ya que la arista (4,6) no está en el subconjunto que exploramos.
La arista escogida, en este caso sería (3,6), por lo que la separación del subconjunto representado por el nodo 2 sería en función de esta arista.

El subconjunto de coger (3,6), quedaría:

nº3	***1***	***3***	***4***	***6***	***min***
1	-	-	5	-	**5**
3	-	-	-	-	**-**
4	5	-	-	-	**5**
6	-	-	-	-	**-**

$\sum$ = **10**

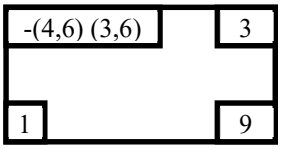
Las filas y columnas de vértices de aristas cogidas tienen valor nulo, por el razonamiento expuesto anteriormente.

c_1 (3) = Φ (10 / (4/2)) = 5

c_2 (3) = c(C_{36}) = 4

cota (3) = $c_1 + c_2 = 5 + 4 = 9$

con lo que el nodo 3 nos queda:



El subconjunto de dejar (3,6), quedaría:

nº4	*1*	*3*	*4*	*6*	*min*
1	-	12	5	8	**5**
3	12	-	7	-	**7**
4	5	7	-	-	**5**
6	8	-	-	-	**8**

$\sum$ = **25**

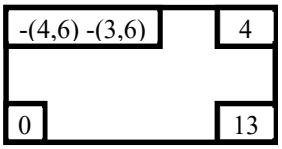
En este caso, sabemos que en el futuro no volveremos a usar las aristas rechazadas, por la que pasa a tener valores nulos.

c_1 (4) = Φ (25 / (4/2)) = 13

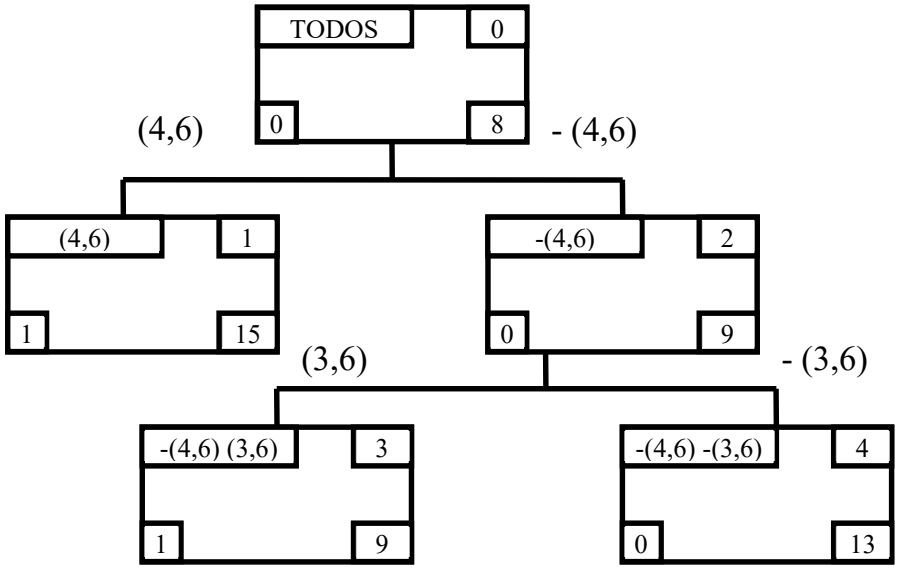
c_2 (4) = 0

cota (4) = $c_1 + c_2$ = 13 + 0 = 13

con lo que el nodo 4 nos queda:



El árbol construido hasta ahora sería:



Tenemos tres posibles nodos a explorar, supongamos que seguimos eligiendo el de menor cota, es decir el nodo 3.

La lista l0 nos quedaría:

l0	-	-	(1,4)	-	-	-

ya que la arista (4,6) no está en el subconjunto y al incluir la arista (3,6), se imposibilita la entrada de cualquier arista que tenga algún vértice en común.

La arista escogida, en este caso sería (1,4), por lo que la separación del subconjunto representado por el nodo 2 sería en función de esta arista.

El subconjunto de coger (1,4), quedaría:

nº5	*1*	*3*	*4*	*6*	*min*
1	-	-	-	-	-
3	-	-	-	-	-
4	-	-	-	-	-
6	-	-	-	-	-

$\sum = \mathbf{0}$

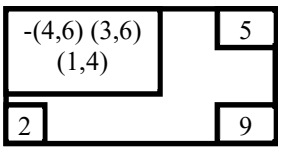
Las filas y columnas de vértices de aristas cogidas tienen valor nulo, ya que dichos vértices ya no pueden participar en otra arista.

c_1 (5) = Φ (10 / (4/2)) = 0

c_2 (5) = c(C_{36}) + c(C_{14}) = 4 + 5 = 9

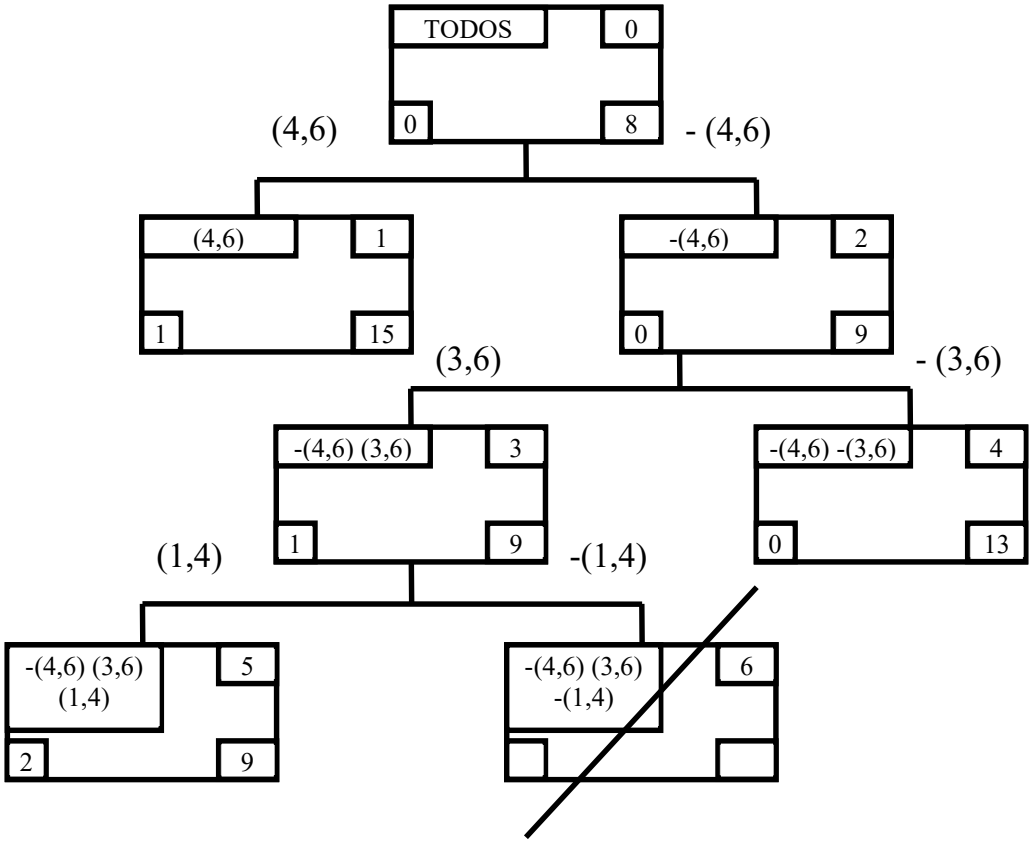
cota (5) = c_1 + c_2 = 0 + 9 = 9

con lo que el nodo 5 nos queda:



Llegados a este punto, el nodo correspondiente a dejar la arista (1,4), no puede formar parte del árbol, ya que como vemos en la lista l0 no nos queda ninguna otra arista para poder coger y formar el acoplamiento. El subconjunto formado sería vacío.

El árbol construido hasta ahora sería:



En este punto del B&B, volvemos a elegir el nodo de menor cota, el nodo 5, de cota 9. Vemos que es un nodo terminal, es decir tiene 2 aristas cogidas, que son la que necesitamos para este acoplamiento (había 4 vértices de grado impar). Esto nos indica que hemos llegado al óptimo, formado por los vértices, de K_I, (3,6) y (1,4).

10. Implementación del algoritmo

10.1. Estructura del algoritmo

En la implementación del algoritmo se ha optado por la técnica de archivos separados, consistente en que cada bloque se almacena y compila por separado, uniéndose posteriormente para formar el archivo ejecutable. Las principales ventajas son una mayor sencillez en el depurado del programa, ya que cada parte puede ser testeada por separado y una mayor rapidez en las compilaciones, ya que solo se compilan las partes que han sido modificadas.

Consta de los siguientes bloques o módulos:

- principal o *main*: es el núcleo del algoritmo, la acción principal que *llama* a las diferentes acciones que forman parte de los distintos módulos (lo que no le impide tener sus propias acciones).
- secundarios o librerías: contienen todo lo necesario (declaraciones de estructuras, acciones,…) para gestionar su trabajo independientemente del *main*. Aparte de las librerías propias del lenguaje C, en este algoritmo se han implementado las siguientes:
    - librería de gestión del árbol usado en el branch and bound.
    - librería de gestión del grafo almacenado en listas de adyacencia.
    - librería con la acción necesaria para hallar circuitos eulerianos.
    - librería de gestión de matrices simétricas utilizando la mínima memoria.
    - librería para hallar el camino mínimo entre toda pareja de vértices según el algoritmo de Fulkerson.

A cada librería le corresponde un fichero *cabecera* donde está toda la información necesaria para el usuario de la librería (el programa *main*).

En algunos casos, el nombre de *librería* no es del todo correcto, ya que solo contiene una acción, pero se ha optado usar la misma denominación.

10.2. Bloque principal

El algoritmo principal está basado directamente en los propuestos en los capítulos 7 y 8 para el problema del cartero chino y procesos branch and bound en general, una exposición con mayor detalle sería la siguiente.

```
inicio
almacenar grafo
hallar los caminos mínimos ∀ pareja de vértices
si (el número de vértices de grado impar > 0)
        iniciar el proceso de B&B para hallar el acoplamiento
                inicializar árbol
        valor_mejor := 4
                mientras árbol no vacío hacer
                        sacar partición del árbol
                        si (cota(partición) < valor_mejor) entonces
                        mientras queden hijos hacer
                                        generar hijo
                                        acotar hijo
                                        si (cota(hijo) < valor_mejor) entonces
                                                si (hijo es terminal) entonces
                                                        valor_mejor := cota(hijo)
                                                        mejor := hijo
                                                        matar a los que no superarán a mejor
                                        finsi
                                                sino
                                                        añadir el hijo al árbol
                                                finsino
                                        finsi
                        finmientras
                        finsi
                finmientras
        fin del proceso de B&B
```

añadir al grafo la solución del B&B

finsi

hallar un ciclo euleriano en el ciclo definitivo

presentar y guardar los resultados

fin

10.3. Estructuras de datos

Aparte de matrices y vectores usados puntualmente en alguna parte del algoritmo, las principales estructuras para el almacenamiento de datos que se han usado son: almacenamiento del árbol, almacenamiento del multigrafo y almacenamiento de matriz simétrica.

10.3.1.Implementación del árbol

Está en el fichero "arbol.c" y su cabecera "arbol.h".

10.3.1.1. Estructura de datos del árbol

Básicamente la estructura de datos que almacena el árbol que contiene los vértices pendientes de ser tratados es una lista circular doblemente enlazada.

Una lista doblemente enlazada se comporta como una lista de nodos ordenados según dos criterios. La implementación circular consiste en que el nodo siguiente al último es el primero.

Al estar implementada usando la asignación dinámica de memoria para optimizar el espacio ocupado necesitamos una serie de punteros:

- Cada nodo tiene cuatro punteros, que indican cuáles son sus nodos anterior y posterior para cada criterio de ordenación. Como en toda lista circular los primeros y últimos se señalan entre sí.
- Para cada criterio de ordenación necesitamos un puntero que indique cuál es el primer elemento de la lista.

En esta implementación tenemos dos ordenaciones:

- Orden 1: ordena la lista por cotas. Un nodo estará delante de otro si el vértice que contiene tiene cota menor. En caso de empate por cota, tendrá prioridad el de mayor número de aristas cogidas.

- Orden 2: ordena la lista por número de aristas cogidas por el vértice del árbol asociado al nodo. Un nodo estará delante de otro si el número de aristas cogidas es mayor. En caso de empate se dará prioridad al nodo de menor cota.

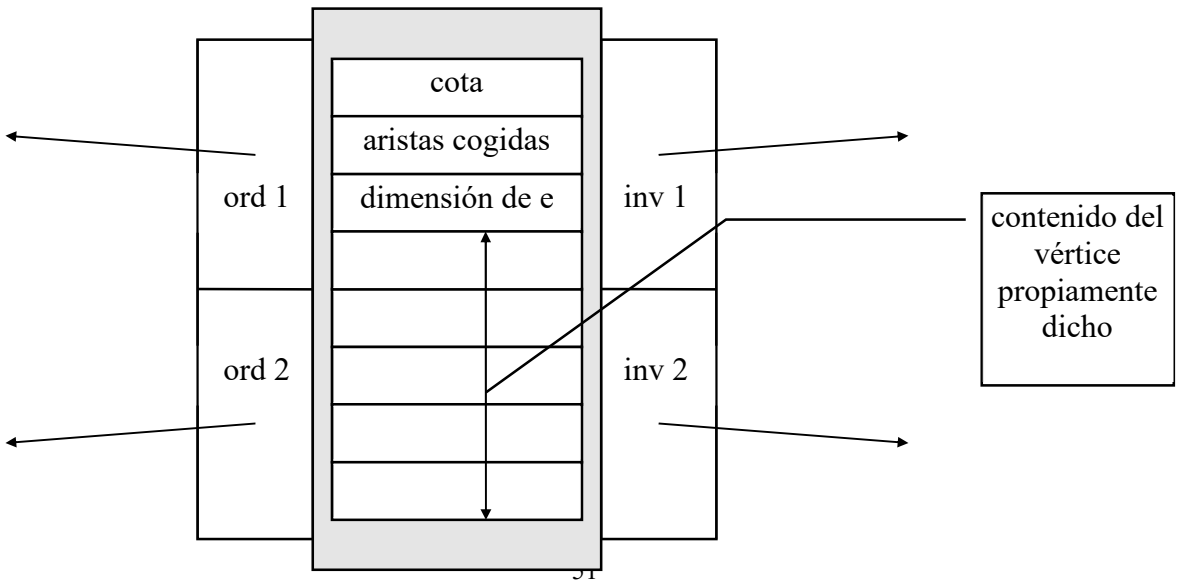
10.3.1.2. Estructura de los elementos del árbol

Los elementos que forman parte del árbol están formados por los siguientes campos:

- ord1: puntero que señala al nodo siguiente según el orden 1.
- ord2: puntero que señala al nodo siguiente según el orden 2.
- inv1: puntero que señala al nodo anterior según el orden 1.
- inv2: puntero que señala al nodo anterior según el orden 2.
- e: vector que contiene al elemento propiamente dicho. Tiene los siguientes componentes:
  - e[0]: contiene la cota del vértice asociado al nodo.
  - e[1]: contiene el número de aristas cogidas en el vértice asociado al nodo.
  - e[2]: contiene la dimensión (dim) del vector e.

Cabe destacar, que debido a la naturaleza del problema a tratar, la *dimensión de e varía de un elemento a otro*, en función del número de aristas cogidas o dejadas. Para conseguir una optimización de los recursos de memoria, se dimensiona cada vez el vector e dependiendo de dicha dimensión.

  - e[3] ... e[dim]: contienen los códigos de las aristas cogidas y dejadas por el vértice asociado al nodo. Dichos códigos son la posición de la arista dentro de la lista l0 antes citada, con signo negativo en caso de ser aristas dejadas. Con esto se consigue representar una arista y su estado (cogida o rechazada) con un solo entero.



10.3.1.3. Operaciones sobre el árbol

Las operaciones que pueden ser realizadas sobre el árbol son las siguientes:

- Poner nodo: inserta un nuevo nodo en el árbol, arreglando los punteros para que quede colocado en la posición que le corresponde por cada orden.

Concretamente los pasos a seguir son:

1 - reservar memoria para un nodo.

2 - dimensionar el vector representante del vértice, en función del número de aristas cogidas y rechazadas.

3 - establecer que en el árbol hay nodos.

4 - copiar el contenido del vértice en el nodo.

5 - gestionar los 6 punteros.

6 - incrementar los contadores.

- Sacar nodo: elige, extrae y libera la memoria de un nodo del árbol. Este nodo se elige de acuerdo con la estrategia de extracción, función del número de elementos contenidos en el árbol.

Sigue los siguientes pasos:

1 - comprobar que el árbol no está vacío.

2 - establecer el criterio de extracción (en función del número de nodos almacenados).

3 - sacar el primero de la lista, copiarlo en el vector de salida y eliminarlo.

4 - gestionar los 6 punteros.

5 - decrementar el contador de elementos actuales.

- Matar nodos: elimina del árbol los elementos con cota superior a una dada.
- Inicializar árbol: Prepara el árbol para que se inserten nodos.
- Finalizar árbol: vacía el árbol y libera la memoria ocupada.

10.3.2.Implementación del almacenamiento del multigrafo

El módulo que gestiona este almacenamiento está en el fichero “grafo.c” y su cabecera es “grafo.h”.

10.3.2.1. Estructura de datos que almacena el multigrafo

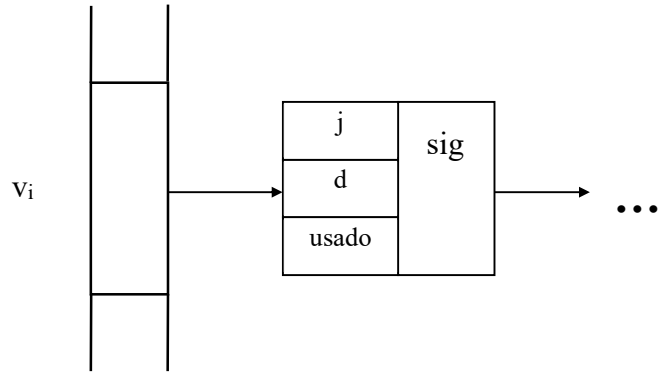
El multigrafo original y las aristas duplicadas a partir del acoplamiento son almacenados mediante un modelo de listas de adyacencia, más concretamente implementado mediante colas.

Para cada vértice $v_i \in V$ existe una lista (gestionada a nivel interno como una cola), formada por todos los vértices $v_j \in V$ adyacentes a v_i.

10.3.2.2. Estructura de los elementos

Los elementos que forman parte de cada lista i tienen los siguientes campos:

- j: indica el vértice v_j unido al v_i mediante la arista contenida en este elemento.
- d: indica el valor c_{ij} de la arista asociada al elemento.
- usado: campo que nos sirve para concretar si la arista en cuestión ha sido usada ya en el recorrido del ciclo euleriano.
- sig: puntero que señala al siguiente elemento de la lista.



10.3.2.3. Operaciones sobre el módulo de almacenamiento del multigrafo

Las operaciones definidas sobre este módulo son las siguientes:

- preparar listas: inicializa n listas vacías, en función del número de vértices del grafo.
- poner en la lista una arista: introduciendo los valores de una arista (simétrica), la almacena en las dos listas necesarias.
- poner en la lista un arco: almacena un arco en la lista correspondiente. En este caso, esta acción tan solo es usada internamente por la acción anteriormente descrita, pero se ha optado por hacerla pública para dar mayor generalidad al módulo.

- establecer todas las aristas almacenadas como no usadas: acción que permite en cualquier momento designar como no usadas las aristas almacenadas. Normalmente es usada al introducir las aristas.
- establecer una arista como usada: permite "marcar" las aristas ya usadas.
- sacar de la lista: saca la primera arista de la lista indicada.
- ver las aristas incidentes: permite conocer cuáles son las aristas incidentes, ya sean usadas, no usadas, o en total, en cualquier vértice en cualquier instante.
- ver un elemento: permite conocer un elemento de la lista, sin sacarlo de la lista.
- número de aristas incidentes: retorna el número de aristas incidentes en un vértice.
- número de aristas incidentes en un determinado estado: retorna el número de aristas incidentes en un vértice dado en estado usado o no usado.
- número de vértices: retorna el número de vértices del grafo tratado.
- eliminar el grafo: libera la memoria ocupada por todas las listas.

10.3.2.4. Almacenamiento en matrices

Una parte del multigrafo inicial, antes de entrar en el proceso de branch and bound, debe ser tratada por el algoritmo de Fulkerson. Tan solo es una parte del multigrafo inicial, porque si entre dos vértices existe más de una arista, tan solo se tendrá en cuenta la de menor valor. Esto, juntamente con que en el momento de la ejecución este paso es anterior al desarrollo del árbol (parte que ocupa la práctica totalidad de la memoria usada por el algoritmo), hace que durante una parte al inicio del algoritmo haya una duplicidad de la información almacenada. Cabe reseñar que dadas las características del problema, esto no es crítico, ya que el espacio ocupado por tres matrices representantes de un problema resoluble es mucho menor que la memoria disponible.

Al salir del algoritmo de Fulkerson, la matriz de datos iniciales es eliminada, la de resultados es almacenada en memoria reducida al ser simétrica y haber un gran número de accesos a ella durante la ejecución del programa, y la matriz de trazas es almacenada en disco, ya que siempre se necesita, pero un número reducido de veces.

10.3.3. Almacenamiento de una matriz simétrica

Al tener que guardar una matriz simétrica en memoria, debido a que existe un número elevado de accesos a ella, es conveniente optimizar la memoria utilizada usando menos de la mitad del espacio requerido para la matriz entera (si la diagonal tiene valor constante).

Concretamente, si la matriz es de dimensión n × n, el número de posiciones de memoria necesarios son $(n^2 - n) / 2$. Este módulo está en el archivo "simetr.c" y en su cabecera "simetr.h"

10.3.3.1. Estructura de datos de la matriz simétrica reducida

La estructura de datos utilizada es simplemente un vector de $(n^2 - n) / 2$ componentes, en el que se almacena la submatriz superior a la diagonal principal.

10.3.3.2. Operaciones disponibles

Las operaciones son dos:

- reducir la matriz: a partir de la matriz original establece el vector de dimensión reducida.
- acceder a la información: a partir de la posición en la matriz original, realiza la descodificación necesaria para hallar la posición actual en el vector y retorna el valor requerido.

10.4. Implementación del algoritmo de caminos eulerianos

Este algoritmo ha sido implementado como una acción compilada en un módulo separado. El fichero fuente está en "eulerian.c" y la cabecera en "eulerian.h". El algoritmo implementado es el descrito en el punto 5.2.2.

10.5. Implementación del algoritmo de Fulkerson de caminos mínimos

Este es otro de los bloques compilados por separado. Se corresponde al fichero fuente "fulker.c" y a la cabecera "fulker.h".

10.5.1.Introducción

Para hallar la matriz de caminos mínimos entre todas las parejas de vértices $(v_i,v_j) \in V$, ha sido utilizado el algoritmo de Fulkerson.

Dicho algoritmo maneja simultáneamente tres matrices de dimensión $|V| x |V|$.

En la matriz L se guardan los datos iniciales del grafo, en P se obtendrá la solución al problema y N permitirá reconstruir el camino mínimo, ya que el elemento n_{ij} es el vértice siguiente al i en el camino de i a j.

Las iteraciones consisten en:

En la iteración q, $\forall$ i y $\forall$ j, hacer:

$p^q_{ij} = \min \{ p^{q-1}_{ij}, \min (c_{ik} + p^{q-1}_{ij}))$

si $p^q_{ij} < p^{q-1}_{ij} \Rightarrow n_{ij} = k$

Es decir, consiste en encontrar el mínimo de todos los caminos posibles entre i y j, formados por un arco entre i y k seguido del camino mínimo desde k hasta j hallado en la iteración anterior. Si en el curso de la iteración actual se ha hallado un camino más corto que los anteriores, se almacena su valor en la matriz P, y el vértice k en la matriz N, el cual da la dirección a seguir para reconstruir este camino.

El algoritmo termina cuando durante una iteración no se ha encontrado ninguna mejora, esto es cuando $\forall$ i ,$\forall$ j , $p^q_{ij} = p^{q-1}_{ij}$

10.5.2.Algoritmo

El algoritmo es el siguiente:

```
inicio
        L := P
        si l_ij < 4 entonces
                m_ij := j
        finsi
        sino
                m_ij := 4
        finsino
```

```
        hacer
                fin := 1
                para i = 1 hasta nv
                        para j = 1 hasta nv
                                para k = 1 hasta nv
                                        si (pij > lik + pkj) hacer
                                                pij := lik + pkj
                                                mij  := k
                                                fin := 0
                                        finsi
                                finpara
                        finpara
                finpara
        hasta fin = 1
fin
```

10.6. Almacenamiento de grafos en ficheros

Los grafos simétricos utilizados en este problema son introducidos via fichero de texto con el siguiente formato:

Nº de vértices
Nº de aristas
Vértice Vértice Coste
Vértice Vértice Coste
Vértice Vértice Coste
…

En los ejemplos propuestos, los ficheros de grafos tienen la extensión GRA.

10.7. Hardware utilizado

El ordenador utilizado tanto para la programación y compilación como para la obtención de los resultados expuestos ha sido un Pentium Intel 150 Mhz, con pipeline cache 256.

10.8. Modelos de compilación

El compilador utilizado (Turbo C) dispone de varios modelos de compilación, en función de las características del programa (dimensión del programa fuente y memoria requerida). En este programa se ha optado por el modelo llamado *Compact*, que proporciona una cantidad de memoria RAM de 640 Kb a cambio de un acceso a cada posición de memoria ligeramente más lento que en los modelos menores. No se ha optado por un doble modelo de compilación *Compact* y otro menor (*Small*) para problemas pequeños, ya que en los problemas que no es necesario el modelo *Compact*, el tiempo de resolución es pequeño para ambos (del orden de unos pocos segundos). Cabe destacar que la ejecución desde el sistema operativo DOS es alrededor de un 8 % más rápido que desde Windows 95.

11. Comportamiento del algoritmo

En general este algoritmo se comporta de la siguiente manera:
Al principio el objetivo es encontrar una solución posible, aunque generalmente no óptima, de forma que cuando es hallada, el algoritmo se dedica a comprobar que sea óptima. Si no lo es, al encontrar una mejor nos la quedamos y sigue el proceso.
En principio, la diferencia entre las soluciones halladas es decreciente, mientras que el tiempo entre encontrar dos soluciones consecutivas es creciente, de manera que para problemas grandes no nos interesará tanto llegar hasta el final de la ejecución como obtener una solución bastante buena, generalmente óptima, aunque no podamos asegurarlo. Esto convierte el procedimiento exacto en una heurística bastante correcta.
Las series de problemas usadas a continuación están descritas en los anexos.

11.1. Criterios de extracción

El comportamiento del programa, por lo que se refiere a los tiempos de obtención de soluciones, puede variar en función del criterio de extracción de las particiones del árbol. Los criterios utilizados son dos:
- extraer el vértice de menor cota. Es una buena estrategia al principio, pero generalmente hace que el árbol crezca demasiado, lo que puede llegar a producir problemas de memoria y una mayor lentitud, tanto para hallar la solución óptima como para terminar el algoritmo.
- extraer el vértice con más aristas cogidas, y que consecuentemente está más cerca de ser terminal. A partir de una cierta dimensión del árbol este criterio es mejor, ya que nos lleva a nodos terminales, obtenemos soluciones más rápidamente y el algoritmo es más rápido al solo tener que comprobar si la mejor solución obtenida es óptima.
A partir de la experimentación del programa, para problemas de distintas características y dimensiones se ha obtenido que un buen número de nodos del árbol para cambiar el criterio de extracción podría ser 100. Cuando el árbol contiene menos de 100 nodos, se sigue el criterio de menor cota, mientras que a partir de 100 nodos, se extrae al que tenga más aristas cogidas. Con esto conseguimos buenas soluciones rápidas en problemas grandes, a cambio de pasar por más soluciones obtenidas hasta llegar al

óptimo en problemas pequeños. Esto no es importante, ya que no perjudica seriamente al tiempo de obtención del óptimo.

11.2. Dimensiones de los problemas

En función del tiempo de ejecución podemos clasificar los problemas en tres grupos (el número de vértices de grado impar indicados puede variar de un caso a otro según lo expuesto en el siguiente punto):

Pequeños: serían los problemas de aproximadamente hasta 14 vértices de grado impar. Los tiempos de resolución son muy pequeños o inapreciables.

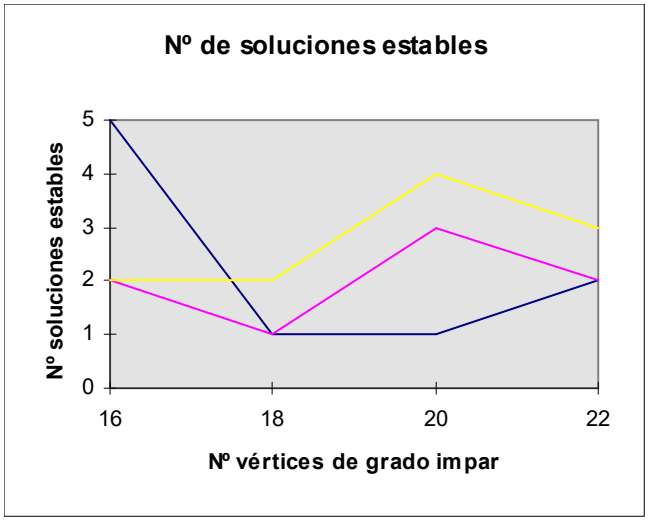
Medianos: Aquí estarían los casos de entre 14 y 22 nodos. Es razonable esperar hasta el final de la ejecución para obtener la solución óptima (alrededor de 75 minutos en el caso máximo)

Grandes: Serían los problemas mayores. Rápidamente obtenemos buenas soluciones, pero el tiempo total de ejecución se dispara exponencialmente.

11.3. Predicción de tiempos de ejecución

Las gráficas siguientes nos llevan a la conclusión de que los tiempos, tanto de hallar una solución estable supuestamente óptima, como total de ejecución, no son fácilmente predictibles a partir de los datos iniciales del problema, como sería el número de vértices de grado impar, el número de aristas o el número total de vértices.

A pesar de que hay una cierta relación exponencial o factorial entre el orden de magnitud del tiempo total de ejecución y el número de vértices de grado impar, dicha relación depende mucho de los datos concretos de cada problema, lo que hace bastante inviable una predicción a priori.

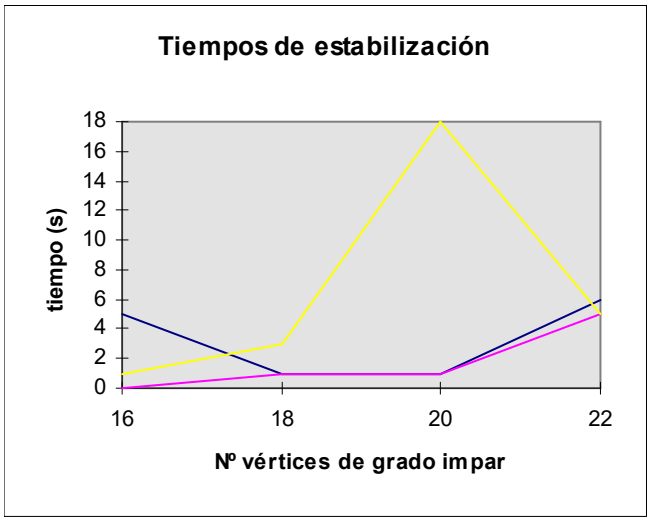


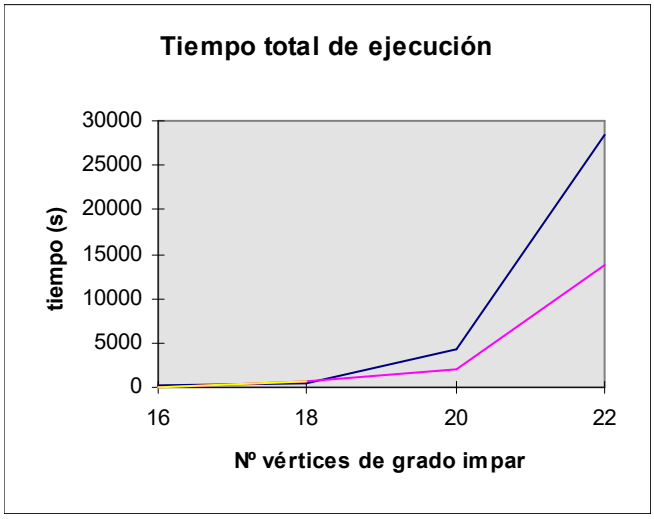
Azul: caso Barri de Mar

Rosa: caso Barri del Port

Amarillo: caso Barri de la via

Todos estos casos se explican en el Anexo 2.





12. Obtención de un procedimiento heurístico

Según lo expuesto en el apartado referente a procedimientos de tipo branch and bound y la obtención de heurísticas, generalmente bastante correctas, prescindiendo de algunas de las condiciones que definen el algoritmo, en nuestro caso podemos indicar algunas reglas *sensatas* para intentar que en los problemas en que los recursos necesarios crezcan por encima de nuestras posibilidades, obtengamos soluciones que aunque no podamos asegurar su optimalidad, podamos saber que no se alejan de ésta.

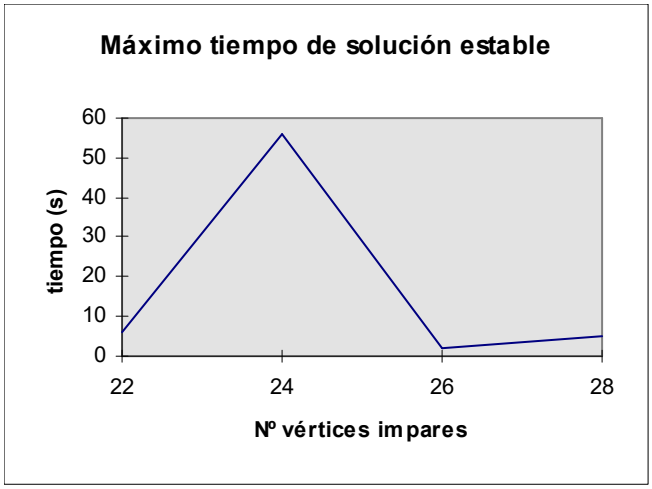
La mayoría de autores suelen proponer distintas reglas en función de cuál sea el recurso limitante en cada caso.

Cuando se dispone de menos memoria que la necesaria para la ejecución total del programa, se puede optar por, o bien eliminar los vértices de peor cota entre los sucesores del vértice que se separa (indicado sobre todo para separaciones no binarias, en las que a partir de un vértice podemos obtener sucesores de cotas muy diversas), o bien eliminar los vértices de peor cota en el momento en que no se dispone de más memoria.

Otro método quizás más sofisticado consistiría en eliminar los vértice en los que la cota es superior al mejor valor conocido hasta el momento más un valor ε, con $\varepsilon > 0$, y que puede ser función de distintos parámetros, como la posición del vértice en el árbol.

En nuestro caso el problema no es de memoria, sino de tiempo. Como ya hemos visto, para problemas normalmente mayores de una veintena de vértices de grado impar, la obtención del óptimo se hace prácticamente inabordable. Es por ello que una heurística sencilla y que no por esto proporciona malos resultados consiste en fijar un tiempo máximo de ejecución suficientemente grande para pasar las primeras soluciones no demasiado buenas y nos lleve cerca del óptimo.

Como ya hemos estudiado anteriormente, la predicción de un tiempo o un número de soluciones a pasar en función de los parámetros del problema es muy variable y depende mucho de cada caso en concreto. Es por esto que aquí se presentan los valores obtenidos a partir de los casos estudiados, para que conjuntamente con la características concretas de cada problema proporcionen una orientación para fijar el tiempo máximo de ejecución en cada caso.



En esta gráfica se observa que el máximo se alcanza para 24 vértices de grado impar. Esta es una característica común en los casos estudiados, lo que podría ser debido a que para los problemas mayores a este no se haya esperado un tiempo suficiente para obtener de forma heurística la solución estable definitiva.

13. Conclusiones

A partir de la implementación y experimentación del comportamiento de un proceso de branch and bound para el cálculo del acoplamiento del problema del cartero chino, podemos obtener las siguientes conclusiones:

Este procedimiento nos proporciona rápidamente el óptimo (en unos pocos segundos) para problemas pequeños y medianos (en general hasta problemas de una veintena de vértices de grado impar), mientras que para problemas mayores se indica una heurística que generalmente proporciona muy buenos resultados.

El hecho de que no podamos asegurar el óptimo para todos los problemas no es tan grave como en un principio pudiera parecer, ya que dependiendo del problema real a resolver nos podemos encontrar con situaciones atenuantes.

La primera es darnos cuenta de que en algunos casos, puede no compensarnos el tiempo necesario para la obtención del óptimo frente a la ligera mejora que obtendríamos al pasar de la segunda mejor solución (obtenida rápidamente) a la solución óptima. Esto se produciría principalmente en la resolución de problemas singulares que por tanto se han de realizar una sola vez (o unas pocas), como podría ser la situación de urgencia en que necesitamos la actuación de una máquina quitanieves.

Otra situación sería la que se produce al aplicar el CPP a un procedimiento rutinario como sería la recogida de basura o el reparto de la correspondencia. En estos casos nos damos cuenta de que el trabajo que se puede realizar en media jornada (o incluso en una jornada laboral entera), después de la cual se debe volver al punto de partida, es claramente limitado. Por tanto, si la longitud de las calles es “normal”, nunca nos encontraremos con problemas gigantescos del estilo de calcular la ruta de la recogida de basura de Barcelona por un solo camión.

Estos dos fenómenos hacen que el método de resolución aquí propuesto se adapte bastante a las necesidades prácticas más usuales de quienes necesiten hallar ciclos eulerianos.

Algunas posibles ampliaciones de este proyecto para acercarlo todavía más a las necesidades prácticas podrían ser:

La resolución de CPP para el caso asimétrico, lo que permitiría su aplicación a un número mucho mayor de casos.

Resolver el CPP (simétrico y/o asimétrico) con la variante de que el variante de que el valor asociado en cada arista variara en función de que fuera o no la primera vez que se recorre. Esto se adaptaría más a la situación de que una misma carretera puede tener una dificultad de recorrerla muy distinta en función de que esté nevada o la nieve esté ya apartada; o que una calle no costará tanto de recorrer con los contenedores recogidos o por recoger.

14. Bibliografía

(AHU 88) AHO, HOPCROFT, ULLMAN, "Estructuras de datos y algoritmos", Addison-Wesley Iberoamericana S.A., Willmington, Delaware, U.S.A., 1988

(AlC 85) ALSPACH, CODSIL, "Cycles in graphs", North-Holland, Amsterdam, 1985

(Bas 93) BASART J., "Grafs: fonaments i algorismes", Servei de publicacions de la Universitat Autònoma de Barcelona, Barcelona, 1993.

(Bru 96) BRUNAT BLAY, J.Mª, "Combinatòria i teoria de grafs", Edicions U.P.C., Barcelona 1996

(CoC 95) COMPANYS, COROMINAS, "Proceso generalizado de branch and bound", CPDA-ETSEIB, Barcelona, 1995

(ClB 85) CLAVEL, BIONDI, "Introducción a la programación. Tomo 2: Estructuras de datos", Masson, S.A., Barcelona, 1985.

(JoM 95) JORBA, MASDEMONT, "Introducció a la simulació", Servei de publicacions de la U.P.C., Barcelona, 1995

(Lab 95) LABORATORIO DE ORGANIZACIÓN DE LA PRODUCCIÓN, "Optimización Combinatoria", CPDA-ETSEIB, Barcelona, 1995

(Mil 93) MILLER, Q., "The joy of C", John Wiley & Sons, Inc., U.S.A., 1993

(Sch 91) SCHILDT, H., "ANSI C a su alcance", McGraw-Hill/Interamericana de España, S. A., Madrid, 1991

(Tar 88) TARJAN, R., "Data structures and network algorithms", P.A. Society for Industrial and Applied Mathematics, Philadelphia, 1988

15. Presupuesto

El presupuesto de la realización de este proyecto es el siguiente:

A) Proyecto

Recogida de información (10 %): 40 h. a 1.500 pts/h.	=	60.000 pts.
Investigación (50 %): 200 h. a 3.500 pts/h.	=	700.000 pts.
Programación (40 %): 120 h. a 3.500 pts/h.	=	420.000 pts.
TOTAL A		**1.180.000 pts.**

B) Mecanografiado

Mecanografiado: 80 h. a 1.500 pts/h	=	**120.000 pts.**

Total del proyecto: A + B = **1.300.000 pts.**

ANEXO 1: listado de programas

MAIN.C

```
#include<alloc.h>
#include<conio.h>
#include<stdio.h>
#include<math.h>
#include<time.h>

#include"arbol.h"
#include"fulker.h"
#include"grafo.h"
#include"eulerian.h"
#include"simetr.h"

typedef struct
{       int i;
        int j;
        int d;
        char fl;
}t_l;

typedef struct
{       int z;
        char fl;
}t_m;

#define INF 32767

void acotar_hijo(int *hijo,t_m **m0,t_l *l0);
void conv_mat(int nv,int **l);
void decodif(t_l *l0,int pos,int *a,int *b);
void flag_l0(int *part,t_l *l0);
void generar_hijo(int modo,int *part,int *hijo,t_l *l0);
void guardar_m(int **m,int nv);
void leer_archivo();
void llenar_l0(t_l *l0,t_m **m0,int imp);
void llenar_m0(t_m **m0,int imp);
void modificar_m0(int *hijo,t_m **m0,t_l *l0);
void ordenar_l0(t_l *l0);
void poner_raiz(t_m **m0,int *final);
void restaurar_m0(t_m **m0);
```

```
int cota(t_m **a);
int impar();
int leer_m(int nv,int a,int b);

FILE *fm;
FILE *f;

void main(void)
{  int i,j,k;
        int arch,heur,entrar;
        int n_sol,esc,na,nv,imp,vm,final,dim,tmax,tiempo,cont;
        int **l,**p,**m;
        int *particion,*hijo,*mejor;
        int *eul;
        float tmax, cont, tiempo;
        char sn,he,fr[20];
        t_m **m0;
        t_l *l0;
        time_t start,sol,forzar,end,anterior,act;
        el_lista e;


        /*LEER EL ARCHIVO DE LOS DATOS DEL PROBLEMA*/
        leer_archivo();

        /*HEURISTICA?*/
        printf("¨Desea limitar el tiempo máximo de ejecución? (s/n) "); he=getche();
        heur=0;
        if(he=='s' || he=='S')
        {       printf("\nTiempo m áximo de ejecucin (s): ");
                scanf("%f",&tmax);
                heur=1;
        }

        /*GUARDAR RESULTADOS*/
        printf("\n¨Desea guardar los resultados en archivo? (s/n) "); sn=getche();
        arch=0;
        if(sn=='s' || sn=='S')
        {  printf("\nFichero de resultados: ");
                scanf("%s",fr);

                if((f=fopen(fr,"w"))==NULL)
                {       printf("\n No se puede crear el archivo de resultados %s\n",fr);
                        exit(1);
                }
                arch=1;
```

```
        }


        /*INICIALIZAR LA FUNCION DE TIEMPO*/
        start=time(NULL);

        /*INICIALIZAR TIEMPO*/
        anterior=clock()/CLK_TCK;
        cont=0;
        tiempo=0;
        gotoxy(60,1); printf("Tiempo= %.0f s.",tiempo);

        /*ESTABLECER LAS ARISTAS COMO NO USADAS*/
        todo_no_usado();

        /*VARIABLES INDICADORAS DEL NUMERO DE ARISTAS Y DE
VERTICES DEL PROBLEMA*/
        na=n_aristas();
        nv=n_vert();

        /*CONTADOR DE SOLUCIONES HALLADAS*/
        n_sol=0;

        /*NO ESCAPAR*/
        esc=0;


        /*RESERVAR MEMORIA PARA LAS MATRICES PARA CALCULAR
CAMINOS MINIMOS*/
  if((l=(int**)malloc((nv+1)*sizeof(int*)))==NULL){puts("\nNo hay memoria para la
matriz l");exit(1);}
        for(i=0;i<nv+1;i++)
if((l[i]=(int*)malloc((nv+1)*sizeof(int)))==NULL){puts("\nNo hay memoria para la
matriz l");exit(1);}

        if((p=(int**)malloc((nv+1)*sizeof(int*)))==NULL){puts("\nNo hay memoria
para la matriz p");exit(1);}
        for(i=0;i<nv+1;i++)
if((p[i]=(int*)malloc((nv+1)*sizeof(int)))==NULL){puts("\nNo hay memoria para la
matriz p");exit(1);}

        if((m=(int**)malloc((nv+1)*sizeof(int*)))==NULL){puts("\nNo hay memoria
para la matriz m");exit(1);}
```

```
        for(i=0;i<nv+1;i++)
if((m[i]=(int*)malloc((nv+1)*sizeof(int)))==NULL){puts("\nNo hay memoria para la
matriz m");exit(1);}

        /*PASAR EL GRAFO A MATRIZ*/
        conv_mat(nv,l);

        /*EJECUTAR EL ALGORITMO DE FULKERSON PARA CAMINOS
MINIMOS*/
        fulk(nv,l,p,m);

        /*ELIMINAR MATRIZ l INICIAL*/
        for(i=0;i<nv+1;i++) free(l[i]);
        free(l);

        /*GUARDAR p (SIMETRICA) EN MEDIA MATRIZ Y ELIMINAR p
ORIGINAL*/
        simetr(nv,p);
        for(i=0;i<nv+1;i++) free(p[i]);
        free(p);

        /*GUARDAR m EN DISCO (POCOS ACCESOS A m) Y ELIMINAR m
ORIGINAL*/
        guardar_m(m,nv);
        for(i=0;i<nv+1;i++) free(m[i]);
        free(m);

        /*VARIABLE N§ DE VERTICES DE GRADO IMPAR*/
        imp=impar();
        gotoxy(1,5);printf("\nN§ de v,rtices impares: %d",imp);

        /*EMPEZAR EL PROCESO SI HAY VERTICES DE GRADO IMPAR*/
        if(imp!=0)
        {
                /*RESERVAR MEMORIA PARA m0 Y l0*/
                if((m0=(t_m**)malloc((imp+1)*sizeof(t_m*)))==NULL){printf("\nNo
hay memoria para m0");exit(1);}
                for(i=0;i<imp+1;i++)
if((m0[i]=(t_m*)malloc((imp+1)*sizeof(t_m)))==NULL){printf("\No hay memoria
para m0[i]");exit(1);}
                if((l0=(t_l*)malloc((((imp*imp-
imp)/2+1)*sizeof(t_l))))==NULL){printf("\No hay memoria para l0");exit(1);}

                /*LLENAR Y ORDENAR*/
                llenar_m0(m0,imp);
                llenar_l0(l0,m0,imp);
```

```
ordenar_l0(l0);


/*ALGORITMO DE BRANCH & BOUND*/
        /*INICIALIZAR EL ARBOL*/
        inicializar();

        /*DECLARACION INICIAL DE VECTORES*/
        if((particion=(int*)malloc(sizeof(int)))==NULL) {printf("\nFalta
memoria para malloc particiȼn (1)");exit(1);}
        if((hijo=(int*)malloc(sizeof(int)))==NULL){printf("\nFalta
memoria para malloc hijo (1)");exit(1);}
        if((mejor=(int*)malloc(sizeof(int)))==NULL){printf("\nFalta
memoria para malloc mejor (1)");exit(1);}

        /*ARBOL={E}*/
        poner_raiz(m0,&final);

        /*VALOR MEJOR=INFINITO*/
        vm=INF;


        /*MIENTRAS (ARBOL NO VACIO Y NO ESCAPAR)
HACER*/
        while(final==0 && esc==0)
        {
                /*MOSTRAR TIEMPO*/
                act=clock()/CLK_TCK;
                if((act-anterior)>=5)
                {       anterior=act;
                        cont++;
                        tiempo=cont*5;
                        gotoxy(60,1); printf("Tiempo= %.0f s.",tiempo);
                        entrar=1;
                }
                else entrar=0;


                /*SACAR PARTICION DEL ARBOL*/
                dim=dimension(&final);

        if((particion=(int*)realloc(particion,dim*sizeof(int)))==NULL) {printf("\nFalta
memoria para sacar particiȼn (%d)",dim);exit(1);}
                        sacar(dim,particion,&final);
```

```
                        /*SI COTA(PARTICION)<VALOR MEJOR
ENTONCES*/
                        if(particion[0]<vm)
                        {
                                /*EJECUTAR PARA COGER Y PARA DEJAR
LA ARISTA*/
                                for(i=1;i>=0;i--)
                                {
                                        /*GENERAR NUEVO HIJO DE LA
PARTICION*/

        if((hijo=(int*)realloc(hijo,(dim+1)*sizeof(int)))==NULL){printf("\nFalta
memoria para generar hijo (%d)",dim+1);exit(1);}
                                        generar_hijo(i,particion,hijo,l0);

                                        /*ACOTAR NUEVO HIJO*/
                                        if(hijo[hijo[2]-1]!=0)
acotar_hijo(hijo,m0,l0);

                                        /*SI COTA(HIJO)<VALOR MEJOR
ENTONCES*/
                                        if((particion[0]<=hijo[0]) &&
        (hijo[0]<vm) && (hijo[hijo[2]-1]!=0))
                                        {
                                                /*SI HIJO ES TERMINAL*/
                                                if(hijo[1]==imp/2)
                                                {
                                                        /*VALOR
MEJOR=COTA(HIJO)*/
                                                        vm=hijo[0];

                                                        /*CONTADOR
SOLUCIONES*/
                                                        n_sol++;

                                                        /*MEJOR=HIJO*/

        if((mejor=(int*)realloc(mejor,(dim+1)*sizeof(int)))==NULL){printf("\nFalta
memoria para guardar mejor (%d)",dim+1);exit(1);}
                                                        for(k=0;k<=dim;k++)
mejor[k]=hijo[k];

                                                        /*MATAR A LOS QUE NO
PUEDEN SER MEJORES QUE EL MEJOR*/
                                                        matar(vm,&final);
                                                        sol=time(NULL);
```

```
                                                gotoxy(1,7);printf("Soluci¢n
actual: ");

        gotoxy(19,7);clreol();printf("%d",vm);

        gotoxy(1,8);clreol();printf("Tiempo sol. actual: %d
segundos",(int)difftime(sol,start));

        gotoxy(1,9);clreol();printf("Soluciones halladas: %d",n_sol);
                                        }

                                        /*SINO*/
                                        else
                                        {
                                                /*A¥ADIR HIJO AL
ARBOL*/
                                                poner(dim+1,hijo,&final);
                                        }
                                }

                                        /*RESTAURAR m0 Y l0*/
                                        restaurar_m0(m0);
                                }
                        }

                        /*CONTROLAR HEURISTICA*/
                        if(heur && entrar)
                        {       if(tiempo>=tmax)
                                {       forzar=time(NULL);
                                        esc=1;
                                }
                        }

                        /*POSIBILIDAD DE ESCAPAR*/
                        if(kbhit())
                        {  forzar=time(NULL);
                                esc=1;
                        }
                }

                        /*FINALIZAR EL ARBOL*/
                        finalizar();
                /*FINAL ALGORITMO BRANCH & BOUND*/

                /*A¥ADIR AL GRAFO LA SOLUCION DEL BRANCH AND
BOUND*/
```

```
            gotoxy(1,10);printf("Aristas resultantes del B&B:\n");
            if(arch) fprintf(f,"Aristas resultantes del B&B:\n");
            for(k=3;k<mejor[2];k++)
            {       if(mejor[k]>0)
                    {       decodif(l0,mejor[k],&i,&j);
                            while(i!=j)
                            {       e.d=leer_p(nv,i,leer_m(nv,i,j));
                                    e.j=leer_m(nv,i,j);
                                    e.usado=0;
                                    pon_lista_sim(e,i);
                                    printf("%d-%d:  %d \t",i,e.j,e.d);
                                    if(arch) fprintf(f,"%d-%d: %d \n",i,e.j,e.d);
                                    i=leer_m(nv,i,j);
                            }
                    }
            }
            printf("\n\nDistancia total a¤adida: %d",mejor[0]);
            if (arch) fprintf(f,"\nDistancia total a¤adida: %d\n",mejor[0]);

            /*LIBERAR m0 Y l0*/
            for(i=0;i<imp+1;i++) free(m0[i]);
            free(m0);
            free(l0);
    }

    /*CERRAR Y ELIMINAR EL FICHERO DE LA MATRIZ m*/
    fclose(fm);
    remove("m.mat");

    /*HALLAR UN CICLO EULERIANO EN EL GRAFO DEFINITIVO*/
    if(impar()!=0){printf("\nError, el n§ de v,rtices es impar");exit(1);}
    na=n_aristas();

    /*DECLARAR EL VECTOR QUE CONTENDRA EL CICLO EULERIANO
SOLUCION*/
    if((eul=(int*)malloc((na+1)*sizeof(int)))==NULL){printf("\No hay memoria
para el vector de la soluci¢n");exit(1);}

    /*FUNCION QUE CALCULA EL CICLO EULERIANO*/
    c_euler(eul);

    /*PRESENTAR RESULTADO EN PANTALLA*/
    printf("\n\nCiruito euleriano:\n");
    for(i=0;i<=na;i++) printf("%d - ",eul[i]);
    printf("\n");
    if(arch)
```

```
        {  fprintf(f,"\nCircuito euleriano:\n");
                for(i=0;i<=na;i++) fprintf(f,"%d - ",eul[i]);
                printf("\n");
        }


        /*MENSAJES FINALES*/
        if(esc==1)
        {       printf("\nPrograma forzado a terminar.\n");
                if(n_sol!=0) printf("\%d segundos desde la £ltima
soluci¢n.\a\n",(int)difftime(forzar,sol));

                if(arch)
                {       fprintf(f,"\n\nPrograma forzado a terminar.\n");
                        if(n_sol!=0) fprintf(f,"\%d segundos desde la £ltima
soluci¢n.\a\n",(int)difftime(forzar,sol));
                }
        }
        else
        {
                /*FINALIZAR TIEMPO DE EJECUCION Y CALCULAR
DIFERENCIA*/
                end=time(NULL);
                printf("\nEl programa ha tardado %d segundos",(int)difftime(end,start));
                if(difftime(end,start)!=0 && imp!=0) printf("\nLa soluci¢n fue
encontrada en un %.2f%% del tiempo
total\a\n",100*difftime(sol,start)/difftime(end,start));
                else printf("\a");

                if(arch)
                {       fprintf(f,"\n\nEl programa ha tardado %d
segundos",(int)difftime(end,start));
                        if(difftime(end,start)!=0 && imp!=0) fprintf(f,"\nLa soluci¢n fue
encontrada en un %.2f%% del tiempo
total\n",100*difftime(sol,start)/difftime(end,start));
                }
        }

        /*CERRAR ARCHIVO DE RESULTADOS*/
        close(f);

        /*ELIMINAR EL VECTOR DEL CICLO EULERIANO*/
        free(eul);

        /*ELIMINAR GRAFO*/
        elim_listas();
```

```
}


void acotar_hijo(int *hijo,t_m **m0,t_l *l0)
{       int i;
        modificar_m0(hijo,m0,l0);
        hijo[0]=cota(m0);
        for(i=3;i<hijo[2];i++)
        {       if (hijo[i]>0)
                {       hijo[0]=hijo[0]+l0[hijo[i]].d;
                }
        }
}

void conv_mat(int nv,int **l)
{       int i,k;
        el_lista e;

        for(i=1;i<=nv;i++)
        {       for(k=1;k<=nv;k++)
                {       if(i==k) l[i][k]=0;
                        else l[i][k]=INF;
                }
        }
        for(i=1;i<=nv;i++)
        {       for(k=1;k<=l_lista(i)+1;k++)
                {       ver_lista(&e,i);
                        if(l[i][e.j]>e.d) l[i][e.j]=e.d;
                }
        }
}

void decodif(t_l *l0,int pos,int *a,int *b)
{       *a=l0[pos].i;
        *b=l0[pos].j;
}

void flag_l0(int *part,t_l *l0)
{       int i,k,x,y,imp;
        imp=l0[0].i;
        for(i=1;i<=(imp*imp-imp)/2;i++) l0[i].fl='0';
        for(i=3;i<part[2];i++)
        {       if(part[i]>0)
                {       decodif(l0,part[i],&x,&y);
                        for(k=1;k<=(imp*imp-imp)/2;k++)
```

```
                        {       if(l0[k].i==x || l0[k].i==y || l0[k].j==x || l0[k].j==y)
l0[k].fl='1';
                        }
                }
                else l0[(-1*part[i])].fl='1';
        }
}

void generar_hijo(int modo,int *part,int *hijo,t_l *l0)
{       int arista,i;

        flag_l0(part,l0);
        arista=elegir_arista(l0);
        hijo[1]=part[1]+modo;
        hijo[2]=part[2]+1;
        for(i=3;i<part[2];i++) hijo[i]=part[i];
        if(modo==1) hijo[i]=arista;
        else hijo[i]=-1*arista;
}


void guardar_m(int **m,int dim)
{       int i,j,aux;

        if((fm=fopen("m.mat","w+b"))==NULL){printf("\nNo se puede crear el
archivo m.mat");exit(1);}
        for(i=1;i<=dim;i++)
        {       for(j=1;j<=dim;j++)
                {       aux=m[i][j];
                        fseek(fm,sizeof(aux)*(((i-1)*dim+j)-1),SEEK_SET);
                        fwrite(&aux,sizeof(aux),1,fm);
                }
        }
}


void leer_archivo()
{       FILE *f1;
        char ff[20];
        int nv,na,a,b,c,i;
        el_lista e;

        clrscr();

        printf("Fichero fuente del grafo: ");
        scanf("%s",ff);
```

```
        if((f1=fopen(ff,"r"))==NULL)
        {       printf("\n No se puede abrir el archivo fuente %s\n",ff);
                exit(1);
        }

        fscanf(f1,"%d %d",&nv,&na);
        prep_listas(nv);

        for(i=1;i<=na;i++)
        {       fscanf(f1,"%d %d %d",&a,&b,&c);
                e.d=c;
                e.j=b;
                pon_lista_sim(e,a);
        }
        fclose(f1);
}




void llenar_l0(t_l *l0,t_m **m0,int imp)
{       int i,j,pos;
        l0[0].i=imp;
        pos=1;
        for(i=1;i<=imp;i++)
        {       for(j=i+1;j<=imp;j++)
                {       l0[pos].d=m0[i][j].z;
                        l0[pos].i=m0[0][i].z;
                        l0[pos].j=m0[0][j].z;
                        l0[pos].fl='0';
                        pos++;
                }
        }
}

void llenar_m0(t_m **m0,int imp)
{       int a,b,i,j,nv;
        int grado_i,grado_j;
        a=1;
        b=1;
        nv=n_vert();

        m0[0][0].z=imp;
        for(i=1;i<=nv;i++)
```

```
        {  grado_i=n_incid(i,0);
                if(grado_i>grado_i/2*2)
                {       m0[0][b].z=i;
                        m0[b][0].z=i;
                        for(j=1;j<=nv;j++)
                        {  grado_j=n_incid(j,0);
                                if(grado_j>grado_j/2*2)
                                {       if(i==j)
                                        {       m0[a][b].fl='0';
                                                m0[a][b].z=0;
                                        }
                                        else
                                        {       m0[a][b].z=leer_p(nv,i,j);
                                                m0[a][b].fl='1';
                                        }
                                        a++;
                                        if(a>imp)
                                        {       a=1;
                                                b++;
                                        }
                                }
                        }
                }
        }
}

void modificar_m0(int *hijo,t_m **m0,t_l *l0)
{       int i,j,k,x,y,imp;
        imp=l0[0].i;
        for(k=3;k<hijo[2];k++)
        {       if(hijo[k]>0)
                {       decodif(l0,hijo[k],&x,&y);
                        for(i=1;i<=imp;i++)
                        {       if(m0[0][i].z==x || m0[0][i].z==y)
                                {       for(j=1;j<=imp;j++)
                                        {       m0[i][j].fl='0';
                                                m0[j][i].fl='0';
                                        }
                                }
                        }
                }
                else
                {       decodif(l0,(-1*hijo[k]),&x,&y);
                        for(i=1;i<=imp;i++)
                        {       if(m0[0][i].z==x)
                                {  for(j=1;j<=imp;j++)
```

```
                                        {       if(m0[0][j].z==y)
                                                {       m0[i][j].fl='0';
                                                        m0[j][i].fl='0';
                                                }
                                        }
                                }
                        }
                }
        }
}




void ordenar_l0(t_l *l0)
{       int n;
        int i,j,jmax,imp;
        t_l temp;

        imp=l0[0].i;

        n=(imp*imp-imp)/2;

        for(i=1;i<n;i++)
        {       jmax=i;
                for(j=i+1;j<=n;j++)
                {       if(l0[j].d<l0[jmax].d)
                        {       jmax=j;
                        }
                }
                if(jmax>i)
                {       temp=l0[i];
                        l0[i]=l0[jmax];
                        l0[jmax]=temp;
                }
        }
}

void poner_raiz(t_m **m0,int *final)
{       int v[3];
        v[0]=cota(m0);
        v[1]=0;
        v[2]=3;
        poner(3,v,final);
}
```

```
void restaurar_m0(t_m **m0)
{  int i,j,imp;
        imp=m0[0][0].z;
        for(i=1;i<=imp;i++)
        {       for(j=1;j<=imp;j++)
                {       if(i!=j) m0[i][j].fl='1';
                }
        }
}

int cota(t_m **a)
{       int i,j,dim;
        float *v,im,c,fc;

        dim=a[0][0].z;

        if((v=(float*)malloc((dim+1)*sizeof(float)))==NULL) {printf("\nFalta
memoria para calcular la cota (%d)",dim+1);exit(1);}
        for(i=1;i<=dim;i++) v[i]=3.e37;
        for(i=1;i<=dim;i++)
        {       for(j=1;j<=dim;j++)
                {       if(((float)a[i][j].z<v[i])&&(a[i][j].fl=='1')) v[i]=a[i][j].z;
                }
                if(v[i]>3.e35) v[i]=0;
        }
        c=0;
        for(i=1;i<=dim;i++) c=c+v[i];
        c=c/(dim/2);
        fc=(float)floor((double)c);
        if(c!=fc) c=fc+1;
        im=0;
        for(j=1;j<=dim;j++) if(v[j]==0) im++;
        if(im/2!=floor(im/2)) c=32700;
        free(v);
        return((int)c);
}

int elegir_arista(t_l *l0)
{       int k,imp;
        k=0;
        imp=l0[0].i;

        while(k<=(imp*imp-imp)/2)
        {       k++;
                if(l0[k].fl=='0') return(k);
```

```
        }
        return(0);
}

int impar()
{       int i,incid,z;
        z=0;
        for(i=1;i<=n_vert();i++)
        {  incid=n_incid(i,0);
                if(incid>incid/2*2) z++;
        }
        return(z);
}

int leer_m(int nv,int a,int b)
{       int aux;

        fseek(fm,sizeof(aux)*(((a-1)*nv+b)-1),SEEK_SET);
        fread(&aux,sizeof(aux),1,fm);

        return(aux);
}
```

ARBOL.C

```
#include<stdio.h>
#include<alloc.h>

#include"arbol.h"

#define P 100

struct arbol *a;
int unsigned maximo;

/*INSERIR UN ELEMENTO*/
void poner(int dim,int *v,int *final)
{  struct nodo *n, *or;
  int i, aux;

        /*malloc NUEVO ELEMENTO*/
        if ((n=(struct nodo*)malloc(sizeof(struct nodo)))==NULL)      {printf("\nFalta
memoria para inserir elementos en la lista."); exit(1);}
        if((n->e=(int*)malloc((dim)*sizeof(int)))==NULL){ printf("\nFalta memoria
para inserir elementos en la lista."); exit(0);}

        /*CONTROLAR FINAL*/
        *final=0;

        /*COPIAR NUEVO ELEMENTO*/
        for (i=0;i<dim;i++) *(n->e+i)=*(v+i);

        /*CONTROLAR PUNTEROS SI LISTA VACIA*/
  if (a->cont==0)
        { n->inv1=n;
          n->inv2=n;
    n->ord1=n;
    n->ord2=n;
    a->ord1=n;
    a->ord2=n;
        }

        /*CONTROLAR SI NO ESTABA VACIA*/
        else
        {
                /*A¥ADIR NODO Y ORDENAR POR ORDEN 1*/
     aux=0;
     or=a->ord1;
```

```
                /* HACER QUE or APUNTE AL ELEMENTO ANTERIOR DONDE
HEMOS DE INSERIR EL NUEVO SEGUN ORDEN 1*/
    while (aux<a->cont)
                {       if((*(or->e)<(*v)) || ((*(or->e)==(*v)) && (*(or->e+1)>*(v+1))))
                        {       aux++;
                                or=or->ord1;
                        }
                        else
                        {  if (aux==0) a->ord1=n;   /* por si el nuevo es primero en ord 1
*/
                                aux=a->cont;
                        }
                }

                /*CONTROLAR PUNTEROS DE ORDEN 1*/
    n->ord1=or;
    n->inv1=or->inv1;
    (or->inv1)->ord1=n;
    or->inv1=n;

                /*ORDENAR POR ORDEN 2*/
    aux=0;
    or=a->ord2;

                /*HACER QUE or APUNTE AL ELEMENTO ANTERIOR DONDE
HEMOS DE INSERIR EL NUEVO SEGUN ORDEN 2*/
                while (aux<a->cont)
                {  if((*(or->e+1)>*(v+1)) || ((*(or->e+1)==*(v+1)) && (*(or-
>e)<(*v))))
                        {       aux++;
                                or=or->ord2;
            }
                        else
                        {  if(aux==0) a->ord2=n; /* por si el nuevo es primero en ord 2 */
                                aux=a->cont;
            }
                }

                /*CONTROLAR PUNTEROS DE ORDEN 2*/
                n->ord2=or;
                n->inv2=or->inv2;
                (or->inv2)->ord2=n;
                or->inv2=n;
        }
```

```
	/*ACTUALIZAR CONTADOR ELEMENTOS ACTUALES*/
	a->cont++;

	/*ACTUALIZAR CONTADOR ELEMENTOS MAXIMOS*/
	if(a->cont>maximo) maximo=a->cont;
}



/*SACAR UN ELEMENTO*/
void sacar (int dim,int *v,int *final)
{  struct nodo *aux;
   int i;

	/*CONTROLAR LISTA VACIA*/
	if (a->cont==0) *final=2;

	/*SACAR ELEMENTO*/
	else
	{
		/*SACAR SEGUN CRITERIO 1*/
		if (a->cont<P)
		{
			/*COPIARLO EN EL VECTOR DE SALIDA*/
			for (i=0;i<dim;i++) *(v+i)=*(a->ord1->e+i);

			/*LIBERAR MEMORIA*/
			free(a->ord1->e);

			/*REORDENAR*/
			aux=a->ord1;
			a->ord1=aux->ord1;
			if (a->ord2==aux) a->ord2=aux->ord2;
			if (a->cont!=1)
			{  a->ord1=aux->ord1;
				if (a->ord2==aux) a->ord2=aux->ord2;
			}
		}

		/*SACAR SEGUN CRITERIO 2*/
		else
		{
			/*COPIARLO EN EL VECTOR DE SALIDA*/
			for (i=0;i<dim;i++) *(v+i)=*(a->ord2->e+i);

			/*LIBERAR MEMORIA*/
```

```
                free(a->ord2->e);

                /*REORDENAR*/
                aux=a->ord2;
                a->ord2=aux->ord2;
                if (a->ord1==aux) a->ord1=aux->ord1;
        }

        /*CONTROLAR SI LA LISTA QUEDA VACIA*/
        if (a->cont==1)
        {
                a->ord1=NULL;
                a->ord2=NULL;
                *final=1;
        }

        /*SI NO ESTA VACIA, ARREGLAR ENCADENAMIENTOS*/
        else
        {  (aux->inv1)->ord1=aux->ord1;
                (aux->ord1)->inv1=aux->inv1;
                (aux->inv2)->ord2=aux->ord2;
                (aux->ord2)->inv2=aux->inv2;
        }

        /*LIBERAR MEMORIA*/
        free(aux);

        /*ACTUALIZAR CONTADOR DE NODOS ACTUALES*/
        a->cont--;
    }
}

/*ELIMINAR TODOS LOS NODOS DEL ARBOL CON UNA COTA IGUAL O
SUPERIOR A LA DADA*/
void matar(int cota,int *final)
{  struct nodo *aux,*or;

    /*CONTROLAR SI LA LISTA YA ESTABA VACIA*/
    if (a->cont==0) *final=1;

    /*MATAR*/
    else
    {
        /*APUNTAR AL ULTIMO SEGUN ORDEN 1*/
        aux=(a->ord1)->inv1;
```

```
                /*PASAR POR LOS QUE VAN A MORIR*/
                while (*(aux->e)>=cota)
                {
                        /*MATAR*/
                        or=aux;

                        /*APUNTAR AL NUEVO ULTIMO*/
                        aux=aux->inv1;

                        /*ACTUALIZAR CONTADOR*/
                        a->cont--;

                        /*SI LA LISTA QUEDA VACIA*/
                        if (a->cont==0)
                        {       a->ord1=NULL;
                                a->ord2=NULL;
                                free(or->e);
                                free(or);
                                break;
                        }

                        /*SI NO, ARREGLAR ENCADENAMIENTOS*/
                        else
                        {  aux->ord1=a->ord1;
                                (a->ord1)->inv1=aux;
                                if (a->ord2==or) a->ord2=or->ord2;
                                (or->inv2)->ord2=or->ord2;
                                (or->ord2)->inv2=or->inv2;
                                free(or->e);
                                free(or);
                        }
                }

                /*CONTROLAR FINAL SI LA LISTA ESTA VACIA*/
                if (a->cont==0) *final=1;
        }
}



/*INICIALIZAR EL ARBOL*/
void inicializar()
{  a=(struct arbol*)malloc(sizeof(struct arbol));
        a->cont=0;
  a->ord1=NULL;
```

```
        a->ord2=NULL;

        maximo=0;
}


/*LIBERAR LA MEMORIA OCUPADA POR EL ARBOL*/
void finalizar()
{  struct nodo *n,*or;
   int i;

        /*SITUALRNOS EN EL PRIMER NODO*/
        n=a->ord1;

        /*RECORRER LOS NODOS LIBERANDO EL CAMPO e Y EL NODO*/
        for (i=1;i<=a->cont;i++)
        {       or=n;
                n=n->ord1;
     free(or->e);
     free(or);
        }
   a->ord1=NULL;
   a->ord2=NULL;
}


/*RETORNA LA DIMESION DEL PROXIMO ELEMENTO A SACAR*/
int dimension(int *final)
{  int k;

        /*CONTROLAR QUE HAYA ELEMENTO A SACAR*/
        if(a->cont==0)
        {       *final=2;
                return(0);
        }

        /*RETORNA EN FUNCION DEL CRITERIO DE EXTRACCION EL
CONTENIDO DE LA POSICION QUE INDICA LA DIMENSION*/
        else
        {  if(a->cont<P)
                {       k=*(a->ord1->e+2);
                        return(k);
                }
                else
                {       k=*(a->ord2->e+2);
                        return(k);
```

```
                }
        }
}

/*RETORNA LA OCUPACION ACTUAL DEL ARBOL*/
int ocup()
{       return(a->cont);
}

/*RETORNA LA OCUPACION MAXIMA ALCANZADA*/
int max()
{       return(maximo);
}
```

FULKER.C

```
#include<stdio.h>
#include<alloc.h>

#include"grafo.h"

#define INF 32700

/*ALGORITMO DE FULKERSON PARA HALLAR CAMINOS MINIMOS*/
void fulk(int nv,int **li,int **pi,int **m)
{       int fin,i,j,k;
        float **l,**p;

        /*DECLARACION DE MATRICES AUXILIARES FLOAT*/
        if((l=(float**)malloc((nv+1)*sizeof(float*)))==NULL){printf("\nNo hay
memoria en la declaraci¢n de l float en la funci¢n fulker.");exit(1);}
        for(i=0;i<=nv+1;i++)
if((l[i]=(float*)malloc((nv+1)*sizeof(float)))==NULL){printf("\nNo hay memoria en
la declaraci¢n de l[i] float en la funci¢n fulker.");exit(1);}
        if((p=(float**)malloc((nv+1)*sizeof(float*)))==NULL){printf("\nNo hay
memoria en la declaraci¢n de p float en la funci¢n fulker.");exit(1);}
        for(i=0;i<=nv+1;i++)
if((p[i]=(float*)malloc((nv+1)*sizeof(float)))==NULL){printf("\nNo hay memoria en
la declaraci¢n de p[i] float en la funci¢n fulker.");exit(1);}

        /*LLENAR LAS MATRICES FLOAT CON LOS VALORES DE LAS INT*/
        for(i=1;i<=nv;i++)
        {       for(j=1;j<=nv;j++)
                {       l[i][j]=(float)li[i][j];
                        p[i][j]=(float)pi[i][j];
                }
        }

        /*INICIALIZACION*/
        for(i=1;i<=nv;i++)
        {       for(j=1;j<=nv;j++)
                {       p[i][j]=l[i][j];
                        if (l[i][j]<INF) m[i][j]=j;
                        else m[i][j]=INF;
                }
        }

        /*BUCLE PRINCIPAL*/
        do
```

```
{
        /*VARIABLE DE FINALIZACION*/
        fin=1;

        /*BUCLE DE FULKERSON*/
        for(i=1;i<=nv;i++)
        {       for(j=1;j<=nv;j++)
                {       for(k=1;k<=nv;k++)
                        {       if(p[i][j]>l[i][k]+p[k][j])
                                {       p[i][j]=l[i][k]+p[k][j];
                                        m[i][j]=k;
                                        fin=0;
                                }
                        }
                }
        }

    /*FINALIZAR CUANDO NO SE MODIFIQUE NINGUN VALOR*/
    } while (!fin);

    /*PASAR LOS VALORES A LAS MATRICES PRINCIPALES*/
    for(i=1;i<=nv;i++)
    {       for(j=1;j<=nv;j++)
            {       li[i][j]=(int)l[i][j];
                    pi[i][j]=(int)p[i][j];
            }
    }

    /*LIBERAR LA MEMORIA DE LAS MATRICES AUXILIARES*/
    for(i=0;i<nv+1;i++)free(p[i]);
    free(p);
    for(i=0;i<nv+1;i++)free(l[i]);
    free(l);
}
```

GRAFO.C

```
#include <alloc.h>
#include <stdlib.h>

#include "grafo.h"

static el_lista **ini_lista, **fin_lista;
static int *lon_lista;
static int num_listas;


/*ELIMINA TODAS LAS LISTAS*/
void elim_listas()
{  int i;

        /*LIBERAR LA MEMORIA RESERVADA PARA LAS LISTAS*/
        free(lon_lista);
        for(i=0;i<num_listas;i++) free(ini_lista[i]);
        free(ini_lista);
        for(i=0;i<num_listas;i++) free(fin_lista[i]);
        free(fin_lista);
}


/*VUELCA EN EL VECTOR v EL N§ DE ARISTAS (VOLCADO EN n)
INCIDENTES EN EL VERTICE x CON ESTADO u*/
void incidentes (int x,int u,el_lista *v,int *n)
{       int i,pos;
        el_lista e;

        pos=0;
        for (i=1;i<=l_lista(x);i++)
        {       ver_lista(&e,x);
                if((u==1)||(u==0))
                {       if(u==e.usado)
                        {  pos++;
                                v[pos]=e;
                        }
                        *n=pos;
                }
                else
                {       if(u==2)        v[i]=e;
                        else {printf("error tipo usado");exit(1);}
                        *n=i;
```

```
                }

        }
}


/*PONE LA ARISTA c EN LA ULTIMA POSICION DE LA LISTA*/
void pon_lista (el_lista c,int k)
{       el_lista *nuevo;

        k=k-1;

        if ((k>=num_listas)||(k<0)){puts("error lista inexistente");exit(1);}

  /*CREA UN NUEVO ESPACIO DE MEMORIAPARA GUARDAR EL NUEVO
ELEMENTO*/
        nuevo=(el_lista*)malloc(sizeof(el_lista));

  /*COPIA EN EL NUEVO ESPACIO DE MEMORIA EL NUEVO ELEMENTO*/
        *nuevo=c;

        /*EL NUEVO ELEMENTO ES EL ULTIMO DE LA LISTA*/
        nuevo->sig=NULL;

        /*SI NO HABIA LISTA LA INICIALIZAMOS CON EL NUEVO*/
        if(lon_lista[k]==0) ini_lista[k]=nuevo;

        /*EL ANTIGUO ULTIMO ELEMENTO APUNTA AL NUEVO*/
        else fin_lista[k]->sig=nuevo;

        /*EL PUNTERO INDICADOR AL ULTIMO ELEMENTO APUNTA AL
NUEVO ULTIMO*/
        fin_lista[k]=nuevo;

        /*INCREMENTA EL VALOR DE LA VARIABLE INDICADORA DE LA
LONGITUD DE LA LISTA*/
        ++lon_lista[k];
}


/*PONE LA ARISTA SIMETRICA e EN LA LISTA k Y EN LA QUE
CORRESPONDE POR e.k*/
void pon_lista_sim(el_lista e,int k)
{       int aux;

        pon_lista(e,k);
```

```
	aux=e.j;e.j=k;
	pon_lista(e,aux);
}


/*INICIALIZA n LISTAS VACIAS*/
void prep_listas (int n)
{  int i;

	num_listas=n;

	ini_lista=(el_lista**)malloc(num_listas*sizeof(el_lista**));
	if(ini_lista==NULL){puts("error 1");exit(1);}
	for(i=0;i<num_listas;i++)ini_lista[i]=NULL;

	fin_lista=(el_lista**)malloc(num_listas*sizeof(el_lista**));
	if(fin_lista==NULL){puts("error 2");exit(1);}
	for(i=0;i<num_listas;i++)fin_lista[i]=NULL;

	lon_lista=(int*)calloc(num_listas,sizeof(int));
	if(lon_lista==NULL){puts("error 3");exit(1);}
}


/*VUELCA EN c LA ARISTA POSICIONADA EN PRIMER LUGAR DE LA
LISTA k*/
void saca_lista(el_lista *c,int k)
{  k=k-1;

	/*CONTROLAR QUE QUEDEN LISTAS PARA SACAR*/
	if(lon_lista[k]==0) {puts("lista de longitud 0");exit(1);}

	/*DISMINUYE EL VALOR DE LA VARIABLE INDICADORA DE LA
LONGITUD DE LA LISTA*/
	--lon_lista[k];

	/*SI LA LISTA QUEDA VACIA CONTROLAMOS EL PUNTERO QUE
CONTROLA EL FIN DE LA LISTA*/
	if (lon_lista[k]==0) fin_lista[k]=NULL;

	/*VUELCA EN c LA ARISTA QUE SALE DE LA LISTA*/
	*c=*ini_lista[k];

	/*LIBERA LA MEMORIA OCUPADA POR EL ELEMENTO QUE SALE*/
	free(ini_lista[k]);
```

```
        /*MUEVE EL INDICADOR DEL PRIMER ELEMENTO DE LA LISTA*/
        ini_lista[k]=c->sig;
}


/*ESTABLECE COMO NO USADAS TODAS LAS ARISTAS*/
void todo_no_usado()
{       int i,j;
        el_lista e;

        for (i=1;i<=num_listas;i++)
        {  for (j=1;j<=l_lista(i);j++)
                {  saca_lista(&e,i);
                        e.usado=0;
                        pon_lista(e,i);
                }
        }
}


/*ESTABLECE COMO USADA LA ARISTA SIMETRICA a-b*/
void usado(int a,int b)
{       int i,aux,fin;
        el_lista e;

        for(i=1;i<=2;i++)
        {       if (i==2) {aux=b;b=a;a=aux;}
                fin=0;
                do
                {       saca_lista(&e,a);
                        if ((e.j==b)&(e.usado==0)) {e.usado=1;fin=1;}
                        pon_lista(e,a);
                } while(!fin);
        }
}


/*VER EL PRIMER ELEMENTO DE LA LISTA k*/
void ver_lista(el_lista *c,int k)
{       saca_lista(c,k);
        pon_lista(*c,k);
}


/*RETORNA EL N§ DE ARISTAS INCIDENTES EN EL VERTICE K*/
int l_lista(int k)
```

```
{  k=k-1;
        return(lon_lista[k]);
}


/*RETORNA EL N§ DE ARISTAS DEL GRAFO TRATADO*/
int n_aristas()
{       int i,na;
        na=0;
        for(i=1;i<=n_vert();i++) na=na+l_lista(i);
        na=na/2;
        return(na);
}


/*RETORNA EL N§ DE ARISTAS INCIDENTES EN EL VERTICE v EN ESTADO
k*/
int n_incid(int v,int u)
{       int n;
        el_lista *m;

        m=(el_lista*)malloc((l_lista(v)+1)*sizeof(el_lista));
        if(m==NULL) {puts("Error n_incid");exit(1);}

        incidentes(v,u,m,&n);
        free(m);
        return(n);
}


/*RETORNA EL N§ DE VERTICES DEL GRAFO TRATADO*/
int n_vert()
{  return(num_listas);
}
```

EULERIAN.C

```
#include<alloc.h>

#include "grafo.h"

/*HALLAR UN CIRCUITO EULERIANO Y DEVOLVERLO EN UN VECTOR*/
void c_euler(int *l)
{       int aux,na,cursor,ult,fin,c,v,y,n,i;
        int *laux;
        el_lista *m;

        /*NUMERO DE ARISTAS*/
        na=n_aristas();

        /*INICIALIZAR CURSORES*/
        cursor=0;
        ult=0;

        /*MIENTRAS NO HAYAMOS RECORRIDO EL NUMERO TOTAL DE
ARISTAS*/
        while(ult<na)
        {
                /*SI NO HEMOS EMPEZADO, EMPEZAR*/
                if(ult==0)
                {       v=1;
                        l[0]=v;
                }

                /*SI HEMOS EMPEZADO, SEGUIR*/
                else
                {
                        fin=0;
                        while(!fin && cursor <=na)
                        {       if (n_incid(l[cursor],0)!=0)
                                {       v=l[cursor];
                                        fin=1;
                                }
                                else cursor++;
                        }
                }


                if((laux=(int*)malloc((na-ult+1)*sizeof(int)))==NULL) {printf("\nFalta
memoria para calcular el circuito euleriano"); exit(1);}
```

```
                c=0;
                aux=n_incid(v,0);
                while(aux!=0)
                {       if((m=(el_lista*)malloc((l_lista(v)+1)*sizeof(el_lista)))==NULL)
{printf("\nFalta memoria para calcular el circuito euleriano"); exit(1);}
                        incidentes(v,0,m,&n);
                        y=m[1].j;
                        laux[c]=v;
                        c++;
                        usado(v,y);
                        v=y;
                        free(m);
                        aux=n_incid(v,0);
                }
                laux[c]=v;

                for(i=ult;i>cursor;i--)
                {       l[i+c]=l[i];
                }
                for(i=1;i<=c;i++)
                {       l[cursor+i]=laux[i];
                }
                ult=ult+c;
                free(laux);
        }
}
```

SIMETR.C

```
#include<stdio.h>
#include<alloc.h>

int *vector_int;

/*CONVIERTE LA MATRIZ SIMETRICA ENTERA m EN UN VECTOR DE
MENOR DIMENSION*/
void simetr(int dim,int **m)
{       int i,j,pos;

        if((vector_int=(int*)malloc(((dim*dim-
dim)/2+1)*sizeof(int)))==NULL){printf("\Falta memoria en mat_sim");exit(1);}
        pos=1;
        for(i=1;i<=dim;i++)
        {       for(j=i+1;j<=dim;j++)
                {       vector_int[pos]=m[i][j];
                        pos++;
                }
        }
}

/*LEE EL VALOR DE LA POSICION (i,j) DE LA ANTIGUA MATRIZ p*/
int leer_p(int dim,int i,int j)
{       int aux;
        if(i>j){aux=i;i=j;j=aux;}
        if(i==j) return(0);
        else
        {       aux=(i-1)*(dim-i)+k(i)+j-i;
                return(vector_int[aux]);
        }
}

static int k(int i)
{       int k,z=0;
        for(k=1;k<i;k++) z=z+k;
        return(z);
}
```