Facultad de Informática
Universidad Politécnica de Madrid

TRABAJO FIN DE CARRERA

Recorrido óptimo de los nodos de una red

Autor: Francisco Ruiz Recuenco

Tutor: Alfonso Mateos Caballero

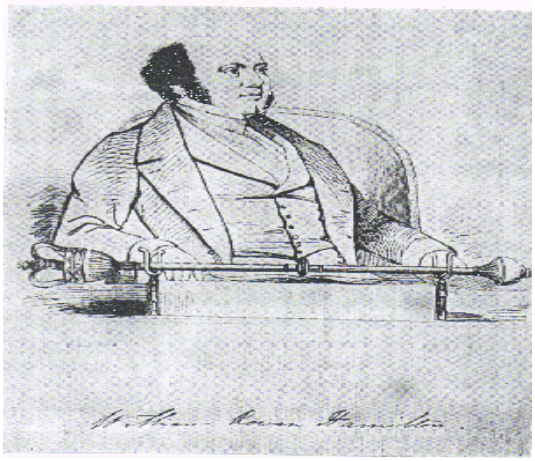
ÍNDICE

1.- Caminos y circuitos hamiltonianos y su optimización

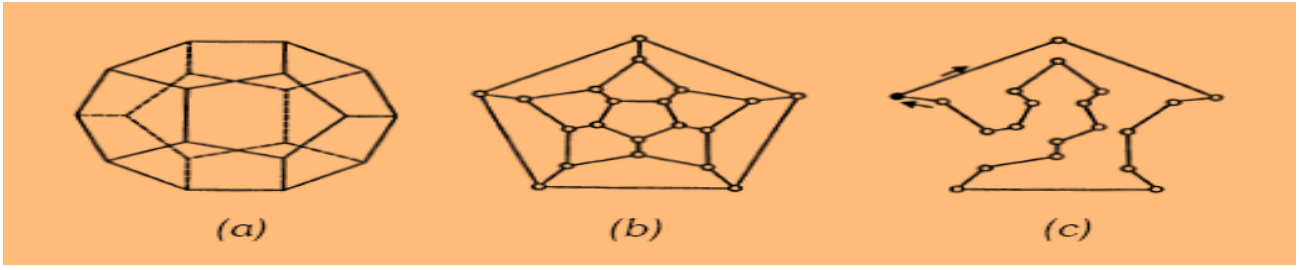
1.1- Introducción, caminos y circuitos hamiltonianos

Orígenes y fundamentación teórica

El origen de la terminología camino hamiltoniano o circuito hamiltoniano es un juego, el juego icosiano, inventado en 1857 por el matemático irlandés Sir William Rowan Hamilton (el científico irlandés más famoso de todos los tiempos nacido en 1805 que hizo importantes contribuciones al desarrollo de la óptica, la dinámica, y el álgebra).



Consistía en un dodecaedro de madera (un poliedro de 12 caras, cada una de las cuales es un pentágono regular) con un alfiler saliendo de cada vértice del dodecaedro y un trozo de cuerda. Los 20 vértices del dodecaedro estaban etiquetados con el nombre de distintas ciudades del mundo. El objetivo del juego era comenzar en una ciudad, viajar siguiendo las aristas del dodecaedro visitando cada una de las otras 19 ciudades exactamente una vez y terminar el viaje en la primera ciudad. El círculo seguido se marcaba utilizando la cuerda y los alfileres.



Es decir, extrapolando esta idea a la teoría de grafos, se trataría de encontrar un circuito en un grafo que pase por cada uno de sus nodos exactamente una vez.

El popular problema del viajante de comercio es un caso particular de otro problema más general conocido como problema de itinerarios cíclicos y cuyo enunciado general se puede describir brevemente diciendo que se trata de encontrar el itinerario óptimo desde un cierto punto de vista (económico, tiempo mínimo, política de relaciones públicas, comodidad, etc), para un viajante o agente de ventas que, saliendo de una determinada ciudad, debe recorrer todas las ciudades de una determinada zona, sin repetir la estancia en ninguna de estas si fuera posible, y volver al punto de partida.

Éste es un problema típico de fenómenos de organización y tiene consecuencias e implicaciones económicas y técnicas importantes.

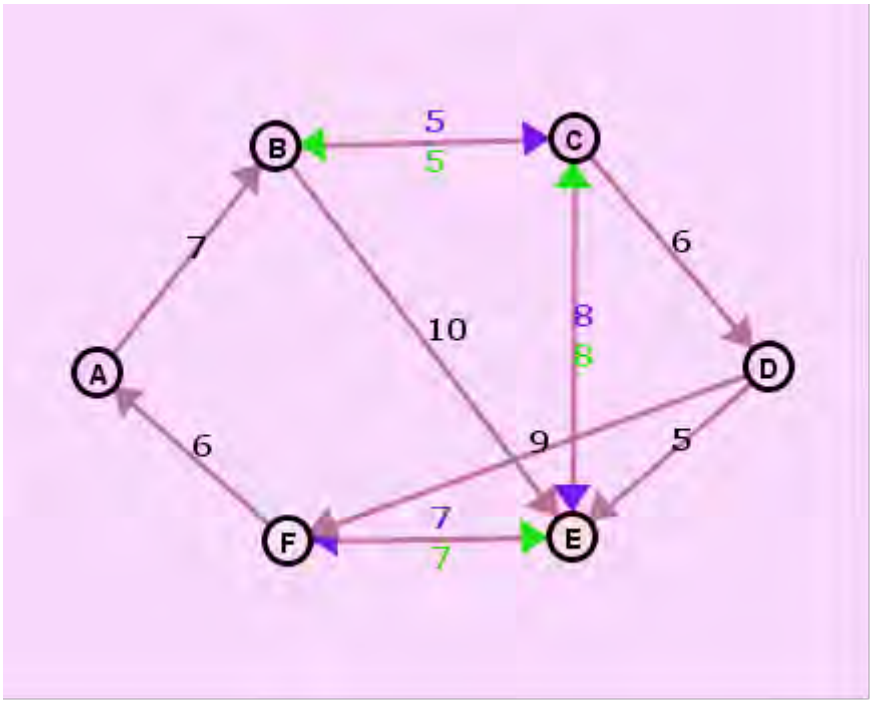
La solución para los problemas antes citados pasa por la determinación de los caminos y/o circuitos hamiltonianos de una red representada por un grafo. Además, los arcos del grafo tendrán un valor asociado para que se puedan calcular los caminos y circuitos hamiltonianos con una suma total de valores de sus arcos máxima o mínima. Esto nos abre un amplio abanico de posibilidades porque tanto los nodos como los arcos pueden simbolizar multitud de situaciones que se pueden optimizar, por ejemplo, los arcos pueden ser distancias, tiempos, costes, beneficios, etapas, etc.

Solamente centrándonos en el problema del viajante nos encontramos con muchas situaciones reales importantes de optimización (**transporte entre ciudades de forma óptima, líneas de autobuses, ruta del cartero en una ciudad, mensajería, carga y descarga en distintos puntos, …**) y esto es sólo una parte de todos los modelos que pueden ser representados y resueltos aplicando la resolución y optimización de caminos y circuitos hamiltonianos sobre un grafo creado.

Ahora, se muestra un ejemplo que dará una idea del tipo de problemas que se pueden resolver. Se dejan las definiciones formales sobre teoría de grafos para más

adelante porque lo que se pretende ahora es sólo dar una idea clara de un ejemplo que resuelve este proyecto fin de carrera.

El siguiente ejemplo muestra el problema del viajante de comercio que tiene que viajar a varias ciudades exactamente una vez en el menor tiempo posible y volver al punto de partida. Esta situación se puede expresar mediante el siguiente grafo:



Dos arcos en sentido contrario entre dos nodos tienen distintos colores.

Los nodos representan las ciudades.

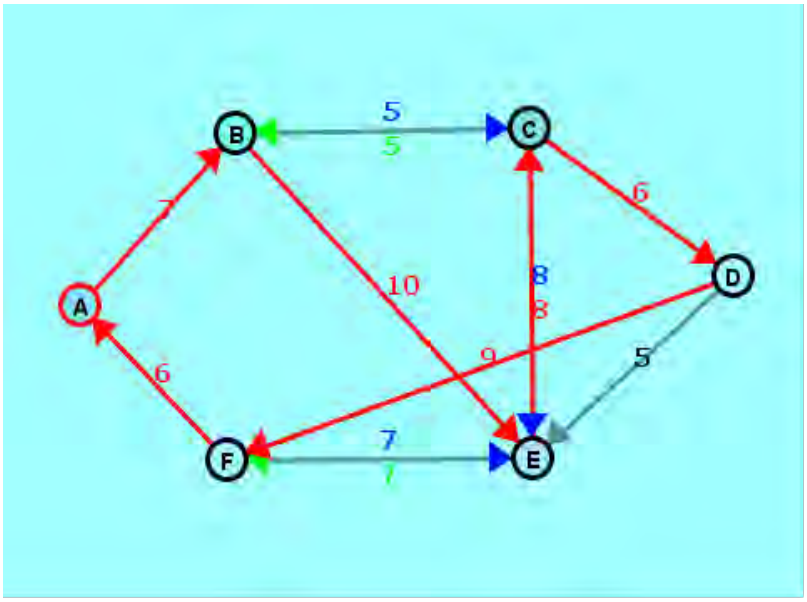
Supongamos que se viaja en avión y no existe problema de enlaces de vuelos. Los arcos representan los vuelos existentes entre las distintas ciudades.

Los valores de los arcos son unidades de tiempo (el tiempo que se tarda en viajar de una ciudad a otra).

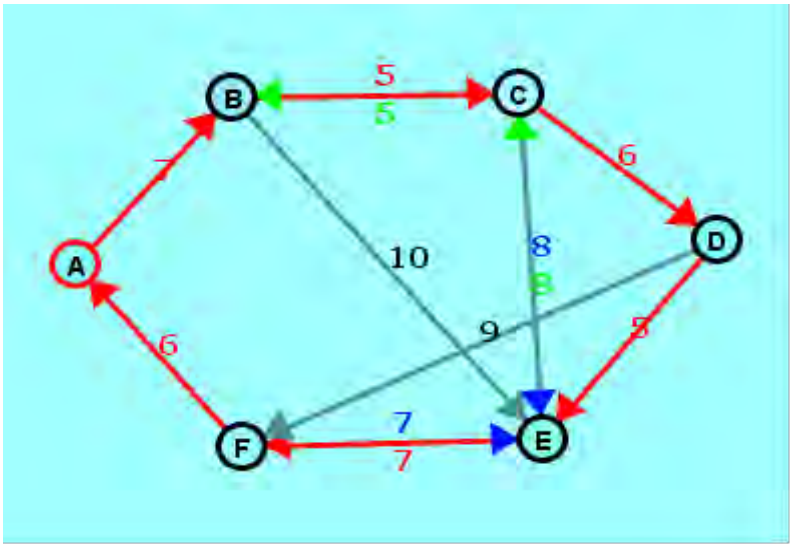
La aplicación calcula todas las soluciones desde cualquier nodo de partida del grafo. Para simplificarlo, el nodo de partida será el nodo “A”.

Se trataría de encontrar un circuito hamiltoniano que partiendo del nodo “A” recorra exactamente una vez el resto de nodos del grafo terminando otra vez en el nodo “A” con un tiempo total de recorrido mínimo.

La aplicación nos muestra dos soluciones, con los arcos de color rojo, de las cuales seleccionaría la segunda por tener un menor tiempo total:



Coste total = 46



Coste total = 36

El proyecto en lo que se refiere a Caminos y Circuitos Hamiltonianos tiene los siguientes **objetivos**:

- Puesto en Internet, permitiría un acceso universal, o lo restringido que se quiera, para gran cantidad de personas porque, además se puede elegir entre el castellano y el inglés para toda la aplicación y la información aparece igual para los cinco navegadores de Internet más importantes.
- Soporte de ayuda para los alumnos que estudien teoría de grafos o cualquier otro uso que le quiera dar la cátedra de Investigación Operativa.
- Documentar un pseudocódigo universal para implementar el cálculo de los caminos y circuitos hamiltonianos y preservar lo fundamental del proyecto para el futuro. Es decir, si los lenguajes de programación que implementan la aplicación cambian o desaparecen con el tiempo, el proyecto se podría perder. Si esto sucede, sería muy fácil volver a implementar el cálculo de los caminos y circuitos hamiltonianos en el nuevo lenguaje de programación partiendo de los programas escritos en pseudocódigo. Para este objetivo, también se documentan los soportes de datos en los que se desarrolla la aplicación, de esta forma, se entiende muy bien lo que hace cada procedimiento en pseudocódigo.

Para realizar los anteriores objetivos, el proyecto consta de las siguientes **fases**:

- Interface gráfica muy desarrollada para que un usuario pueda crear un grafo dirigido que represente el modelo del problema a resolver.
- Cálculo de los caminos y circuitos hamiltonianos para cada nodo del grafo mediante el algoritmo de Kaufmann y Malgrange.
- Optimización de los caminos hamiltonianos y circuitos hamiltonianos calculados para cada nodo del grafo (caminos) y para cada circuito.
- Información teórica para entender los procesos que se calculan y la ayuda para manejar la aplicación.

El usuario dispone de una interface gráfica muy completa para construir el grafo que le permitirá crearlo, modificar su tamaño y la posición de sus nodos, ampliarlo, reducirlo, cambiar los valores de sus arcos, etc siempre guiado por los mensajes que aparecen en la barra de estado. Cuando se calculan los caminos o circuitos hamiltonianos, las soluciones aparecen en una ventana y, simultáneamente, en el grafo los arcos de cada solución y sus valores asociados pasan a tener el color rojo. También se puede hacer un seguimiento de los pasos intermedios que realiza el algoritmo de Kaufmann y Malgrange para calcular los caminos hamiltonianos pudiendo ver las matrices intermedias que se van generando (caminos elementales) hasta finalizar con los caminos hamiltonianos.

Entre cada par de nodos puede haber a lo sumo dos arcos, uno en cada sentido y cada uno con su valor asociado. Esto puede ocasionar, si no se remedia, un grafo engorroso con información agrupada de arcos y sus valores asociados que puede dar origen a confusiones, sobretodo en el sentido de que un valor no se asocie con su verdadero arco al ver el grafo. Para arreglar esta situación el grafo se construye de la siguiente manera:

- Cuando entre dos nodos sólo exista un arco, el color del arco será gris y su valor asociado será negro.
- Cuando entre dos nodos existan dos arcos, los dos arcos se convertirán en una sola arista de color gris con dos puntas de flecha cada una apuntando a su nodo de destino. Una punta de flecha y su valor asociado serán de color azul siendo la otra punta de flecha y su valor asociado de color verde.

De esta manera, dos arcos en sentido contrario entre dos nodos se reducen a una sola arista y se puede distinguir con claridad el valor asociado de cada arco por los colores. Esto simplifica el grafo y permite verlo con más claridad y sencillez. La interface gráfica es bastante intuitiva y gracias, sobre todo, a los mensajes que aparecen en la barra de estado, el usuario sabe en todo momento lo que está haciendo y lo que tiene que hacer. Al situar el ratón sobre los botones de la interface, aparecen

mensajes explicativos del botón sobre el que está el ratón, además, existe un botón de “Ayuda” que nos explica la función de cada elemento en la interface gráfica y la secuencia de pasos que hay que seguir para construir el grafo y calcular los caminos y circuitos hamiltonianos.

La documentación teórica, que también se puede acceder a ella mediante un enlace en la aplicación puesta en Internet, nos explica los orígenes del problema que se quiere resolver, su fundamentación teórica y el procedimiento a seguir, además de los ejemplos.

Por último, se presenta un conjunto de casos resueltos paso a paso.

El código real del programa se expone en un fichero independiente por su gran volumen y porque no está ligado de forma inseparable al resto del proyecto, realmente, lo que va a perdurar en el tiempo es todo lo demás y el código real sólo pasa al ordenador lo desarrollado en pseudocódigo, soporte de datos, etc.

1.2- Métodos para determinar caminos y circuitos hamiltonianos

Respecto a los caminos hamiltonianos de un grafo, se verá primero como identificar grafos hamiltonianos, a continuación, y antes de ver el algoritmo de Kaufmann y Malgrange, se explicarán paso a paso los procedimientos para calcular multiplicaciones de matrices y multiplicación latina de matrices. Se explicará como pasar la información de un grafo a una matriz y a partir de ésta ir creando matrices para calcular los caminos elementales del grafo. Por último, a través del algoritmo de Kaufmann y Malgrange se calcularán todos los caminos hamiltonianos, si existen.

Con los caminos hamiltonianos encontrados, se calcularán los que tengan la suma total mayor y la suma total menor de sus arcos partiendo de cada nodo del grafo.

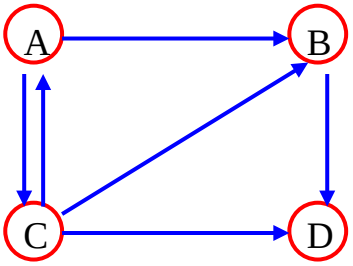
Con respecto a los circuitos hamiltonianos de un grafo, seguiremos aplicando el mismo algoritmo de Kaufmann y Malgrange con una pequeña variación para calcularlos y por último, se obtendrán los que tengan una suma mayor y menor en sus arcos.

Antes de comenzar, es necesario dar una serie de definiciones:

Grafo:

Conjunto de nodos unidos por un conjunto de líneas (aristas) o flechas (arcos). Formalmente, un grafo es una pareja $G = < N , A >$, en donde *"N"* es un conjunto de nodos y *"A"* es un conjunto de aristas o arcos.

Ejemplo:

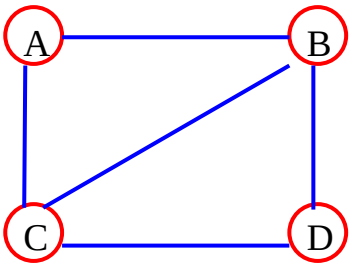


G = < N , A >, en donde

N = {A , B , C , D}

A = { (A , B) , (A , C) , (B , D) ,
(C , A) , (C , B) , (C , D) }

Si este grafo fuese no dirigido tendríamos:



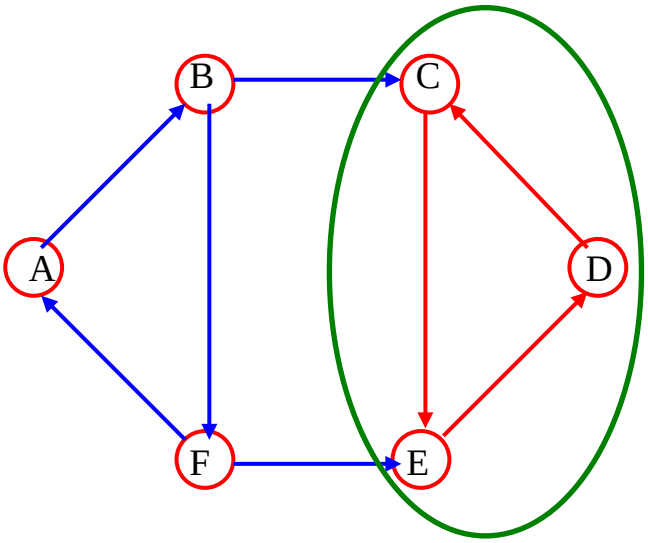
G = < N , A >, en donde

N = {A , B , C , D}

A = { {A , B} , {A , C} , {B , C}, {B , D} , {C , D} }

Subgrafo:

Un grafo es un subgrafo de otro si todos sus nodos y aristas (o arcos) están en el segundo grafo. Por ejemplo:



El grafo G' = < N', A'> en donde:

N’ = {C, D, E}

A’ = {(C, E), (D, C), (E, D)}

Es subgrafo del grafo G = <N, A>

N = {A, B, C, D, E, F}

A = {(A, B), (B, C), (B, F), (C, E), (D, C), (E, D), (F, A), (F, E)}

Grafo dirigido:

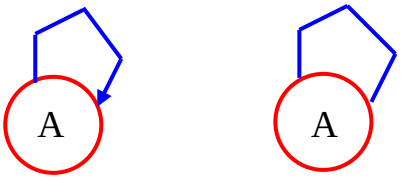
Sus nodos están unidos mediante arcos con indicación de dirección. Nunca hay más de dos flechas que unan dos nodos, y si hay dos flechas, entonces tienen que ir en sentidos opuestos. Un arco que vaya desde el nodo *“A”* hasta el nodo *“B”* será expresado mediante el par ordenado *(A , B).* Ver primer ejemplo en la definición de grafo.

Grafo no dirigido:

Sus nodos están unidos mediante aristas sin indicación de dirección (por lo que la dirección está en ambos sentidos). Nunca hay más de una arista que una dos nodos. Una arista que una los nodos *“A”* y *“B”* será expresada mediante el conjunto *{A , B}.* Ver segundo ejemplo en la definición de grafo.

Bucle:

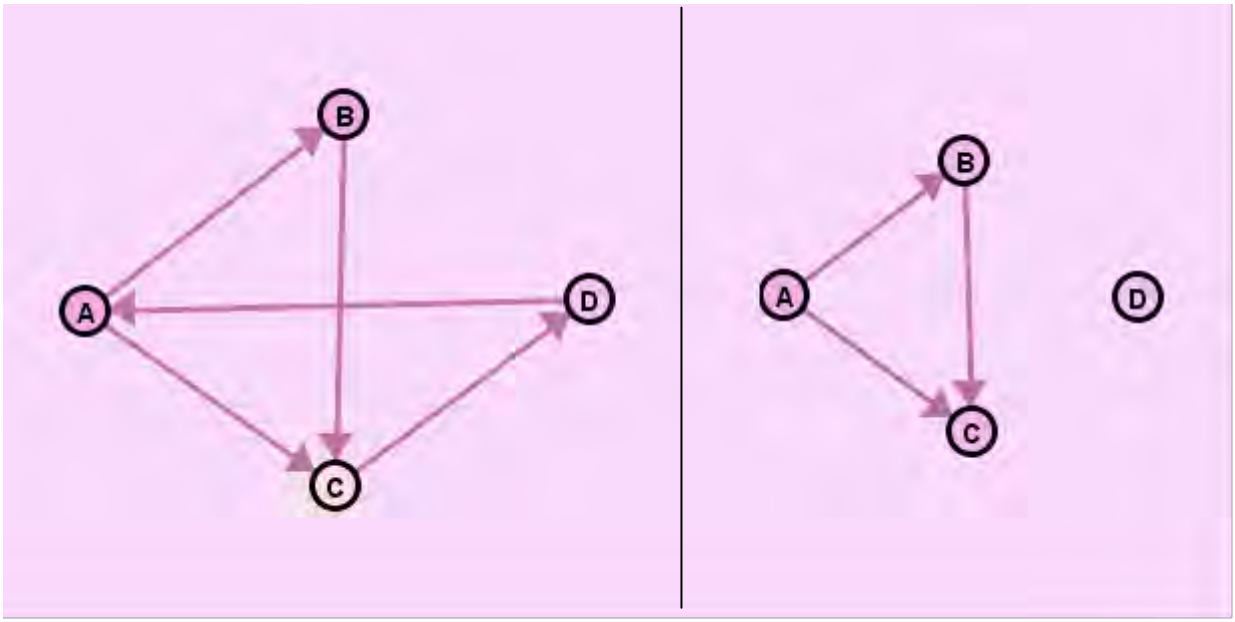
Arista o arco en el que el nodo inicial y el nodo final son el mismo.



Grafo conexo:

Se puede llegar desde cualquier nodo hasta cualquier otro siguiendo una secuencia de aristas (en el caso de un grafo dirigido se permite circular en sentido inverso a lo largo de una flecha).

Grafo conexo: Grafo no conexo:

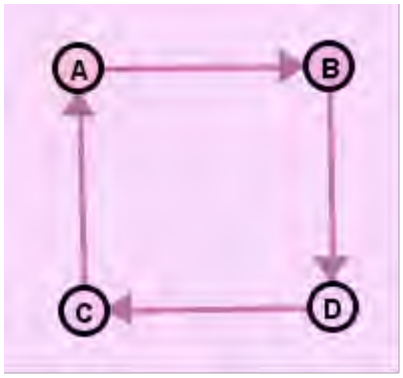


Componente conexa:

Se denomina “componente conexa” a un subconjunto de nodos de un grafo que sea conexo. En el ejemplo anterior, para el grafo conexo, cualquier subconjunto de él sería una componente conexa y para el grafo no conexo el nodo *“D”* sería una componente conexa y cualquier subconjunto de los nodos *“A”*, *“B”*, *“C”* sería una componente conexa.

Grafo dirigido fuertemente conexo:

Se puede pasar desde cualquier nodo hasta cualquier otro siguiendo una secuencia de arcos, pero respetando el sentido de las flechas. Por ejemplo:



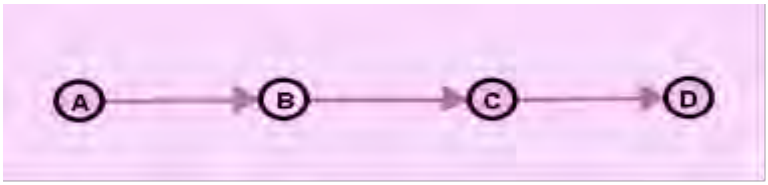
Nodos adyacentes:

El nodo *"i"* y el nodo *"j"* son adyacentes si están unidos por una arista o arco. En el ejemplo anterior, el nodo *"A"* es adyacente a los nodos *"B"* y *"C"*.

Camino:

Sucesión finita de arcos tal que el extremo final de uno de ellos coincida con el extremo inicial del siguiente.

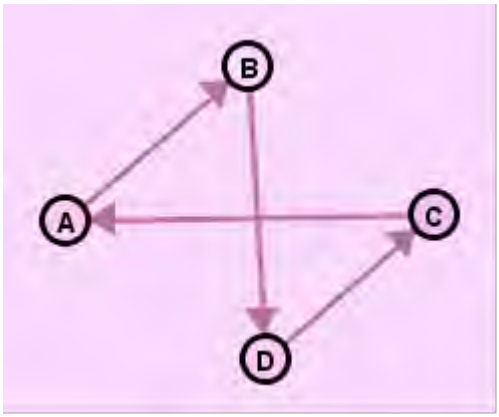
Camino de arcos *{(a, b) (b, c) (c, d)}:*



Circuito:

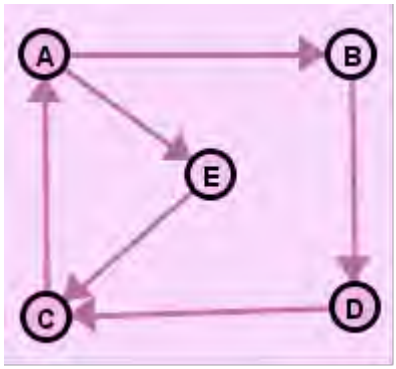
Camino en el que el nodo final coincide con el nodo inicial.

Circuito *{(a, b), (b, d), (d, c), (c, a)}:*



Camino simple:

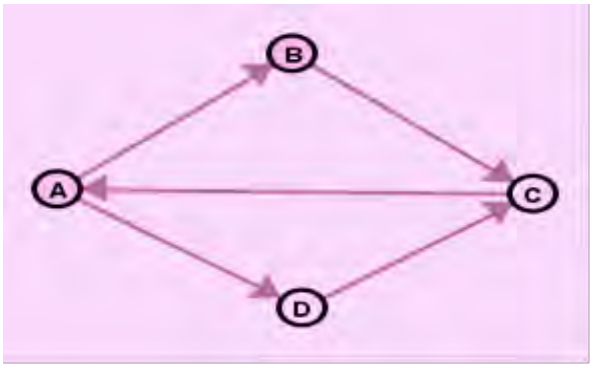
Si no se atraviesa ningún arco dos veces.



El camino *{(A, B), (B, D), (D, C), (C, A), (A, E), (E, C)}* es un camino simple.

Camino elemental:

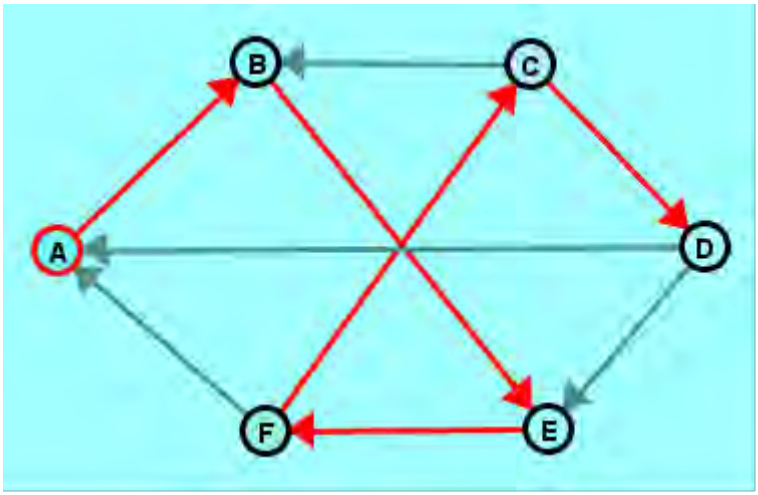
Si no se atraviesa ningún nodo dos veces.



El camino *{(A, B), (B, C)}* es un camino elemental.

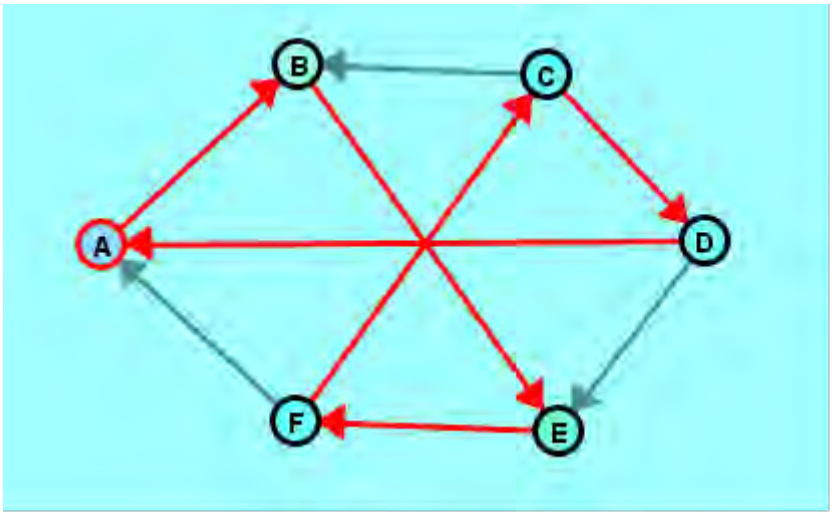
Camino Hamiltoniano:

Si se pasa exactamente una vez por cada nodo de un grafo (lógicamente, el grafo tiene que ser conexo). Los arcos en rojo nos muestran un camino hamiltoniano.



Circuito Hamiltoniano:

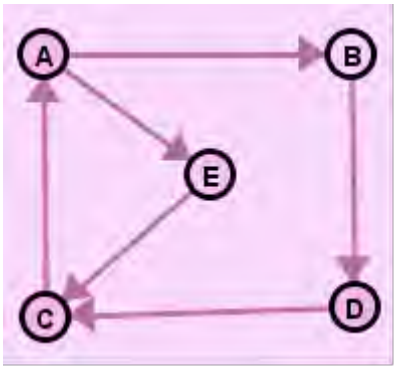
Camino Hamiltoniano en el que el nodo inicial coincide con el nodo final.



Camino Euleriano:

Aquel en el que se pasa exactamente una vez por cada arco.

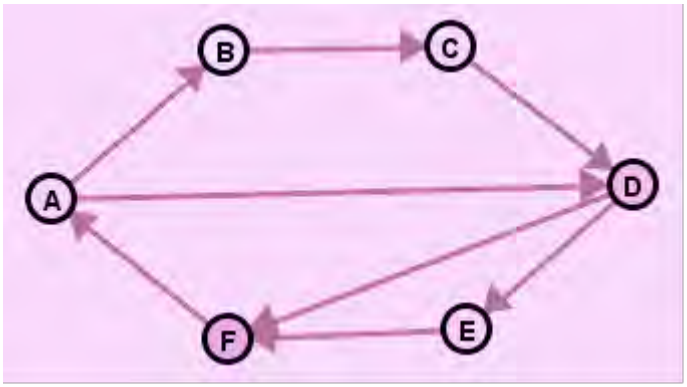
El camino *{(A, B), (B, D), (D, C), (C, A), (A, E), (E, C)}* es un camino euleriano.



Camino prehamiltoniano:

Aquel en el que se pasa al menos una vez por cada nodo.

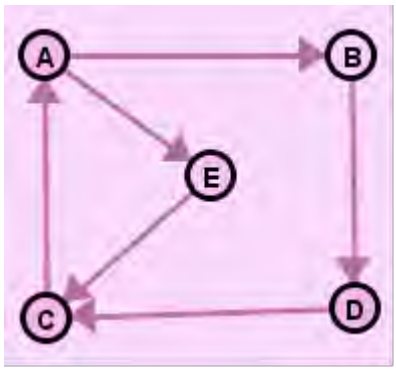
El camino *{(A, B), (B, C), (C, D), (D, E), (E, F), (F, A), (A, D), (D, F)}* es prehamiltoniano.



Camino preeuleriano:

Aquel en el que se pasa al menos una vez por cada arco.

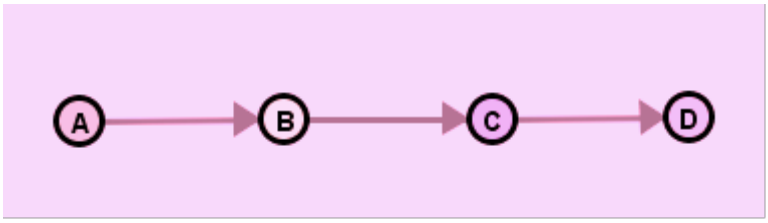
El camino *{(A, B), (B, D), (D, C), (C, A), (A, E), (E, C), (C, A), (A, B)}* es un camino preeuleriano.



Longitud u orden de un camino:

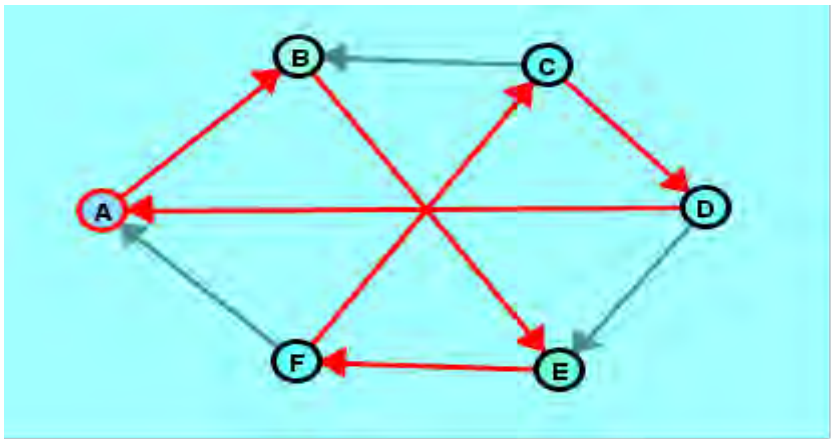
Número de arcos que tiene ese camino.

El camino *{(A, B), (B, C), (C, D)}* es de longitud u orden tres.



Circuitos equivalentes:

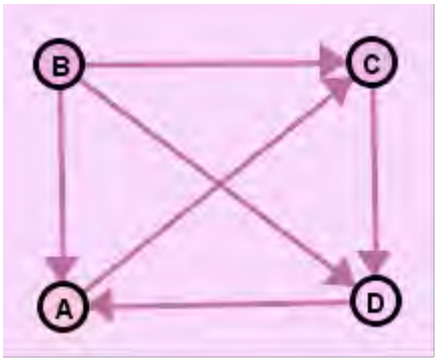
Dado un circuito hamiltoniano, son los circuitos que resultan de él por permutación circular de sus nodos.



Los circuitos hamiltonianos *{(A, B), (B, E), (E, F), (F, C), (C, D), (D, A)}* y *{(E, F), (F, C), (C, D), (D, A), (A, B), (B, E)}* son circuitos equivalentes.

Grafo simple:

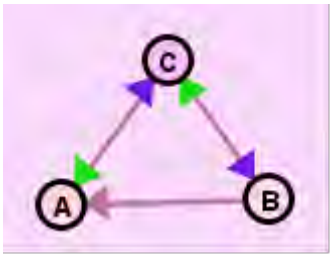
Es un grafo en el que entre dos nodos solo hay un arco:



Multigrafo:

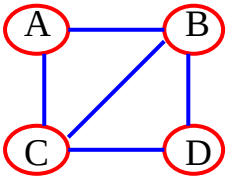
Es un grafo en el que entre dos nodos puede haber más de un arco.

En el siguiente ejemplo, el color de las flechas de los arcos nos indican su dirección.



Grado de un nodo i:

Número de aristas que inciden en ese nodo *"i"*. Se representa por *"g(i)"*.

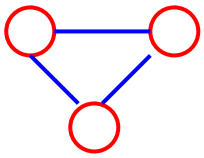


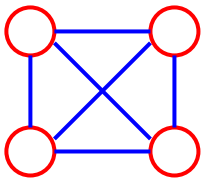
g(A) = 2; g(B) = 3

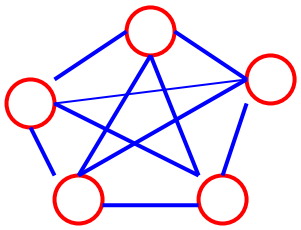
g(C) = 3; g(D) = 2

Grafo completo:

Aquel con una arista entre cada par de vértices. Un grafo completo con *"n"* vértices se representará por *"G_n":*

G_3: 

G_4: 

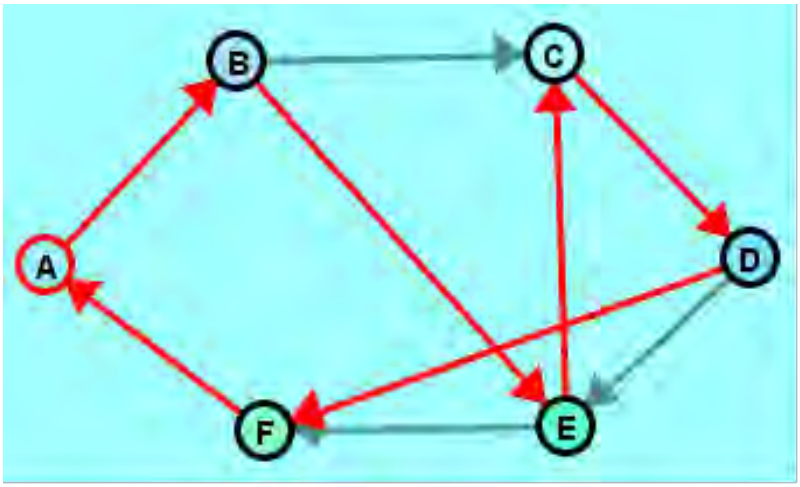
G_5: 

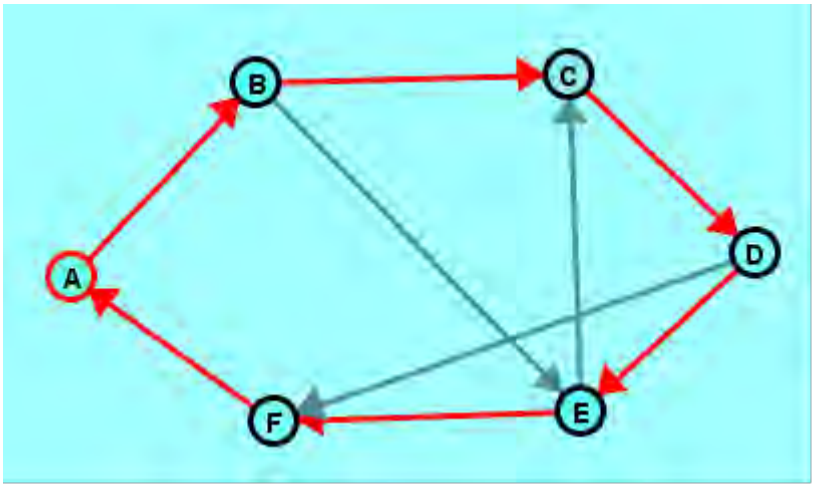
Los grafos completos tienen la siguiente propiedad: si el número de nodos del grafo es *"n"*, entonces tiene *"[n (n - 1)] / 2"* aristas.

Grafo hamiltoniano:

Es aquel que contiene al menos un circuito hamiltoniano.

El siguiente grafo hamiltoniano contiene dos circuitos hamiltonianos (con los arcos en rojo).




B
C
A
D
F
E

1.2.1- Caminos hamiltonianos de un grafo

Los caminos hamiltonianos se podrán calcular partiendo de cada nodo del grafo y si de un nodo comienza al menos un camino hamiltoniano, se calculará también el camino hamiltoniano que parte de él con la mayor suma de los valores de sus arcos y también el camino hamiltoniano con la menor suma. Posteriormente, si existen varios circuitos hamiltonianos, se calculará también los circuitos con mayor y menor suma en sus arcos.

1.2.1.1- Ayudas para identificar Grafos Hamiltonianos

Aunque no se conocen condiciones necesarias y suficientes útiles para la existencia de circuitos hamiltonianos, se han encontrado bastantes condiciones suficientes además de ciertas propiedades que se puede utilizar para demostrar que un grafo no contiene ningún circuito hamiltoniano. Por ejemplo, un grafo con un nodo en el que incide o del que sale un sólo arco no puede contener un circuito hamiltoniano ya que en un circuito hamiltoniano cada nodo tiene dos arcos del circuito (uno entrante y otro saliente). Además, si un nodo del grafo tiene sólo dos arcos, uno entrante y otro saliente, entonces esos dos arcos tienen que ser parte del recorrido de cualquier circuito hamiltoniano. A medida que añadimos arcos a un grafo, vamos haciendo más y más probable que exista un circuito hamiltoniano en ese grafo, por ello, es de esperar que haya condiciones suficientes para la existencia de circuitos hamiltonianos que dependan de que el grado de los vértices sea suficientemente grande. Dos de estas condiciones suficientes más importantes fueron descubiertas por Gabriel A. Dirac en 1952 y por Oystein Ore en 1960.

Los teoremas de Dirac y Ore nos proporcionan condiciones suficientes para que exista un circuito hamiltoniano. Sin embargo, estos teoremas no dan condiciones necesarias para la existencia de un circuito hamiltoniano, es decir, puede existir un

grafo que tenga algún circuito hamiltoniano y no cumpla las hipótesis de Ore ni de Dirac.

Teorema de Dirac: Dado un grafo $G = < N , A >$ simple con *"n"* nodos, con un número de nodos $|N| \geq 3$. Si $\forall v \in N$ se verifica que $g(v) \geq n / 2$, entonces existe al menos un circuito hamiltoniano.

Teorema de Ore: Dado un grafo $G = < N , A >$ simple con *"n"* nodos, cuyo número de nodos sea $|N| \geq 3$. Si se verifica que $g(v) + g(w) \geq n$, $\forall v, w \in N$ con $v \neq w$ y no adyacentes, entonces el grafo tiene, al menos, un circuito hamiltoniano.

Los siguientes puntos nos pueden ayudar para identificar grafos hamiltonianos:

1. Todo grafo G completo con $n >= 3$ posee un circuito hamiltoniano.
2. Todo grafo con un punto de corte no es hamiltoniano.
3. Si G no es conexo no posee ningún circuito hamiltoniano.
4. Un camino hamiltoniano contiene *"n-1"* arcos.
5. Un circuito hamiltoniano contiene n arcos.
6. Si G posee un circuito hamiltoniano todo nodo del grafo debe tener un *Grado >= 2*.
7. Si existen nodos de grado dos en el grafo necesariamente sus aristas incidentes deben estar en el ciclo de hamilton.
8. Cuando se construye un circuito hamiltoniano y este circuito ha pasado por un nodo, pueden descartarse todas las aristas incidentes en ese nodo que no sean las dos usadas en el circuito.

1.2.1.2.- Método para encontrar todos los caminos hamiltonianos de un grafo

Primero se explicará el procedimiento utilizado para calcular caminos elementales lo cual nos ayudará mucho a la hora de entender el algoritmo de Kaufmann para calcular los caminos hamiltonianos.

Y para finalizar este apartado, a través del algoritmo de Kaufmann y Malgrange se calcularán todos los caminos hamiltonianos del grafo.

1.2.1.2.1- Determinación de caminos elementales

Se aclararán algunos conceptos sobre las distintas operaciones que se realizarán con matrices. Primero se empezará por una multiplicación normal de matrices para ver el orden de interacción de cada elemento de la primera matriz con los elementos de la segunda matriz. Una vez conocido el orden de interacción entre dos matrices, se dará paso a la multiplicación latina de matrices centrándonos en la operación resultante entre dos elementos (cadenas). A continuación, veremos como se representa un grafo en una matriz y como se calculan los caminos elementales hasta que estos caminos se conviertan en caminos hamiltonianos utilizando la multiplicación latina de matrices.

Multiplicación de matrices

Para que dos matrices *A* y *B* se puedan multiplicar, se requiere que el número de columnas de la primera matriz *A* sea igual al número de filas de la segunda matriz *B*.

Cada elemento resultante de la multiplicación (matriz *C*) es el resultado de aplicar la siguiente fórmula:

$$C(i,j) = \sum_{K=1}^{n} A(i,k) * B(k,j)$$

Siendo "*n*" el número de columnas de la matriz *A*.

Por ejemplo:

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix} \quad ; \quad B = \begin{pmatrix} 1 & 5 & 10 \\ 2 & 6 & 11 \\ 3 & 7 & 12 \\ 4 & 8 & 13 \end{pmatrix}$$

El resultado de la multiplicación de las matrices *A* y *B* es:

$$C = \begin{pmatrix} 30 & 70 & 120 \\ 70 & 174 & 304 \\ 110 & 278 & 488 \end{pmatrix}$$

Para obtener el elemento *C(2,2)* se han tenido que efectuar las siguientes operaciones:

C (2 , 2)	=	A (2 , 1)	*	B (1 , 2)	=	5 * 5
		A (2 , 2)	*	B (2 , 2)	=	6 * 6
		A (2 , 3)	*	B (3 , 2)	=	7 * 7
		A (2 , 4)	*	B (4 , 2)	=	8 * 8
						174 Suma.

Multiplicación latina de matrices

Esta operación sigue el procedimiento de una multiplicación normal de matrices pero los elementos de las matrices no se multiplican (no son valores numéricos, representan cadenas). La operación que se realiza entre ellos es una multiplicación latina:

Si se tiene dos cadenas elementales:

S = (i1, i2, …, ip)

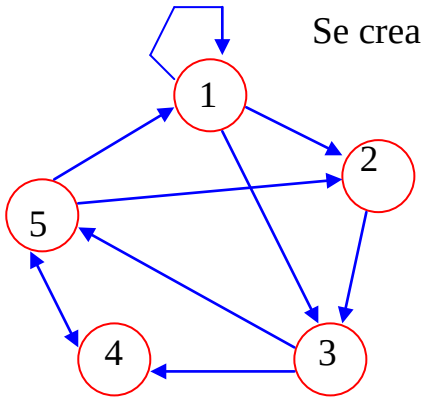
T = (j1, j2, …, jq)

La multiplicación latina *(XL)* de ambas cadenas será:

$$\begin{cases} S \; XL \; T = \{(i1, i2, \ldots, ip, j2, j3, \ldots, jq)\} \\ Si \quad S \sqcap T = ip \quad y \quad ip = j1 \\ \\ Y \quad \text{“”} \quad \text{en los demás casos.} \end{cases}$$

Matrices utilizadas, multiplicación latina sobre ellas y obtención de caminos elementales

Para calcular los caminos hamiltonianos de un grafo, primero hay que obtener la matriz inicial de ese grafo con la que vamos a trabajar. Siguiendo un grafo de ejemplo:



Se crea la matriz $[M]^1$ de orden uno cuyos elementos son:

$$m^1_{ij} = \begin{cases} ij & \text{Si } (i, j) \text{ es un arco del grafo.} \\ \text{“”} & \text{En caso contrario.} \end{cases}$$

El carácter “#” es un simple separador de nodos.

$$[M]^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} \text{“”} & 1\#2 & 1\#3 & \text{“”} & \text{“”} \\ \text{“”} & \text{“”} & 2\#3 & \text{“”} & \text{“”} \\ \text{“”} & \text{“”} & \text{“”} & 3\#4 & 3\#5 \\ \text{“”} & \text{“”} & \text{“”} & \text{“”} & 4\#5 \\ 5\#1 & 5\#2 & \text{“”} & 5\#4 & \text{“”} \end{pmatrix} \end{array}$$

Los bucles se eliminan.

Esta matriz nos dará los caminos elementales de orden uno.

Una vez que se tiene la matriz inicial, la multiplicación latina de esta matriz consigo misma es:

$[M]^2 = [M]^1$ XL $[M]^1 =$

	1	2	3	4	5
1	“”	“”	1#2#3	1#3#4	1#3#5
2	“”	“”	“”	2#3#4	2#3#5
3	3#5#1	3#5#2	“”	3#5#4	3#4#5
4	4#5#1	4#5#2	“”	“”	“”
5	“”	5#1#2	5#1#3 5#2#3	“”	“”

Esta matriz nos dará los caminos elementales de orden dos.

Por ejemplo,

Si un elemento de la primera matriz está compuesto por 1 # 2

y un elemento de la segunda matriz está compuesto por 2 # 3

se ve que el único elemento común en las dos cadenas es el 2, la multiplicación latina nos dará como resultado: *1#2#3*

Si se encuentra un caso como éste: *2#3#4* en la primera matriz con *4#5#2* en la segunda matriz, el resultado sería “” porque aunque el elemento 4 es coincidente, se repite el elemento 2.

Se observa que el elemento m^2_{53} tiene dos componentes *5#1#3* y *5#2#3*, es decir, la matriz $[M]^2$ tiene, o puede tener, una dimensión más.

Más formalmente:

$$[M]^p = \begin{pmatrix} m^p_{11} & m^p_{12} & m^p_{13} & \ldots\ldots\ldots\ldots & m^p_{1n} \\ & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & & & \\ m^p_{n1} & m^p_{n2} & m^p_{n3} & \ldots\ldots\ldots\ldots & m^p_{nn} \end{pmatrix}$$

$$[M]^q = \begin{pmatrix} m^q_{11} & m^q_{12} & m^q_{13} & \ldots\ldots\ldots & m^q_{1n} \\ & & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & & \\ m^q_{n1} & m^q_{n2} & m^q_{n3} & \ldots\ldots\ldots\ldots & m^q_{nn} \end{pmatrix}$$

Un elemento m^p_{ij} de la matriz $[M]^p$ representa un camino elemental de "*i*" a "*j*" de orden *p*.

Un elemento m^q_{jk} de la matriz $[M]^q$ representa un camino elemental de "*j*" a "*k*" de orden *q*.

La multiplicación latina de estos dos elementos m^p_{ij} XL m^q_{jk} representa los caminos elementales de "*i*" a "*k*" de orden $p + q$.

1.2.1.2.2- Determinar todos los caminos hamiltonianos (algoritmo de Kaufmann y Malgrange)

Enumera sin redundancia todos los caminos elementales de longitudes *1, 2, …* , hasta *"n – 1"* (siendo *"n"* el número de nodos del grafo) que parten de cada nodo, si es que existen dichos caminos.

1.- Iniciación:

A partir de un grafo, nos creamos la matriz $[M]^1$ cuyos elementos son:

$$m^1_{ij} = \begin{cases} ij & \text{Si } (i\,,j) \text{ es un arco del grafo.} \\ \text{""} & \text{En caso contrario.} \end{cases}$$

Los bucles se eliminan.

2.- $r = 1$; siendo *"r"* el índice de la matriz.

3.- Se realiza la multiplicación latina de matrices:

$$[M]^{r+1} = [M]^r \text{ XL } [M]^1$$

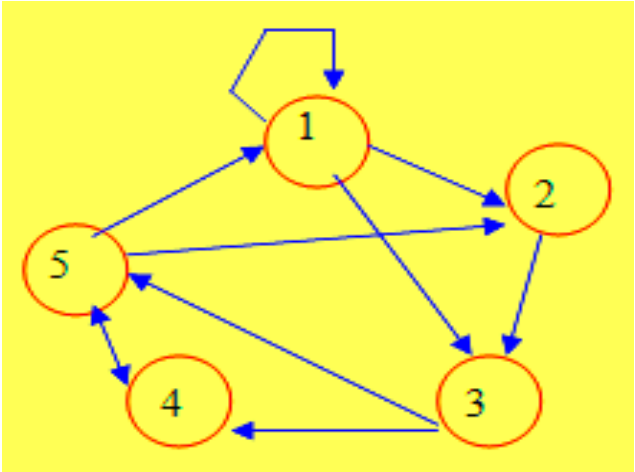
4.- Finalización:

Siendo *"n"* el número de nodos del grafo,

- Si $r = n - 1$ Fin ; la matriz $[M]^{n-1}$ contiene todos los caminos hamiltonianos del grafo. La fila "*i*" de esta matriz corresponde a los caminos hamiltonianos que parten del nodo "*i*".

- Y sino, $r = r + 1$ y se vuelve al paso *3º*.

Para ver el desarrollo práctico de este algoritmo, se seguirá el mismo ejemplo del grafo anterior para calcular todos sus caminos hamiltonianos.



Pasos 1º y 2º:

Para nuestro grafo, la matriz $[M]^1$ de índice uno ($r = 1$) será:

	1	2	3	4	5
1	“”	1#2	1#3	“”	“”
2	“”	“”	2#3	“”	“”
3	“”	“”	“”	3#4	3#5
4	“”	“”	“”	“”	4#5
5	5#1	5#2	“”	5#4	“”

Paso 3º:

Realizamos la multiplicación latina $[M]^{r+1} = [M]^r$ XL $[M]^1$, ya hecha anteriormente:

$[M]^2 = [M]^1$ XL $[M]^1 =$

	1	2	3	4	5
1	“	“	1#2#3	1#3#4	1#3#5
2	“	“	“	2#3#4	2#3#5
3	3#5#1	3#5#2	“	3#5#4	3#4#5
4	4#5#1	4#5#2	“	“	“
5	“	5#1#2	5#1#3 5#2#3	“	“

$r = 2$ y $n = 5$, luego no se cumple que $r = n - 1$ y se vuelve al paso 3º para calcular $[M]^3$:

$[M]^3 = [M]^2$ XL $[M]^1 =$

“	1#3#5#2	“	1#2#3#4 1#3#5#4	1#2#3#5 1#3#4#5
2#3#5#1	“	“	2#3#5#4	2#3#4#5
3#4#5#1	3#5#1#2	“	“	“
“	4#5#1#2	4#5#1#3 4#5#2#3	“	“
“	“	5#1#2#3	5#1#3#4 5#2#3#4	“

Como todavía el valor de *“r”* no es *“n-1”* , se vuelve al paso *3º* y se calcula $[M]^4$:

$$[M]^4 = [M]^3 \text{ XL } [M]^1 = \begin{pmatrix} \text{““} & \text{1\#3\#4\#5\#2} & \text{““} & \text{1\#2\#3\#5\#4} & \text{1\#2\#3\#4\#5} \\ \text{2\#3\#4\#5\#1} & \text{““} & \text{““} & \text{““} & \text{““} \\ \text{““} & \text{3\#4\#5\#1\#2} & \text{““} & \text{““} & \text{““} \\ \text{““} & \text{““} & \text{4\#5\#1\#2\#3} & \text{““} & \text{““} \\ \text{““} & \text{““} & \text{““} & \text{5\#1\#2\#3\#4} & \text{““} \end{pmatrix}$$

Ahora $r = n - 1 = 4$ y por tanto se ha terminado de calcular todos los caminos hamiltonianos del grafo. Estos caminos son:

Desde el nodo	Caminos hamiltonianos del grafo
1	1#3#4#5#2 1#2#3#5#4 1#2#3#4#5
2	2#3#4#5#1
3	3#4#5#1#2
4	4#5#1#2#3
5	5#1#2#3#4

Se puede hacer una modificación para disminuir considerablemente, sobre todo si el grafo es grande, el número de veces que se realiza la multiplicación latina:

1.- Iniciación:

Siendo “*n*” el número de nodos del grafo,

A partir de un grafo, se crea la matriz $[M]^1$ cuyos elementos son:

$$m^1_{ij} = \begin{cases} ij & \text{Si } (i\ ,\ j\ ) \text{ es un arco del grafo.} \\ \text{“”} & \text{En caso contrario.} \end{cases}$$

Los bucles se eliminan.

2.- $r = 1$; siendo “r” el índice de la matriz.

3.- Calculamos la matriz $[M]^2$:

$[M]^2 = [M]^1 \text{ XL } [M]^1$

$r = 2$

4.- Según sea el valor del índice r se hará una cosa u otra.

- Si el índice r de $[M]^r$ cumple $r + r <= n - 1$:

  Se calcula $[M]^{r+r} = [M]^r \text{ XL } [M]^r$

  $r = r + r$

- Si el índice r de $[M]^r$ cumple $r < n - 1$ y $r + r > n - 1$:

  Se calcula $[M]^{r+1} = [M]^r \text{ XL } [M]^1$

  $r = r + 1$

5.- Finalización:

- Si $r = n - 1$ Fin ; la matriz $[M]^{n-1}$ contiene todos los caminos hamiltonianos del grafo. La fila "*i*" de esta matriz corresponde a los caminos hamiltonianos que parten del nodo "*i*".

- Y sino, se vuelve al paso 4^o.

Si se siguiera el ejemplo anterior, se obtendría $[M]^1$ del grafo, después $[M]^2$, y por último $[M]^4$.

Un dato muy importante es que cuando se trabaja con grafos muy grandes y muchos arcos el cálculo de las matrices puede tardar un poco (los recursos del ordenador son finitos). Para esta inevitable situación, en la barra de estado de la aplicación informática se muestra en qué punto está el proceso (que matriz se está calculando) y el usuario tiene dos opciones, interrumpir el cálculo de los caminos hamiltonianos haciendo clic con el ratón en un botón sin que se produzca un error, o simplemente, dejar pasar el tiempo hasta que se termine el cálculo de las matrices. En caso de que la memoria del ordenador se desborde, simplemente aparece una ventana con un mensaje explicándolo interrumpiendo el proceso de forma normal y sin que se produzcan errores. Estas dos situaciones son muy excepcionales pero se tienen que controlar.

1.2.1.3- Optimización de los caminos hamiltonianos

Partiendo de cada nodo del grafo, se recorren todas las soluciones obtenidas (caminos hamiltonianos) y para cada una de ellas, si el valor de la suma total de sus arcos es mayor que los valores totales de los caminos hamiltonianos ya vistos (desde el mismo nodo de partida), se copia en un vector de máximos el nodo de inicio del camino hamiltoniano y su suma total. Se hace el mismo procedimiento para los valores menores. Es decir, para cada nodo del grafo que sea el inicio de un camino hamiltoniano, existirá un vector que contenga dicho nodo y la suma total de los arcos de un camino hamiltoniano que parte de él de tal manera que esa suma sea la mayor de todos los caminos hamiltonianos que parten de dicho nodo. Lo mismo hay que decir para los valores mínimos. Por ejemplo, los vectores podrían ser para un caso dado:

MayorValorSolución $\longrightarrow$ *{{“A”, 15,3}, {“B”, 8}, {“C”, 14}}*

MenorValorSolución $\longrightarrow$ *{{“A”, 6}, {“B”, 8}, {“C”, 2,5}}*

Puede haber varias soluciones para un mismo nodo, estos caminos hamiltonianos se muestran ordenados en la ventana de soluciones máximas o mínimas de la aplicación.

1.2.2- Cálculo de los circuitos hamiltonianos y su optimización

Dado un grafo *G = (N , A)* ; siendo *"N"* el conjunto de nodos y *"A"* el conjunto de arcos, primero se calculará mediante el algoritmo de Kaufmann los caminos hamiltonianos del grafo, es decir, la matriz $[M]^{n-1}$, siendo *"n"* en número de nodos del grafo.

Si ahora se calcula $[M]^n = [M]^1$ XL $[M]^{n-1}$ pero con la modificación de que al hacer la multiplicación latina también será válido unir caminos elementales con los nodos primero de la matriz $[M]^1$ y último de la matriz $[M]^{n-1}$ coincidentes, resultará una matriz $[M]^n$ cuyos componentes son todos "" menos en la diagonal principal en la que se encontrarán todos los circuitos hamiltonianos del grafo.

Cada elemento de la diagonal tiene <u>todos</u> los circuitos hamiltonianos del grafo, y en cada fila "*i*" los circuitos empezarán por el nodo "*i*". En la práctica, sólo será necesario iterar la primera fila de $[M]^1$ con la primera columna de $[M]^{n-1}$.

Siguiendo el ejemplo del grafo anterior, se calcula sus circuitos hamiltonianos, es decir, se calcula $[M]^5 = [M]^1$ XL $[M]^4$

$$[M]^5 = \begin{pmatrix} 1\#2\#3\#4\#5\#1 & \text{““} & \text{““} & \text{““} & \text{““} \\ \text{““} & 2\#3\#4\#5\#1\#2 & \text{““} & \text{““} & \text{““} \\ \text{““} & \text{““} & 3\#4\#5\#1\#2\#3 & \text{““} & \text{““} \\ \text{““} & \text{““} & \text{““} & 4\#5\#1\#2\#3\#4 & \text{““} \\ \text{““} & \text{““} & \text{““} & \text{““} & 5\#1\#2\#3\#4\#5 \end{pmatrix}$$

En esta matriz se puede ver que sólo existe un circuito hamiltoniano, dependiendo de la fila "*i*" en que esté el circuito el nodo de partida será el nodo "*i*" pero es el mismo circuito en todas las filas.

Al igual que para los caminos hamiltonianos, con grafos grandes, se resuelven igualmente los problemas de un proceso lento o un desbordamiento de memoria.

Para obtener los circuitos hamiltonianos óptimos (los de mayor y menor suma de sus arcos), el nodo de partida es irrelevante al tratarse de un circuito, simplemente se calculan los circuitos de mayor y menor suma total.

1.3- Ejemplo completo

Una persona quiere hacer un recorrido en coche por todas las capitales de provincia de Andalucía. En principio, la ciudad de partida puede ser cualquiera y la última ciudad del recorrido tiene que ser la misma ciudad de inicio del circuito. De lo que se trata, es de pasar exactamente una vez por cada ciudad andaluza y volver a la ciudad de partida habiendo recorrido el menor número de kilómetros en total.

El primer paso, es hacernos con un mapa de Andalucía y ver la red de carreteras existente entre las capitales de provincia (marcadas con una flecha en rojo).

A continuación tenemos que buscar los kilómetros reales que hay entre las capitales de provincia conectadas entre sí por carretera según el mapa:

Ciudades	Km
Sevilla – Cádiz	125
Sevilla – Huelva	94
Sevilla – Córdoba	138
Sevilla – Granada	256
Sevilla – Málaga	219
Málaga – Cádiz	265
Málaga – Córdoba	187
Málaga – Granada	129
Málaga – Almería	219
Granada – Jaén	99
Granada – Almería	166
Córdoba – Jaén	104

Ya tenemos toda la información necesaria para construir un grafo que represente a las ciudades andaluzas y sus conexiones en kilómetros por carretera.

Desde la página principal de la aplicación, en su parte superior nos vamos al enlace "Applets Algoritmos":

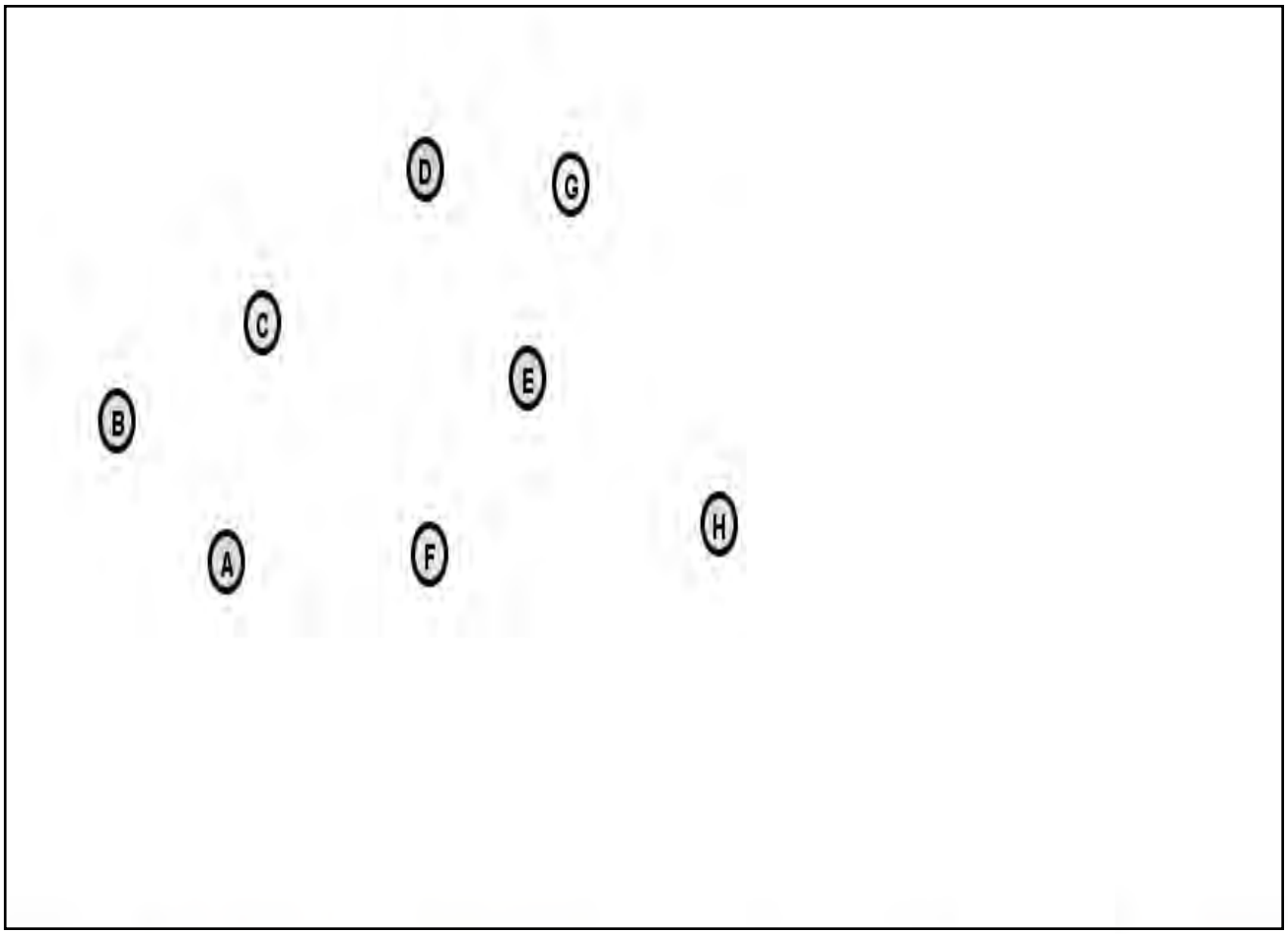
Esta nueva ventana nos da información útil para manejar la aplicación. Ahora lo que nos interesa es ver la interface de usuario para poder trabajar. Para ello hacemos clic con el ratón en el enlace "**aquí**" referente a los caminos y circuitos hamiltonianos.

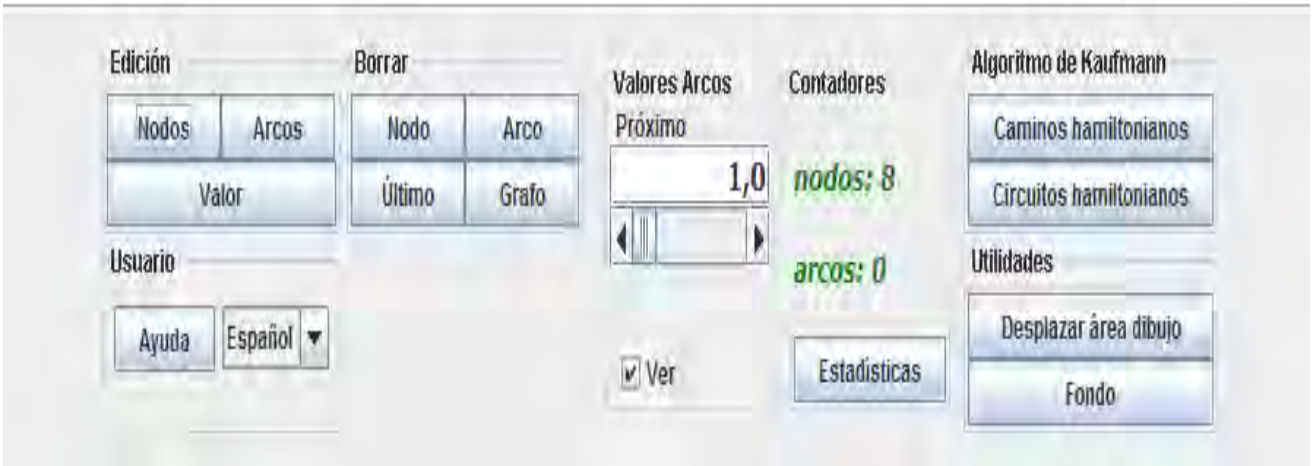
Y se nos visualizará la ventana de trabajo:

En el marco “Usuario”, ya está seleccionado el idioma “Español” y suponemos que sabemos manejar esta ventana, por tanto, no se va a utilizar el marco “Usuario”.

Ejemplo completo

Siempre guiados por la información que aparece en la barra de estado de la parte inferior, primero crearemos las ocho ciudades andaluzas representadas por ocho nodos. En el marco “Edición”, el botón “Nodos” está activado por defecto al abrirse esta ventana y sólo hay que hacer clic con el ratón ocho veces en el área de dibujo en distintas posiciones:





Nodo “A” ⟹ Cádiz.

Nodo “B” ⟹ Huelva.

Nodo “C” ⟹ Sevilla.

Nodo “D” ⟹ Córdoba.

Nodo “E” ⟹ Granada.

Nodo “F” ⟹ Málaga.

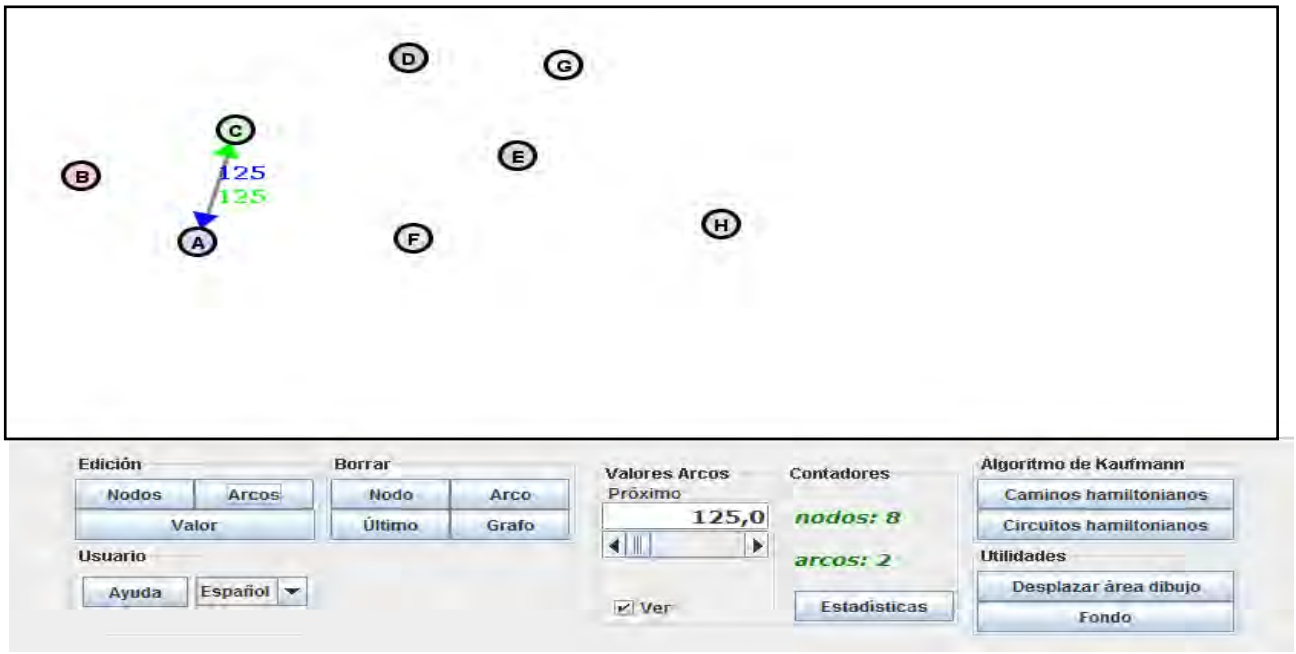
Nodo “G” ⟹ Jaén.

Nodo “H” ⟹ Almería.

A continuación crearemos los arcos que representan las carreteras y pondremos sobre ellos los kilómetros entre ciudad y ciudad. Por ejemplo, si queremos unir las ciudades de Sevilla y Cádiz, en el marco “Valores arcos” ponemos los kilómetros entre ambas ciudades “125”. A continuación, en el marco “Edición”, hacemos clic con el ratón en el botón “Arcos”.

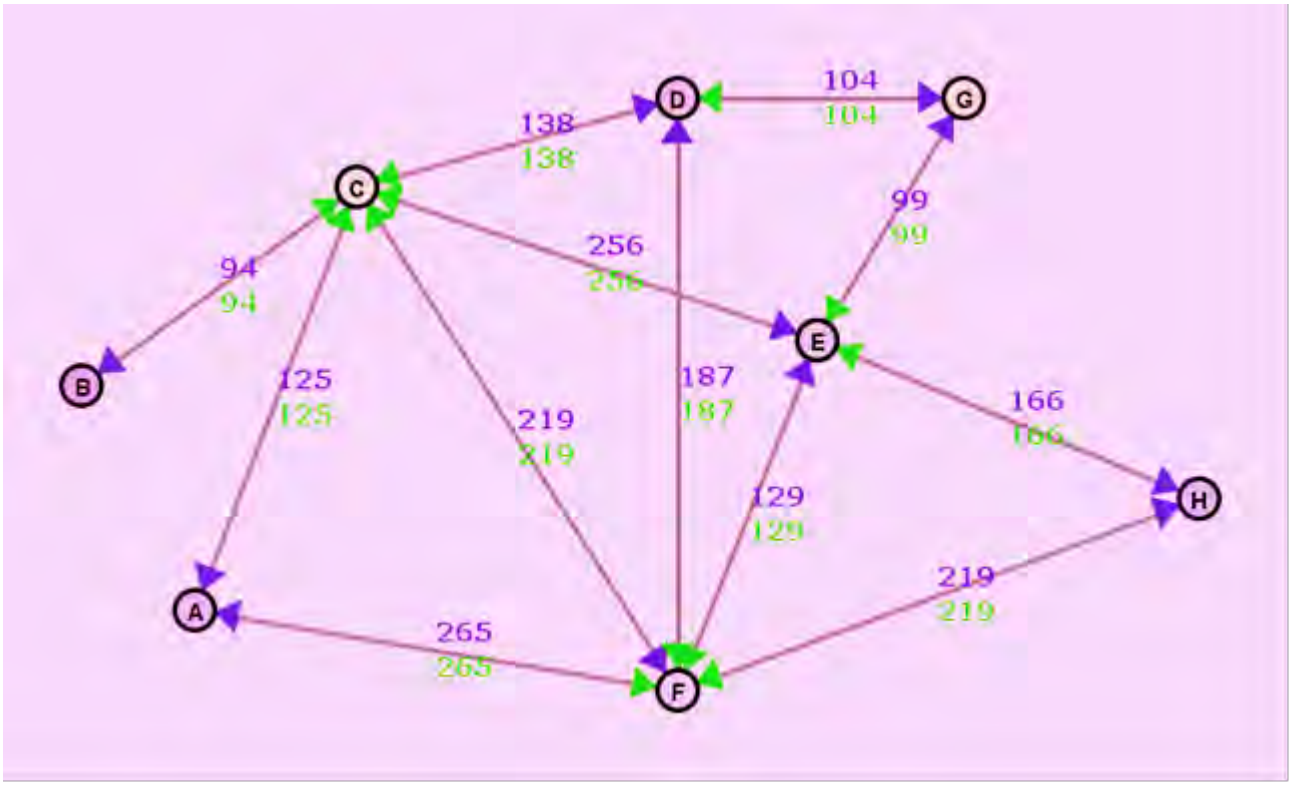
Ahora, el trazar un arco es tan sencillo como hacer clic con el ratón en el nodo *“C”* (Sevilla) y a continuación hacer clic otra vez sobre el nodo *“A”* (Cádiz), aparecerá un arco de Sevilla a Cádiz con los 125 kilómetros que hay entre ambas ciudades.

También hay que trazar un arco en sentido contrario y del mismo valor, ya que también se puede ir de Cádiz a Sevilla. Para ello, simplemente hacemos clic en el nodo *“A”* (Cádiz) y después otro clic en el nodo *“C”* (Sevilla).



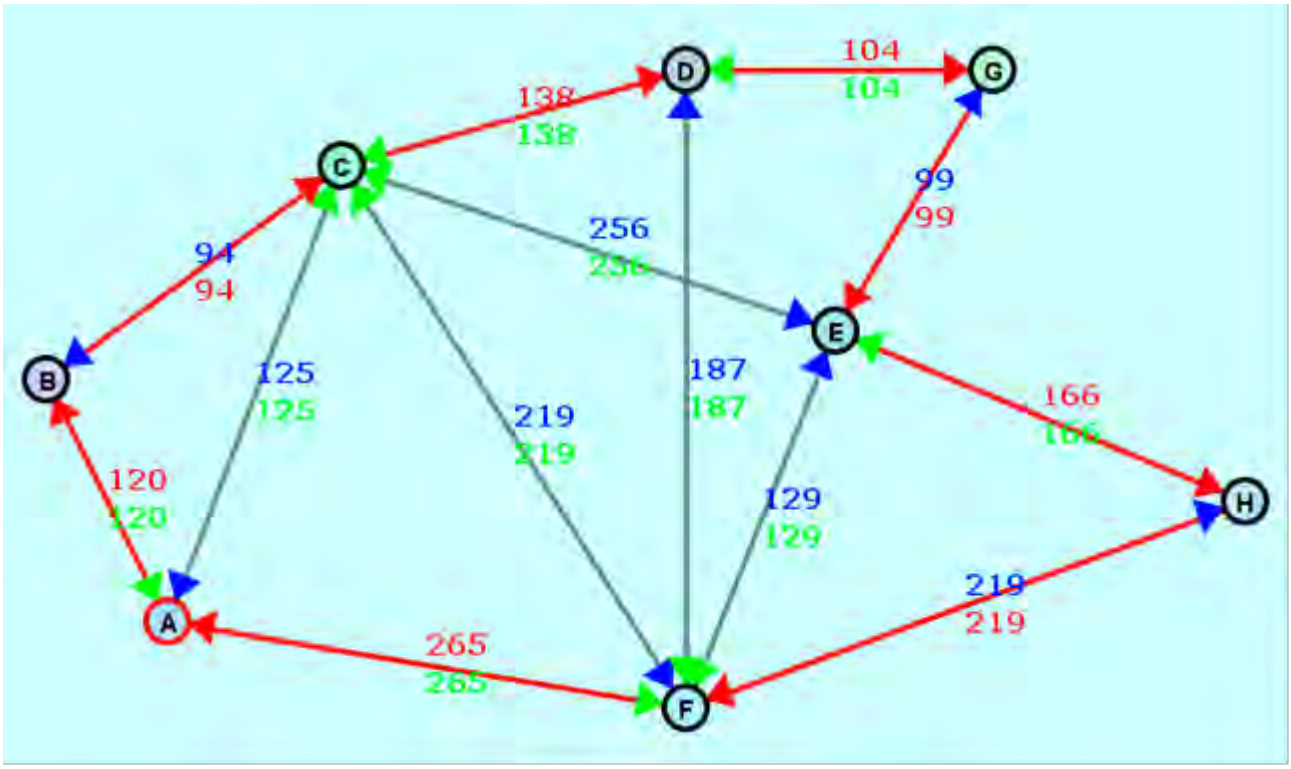
Ejemplo completo

Si repetimos estas operaciones para trazar el resto de arcos nos quedará el siguiente grafo:

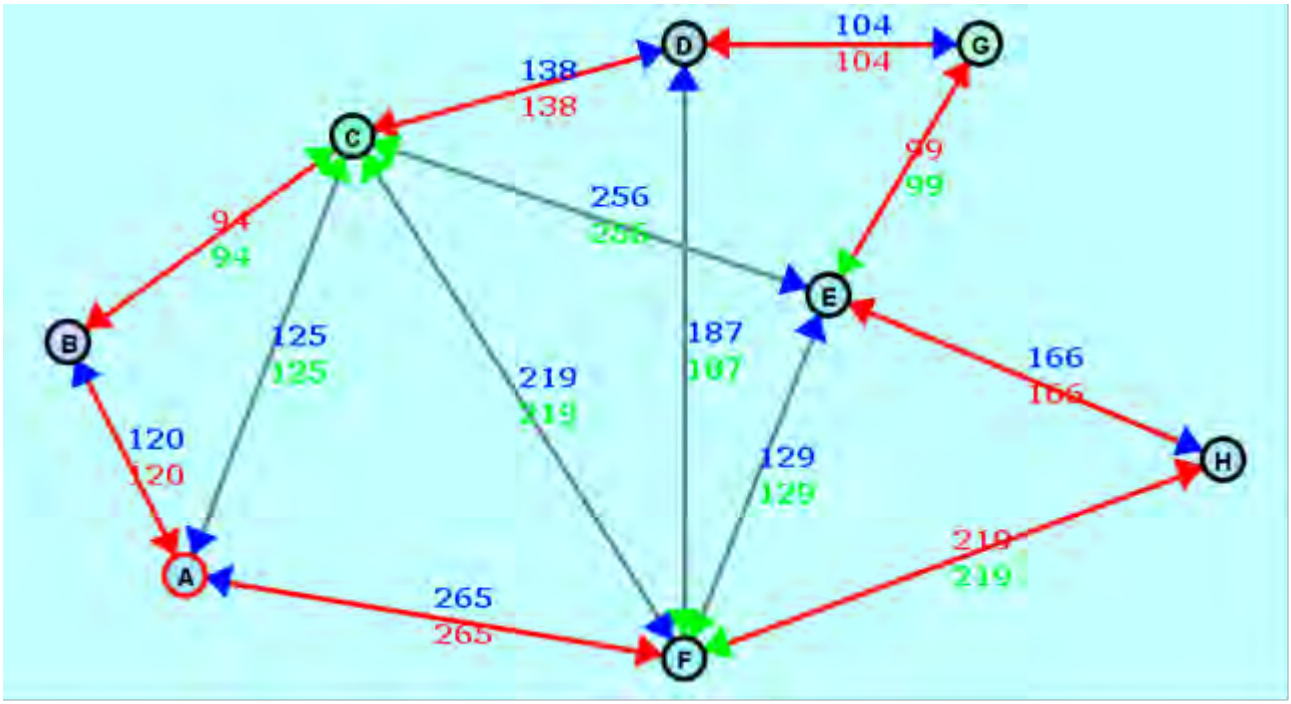


Sólo queda ya hacer clic en el botón “Circuitos hamiltonianos”. Se puede comprobar que no existe ningún circuito hamiltoniano para este grafo, habrá que ir por carreteras secundarias para que aparezca alguno. Vamos a suponer que de Cádiz a Huelva se puede ir directamente recorriendo una distancia de 120 kilómetros. Añadimos dos arcos en sentido contrario entre los nodos *“A”* y *“B”* con un valor de 120 y a continuación hacemos clic en el botón “Circuitos hamiltonianos”.

Nos aparecen dos circuitos con el mismo total de kilómetros recorridos, en realidad es el mismo circuito recorrido en un sentido o en otro, partiendo de cualquier ciudad.



Solución 1



Solución 2

Vamos a añadir una carretera entre Jaén y Almería de 200 kilómetros de distancia trazando dos arcos en sentido contrario desde los nodos *"G"* y *"H"* con un valor de 200 y volvemos a pulsar el botón de "Circuitos hamiltonianos".

Nos salen 6 circuitos hamiltonianos. Pulsando el botón de siguiente vamos viendo las distintas soluciones y si pulsamos el botón “Soluciones menores” situado en la ventana de soluciones, nos aparecen dos con un valor total mínimo de kilómetros recorridos de 1205. Estas dos soluciones son también las soluciones 1 y 2 ya vistas anteriormente.

Para volver a la edición del grafo, hacemos clic en el botón “Terminar”.

Para volver a la página principal, hacemos clic en el enlace “cerrar ventana” de la parte superior.

1.4- Estructuración del programa

Lo primero que hay que hacer antes de programar la aplicación en el ordenador es saber que soportes de datos se van a utilizar y su relación entre ellos. Con esta ayuda, la programación se centra en resolver y explicar los algoritmos de una forma más sencilla sin entrar en detalles que pueden enturbiar su seguimiento y que ya se han aclarado antes de meternos en el pseudocódigo.

El pseudocódigo está basado en una programación estructurada muy fácil de seguir y dividida en procedimientos para simplificar y modular toda la aplicación.

1.4.1- Soporte de datos

Se ha elaborado documentación sobre los soportes de datos que se utilizan y su interrelación, lo que ayudará mucho a comprender como se ha programado. Esto añadido a la documentación en pseudocódigo, hará muy viable la implementación de la aplicación en cualquier lenguaje de programación.

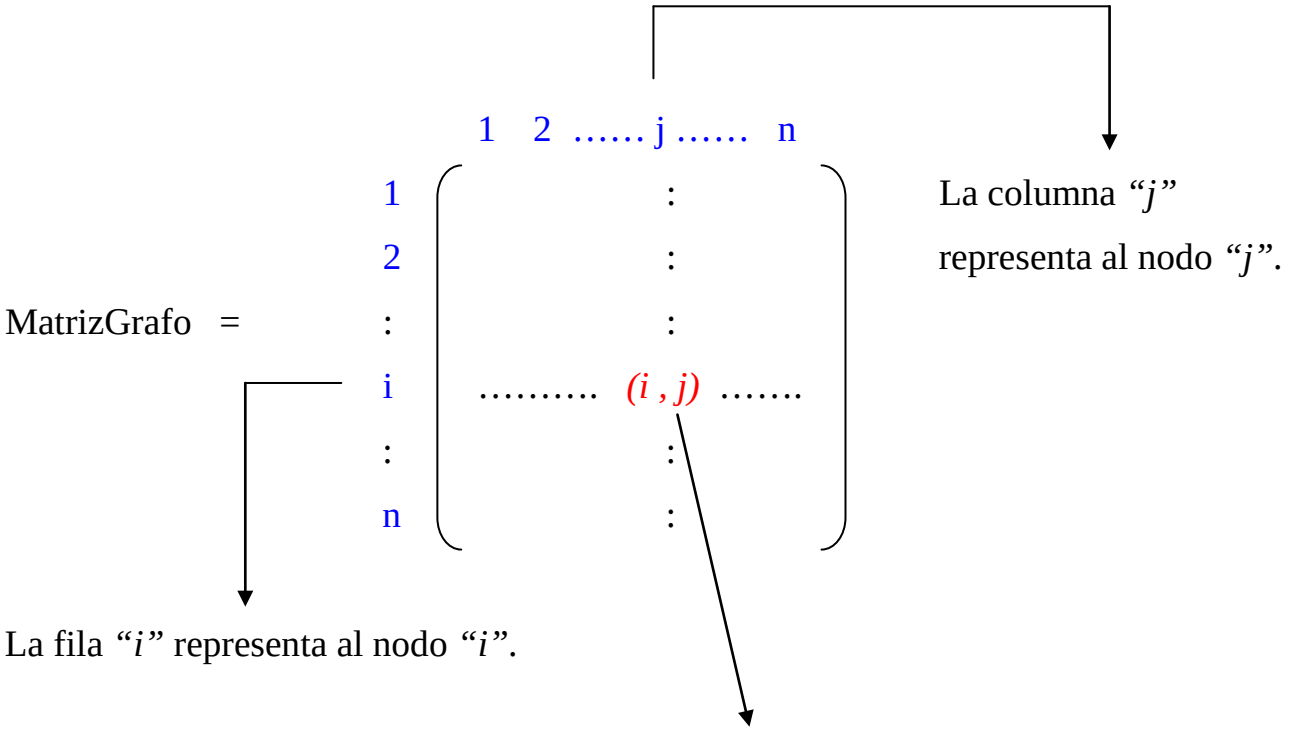
La interfaz gráfica que maneja el usuario proporciona un grafo de partida sobre el que se trabaja para obtener sus caminos y circuitos hamiltonianos.

El primer paso es representar la información del grafo en una matriz de cadenas de caracteres que se llamará "**MatrizGrafo**" equivalente a la matriz $[M]^1$ vista al explicar el algoritmo de Kaufmann.

La estructura de esta matriz es la siguiente:

Si el número de nodos del grafo es "*n*", será una matriz bidimensional cuadrada de "$n \times n$" elementos.

Si se numeran las filas y columnas desde uno al número de nodos:



$$(i\ ,\ j) = \begin{cases} i \# j & \text{Si existe un arco que va desde el nodo “}i\text{” hacia el nodo “}j\text{”.} \\ \text{“”} & \text{En los demás casos.} \end{cases}$$

Es una matriz bidimensional porque no puede haber más de un arco con la misma dirección entre dos nodos.

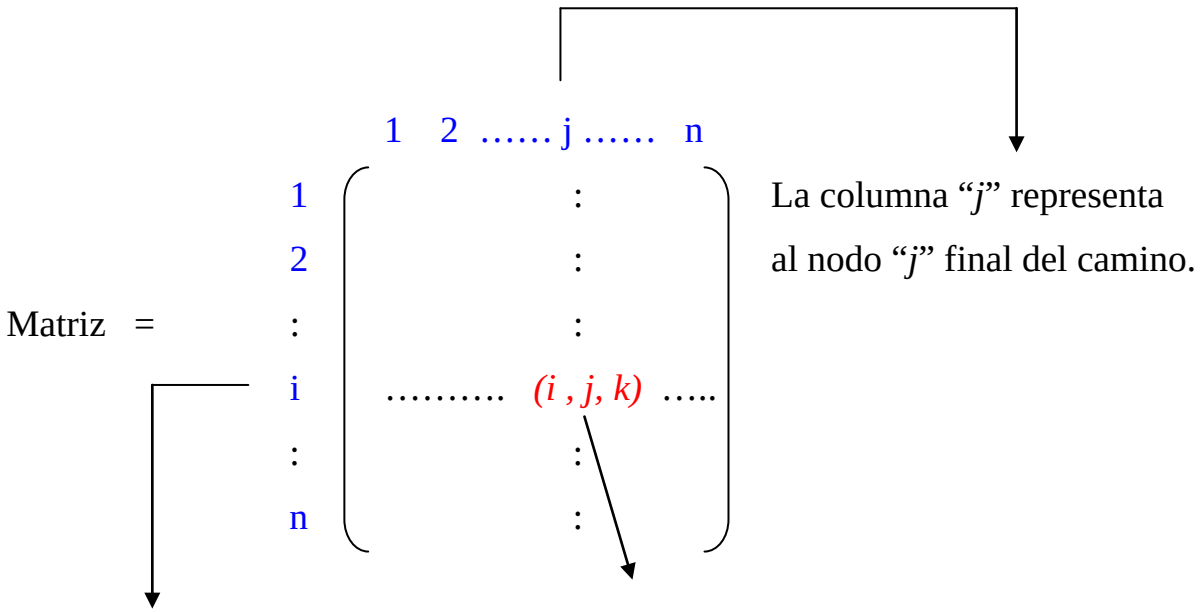
Aunque en la interfaz de usuario los nodos del grafo aparecen con letras, en las matrices se representarán, para trabajar mejor, mediante números enteros a partir del uno.

Para poder distinguir un nodo de otro en una cadena de caracteres, se utilizará un separador dentro de la cadena (se usará el separador “#”).

Los caminos hamiltonianos y los resultados intermedios (caminos elementales de orden menor que " $n - 1$ ") se almacenarán en la matriz de caracteres de tres dimensiones denominada "**Matriz**", equivalente a las matrices M^2, M^3, … ,M^{n-1} vistas al explicar el algoritmo de Kaufmann.

La estructura de esta matriz es la siguiente:

Es una matriz de "$n \times n \times m$" elementos. Si se numeran las filas y columnas de las dos primeras dimensiones desde uno al número de nodos:



$$(i\ ,\ j,\ k) = \begin{cases} i\ \#\ k_1\ \#\ k_2\ \ldots\ \#\ j & \text{Si existe un camino elemental que va desde el nodo "}i\text{" hasta el nodo "}j\text{" pasando por los nodos } k_1, k_2, \ldots, \text{ para todo conjunto de nodos } i, j, k_1, k_2, \ldots \text{ del grafo.} \\ \text{""} & \text{En los demás casos.} \end{cases}$$

Cuando "*Matriz*" es de orden " $n - 1$ ", sus elementos son los caminos hamiltonianos del grafo.

Las dos primeras dimensiones tienen un rango de valores desde *1* al número de nodos del grafo *"n"*.

Como puede haber más de un camino elemental entre dos nodos del grafo, se necesita una tercera dimensión para que cada par de nodos tenga almacenados todos los caminos elementales en esta matriz. El rango de valores que puede tomar la tercera dimensión dependerá del número máximo de caminos elementales que pueda haber entre cada par de nodos en cualquier momento del desarrollo del algoritmo de Kaufmann, es decir, el margen inferior del rango será "" y el margen superior es desconocido a priori. Esto no resulta un problema a la hora de programarlo en un lenguaje de programación concreto porque se asigna espacio a esta tercera dimensión dinámicamente.

La explicación del por qué se pone el separador # entre los nodos pertenecientes a un elemento de las matrices que se están procesando es la siguiente:

Aunque sí se conoce el nodo de origen y el nodo final porque lo indica la fila y la columna de la matriz, no se puede diferenciar entre el resto de nodos del camino elemental, por ejemplo, en la cadena "127453" si los nodos iniciales y final son el "1" y el "3" respectivamente, los nodos intermedios podrían ser 27, 4, 5 o bien 2, 7, 45. Este problema no existe en "MatrizGrafo" porque los elementos de ésta que no valen "" tienen exactamente dos nodos (el nodo origen y el nodo final indicados ambos por la fila y la columna de la matriz respectivamente) pero también se ha puesto el separador # en esta matriz para simplificar la codificación del algoritmo.

La matriz "**MatrizResultado**" servirá como matriz intermedia durante el desarrollo del algoritmo de Kaufmann en el cálculo de los caminos hamiltonianos para almacenar los resultados de forma temporal y también para obtener al final del proceso los circuitos hamiltonianos (finalmente, es equivalente a la matriz $[M]^n$ vista al explicar el algoritmo de Kaufmann).

La estructura, tipo de elementos, dimensiones, etc. de esta matriz es idéntica a "*Matriz*".

Al final del proceso, contendrá la siguiente información:

- Si MatrizResultado $(1, 1, 1)$ = “”

{ No existen circuitos hamiltonianos.

- En caso contrario:

{ Elementos de MatrizResultado $(1, 1, k_{st})$ { $1 \# k_{12} \# k_{13} \# \ldots k_{1n} \# 1$; $1 \# k_{22} \# k_{23} \# \ldots k_{2n} \# 1$; :; $1 \# k_{p2} \# k_{p3} \# \ldots k_{pn} \# 1$

y contendrán todos los circuitos hamiltonianos del grafo.

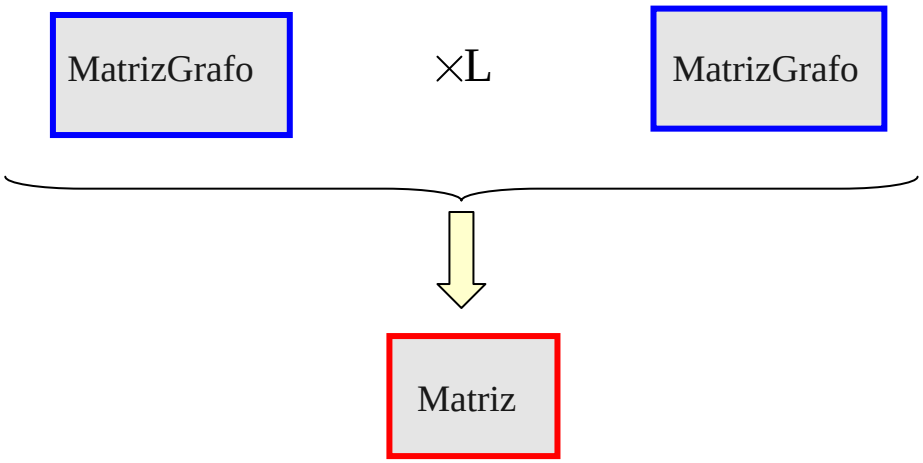
Siendo “p” el número de circuitos hamiltonianos del grafo,
“s” tomará los valores $1, 2, \ldots, p$.
“t” tomará los valores $2, 3, \ldots, n$.

Y el resto de elementos de *MatrizResultado* es “”.

Estas tres matrices se utilizarán en la ejecución del algoritmo de Kaufmann de la siguiente manera:

1.- A partir del grafo creado por el usuario se crea *MatrizGrafo*.

2.- Después, por multiplicación latina de *MatrizGrafo* por sí misma se calcula $[M]^2$ y se almacena el resultado en *Matriz*:

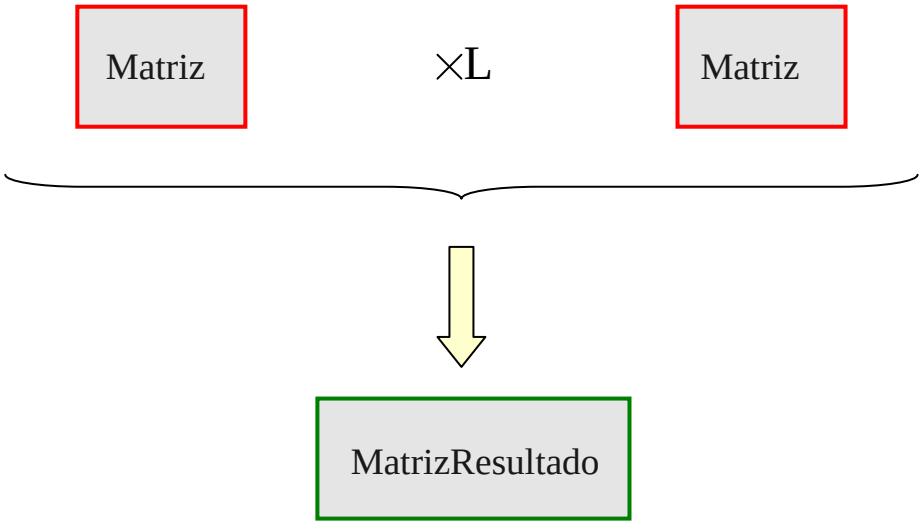


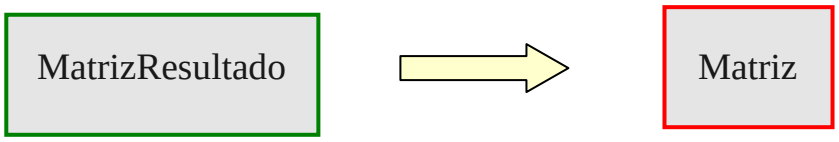
Al índice de "*Matriz*" se le llamará con la palabra "*índice*", en este momento, "*Matriz*" tiene índice 2.

3.- Si *índice* = *n* - *1* , Matriz contiene los caminos hamiltonianos y se salta al paso 4.

Si *índice* < *n* - *1* , se continúa calculando los caminos hamiltonianos:

- Si *índice* + *índice* < *n*



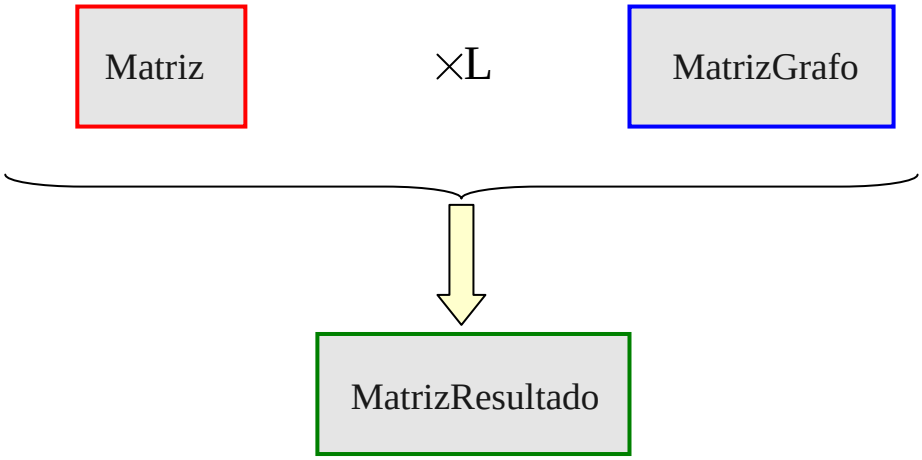


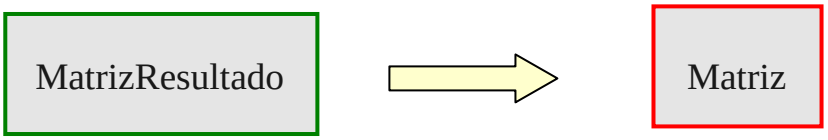
Y después, se inicializan todos los elementos de *MatrizResultado* a “”.

Ahora, el índice de *Matriz* se ha incrementado en *“índice + índice”*.

Se vuelve al paso 3.

- Si *índice + índice >= n*





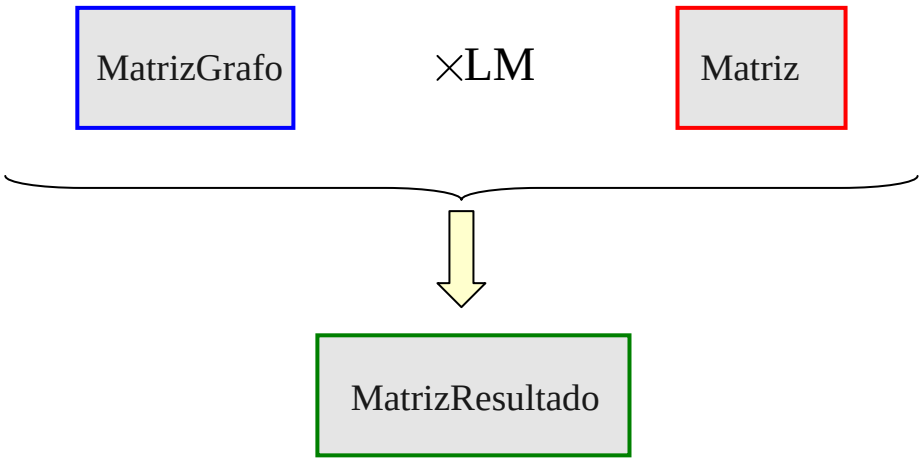
Y después, se inicializan todos los elementos de *MatrizResultado* a “”.

Ahora, el índice de *Matriz* se ha incrementado en uno.

Se vuelve al paso 3.

4.- En *Matriz* se tiene los caminos hamiltonianos y por último se calculan los circuitos hamiltonianos por multiplicación latina, con una modificación, sólo entre

la primera fila de *MatrizGrafo* y la primera columna de *Matriz*, dejando los resultados en *MatrizResultado* $(1, 1, k_{st})$:



En este punto, quedan los caminos hamiltonianos en "*Matriz*" y los circuitos hamiltonianos en *MatrizResultado* $(1, 1, k_{st})$.

1.4.2- Desarrollo del algoritmo del proyecto en pseudocódigo

El pseudocódigo está pensado para una programación estructurada y no para una programación orientada a objetos por una mayor sencillez y simplicidad (con una programación orientada a objetos se complicaría mucho el seguimiento del programa). La interface gráfica no está incluida en el pseudocódigo por su gran envergadura ya que se crearía demasiado pseudocódigo y sería imposible hacer un seguimiento de lo que realmente nos interesa que es la resolución de los algoritmos que calculan los caminos y circuitos hamiltonianos.

Se trata de explicar con claridad e implementar en pseudocódigo todos los pasos de este proyecto, con lo cual los soportes de información, la forma de estructurar el programa, los algoritmos empleados y todas las ideas sobre la posterior realización en código real son independientes del lenguaje o lenguajes de programación que se utilicen después. De esta forma, nos aseguramos de que este proyecto no muera en el tiempo cuando desaparezcan o, simplemente, se modernicen los lenguajes de programación que lo implementan.

Las sentencias que se van a utilizar en pseudocódigo (además de las sentencias triviales, como las de asignación, etc.) serán explicadas a continuación:

Instrucción condicional simple

Si <condición> , entonces

<sentencias>

Finsi

---

condición ⟶ Variable o expresión booleana, o bien, la comparación de dos expresiones, dando siempre un resultado booleano de "*True*" o "*False*".

Si la condición es "*True*", se ejecutan las "sentencias" y si es "*False*" esta instrucción no hace nada.

Instrucción condicional compuesta

Si <condición> , entonces

<sentencias1>

Sino

<sentencias2>

Finsi

---

condición ⟶ Tiene el mismo significado que en la instrucción anterior.

Si la "condición" es "*True*", se ejecutan las "sentencias1", y si es "*False*" se ejecutan las "sentencias2".

Instrucción bucle “mientras”

Mientras <condición> , hacer

<sentencias>

Finmientras

condición ⟶ Tiene el mismo significado que en las instrucciones anteriores.

1º.- Se comprueba si la “condición” es “*True*” o “*False*”.

2º.- Si la “condición” es “*True*” se ejecutan las “sentencias” y se vuelve al paso 1º, y si es “*False*” se termina la instrucción.

Instrucción bucle “hasta”

Repetir

<sentencias>

Hasta <condición>

condición ⟶ Tiene el mismo significado que en las instrucciones anteriores.

1º.- Se ejecutan las “sentencias”.

2º.- Si la condición es “*True*” se vuelve al paso 1º, y si es “*False*” se termina la instrucción.

Instrucción bucle acotado

Para variable = inicio hasta fin , hacer

<sentencias>

Finpara

---

inicio
fin
⟹ Son valores de tipo entero.

Se ejecutan las “sentencias” un número de veces igual a la resta entre los valores “fin” menos “inicio”.

Esta instrucción puede tomar otra forma:

Para variable = inicio hasta fin , salto valor , hacer

<sentencias>

Finpara

valor ⟶ Número entero.

Las sentencias se ejecutan de la misma forma pero el salto que antes era de una unidad ahora lo da “valor”. Por ejemplo:

Para h = 5 hasta 11 , salto 2 , hacer

<sentencias>

Finpara

1º.- Se ejecutan las sentencias (h = 5).

2º.- Se ejecutan las sentencias (h = 7).

3º.- Se ejecutan las sentencias (h = 9).

2º.- Se ejecutan las sentencias (h = 11).

Se acaba la instrucción.

Es decir, si no se pone el salto, las sentencias se ejecutarían 7 veces (para h = 5, 6, 7, 8, 9, 10, 11) y con el salto se ejecutarían 4 veces.

Instrucción comparativa

Seleccionar opción <referencia>

opción1: <comparación1>

<sentencias1>

opción2: <comparación2>

<sentencias2>

:

:

:

opciónk: <comparaciónk>

<sentenciask>

else:

<sentencias else>

Finseleccionar

---

referencia ⟶ Es una variable o expresión.

comparación ⟶ Variable o expresión que se compara con “referencia” dando un resultado booleano de “*True*” o “*False*”.

sentencias ⟶ Si la comparación “*i*” es “*True*”, se ejecutan sus “sentenciasi” asociadas y se termina la instrucción.

else ⟶ Si todas las comparaciones son “*False*” se ejecutan las “sentencias else” y se termina la instrucción.

Las comparaciones se hacen en orden descendente, es decir, <comparación1> <comparación2> … <comparaciónk>, lo que significa que si hay varias comparaciones coincidentes con <referencia> se ejecutarán las sentencias correspondientes a la primera comparación que coincida.

Llamada a un procedimiento

Procedimiento <nombre del procedimiento>

Esta instrucción da un salto en la secuencia de ejecución del programa y continúa la ejecución al comienzo del procedimiento indicado por su nombre. Al finalizar las instrucciones del procedimiento vuelve la ejecución del programa a la instrucción siguiente a “Procedimiento <nombre del procedimiento>”.

Definición de variables

Con objeto de aportar más información y acercarse más a la programación final en un lenguaje de programación, se definen algunas variables del programa en pseudocódigo. Estas definiciones seguirán un modelo sencillo, el modelo de declaración de variables de Visual Basic porque seguir definiendo en castellano también las declaraciones de variables puede resultar excesivo. No se va a aludir el ámbito de las variables ni se van a declarar todas para no complicar más el pseudocódigo. La idea es precisamente hacerlo más comprensible y más cercano al código real que se utilizará después.

```
Dim h As Short
```

La variable “h” se define a nivel local (Dim) como (As) una variable entera corta (Short).

```
Dim caracter As Char
```

La variable “caracter” se define a nivel local como una variable de caracteres.

```
Dim nodo1 As String
```

La variable “nodo1” se define a nivel local como una variable de cadena de caracteres.

Operadores

& ⟶ Concatena o une dos cadenas de caracteres.

<> ⟶ Distinto.

Funciones

Len (cadena)

cadena ⟶ Cualquier expresión de cadena válida.

Devuelve el número de caracteres en una cadena.

Mid (cadena de origen, posición inicial dentro de la cadena de origen, nº de caracteres a extraer)

La función "Mid" obtiene una cadena de caracteres de otra a partir de una posición determinada, especificando el número de caracteres a obtener.

Asc (caracter)

carácter ⟶ Cualquier carácter incluido en el código de caracteres ANSI.

Pasa un carácter a su valor numérico correspondiente ANSI.

Comentarios

Para explicar las instrucciones en pseudocódigo y poder hacer un mejor seguimiento del programa se emplearán comentarios.

Un comentario se distingue por empezar con el carácter “ ’ ” seguido de un texto en color verde.

Una vez explicada la sintaxis del pseudocódigo, se pasará a desarrollarla para realizar el proyecto.

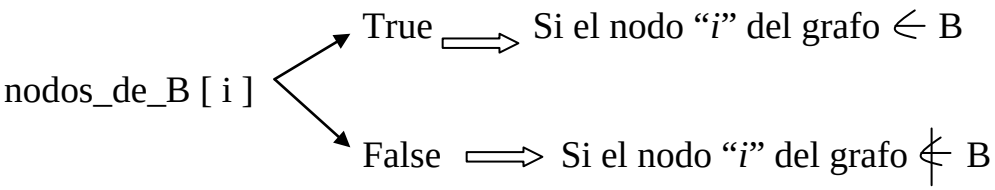
Comprobar que el grafo es conexo

Después de finalizar el diseño completo del grafo, se pasaría a realizar los algoritmos deseados. Es al pasar del grafo a los algoritmos cuando se debe comprobar que el grafo creado es conexo, ya que no se puede calcular caminos o circuitos hamiltonianos con un grafo no conexo. En este punto se ejecutará en el lenguaje de programación elegido (en este caso Java) equivalente al pseudocódigo que se presenta en esta documentación.

La idea es la siguiente:

Inicialmente, se crea una componente conexa *B* de un solo nodo (cualquier nodo del grafo), por ejemplo, el nodo 1. Se van añadiendo nodos del grafo a *B* con la condición de que *B* siga siendo una componente conexa y si al final todos los nodos del grafo están en *B*, el grafo es conexo. Si no se puede conseguir esto porque hay algún nodo o nodos aislados del resto de nodos del grafo, el grafo no es conexo.

Se crea la matriz booleana de una dimensión "*nodos_de_B*" con un número de elementos igual al número de nodos del grafo. Cada elemento "*i*" de esta matriz tomará los valores:



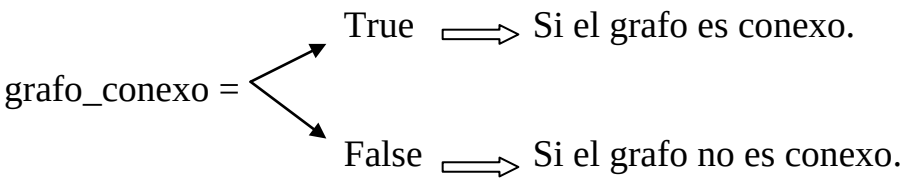
La variable "*cont*" indica, en todo momento, el número de nodos del grafo que están en *B*. Lógicamente, en el momento en que "*cont* = *n*" siendo "*n*" el número de nodos del grafo, el grafo es conexo.

Por simplificación, se hará para un grafo con aristas.

La implementación en pseudocódigo se muestra en la función "*grafo_conexo*".

Función **“grafo conexo”**

Función booleana que devuelve los valores:



El pseudocódigo será:

(se inicializa el conjunto B con el nodo 1)

nodos_de_B [1] ⟵ True

para i = 2 hasta n , hacer

nodos_de_B [i] ⟵ False

Finpara

grafo_conexo ⟵ True

(como el conjunto B tiene un nodo, inicializamos
el número de nodos de B a 1 en la variable “cont”)

cont ⟵ 1

```
para  k = 1  hasta  n-1  ,  hacer
    si  grafo_conexo  ,  entonces
        grafo_conexo  ←  False
        para  i = 1  hasta  n  ,  hacer
            si  nodos_de_B [ i ]  ,  entonces
                para  j = 1  hasta  n  ,  hacer
                    si  ( no ( nodos_de_B [ j ] ) )  ,  entonces
                        si  valor_arista [ i , j ]  ≥  0  ,  entonces
                            (existe una arista entre un nodo i ∈ B
                                y un nodo  j  ∉ B)
                            (añadimos el nodo "j" a B y sumamos 1
                                al número de nodos de B)
                            nodos_de_B [ j ]  ←  True
                            cont  ←  cont + 1
                            (como se ha añadido un nodo más a B, sigue habiendo
                                posibilidades de que el grafo sea conexo)
                            grafo_conexo  ←  True
                            si  cont = n  ,  entonces
                                "El grafo es conexo" ; "Fin de la función"
                            finsi
                        finsi
                    finsi
                finsi
            finsi
        finsi
    si no
        "Grafo no conexo" ; "Fin de la función"
    Finsi
finpara
```

Desarrollo del programa principal

Una vez que se ha comprobado que el grafo es conexo, primero se describen ordenadamente los bloques en los que se divide el programa y después se desarrolla cada uno de ellos:

1.- Se carga la matriz inicial "*MatrizGrafo*" a partir del grafo.

2.- Se obtiene la matriz de orden 2 por multiplicación latina de "*MatrizGrafo*" por sí misma. El resultado se guarda en "*Matriz*".

3.- Se calculan los caminos hamiltonianos, si existen, y se guardan en "*Matriz*".

4.- Se calculan los circuitos hamiltonianos y, si existen, se guardan en los elementos *(0, 0, h)* de "*MatrizResultado*", para cada valor de "*h*" tenemos un circuito hamiltoniano.

5.- Se muestra la salida de resultados.

El primer y último bloques dependen de cómo se implemente la interfaz gráfica con lo que no afecta al desarrollo del algoritmo.

Se denominará *"n"* al número de nodos del grafo.

Para una matriz con una dimensión de rango *"n"*, sus elementos estarán entre 0 y *"n – 1"* para facilitar la posterior programación.

Se llamará "m" al máximo valor que puede tomar la tercera dimensión de una matriz en cualquier momento del proceso para cualquiera de sus elementos. Aunque no se sabe su valor hasta el final del programa, esto no será un problema porque se programará de forma dinámica, es decir, para explicar el funcionamiento del proceso de forma genérica y estructurada es mejor suponer que sabemos el valor de "*m*" a priori.

Obtención de la matriz de orden 2 por multiplicación latina de "MatrizGrafo" por sí misma

```
'Inicializo la matriz que me dará los caminos hamiltonianos.
Para i = 0 hasta n – 1  hacer
    Para j = 0 hasta n – 1  hacer
        Para k = 0 hasta m – 1  hacer
            Matriz (i, j, k) ⟵  ""
        Finpara k
    Finpara j
Finpara i
'--------------------------------
Para i = 0 hasta n – 1  hacer
    Para j = 0 hasta n – 1  hacer
        Para k = 0 hasta n – 1  hacer
            'Sólo cuando coinciden 2 valores distintos
            'de  ""  hago la multiplicación latina.
            Procedimiento multiplicacion_latina_M2
        Finpara k
    Finpara j
Finpara i
```

Procedimiento “multiplicacion_latina_M2”

Multiplicación latina de “*MatrizGrafo*” por sí misma para calcular la matriz $[M]^2$.

```
Dim h As Short

'Pongo en la variable "ultimo" el último nodo del elemento de la primera matriz.
Procedimiento   capturar_ultimo_matriz_grafo()

'Pongo en la variable "primero" el primer nodo del elemento de la segunda matriz.
Procedimiento   capturar_primero_matriz_grafo()

Si (ultimo = primero) entonces

    'Como sólo hay 2 nodos en cada cadena, Compruebo si el primer nodo de la 1ª
    'cadena es distinto al segundo nodo de la segunda cadena. El resultado se
    'guarda en la variable booleana "distintos". También calculo la variable
    "nodo_primera" que contiene la primera cadena excepto el último nodo.
    Procedimiento   resto_no_coincide_M2()

    Si distintos entonces

        'Calculo el elemento resultante.
        elemento  ⟵  nodo_primera & "#" & Matriz_grafo (k, j)

        'Introduzco el nuevo elemento en la posición h adecuada de la tercera
        'dimensión de "Matriz" que guardará los caminos hamiltonianos al final
        'del proceso, y que de momento representará a la matriz de orden 2 "M2".
        Para h = 0 hasta m – 1  hacer

            Si   Matriz (i, j, h) = "" entonces

                 Matriz (i, j, h)  ⟵  elemento

                 Exit Para

            Finsi

        Finpara

    Finsi

Finsi
```

Procedimiento “capturar_ultimo_matriz_grafo()”

```
'Obtengo el último nodo de la primera cadena.

Dim v As Short
Dim t As Short
Dim caracter As Char

Longitud1 ⟵ Len (Matriz_grafo (i, k))

Para  t = 1  hasta  Longitud1 , hacer
'La variable "caracter" captura caracter a caracter desde la primera
'posición de la cadena hasta encontrarme el caracter de separeción "#".

    caracter ⟵ Mid (Matriz_grafo (i, k), t, 1)

    Si  (Asc (caracter) = 35)  entonces
        'Cuando encuentro el caracter "#", en la variable "último"
        'guardo el nodo que hay desde "#" hasta el final de la cadena.

        ultimo ⟵ ""

        Para  v = t + 1  hasta  Longitud1 , hacer

            carácter ⟵ Mid(Matriz_grafo(i, k), v, 1)

            ultimo ⟵ ultimo & caracter

        Finpara v

        Exit Para

    Finsi

Finpara t
```

Procedimiento “capturar_primero_matriz_grafo()”

```
'Obtengo el primer nodo de la segunda cadena.

Dim t As Short
Dim caracter As Char

Longitud2  ←  Len (Matriz_grafo (k, j))

primero  ←  ""

Para  t = 1  hasta  Longitud2  hacer
'La variable "caracter" captura caracter a caracter
'desde el principio hasta encontrarme el caracter de separeción "#".

    caracter  ←  Mid (Matriz_grafo (k, j), t, 1)

    Si  (Asc (caracter) <> 35)  entonces
        'Mientras no encontremos el caracter # vamos
        'cargando el nodo en la variable "primero".

        primero  ←  primero & caracter

    Sino

        Exit Para

    Finsi

Finpara t
```

Procedimiento “resto_no_coincide_M2()”

```
'Compruebo si el primer nodo de la primera cadena
'es distinto al segundo nodo de la segunda cadena.

Dim t As Short
Dim caracter As Char
Dim v As Short

nodo_primera ⟵ ""

Para  t = 1  hasta  Longitud1  hacer
'La variable "caracter" captura caracter a caracter
'desde el primero hasta encontrarme el caracter de separeción "#".

    caracter ⟵ Mid (Matriz_grafo (i, k), t, 1)

    Si  (Asc (caracter) <> 35)  entonces

        nodo_primera ⟵ nodo_primera & caracter

    Sino

        Exit For

    Finsi

Finpara t

'---------------------------

Para  t = 1  hasta  Longitud2  hacer
'La variable "caracter" captura caracter a caracter
'desde el principio hasta encontrarme el caracter de separeción "#".

    caracter ⟵ Mid (Matriz_grafo (k, j), t, 1)

    Si (Asc (caracter) = 35) entonces
    'Cuando encuentro el caracter "#", en la variable "nodo_segunda"
    'guardo el nodo que hay desde "#" hasta el final de la cadena.

        nodo_segunda ⟵ ""
```

```
        Para   v = t + 1   hasta   Longitud2  ,  hacer
            caracter  ←  Mid(Matriz_grafo(k, j), v, 1)
            nodo_segunda  ←  nodo_segunda & caracter
        Finpara v
        Exit Para
    Finsi
Finpara t
'----------------------------
'Compruebo si se puede crear un nuevo elemento o no.
Si   nodo_primera = nodo_segunda   entonces
        distintos  ←  False
Sino
        distintos  ←  True
Finsi
```

Cálculo de los caminos hamiltonianos

```
'Calculo los caminos hamiltonianos y los guardo en la matriz de tres dimensiones
"Matriz".

'Ya tenemos calculada la matriz M2 de índice 2.
Indice ← 2

'El número de nodos del grafo es n. Tenemos que calcular la matriz de índice n - 1.
'El último cálculo será cuando Indice = n - 2, y se calculará la matriz de orden n - 1.
Mientras  Indice < n – 1  hacer

    'Inicializo la matriz temporal de resultados.
    Procedimiento inicializar_matriz_resultados()

    Si  (Indice + Indice) < n  entonces

        'Multiplicación latina de "Matriz" por sí misma.
        Procedimiento por_si_misma()

        Indice ← Indice + Indice

    Sino

        'Multiplicación latina entre "Matriz" y "Matriz_grafo".
        Procedimiento  por_Matriz_grafo()

        Indice ← Indice + 1

    Finsi

    '"Matriz" toma los valores de la matriz en donde
    'he guardado los resultados "Matriz_resultado".
    Procedimiento  Actualizo_Matriz()

Finmientras

'Pongo el valor de cadena vacía "" a todos los elementos de
'"Matriz_resultado" para que en la próxima iteracción pueda guardar
'los resultados de los caminos elementales de orden "Indice".
Procedimiento  inicializar_matriz_resultados()
```

Procedimiento “inicializar_matriz_resultados”

```
'Pongo el valor de cadena vacía "" a todos los elementos de la matriz
'"Matriz_resultado".

Dim a As Short
Dim b As Short
Dim c As Short

Para a = 0 hasta n – 1 hacer
    Para b = 0 hasta n – 1 hacer
        Para c = 0 hasta m – 1 hacer
            Matriz_resultado (a, b, c) ←  ""
        Finpara c
    Finpara b
Finpara a
```

Procedimiento “por_si_misma”

```
'Multiplicación latina de "Matriz" por sí misma y dejo el resultado en
'"Matriz_resultado" que tendrá los caminos elementales de orden "Indice".

p ⟵ 0
q ⟵ 0

Para  i = 0  hasta n – 1  hacer
    Para  j = 0  hasta  n – 1  hacer
        Para  k = 0  hasta n – 1  hacer
            Mientras  (Matriz (i, k, p) <> "")  And  (Matriz (k, j, q) <> "")  ,  hacer
            'Cuando coinciden 2 valores distintos de "" hago la multiplicación latina.
                Procedimiento  multip_latina_por_si_misma()

                'Con la misma cadena de la primera matriz,
                'Busco la siguiente cadena de la segunda matriz.
                q ⟵ q + 1
                Si  (Matriz  (k, j, q) = "")  entonces
                'Cuando se terminan las cadenas de la tercera dimensión en la
                 'segunda matriz inicializo el índice y sumo 1 al índice de la primera
                 'matriz.
                    q ⟵ 0

                    'Paso a la siguiente cadena de la primera matriz.
                    p ⟵ p + 1
                Finsi
            Finmientras
            p ⟵ 0
        Finpara k
    Finpara j
Finpara i
```

Procedimiento “multip_latina_por_si_misma()”

```
Dim   h   As Short

'Capturo el último nodo de la cadena de la primera matriz.
'Guardo el resultado en la variable "ultimo".
Procedimiento   ultimo_nodo()

'Capturo el primer nodo de la cadena de la segunda matriz.
'Guardo el resultado en la variable "primero".
Procedimiento   primer_nodo_Matriz()

Si (ultimo = primero) entonces

    'Tengo que comprobar que los nodos de la 1ª cadena son todos distintos
    'a los nodos de la 2ª cadena excepto el primero de la segunda cadena.
    'El resultado se guarda en la variable booleana "distintos".
    Procedimiento   Resto_no_coincide_Caminos()

    'La variable booleana "distintos" nos dice si creamos una nueva cadena más
    larga.
  Si   distintos   entonces

      'Calculo la cadena resultante y la guardo en la variable "elemento".
      Procedimiento   calcular_elemento()

      'Introduzco el nuevo elemento en la posición h adecuada de la
      terceradimensión de la matriz "Matriz_resultado" que guardará los
      caminoselementales de orden "Indice".
    Para   h = 0   To   m – 1   hacer

      Si   Matriz_resultado (i, j, h) = ""   entonces

            Matriz_resultado(i, j, h) ⟵ elemento

            Exit For

      Finsi

    Finpara h

  Finsi

Finsi
```

Procedimiento “ultimo_nodo()”

```
'Capturo el último nodo de la cadena de la primera matriz.

Dim  t  As Short
Dim  v  As Short
Dim caracter As Char

Longitud1  ←  Len (Matriz (i, k, q))

Para  t = Longitud1  hasta  1  Step -1  hacer

    caracter  ←  Mid (Matriz (i, k, q), t, 1)

    Si  (Asc (caracter) = 35)  entonces

        ultimo  ←  ""

        Para  v = t + 1  hasta  Longitud1  ,  hacer

            caracter  ←  Mid (Matriz (i, k, q), v, 1)

            ultimo  ←  ultimo & caracter

        Finpara v

        Exit Para

    Finsi

Finpara t
```

Procedimiento “primer_nodo_Matriz()”

```
'Capturo el primer nodo de la cadena de la segunda matriz.

Dim  t  As  Short
Dim  caracter  As Char

primero  ←  ""

Longitud2  ←  Len (Matriz (k, j, q))

Para  t = 1  hasta  Longitud2  hacer

      caracter  ←  Mid (Matriz (k, j, q), t, 1)

   Si  Asc (caracter) = 35  entonces

         Exit Para

   Sino

         primero  ←  primero  &  caracter

   Finsi

Finpara t
```

```
'Capturo el primer nodo de la cadena de la segunda matriz.

Dim  t  As  Short
Dim  caracter  As  Char

primero  ←  ""

Longitud2  ←  Len (Matriz (k, j, q))

Para  t = 1  hasta  Longitud2  hacer

      caracter  ←  Mid (Matriz (k, j, q), t, 1)

      Si  Asc (caracter) = 35  entonces

            Exit Para
```

Sino

primero ← primero & caracter

Finsi

Finpara t

Procedimiento “Resto_no_coincide_Caminos()”

```
'Compruebo que los nodos de la 1ª cadena son todos distintos
'a los nodos de la 2ª cadena excepto el primer nodo de la segunda cadena.
'Si "distintos" es verdadero, se puede crear una cadena más grande.

Dim  f  As  Short
Dim  g  As  Short
Dim  caracter  As  Char
Dim  nodo1  As  String
Dim  nodo2  As  String

distintos ←— True

nodo1 ←— ""

Longitud1 ←— Len (Matriz (i, k, p))

Para  f = 1  hasta  Longitud1 ,  hacer

    caracter ←— Mid (Matriz (i, k, p), f, 1)

    '"nodo1" se irá cargando con todos los nodos de Matriz(i, k, p) excepto el último.
    Si  (Asc (caracter) = 35)  entonces
            'Ya tengo completo el nodo1.

            nodo2 ←— ""

            Longitud2 ←— Len (Matriz (k, j, q))

            'Comparo con todos los nodos de Matriz(k, j, q).
            Para  g = 1  hasta  Longitud2  hacer

               caracter ←— Mid (Matriz (k, j, q), g, 1)

               Si  (Asc (caracter) = 35)  Or  (g = Longitud2)  entonces
                'Ya tengo completo el nodo2 y puedo comparar.

                Si  g = Longitud2  entonces

                      nodo2 ←— nodo2 & caracter

                Finsi
```

```
            Si  nodo1 = nodo2  entonces
            'No se puede hacer la multiplicación latina.

                distintos ⟵ False

                Exit Sub

            Sino

                'Iniciamos otra comparación con otro "nodo2".
                nodo2 ⟵ ""

            Finsi

        Sino

            nodo2 ⟵ nodo2 & caracter

        Finsi

    Finpara g

    nodo1 ⟵ ""

Sino

    nodo1 ⟵ nodo1 & caracter

Finsi

Finpara f
```

Procedimiento “calcular_elemento()”

```
Dim  w  As  Short
Dim  caracter  As  Char
Dim  menos_ultimo  As  String
Dim  aux  As  String

aux ⟵ ""

menos_ultimo ⟵ ""

'En la variable "menos_ultimo" obtendré todos los nodos
'del elemento de Matriz(i, k, p) excepto el último.

Para  w = 1  hasta  Longitud1 ,  hacer

    caracter ⟵ Mid (Matriz (i, k, p), w, 1)

    aux ⟵ aux  &  caracter

  Si  Asc (caracter) = 35  entonces

      menos_ultimo ⟵ menos_ultimo  &  aux

      aux ⟵ ""

  Finsi

Finpara w

elemento ⟵ menos_ultimo  &  Matriz (k, j, q)
```

Procedimiento “por_Matriz_grafo()”

```
'Calculo la matriz de orden "Indice" en donde "Indice" se ha sumado en una unidad.
'El resultado se guarda temporalmente en la matriz "Matriz_resultado".

p ⟵ 0

Para  i = 0  hasta  n – 1  ,  hacer

    Para  j = 0  hasta  n – 1  ,  hacer

        Para  k = 0  hasta  n – 1  ,  hacer

            Mientras  (Matriz (i, k, p) <> "")  And  (Matriz_grafo (k, j) <> "") ,hacer

                'Cuando coinciden 2 valores distintos de "" hago la multiplicación
                latina.
                Procedimiento  multip_latina_por_Matriz_grafo()

                'Paso a la siguiente cadena de "Matriz".
                p ⟵ p + 1

            Finmientras

            p ⟵ 0

        Finpara k

    Finpara j

Finpara i
```

Procedimiento “multip_latina_por_Matriz_grafo()”

```
Dim  h  As  Short

'Capturo el último nodo de la cadena de la primera matriz (Matriz).
'Guardo el resultado en la variable "último".
Procedimiento  ultimo_nodo()

'Capturo el primer nodo de la cadena de la segunda matriz (Matriz_grafo).
'Guardo el resultado en la variable "primero".
Procedimiento  capturar_primero_matriz_grafo()

Si  (ultimo = primero)  entonces

    'Tengo que comprobar que los caracteres de la 1ª cadena son todos distintos
    'a los caracteres de la 2ª cadena excepto el primero de la segunda cadena.
    'El resultado se guarda en la variable booleana "distintos".
    Procedimiento  Resto_elemento_MG_no_coincide()

    Si  distintos  entonces

        'Calculo la cadena resultante.
        Procedimiento  calcular_elemento_MG()

        'Introduzco el nuevo elemento en la posición h adecuada de la tercera
        'dimensión de la matriz "Matriz_resultado" que guardará los caminos
        'elementales de orden "Indice".
        Para  h = 0  hasta  m – 1  ,  hacer

            Si  Matriz_resultado (i, j, h) = ""  entonces

                Matriz_resultado (i, j, h)  ←  elemento

                Exit Para

            Finsi

        Finpara

    Finsi

Finsi
```

Procedimiento “Resto_elemento_MG_no_coincide()”

```
'Compruebo que los caracteres de la 1ª cadena son todos distintos
'a los caracteres de la 2ª cadena excepto el primer caracter de la segunda cadena.
'Si "distintos" es verdadero, se puede hacer la multiplicación latina.

Dim  f  As  Short
Dim  g  As  Short
Dim  caracter  As  Char
Dim  nodo1  As  String
Dim  nodo2  As  String

distintos  ←  True

nodo1  ←  ""

Longitud1  ←  Len (Matriz (i, k, p))

Para  f = 1  hasta  Longitud1  ,  hacer

    caracter  ←  Mid (Matriz (i, k, p), f, 1)

    '"nodo1" se irá cargando con todos los nodos de Matriz(i, k, p) excepto el último.
    Si  (Asc (caracter) = 35)  entonces
    'Ya tengo completo el nodo1.

        nodo2  ←  ""

        Longitud2  ←  Len (Matriz_grafo (k, j))

        'Comparo con todos los nodos de Matriz(k, j, q).
        Para  g = 1  hasta  Longitud2  ,  hacer

            caracter  ←  Mid (Matriz_grafo (k, j), g, 1)

            Si  (Asc (caracter) = 35)  Or  (g = Longitud2)  entonces
              'Ya tengo completo el nodo2 y puedo comparar.

                Si  g = Longitud2  entonces

                    nodo2  ←  nodo2  &  caracter

                Finsi
```

```
            Si   nodo1 = nodo2   entonces
            'No se puede hacer la multiplicación latina.
                distintos  ←  False
                Exit Sub
            Sino
            'Iniciamos otra comparación con otro "nodo2".
                nodo2  ←  ""
            Finsi
        Sino
            nodo2  ←  nodo2  &  caracter
        Finsi
    Finpara g
    nodo1  ←  ""
Sino
    nodo1  ←  nodo1  &  caracter
Finsi
Finpara f
```

Procedimiento “calcular_elemento_MG()”

```
Dim  w  As  Short
Dim  caracter  As  Char
Dim  menos_ultimo  As  String
Dim  aux  As  String

aux ←—  ""

menos_ultimo ←— ""

'En la variable "menos_ultimo" obtendré todos los nodos
'del elemento de Matriz(i, k, p) excepto el último.

Para  w = 1  hasta  Longitud1  ,  hacer

      caracter ←—  Mid (Matriz (i, k, p), w, 1)

      aux ←— aux  &  caracter

   Si  Asc (caracter) = 35  entonces

         menos_ultimo ←—  menos_ultimo  &  aux

         aux ←—  ""

   Finsi

Finpara w

elemento ←—  menos_ultimo  &  Matriz_grafo (k, j)
```

Procedimiento “Actualizo_Matriz()”

```
'"Matriz" toma los valores de la matriz en donde
'he guardado los resultados "Matriz_resultado".

Dim a As Short
Dim b As Short
Dim c As Short

Para a = 0 hasta n – 1 , hacer
    Para b = 0 hasta n – 1 , hacer
        Para c = 0 hasta m – 1 , hacer
            Matriz (a, b, c) ← Matriz_resultado (a, b, c)
        Finpara c
    Finpara b
Finpara a
```

Procedimiento “inicializar_matriz_resultados()”

```
'Pongo el valor de cadena vacía "" a todos los elementos de la matriz
'"Matriz_resultado".

Dim  a  As  Short
Dim  b  As  Short
Dim  c  As  Short

Para  a = 0  hasta  n – 1 ,  hacer
    Para  b = 0  hasta  n – 1  ,  hacer
        Para  c = 0  hasta  m – 1  ,  hacer
            Matriz_resultado (a, b, c)  ←  ""
        Finpara c
    Finpara b
Finpara a
```

Cálculo de los circuitos hamiltonianos

```
'Basta con hacer la multiplicación latina sólo para la primera fila de la
'matriz "Matriz_grafo" y para la primera columna de la matriz "Matriz".

p ⟵ 0

Para  k = 0  hasta  n – 1  ,  hacer

    Mientras  (Matriz_grafo (0, k) <> "")  And  (Matriz (k, 0, p) <> "")  hacer

        'Cuando coinciden 2 valores distintos de "" hago la multiplicación latina.
        Procedimiento  multip_latina_circuitos()

        'Paso a la siguiente cadena de la segunda matriz.
        p ⟵ p + 1

    Finmientras

    p ⟵ 0

Finpara k
```

Procedimiento “multip_latina_circuitos()”

```
Dim  h  As  Short

'Capturo el último nodo de la cadena de la primera matriz "Matriz_grafo".

i  ←  0

Procedimiento  capturar_ultimo_matriz_grafo()

'Capturo el primer nodo de la cadena de la segunda matriz "Matriz".

j  ←  0

q  ←  p

Procedimiento  primer_nodo_Matriz()

Si  (ultimo = primero)  entonces

    'Tengo que comprobar que los caracteres de la 1ª cadena son todos distintos
    'a los caracteres de la 2ª cadena excepto el primero de la segunda cadena.
    'El resultado se guarda en la variable booleana "distintos".
    Procedimiento  Resto_elemento_Cir_no_coincide()

    Si  distintos  entonces

        'Calculo la nueva cadena más larga en la variable "elemento".
        Procedimiento  calcular_elemento_circuitos()

        'Introduzco el nuevo elemento en la posición h adecuada de la tercera
        'dimensión de la matriz "Matriz_resultado" que guardará los
        'circuitos hamiltonianos en las posiciones (0,0,h).
        Para  h = 0  hasta  m – 1  ,  hacer

            Si  Matriz_resultado (0, 0, h) = ""  entonces

                Matriz_resultado (0, 0, h)  ←  elemento

                Exit Para

            Finsi

        Finpara h
```

```
    Finsi
Finsi
```

Procedimiento “Resto_elemento_Cir_no_coincide”

```
'Compruebo que los caracteres de la 1ª cadena son todos distintos a los caracteres
'de la 2ª cadena excepto el primer caracter de la segunda cadena y además, deben
'coincidir el primer elemento de la primera cadena con el último elemento de la
'segunda cadena. Si "distintos" es verdadero, se puede hacer la multiplicación latina.

Dim  f  As  Short
Dim  g  As  Short
Dim  caracter  As  Char
Dim  nodo1  As  String
Dim  nodo2  As  String

distintos ⟵ True

nodo1 ⟵ ""

Para  f = 1  hasta  Longitud1 ,  hacer

    caracter ⟵ Mid (Matriz_grafo (0, k), f, 1)

    '"nodo1" se irá cargando con todos los nodos de Matriz(i, k, p) excepto el
    'último.
    Si  (Asc (caracter) = 35)  entonces
        'Ya tengo completo el nodo1.

        nodo2 ⟵ ""

        'Comparo con todos los nodos de Matriz(k, 0, p).
        Para  g = 1  hasta  Longitud2 ,  hacer

            caracter ⟵ Mid (Matriz (k, 0, p), g, 1)

            Si  (Asc (caracter) = 35)  entonces
            'Ya tengo completo el nodo2 y puedo comparar.

            ┌ Si  g = Longitud2  entonces

                    nodo2 ⟵ nodo2  &  caracter

            └ Finsi
```

```
            Si   nodo1 = nodo2   entonces
            'No se puede hacer la multiplicación latina.

                distintos  ←  False

                Exit Sub

            Sino

                'Iniciamos otra comparación con otro "nodo2".
                nodo2  ←  ""

            Finsi

        Sino

            nodo2  ←  nodo2  &  caracter

        Finsi

    Finpara g

    nodo1  ←  ""

Sino

    nodo1  ←  nodo1  &  caracter

Finsi

Finpara f
```

Procedimiento “calcular_elemento_circuitos”

```
'Calculo un circuito hamiltoniano y lo guardo en la variable "elemento".

Dim w As Short
Dim caracter As Char
Dim menos_ultimo As String
Dim aux As String

aux ← ""

menos_ultimo ← ""

'En la variable "menos_ultimo" obtendré el primer nodo
'del elemento de Matriz_grafo(0, k, p).

Para w = 1 hasta Longitud1 , hacer
        caracter ← Mid (Matriz_grafo (0, k), w, 1)
        aux ← aux & caracter
    Si Asc (caracter) = 35 entonces
            menos_ultimo ← menos_ultimo & aux
            aux ← ""
    Finsi
Finpara w

elemento ← menos_ultimo & Matriz (k, 0, p)
```

1.5- Interface gráfica

El proyecto tiene una interface gráfica muy completa y elaborada para que al usuario le resulte sencillo utilizarla. Podríamos hacer la siguiente división en cuanto a su funcionalidad:

- Selección del idioma (español o inglés).
- Ayuda elemento a elemento y paso a paso de la utilización del interface.
- Creación y edición de grafos.
- Información, seguimiento y orientación al usuario de forma interactiva de lo que está haciendo y de lo que tiene que hacer.
- Aplicación de los algoritmos mostrando todos los resultados de forma clara (agrupados o desglosados).
- Explicaciones teóricas sobre grafos, cálculo de los caminos y circuitos hamiltonianos.
- Misma visualización en cinco navegadores de Internet distintos, enlaces, etc.

Una interface de usuario está sujeta a cierta subjetividad y es seguro de que en el futuro, con medios más avanzados, se podrán crear con mayor sencillez y con mejores resultados. El desarrollo de esta interface ha requerido mucho tiempo de elaboración porque además de cumplir con sus objetivos expresados en los puntos anteriores debe tener una calidad, presentación y resolución buenas.

Sería muy larga y compleja una explicación exhaustiva de cómo se ha hecho toda la interface, simplemente se expondrá como se trazan los arcos y, a modo de ejemplo y para dar una idea de lo elaborada y cuidada que está esta interface, como redibujar el grafo cuando hay modificaciones en el área de dibujo.

Un dato importante es el número máximo de nodos que permite crear la aplicación, los nodos aparecen etiquetados con un carácter que puede ser una letra mayúscula (A), minúscula (a), mayúscula con una comilla (A’) y minúscula con una

comilla (a'), haciendo un total de 104 nodos como máximo. Aunque se puedan poner todos los nodos sin que se superpongan unos con otros, pero muy juntos entre sí, el problema surge al trazar los arcos, supongamos que desde un nodo salen cuarenta arcos, el caos originado en el grafo no nos permitiría verlo bien incluso si el ordenador se conecta a un televisor. En definitiva, 104 nodos es un tope más que suficiente para la aplicación.

Trazado de arcos:

Un arco está compuesto de tres elementos gráficos: la arista que une los nodos entre los que se tiende (un segmento), la punta de flecha que indica la dirección de la arista convirtiéndola en un arco (un triángulo) y el valor (un número decimal entre cero y un millón).

Dibujado de la arista (segmento del arco) y el valor:

Partiendo de las coordenadas del centro de los dos nodos asociados al arco, siendo estas *n1 (x1, y1)* y *n2 (x2, y2)*, se dibujan cinco líneas para que la línea resultante sea más gruesa y más fácilmente visible, uniendo los puntos siguientes:

L1: { (x1+1, y1), (x2+1, y2) }
L2: { (x1-1, y1), (x2-1, y2) }
L3: { (x1, y1), (x2, y2) }
L4: { (x1, y1-1), (x2, y2-1) }
L5: { (x1, y1+1), (x2, y2+1) }

De estar activa la opción de mostrar los valores de los arcos, se calcula a continuación su posición a partir de los puntos *n1* y *n2*, hallando el punto medio en ambos ejes de coordenadas.

Dibujado de la punta de flecha:

La punta de flecha es un triángulo. Considerando dos de sus vértices como los límites del segmento del mismo que es perpendicular a la arista del arco, al que nos referiremos como base de la punta de flecha, designadas sus coordenadas por *p1(xp1, yp1)* y *p2(xp2, yp2)*, el vértice restante situado en el segmento formado por la arista del arco y designado por el punto *pf(xf, yf)*.

Se define *"d"* como la distancia entre *n1* y *n2* a partir del cálculo $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$, y *D* como *d+r*, siendo *"r"* el radio de los nodos.

Se calcula la dirección de la recta que contendrá la base de la punta de flecha hallando el vector unitario de la misma, designado por *(xu,yu)*, mediante las siguientes equivalencias:

xu = (x2-x1) / D

yu = (y2-y1) / D

A partir de aquí, el cálculo de las coordenadas de *(xf, yf)* se realiza de la siguiente manera:

xf = xu * d + x1

yf = yu * d + y1

Asumiendo *d2 = d - 15*, se calcula las coordenadas del punto de corte de la base de la punta de flecha con la arista del arco, definido por *(xp, yp)*, como:

xp = xu * d2 + x1

yp = yu * d2 + y1

Teniendo este punto, ya se puede calcular las coordenadas de los dos puntos restantes de la punta de flecha, *p1* y *p2*, de esta forma:

xp1 = -yu * 10 + xp

yp1 = xu * 10 + yp

xp2 = yu * 10 + xp

yp2 = -xu * 10 + yp

Por último, se dibuja un polígono de tres lados utilizando los puntos *{pf,p1,p2}*.

Redibujado en el grafo:

Las operaciones de dibujo son complejas y lleva tiempo (refiriéndonos a tiempo de ordenador) cambiar una imagen por otra. Cuando el usuario, por ejemplo, arrastra un nodo con todos sus arcos, el proceso normal sería borrar una imagen completa y sustituirla por otra nueva repitiendo este proceso en espacios de tiempo pequeñísimos según se va modificando el grafo. Es decir, repetir sucesivamente el proceso de borrar el dibujo antiguo y sustituirlo por el nuevo con la modificación de lugar, dividiendo el desplazamiento en instantáneas que a modo de película van mostrándose en el área de dibujo.

Borrar y dibujar todo el área de dibujo (especialmente para grafos grandes) origina parpadeos en la imagen del grafo y aunque casi no se aprecian, pueden resultar molestos cuando estamos trabajando cierto tiempo en su edición. Para evitar esto, creamos las instantáneas y las almacenamos en un objeto imagen en memoria sin borrar la imagen antigua y hasta que no se copia la nueva instantánea permanece el dibujo anterior. Aunque el tiempo empleado por los dos procedimientos es prácticamente el mismo, al evitar el borrado se evita el parpadeo.

Independientemente de esto, otro problema que se plantea también desde el principio es el siguiente: si copiamos todo el área de trabajo completa tardaría en actualizarse la imagen, para reducir al mínimo la cantidad de píxeles a cambiar,

copiamos solamente el área rectangular más pequeña que contenga la parte del grafo que ha cambiado.

El cálculo del área de dicho rectángulo mínimo, depende de los elementos que intervengan en los cambios del dibujo. Por ejemplo, si arrastramos un nodo aislado, es decir, del que no parten arcos y no es destino de ningún arco, el rectángulo será el mínimo que abarque al nodo. Si, por el contrario, arrastramos un nodo del que salen o entran arcos, consideramos el nodo que se mueve, los arcos adyacentes (entrantes y salientes) y sus nodos adyacentes. Sabemos las coordenadas de cada nodo, se calculan los nodos extremos que serán los límites del rectángulo y se hace un barrido desde la esquina superior izquierda del rectángulo hasta su esquina inferior derecha guardando esta imagen en memoria.

Los valores numéricos de las aristas se pueden escapar del área rectangular. Para solucionar esto, se calculan rectángulos pequeños uno por cada valor del arco y después se comparan con el rectángulo principal viendo si sobrepasan sus límites. Si es así, se amplía el rectángulo principal hasta abarcar los valores de los arcos también. Para calcular los rectángulos pequeños hay que tener en cuenta si hay dos arcos entre dos nodos o uno sólo y el número de dígitos del valor de cada arco, además de la fuente empleada.

Por último, si el área de cambios nueva es menor que el área con los mismos elementos del fotograma anterior, el rectángulo calculado será menor que el rectángulo con los mismos elementos del fotograma anterior y en este caso, el rectángulo resultante tiene que abarcar los dos rectángulos, el del fotograma anterior y el nuevo calculado.

Estas explicaciones nos dan una idea de lo cuidado que está la interface de usuario y de su grado de optimización.

2.- Árboles de recubrimiento mínimo

En esta segunda parte del proyecto se trabajará con grafos cuyas conexiones entre nodos son aristas y no arcos y, por tanto, el recorrido del grafo se puede hacer a través de las aristas en cualquier sentido. Esta parte del trabajo tiene una documentación más técnica debido a la gran profundidad de la resolución de la aplicación informática y al hecho de **crear dos algoritmos originales** que calculan todas las soluciones de los árboles de recubrimiento mínimo. La complejidad en esta parte del trabajo es extrema tanto en la resolución teórica nueva del cálculo de todas las soluciones como en la implementación informática posterior. El ordenador tiene recursos finitos y la programación se ha tenido que hacer bajo este punto de vista con un estudio de investigación y optimización sobre el proyecto terminado y programado para hacerlo real y operativo.

También se ha generado un pseudocódigo universal para que no se pierda lo que está programado cuando estas instrucciones den problemas con el paso del tiempo (no se actualiza el código real del programa o simplemente desaparece el lenguaje de programación).

Por todo ello, la documentación de esta parte del proyecto es bastante técnica tanto por la complejidad del propio proyecto como por la utilización del pseudocódigo que requiere conocimientos de análisis y programación informática.

Esta parte se puede dividir en dos: por un lado la resolución de un árbol de recubrimiento mínimo por parte de un usuario mediante los algoritmos de Kruskal o Prim y por otro lado el cálculo de todos los árboles de recubrimiento mínimo sin la intervención del usuario (salvo para construir el grafo).

2.1.- Introducción, árboles de recubrimiento mínimo

Se trata de encontrar un recorrido que una todos los nodos de un grafo y que la suma de valores de las aristas sea mínima (lo que se denomina un árbol de recubrimiento mínimo). Para ello, se utilizan los algoritmos de Kruskal y Prim. También se calculan todas las soluciones existentes mediante dos algoritmos originales de este proyecto.

Para aplicar estos algoritmos el grafo debe ser:

- Conexo.
- No dirigido, es decir, sus nodos están unidos por aristas y no por arcos (una arista se puede recorrer en ambos sentidos).
- Cada arista debe tener un valor no negativo (que puede ser una distancia, un tiempo, un coste, un beneficio, una etapa, etc.).

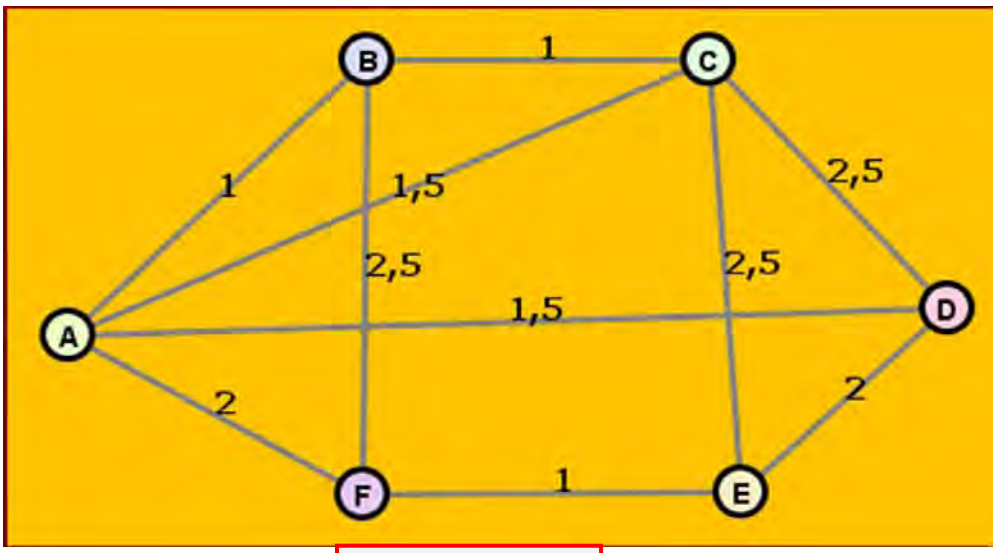


Figura 1

Una solución al grafo de la *"figura 1"* está en la *"figura 2"*.

Si existe un ciclo como en el grafo pequeño de la *"figura 3"*, se puede quitar una arista y los nodos siguen conectados entre sí, luego las soluciones que buscamos no

tienen ciclos (debe tener el menor número de aristas posible para que su suma sea mínima).

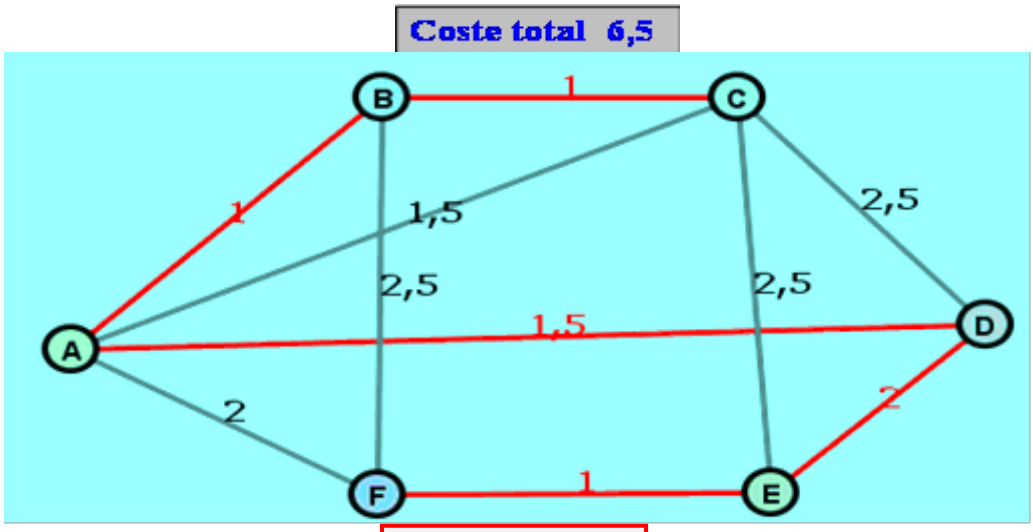


Figura 2

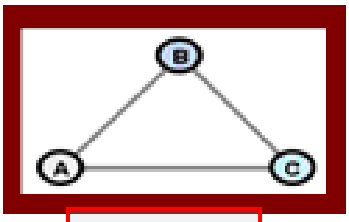


Figura 3

También aquí los nodos y las aristas pueden simbolizar lo que queramos para resolver un determinado tipo de problema, son ejemplos reales y muy importantes: ***Diseño de redes de transporte, diseño de redes de telecomunicaciones, televisión por cable, sistemas distribuidos, interpretación de datos climatológicos, visión artificial, análisis de imágenes, extracción de rasgos de parentesco, análisis de clusters y búsqueda de superestructuras de quásar, plegamiento de proteínas, reconocimiento de células cancerosas, construcción de carreteras y otros***.

Este problema fue considerado originalmente por Otakar Boruvka (1926) mientras estudiaba la necesidad de electrificación rural en el sur de Moldavia.

Joseph Kruskal nació en Estados Unidos en 1928 hermano del matemático y estadista William Kruskal (autor de la prueba de Kruskal-Wallis) y del matemático y físico Martin Kruskal (autor de las coordenadas de Kruskal-Szekeres).

En 1956, descubrió su algoritmo para la resolución del problema del árbol de coste total mínimo.

Robert Prim, nació en Estados Unidos en 1921, matemático e ingeniero informático. El algoritmo que lleva su nombre se conoce también por el algoritmo DJP, iniciales que corresponden a tres científicos: Este algoritmo fue diseñado en 1930 por el matemático Vojtech Jarnik y luego de manera independiente por el científico computacional Robert Prim en 1957 y redescubierto por Dijkstra en 1959.

Kruskal y Prim coincidieron en los laboratorios de investigación Bell y fue allí donde descubrieron sus algoritmos (Kruskal en 1956 y Prim en 1957).

Joseph B. Krukal

Robert C. Prim

Este proyecto tiene como objetivo fundamental, además del cálculo de aristas con una suma total mínima, el perdurar a lo largo del tiempo, y para ello se basa en dos pilares fundamentales:

- Una documentación extensa y clara para tener un conocimiento teórico y de la implementación de los métodos y procedimientos empleados.
- Un desarrollo en pseudocódigo que permita utilizar después la aplicación, basándose en cualquier lenguaje de programación.

Mediante los algoritmos de Kruskal y Prim se calcula una solución. Hay que destacar que el cálculo de todas las soluciones es original de este proyecto, es decir, primero se hace un estudio del porqué se producen varias soluciones tanto siguiendo el algoritmo de Kruskal como el de Prim, y a partir de este estudio se idean dos nuevos algoritmos completos que contemplen todas las soluciones óptimas posibles. Siguiendo estos dos algoritmos, uno utilizando el procedimiento de Kruskal para calcular una solución y el otro utilizando el procedimiento de Prim para una solución también, se llega al cálculo de todas las soluciones y es grato comprobar que, además de la demostración explicada en esta documentación, el número de soluciones calculado por ambos caminos, a veces es un número grande, coincide siempre (en la aplicación informática hay que pulsar los botones de "Kruskal automático" y "Prim automático"). Creo que es importante resaltar el hecho de que no he encontrado nada de información acerca del cálculo de todas las soluciones, lo que convierte este proyecto, en buena parte, en un proyecto de investigación por la gran complejidad del cálculo de todas las soluciones y el enorme esfuerzo empleado como se demuestra en la documentación aportada.

La idea de desarrollar primero un pseudocódigo universal, orientado a programación estructurada, nos permite también explicar paso a paso el proyecto sin escaparse ningún detalle (cosa que con el código de un lenguaje de programación orientado a objetos sería imposible).

El nivel de desarrollo del pseudocódigo es completo y cada proceso, además, viene detallado por organigramas funcionales que nos explican de forma gráfica el orden de

las acciones a seguir. Este es otro motivo por el cual la documentación es muy amplia. Al pasarlo a un lenguaje de programación sólo nos quedaría por implementar los detalles particulares de cada lenguaje ya que no habría que añadir nada a la resolución de los algoritmos. Para organizar toda la documentación, se divide de forma coherente para analizar cada parte por separado y simplificar la comprensión de todo el desarrollo.

Dada la importancia del trabajo y también para que no se pierda este trabajo cuando ya no se utilice el lenguaje de programación empleado, esta parte del proyecto está fundamentalmente enfocada a personas que sepan de análisis y programación informática, es decir, es una documentación bastante técnica.

El proyecto en lo que se refiere al cálculo de árboles de recubrimiento mínimo tiene los siguientes **objetivos**:

- Puesto en Internet, permitiría un acceso universal, o lo restringido que se quiera, para gran cantidad de personas porque, además se puede elegir entre el castellano y el inglés para toda la aplicación y la información aparece igual para los cinco navegadores de Internet más importantes.
- Soporte de ayuda para los alumnos que estudien teoría de grafos o cualquier otro uso que le quiera dar la cátedra de Investigación Operativa.
- Documentar un pseudocódigo universal para implementar el cálculo de los árboles de recubrimiento mínimo y preservar lo fundamental del proyecto para el futuro. Es decir, si los lenguajes de programación que implementan la aplicación cambian o desaparecen con el tiempo, el proyecto se podría perder. Si esto sucede, sería muy fácil volver a implementar el cálculo de los árboles de recubrimiento mínimo en el nuevo lenguaje de programación partiendo de los programas escritos en pseudocódigo. Para este objetivo, también se documentan los soportes de datos en los que se desarrolla la aplicación, de esta forma, se entiende muy bien lo que hace cada procedimiento en pseudocódigo.

- **Desarrollar dos algoritmos originales que calculen todas las soluciones (todos los árboles de recubrimiento mínimo).**

Para realizar estos objetivos, el proyecto consta de las siguientes **fases**:

- Interface gráfica muy desarrollada para que un usuario pueda crear un grafo no dirigido que represente el modelo del problema a resolver.
- Cálculo de un árbol de recubrimiento mínimo mediante los algoritmos de Kruskal y Prim. El usuario, a través de la interface gráfica seleccionará la solución óptima que quiera (si existe).
- Calcular todas las soluciones sin intervención del usuario (simplemente pulsando los botones que calculan todas las soluciones) por dos caminos diferentes. Creación de estos dos algoritmos.
- Información teórica para entender los procesos que se calculan y la ayuda para manejar la aplicación.

El usuario dispone de una interface gráfica muy completa para construir el grafo que le permitirá crearlo, modificar su tamaño y la posición de sus nodos, ampliarlo, reducirlo, cambiar los valores de sus aristas, etc siempre guiado por los mensajes que aparecen en la barra de estado. Cuando se calculan los árboles de recubrimiento mínimo, las soluciones aparecen en una ventana y, simultáneamente, en el grafo las aristas de cada solución y sus valores asociados pasan a tener el color rojo. La interface gráfica es bastante intuitiva y gracias, sobre todo, a los mensajes que aparecen en la barra de estado, el usuario sabe en todo momento lo que está haciendo y lo que tiene que hacer. Al situar el ratón sobre los botones de la interface, aparecen mensajes explicativos del botón sobre el que está el ratón, además, existe un botón de "Ayuda" que nos explica la función de cada elemento en la interface gráfica y la secuencia de pasos que hay que seguir para construir el grafo y calcular los árboles de recubrimiento mínimo.

La documentación teórica, que también se puede acceder a ella mediante un enlace en la aplicación puesta en Internet, nos explica los orígenes del problema que se quiere resolver, su fundamentación teórica y el procedimiento a seguir, además de desarrollar ejemplos completos paso a paso.

El código real del programa se expone en un fichero independiente por su gran volumen y porque no está ligado de forma inseparable al resto del proyecto, realmente, lo que va a perdurar en el tiempo es todo lo demás y el código real sólo pasa al ordenador lo desarrollado en pseudocódigo, soporte de datos, etc.

Definiciones de teoría de grafos

Para comenzar, es necesario dar una serie de definiciones:

Grafo:

Conjunto de nodos unidos por un conjunto de líneas o flechas llamadas aristas. Formalmente, un grafo es una pareja $G = < N , A >$, en donde *"N"* es un conjunto de nodos y *"A"* es un conjunto de aristas.

Grafo dirigido:

Sus nodos están unidos mediante aristas con indicación de dirección. Nunca hay más de dos flechas que unan dos nodos, y si hay dos flechas, entonces tienen que ir en sentidos opuestos. Una arista que vaya desde el nodo *"A"* hasta el nodo *"B"* la expresaré mediante el par ordenado *(A , B)*.

Grafo no dirigido:

Sus nodos están unidos mediante aristas sin indicación de dirección. Nunca hay más de una arista que una dos nodos. Una arista que una los nodos *"A"* y *"B"* la expresaré mediante el conjunto *{A , B}*.

Grafo conexo:

Se puede llegar desde cualquier nodo hasta cualquier otro siguiendo una secuencia de aristas (en el caso de un grafo dirigido se permite circular en sentido inverso a lo largo de una flecha).

Componente conexa:

Llamaremos "componente conexa" a un subconjunto de nodos de *G* que sea conexo.

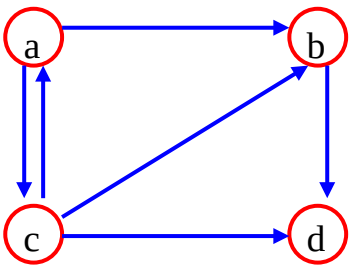
Grafo dirigido fuertemente conexo:

Se puede pasar desde cualquier nodo hasta cualquier otro siguiendo una secuencia de aristas, pero respetando el sentido de las flechas.

Nodos adyacentes:

El nodo *"i"* y el nodo *"j"* son adyacentes si están unidos por una arista.

Ejemplo:

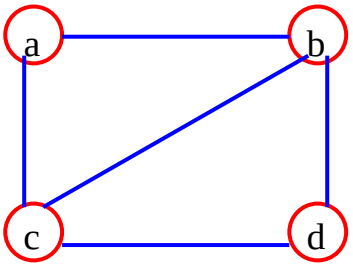


G = < N , A >, en donde

N = {a , b , c , d}

A = { (a , b) , (a , c) , (b , d) ,
(c , a) , (c , b) , (c , d) }

Si este grafo fuese no dirigido tendríamos:



G = < N , A >, en donde

N = {a , b, c , d}

A = { {a , b} , {a , c} , {b , d} ,
{c , a} , {c , b} , {c , d} }

Subconjunto propio (inclusión estricta):

Si $A \subseteq B$, y además, existen elementos de B que no pertenecen a A, se dice que A es un subconjunto propio de B. Entonces, la inclusión de A en B es estricta.

Árboles:

Un árbol (hablando con propiedad un árbol libre) es un grafo acíclico, conexo y no dirigido, es decir, un grafo no dirigido en el cual existe exactamente un camino entre todo par de nodos dado.

Propiedades:

- Un árbol con *"n"* nodos posee exactamente *"n – 1"* aristas.
- Si se añade una única arista a un árbol, entonces el grafo resultante contiene un único ciclo.
- Si se elimina una única arista de un árbol, entonces el grafo resultante ya no es conexo.

Árboles de recubrimiento mínimo (o maximales minimales) de un grafo:

El grafo $G = < N , A >$ tiene que ser:

- Conexo.
- No dirigido.
- Cada arista debe tener un valor no negativo.

Para obtener un árbol de recubrimiento mínimo del grafo anterior hay que hallar un subconjunto *T* de aristas de *G* tal que utilizando solamente las aristas de *T*, todos los nodos deben quedar conectados y además la suma de los valores asociados a cada arista de *T* debe ser tan pequeña como sea posible. Dado que *G* es conexo, debe existir al menos una solución. Dadas dos soluciones de igual valor total en la suma de sus aristas, preferimos la que contenga menos aristas. Incluso con esta condición el problema puede tener varias soluciones diferentes y de igual valor.

Sea *G' = < N ,T >* el grafo parcial formado por los nodos de *G* y las aristas de *T* y supongamos que en *N* hay *"n"* nodos. Un grafo conexo con *"n"* nodos debe tener al menos *"n – 1"* aristas, así que este es el número mínimo de aristas que puede haber en *T*. Por otra parte, un grafo con *"n"* nodos y más de *"n – 1"* aristas contiene al menos un ciclo. Por tanto, si *G'* es conexo y *T* tiene más de *"n – 1"* aristas se puede eliminar al menos una arista sin desconectar *G'*, siempre y cuando seleccionemos una arista que forme parte de un ciclo, luego un conjunto *T* con *"n"* o más aristas no puede ser óptimo. Se deduce que *T* debe tener exactamente *"n – 1"* aristas, y como *G'* es conexo, tiene que ser un árbol.

El grafo *G'* se denomina árbol de recubrimiento mínimo para el grafo *G*.

Conjunto de aristas prometedor:

Diremos que un conjunto de aristas factible es "prometedor" si se puede extender para producir no sólo una solución, sino una solución óptima para nuestro problema. En particular, el conjunto de aristas vacío siempre es prometedor. Además, cuando un conjunto de aristas prometedor ya es una solución, esta solución debe ser óptima.

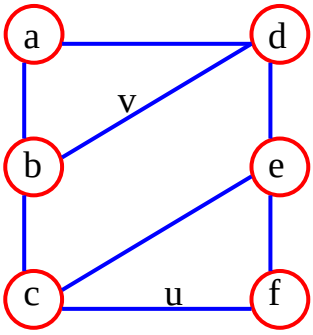
Lema para construir un conjunto de aristas prometedor:

Sea $G = < N , A >$ un grafo conexo no dirigido en el cual está dada la longitud de todas las aristas. Sea $B \subset N$ un subconjunto estricto de los nodos de G. Sea $T \subseteq A$ un conjunto prometedor de aristas tal que no haya ninguna arista de T que sale de B. Sea "v" la arista más corta que salga de B (o una de las más cortas si hay empates). Entonces $T \cup \{v\}$ es prometedor.

Demostración:

Sea U un árbol de recubrimiento mínimo de G tal que el conjunto de aristas $T \subseteq U$. Este U tiene que existir puesto que T es prometedor por hipótesis. Queremos añadir una arista "v", la más corta que sale de B, a T y que siga siendo T prometedor.

Siguiendo un ejemplo:



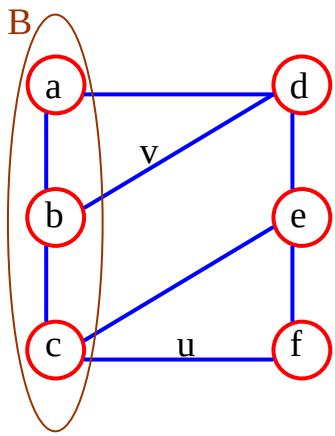
Grafo G = < N , A >

N = { a , b , c , d , e , f }

A = { { a , b } , { a , d } , { b , c } , v , { c , e } ,
u , { d , e } , { e , f } }

Llamamos a la arista { b , d } como "v", y a la arista { c , f } como "u", por simplificar los términos.

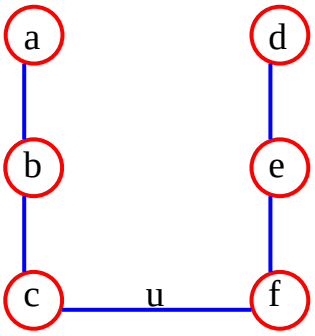
En este grafo tenemos:



B = { a , b , c }

T = { { a , b } , { b , c } }

El árbol *U* de recubrimiento mínimo de *G* es, por ejemplo:



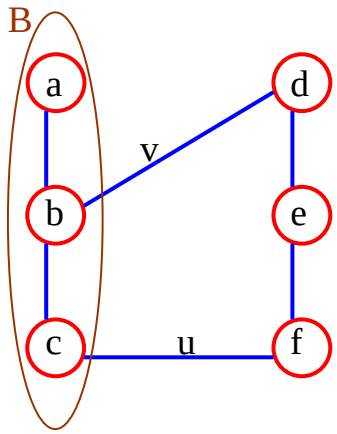
U = < N' , A' >

N' = { a , b , c , d , e , f }

A' = { { a , b } , { b , c } , u , { d , e } , { e , f } }

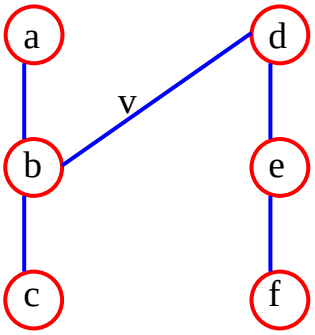
"*v*" sale de *B* por hipótesis. Si $v \notin U$ entonces no hay nada que probar, porque de lo que se trata es de extender *T* hasta conseguir un árbol de recubrimiento mínimo de *G* (por definición de conjunto de aristas prometedoras). Si $v \notin U$, cuando añadimos "*v*" a *U* creamos exactamente un ciclo (esta es una propiedad de los árboles). Si "*v*" sale de *B* y existe un ciclo, existe necesariamente al menos otra arista "*u*" que también sale de *B*, ya que si la arista "*v*" ha producido el ciclo y sale de *B*, tendrá que haber, por lo menos, una arista que sale de *B*, además de "*v*", para que *U* fuese conexo antes de añadir "*v*".

Añadimos "*v*" a *U*:



U deja de ser un árbol al tener un ciclo.

Si ahora eliminamos *"u"*, el ciclo desaparece y obtenemos un nuevo árbol *V* que abarca *G*. Como el valor de *"v"*, por hipótesis, no es mayor que el valor de *"u"*, y por tanto el valor total de las aristas de *V* no sobrepasa el valor total de las aristas de *U*, entonces *V* es también un árbol de recubrimiento mínimo de *G* y contiene a *"v"*. $T \leq V$ porque la arista *"u"* que se ha eliminado sale de *B* con un valor no menor que *"v"*, y por tanto se puede descartar como arista de *T*.



Árbol *V* que abarca a *G* al eliminar la arista *"u"*.

Algoritmo de Kruskal

Dado un grafo:

- Conexo.
- No dirigido.
- Cada arista debe tener un valor no negativo (que puede ser una distancia, un tiempo, un coste, etc.).

Se trata de encontrar un recorrido que una todos los nodos del grafo y que la suma de valores de las aristas sea mínima.

El algoritmo es:

1º.- Se ordenan las aristas del grafo *G* original de menor a mayor valor.

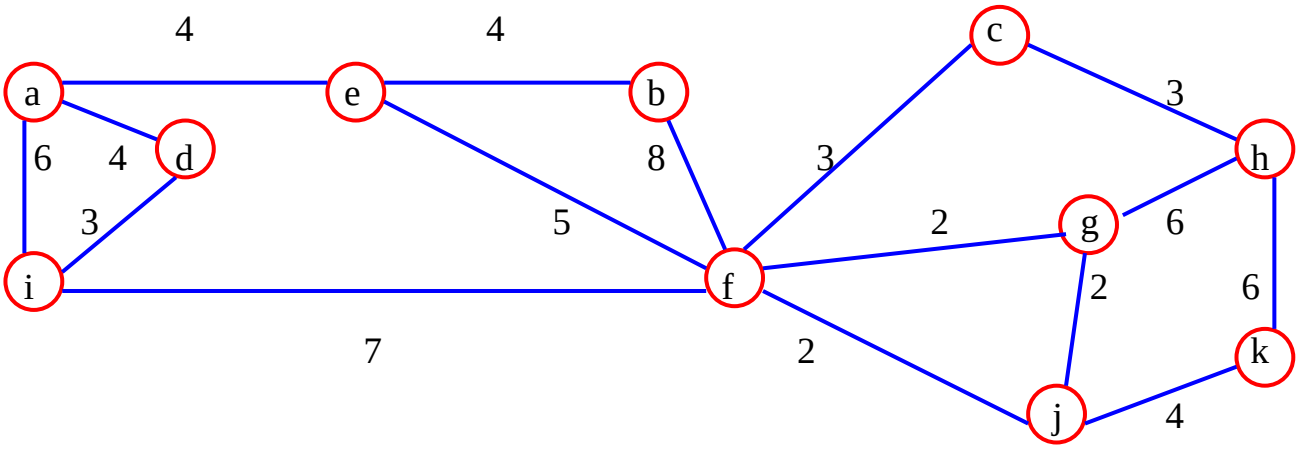
2º.- Se toma la arista con el menor valor y se le van añadiendo las restantes aristas, en orden creciente, siempre y cuando la conexión de una nueva arista no de lugar a un ciclo , en cuyo caso no se añadirá.

3º.- Cuando todos los nodos estén conectados , paramos y ya tenemos la solución.

Como demostraremos más adelante, el algoritmo de Kruskal halla un árbol de recubrimiento mínimo, con lo cual, el grafo solución tendrá exactamente *"n"* nodos y *"n-1"* aristas (siendo *"n"* el número de nodos del grafo original).

Ejemplo:

Sea *G* el grafo de partida que se muestra en la figura:

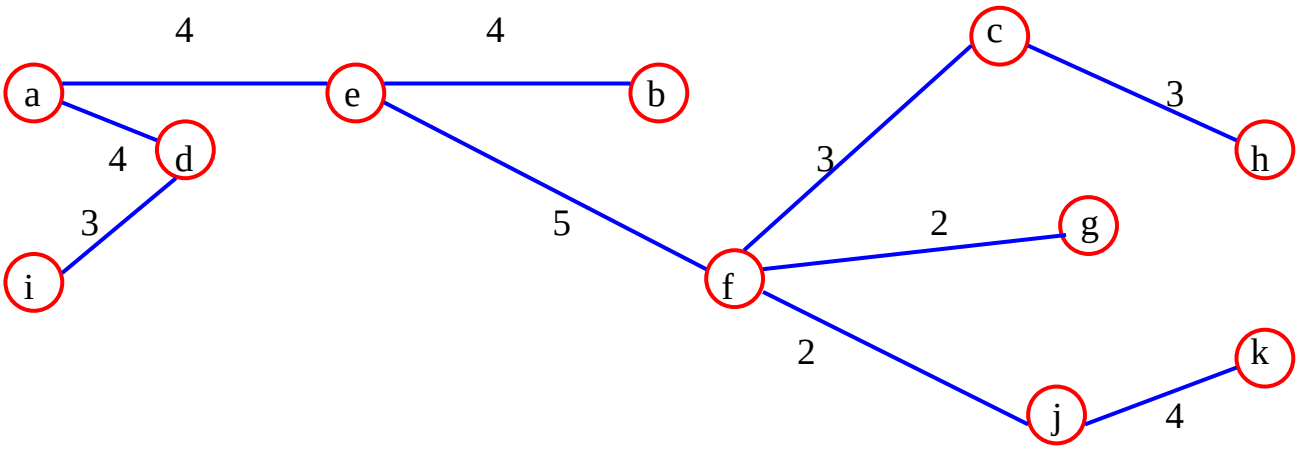


G posee 16 aristas y 11 nodos; el árbol de recubrimiento mínimo tendrá por tanto 11 - 1 = 10 aristas y 11 nodos. Ordenando las aristas en orden creciente de valores tenemos:

ARISTAS	fg	fj	gj	di	fc	ch	ae	ad	eb	jk	ef	ai	kh	gh	if	bf
VALORES	2	2	2	3	3	3	4	4	4	5	6	6	6	7	8	
¿AÑADIR?	Si	Si	No	Si	Si	Si	Si	Si	Si	Si	Si	No	No	No	No	No	

A partir de aquí ya tenemos 10 aristas con *"Si"* con lo cual ya hemos terminado porque el resto de aristas son todas *"No"*.

Se van tomando las aristas indicadas con *"Si"* hasta tener 10 y de manera que no se formen ciclos (aristas con *"No"*). Al final se habrá obtenido el árbol de recubrimiento mínimo *G'* cuya suma de valores es 34. El árbol *G'* posee 11 nodos y 10 aristas.



Teorema: el algoritmo de Kruskal haya un árbol de recubrimiento mínimo

La demostración se hace por inducción matemática sobre el número de aristas que hay en el conjunto *T*. Mostraremos que si *T* es prometedor entonces sigue siendo prometedor en cualquier fase del algoritmo cuando se le añade una arista adicional. Cuando se detiene el algoritmo, *T* da una solución de nuestro problema; puesto que también es prometedora, la solución es óptima.

- Base: Partimos de un conjunto *T* de aristas vacío. El conjunto de aristas vacío es prometedor porque *G* es conexo y por tanto tiene que existir una solución.
  Inicialmente, cada nodo de *G* por separado es una componente conexa (trivial).
  Es decir, inicialmente: $\begin{cases} T = \{ \emptyset \}. \\ \text{Componentes conexas de } G \longrightarrow \text{cada nodo de } G. \end{cases}$
- Paso de inducción: Supongamos que *T* es prometedor inmediatamente antes de que el algoritmo añada una nueva arista $e = \{ u , v \}$. El nodo *"u"* y el nodo *"v"* deben estar en componentes conexas distintas ya que en caso contrario se formaría

un ciclo. Sea *B* el conjunto de nodos de la componente conexa que contiene a *"u"*, ahora:

- El conjunto *B* es un subconjunto estricto de los nodos de *G* (no incluye al nodo *"v"*, por ejemplo).
- *T* es un conjunto prometedor de aristas tal que ninguna arista de *T* sale de *B*.
- *"e"* es la arista más corta o una de las aristas más cortas, si hay empates, que salen de *B* (porque todas las aristas estrictamente más cortas ya se han examinado y, o bien ya se han añadido a *T*, o bien se han rechazado porque tenían los dos extremos en la misma componente conexa y formarían un ciclo).

Entonces se cumplen las condiciones del "lema" antes visto y concluimos que el conjunto *T U { e }* también es prometedor.

Esto completa la demostración por inducción matemática de que el conjunto *T* es prometedor en todas las fases del algoritmo, y por tanto cuando se detiene el algoritmo, *T* no da meramente una solución de nuestro problema, sino una solución óptima.

En el algoritmo de Kruskal, se escogen las aristas por orden creciente de sus valores, sin preocuparse demasiado por su conexión con las aristas seleccionadas anteriormente, salvo que se tiene cuidado para no formar nunca un ciclo. El resultado es un conjunto de árboles (componentes conexas) que crece al azar, hasta que finalmente todas las componentes conexas se fusionan en un único árbol.

Algoritmo de Prim

Dado un grafo $G = < N , A >$:

- Conexo.
- No dirigido.
- Cada arista debe tener un valor no negativo (que puede ser una distancia, un tiempo, un coste, etc.).

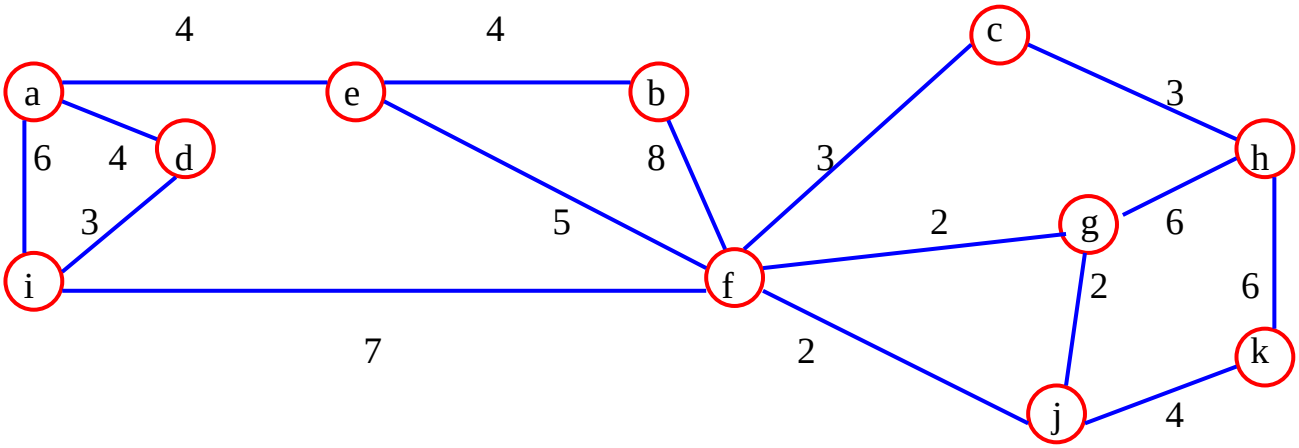
Encuentra un recorrido que une todos los nodos del grafo con la suma de los valores de las aristas mínimas.

También halla un árbol de recubrimiento mínimo de un grafo, pero la forma de crearlo es a partir de un nodo arbitrario (nodo raíz) y en cada fase se añade una nueva rama al árbol ya construido; el algoritmo se detiene cuando se han alcanzado todos los nodos.

Sea *B* un subconjunto de nodos de *G*, y sea *T* un conjunto de aristas. Inicialmente, *B* contiene un único nodo arbitrario de *G* y *T* está vacío. En cada paso, el algoritmo de Prim busca la arista más corta posible $\{ i , j \}$ tal que $i \in B$ y $j \notin B$. Entonces añade el nodo *"j"* a *B* y la arista $\{ i , j \}$ a *T*. De esta manera las aristas de *T* forman en todo momento un árbol de recubrimiento mínimo para los nodos de *B*. Continuamos mientras *B* sea distinto de *N*.

Ejemplo:

Utilizamos el mismo grafo que se utilizó en el ejemplo del algoritmo de Kruskal:



Empezamos por el nodo *"f"*, por ejemplo:

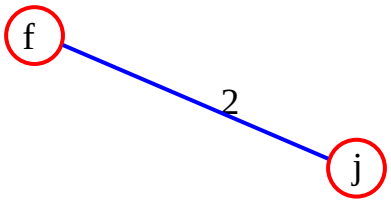
B = {f}

T = { Ø }

Buscamos la arista de menor valor que sale de *B*:

Hay dos posibilidades, la arista *{ f , j }* o la arista *{ f , g }*.

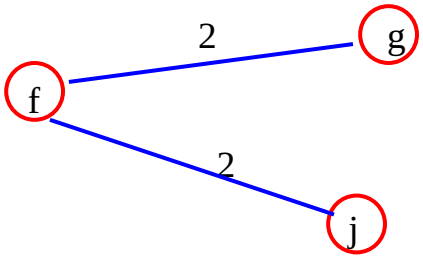
Elegimos la arista *{ f , j }*, por ejemplo:



B = { f , j }

T = { { f , j } }

Seguimos el proceso eligiendo la arista más corta que sale de *B*. Hay también dos opciones y elegimos una cualquiera:



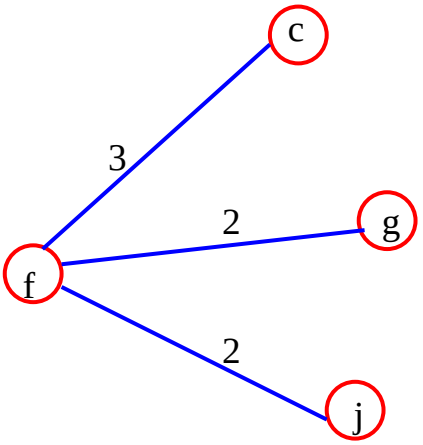
B = { f , j , g }

T = { { f , j } , { f , g } }

Seguimos el mismo proceso hasta que *B* sea igual al conjunto *N* de nodos del grafo:

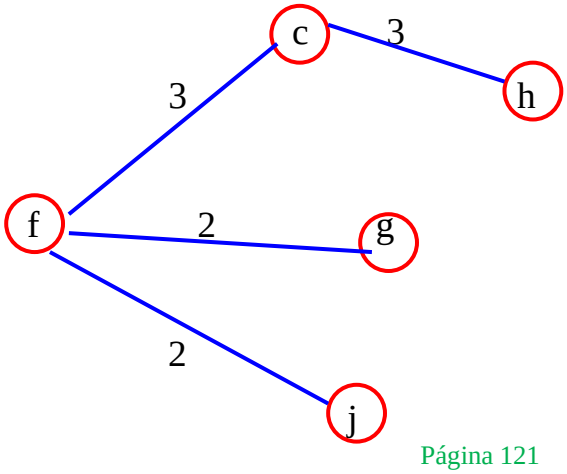
B = { f , j , g , c }

T = { { f , j } , { f , g } , { c , f } }



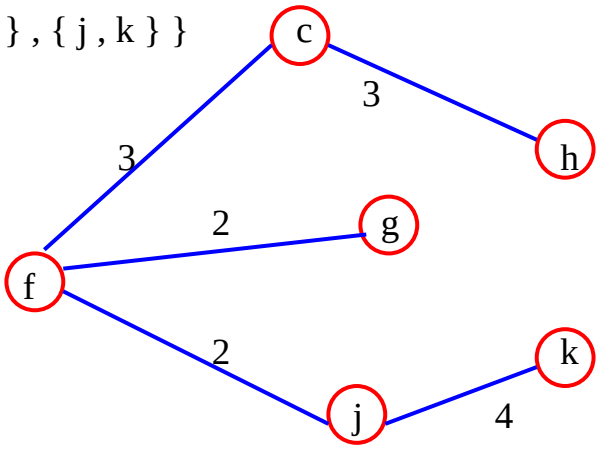
B = { f , j , g , c , h }

T = { { f , j } , { f , g } , { c , f } , { c , h } }



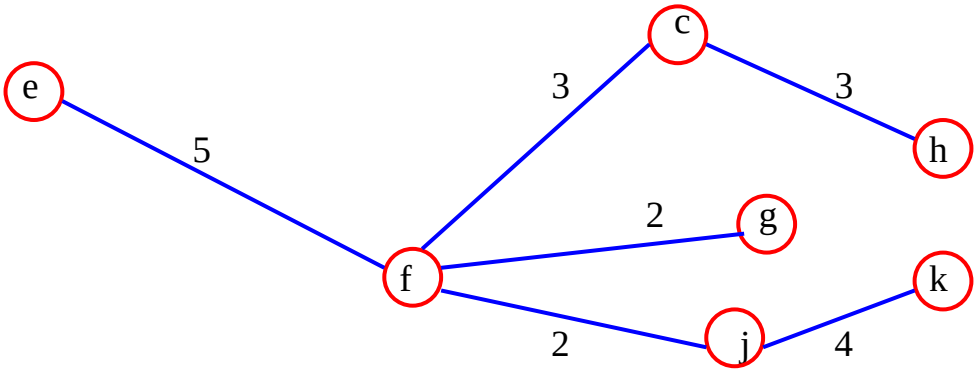
B = { f , j , g , c , h , k }

T = { { f , j } , { f , g } , { c , f } , { c , h } , { j , k } }



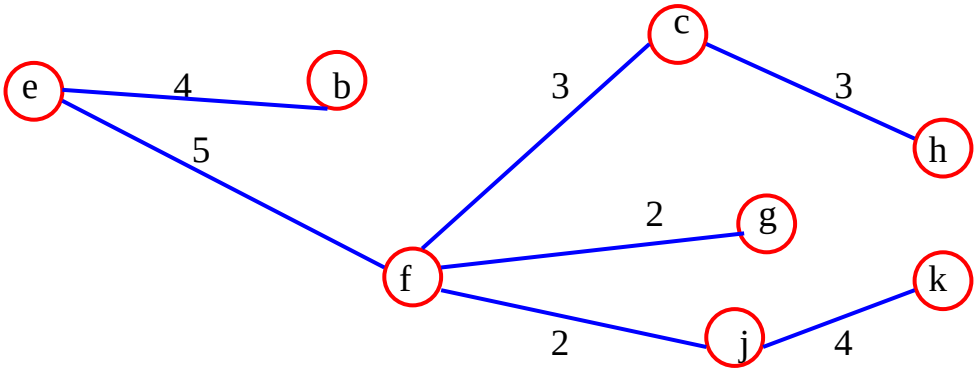
B = { f , j , g , c , h , k , e }

T = { { f , j } , { f , g } , { c , f } , { c , h } , { j , k } , { e , f } }



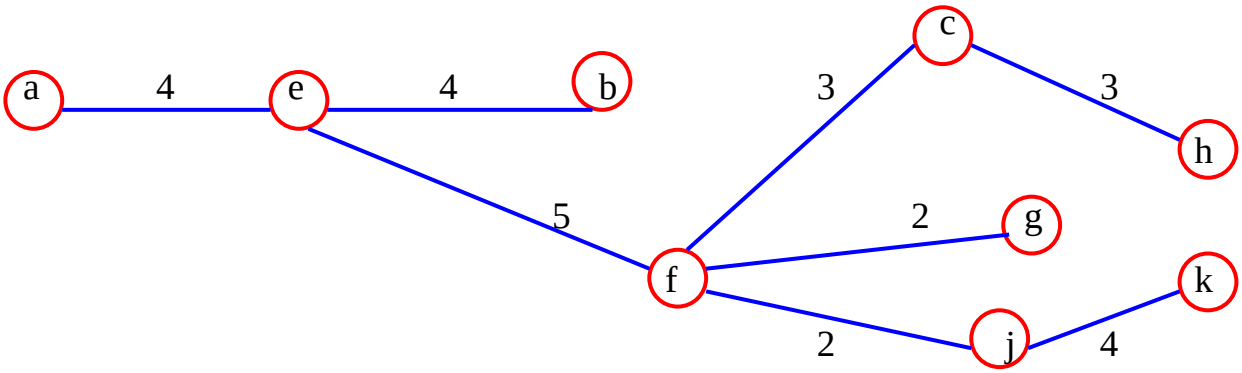
B = { f , j , g , c , h , k , e , b }

T = { { f , j } , { f , g } , { c , f } , { c , h } , { j , k } , { e , f } , { b , e } }



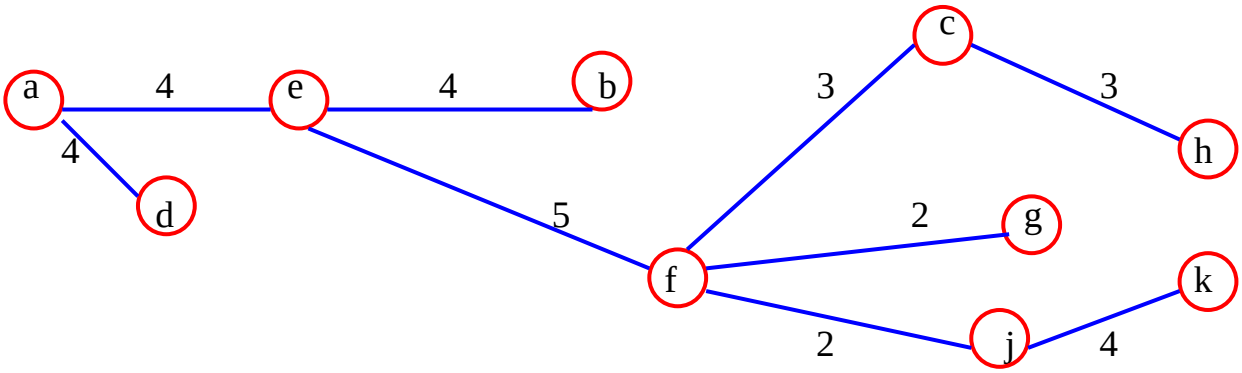
B = { f , j , g , c , h , k , e , b , a }

T = { { f , j } , { f , g } , { c , f } , { c , h } , { j , k } , { e , f } , { b , e } , { a , e } }



B = { f , j , g , c , h , k , e , b , a , d }

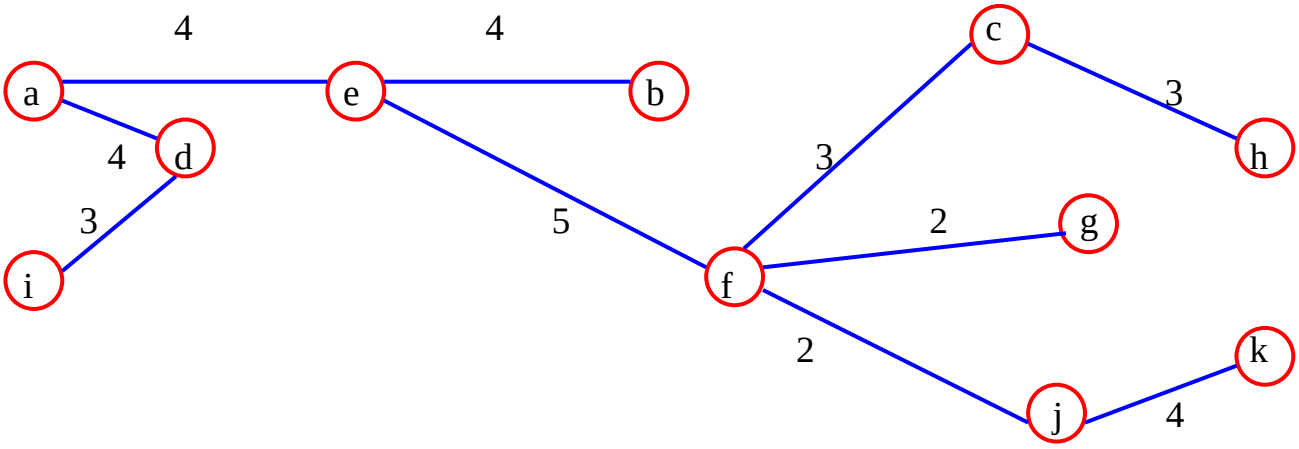
T = { { f , j } , { f , g } , { c , f } , { c , h } , { j , k } , { e , f } , { b , e } , { a , e } ,
{ a , d } }



B = { f , j , g , c , h , k , e , b , a , d , i }

T = { { f , j } , { f , g } , { c , f } , { c , h } , { j , k } , { e , f } , { b , e } , { a , e } ,
{ a , d } , { d , i } }

Ahora $B = N$, luego T nos da la solución final:



G’ = < N , T >

N = { a , b , c , d , e , f , g , h , i , j , k }

T = { { f , j } , { f , g } , { c , f } , { c , h } , { j , k } , { e , f } , { b , e } , { a , e } ,
{ a , d } , { d , i } }

Teorema: El algoritmo de Prim halla un árbol de recubrimiento mínimo

La demostración es parecida a la demostración del algoritmo de Kruskal.
Se hace por inducción matemática sobre el número de aristas que hay en el conjunto *T*. Demostramos que si *T* es prometedor en alguna fase del algoritmo, entonces sigue siendo prometedor al añadir una arista adicional. Cuando se detienen el algoritmo, *T* da una solución a nuestro problema; puesto que también es prometedora, la solución es óptima.

- Base: El conjunto vacío es prometedor.
- Paso de inducción: Suponemos que *T* es prometedor inmediatamente antes de que el algoritmo añada una nueva arista $e = \{ i , j \}$. Ahora *B* es un subconjunto estricto de *N* (porque el algoritmo se detiene cuando *B* = *N*), *T* es un conjunto de aristas prometedor por hipótesis de inducción, y *“e”* es por definición una de las aristas más cortas que salen de *B*. Entonces las condiciones del lema ya visto se cumplen y $T \cup \{ e \}$ también es prometedor.

Esto completa la demostración por inducción matemática de que el conjunto *T* es prometedor en todas las fases del algoritmo. Por tanto, cuando se detiene el algoritmo, *T* ofrece una solución óptima de nuestro problema.

2.2.- Implementación del proyecto en pseudocódigo

Se trata de explicar con claridad e implementar en pseudocódigo todos los pasos de este proyecto, con lo cual, los soportes de información, la forma de estructurar el programa, los algoritmos empleados y todas las ideas sobre la posterior realización en código son independientes del lenguaje de programación que utilicemos después.

Voy a dividir la implementación en pseudocódigo en cinco partes:

- Elementos comunes a los algoritmos de Kruskal y Prim.
- Una sola solución: Resolución automática del algoritmo de Kruskal y resolución también a través de un usuario.
- Todas las soluciones: Resolución automática del algoritmo de Kruskal.
- Una sola solución: Resolución automática del algoritmo de Prim y resolución también a través de un usuario.
- Todas las soluciones: Resolución automática del algoritmo de Prim.

Las sentencias que voy a utilizar en pseudocódigo (además de las sentencias triviales, como las de asignación, etc.) las explicaré a continuación:

Instrucción condicional simple

si <condición> , entonces

<sentencias>

finsi

condición ⟶ Variable o expresión booleana, o bien, la comparación de dos expresiones, dando siempre un resultado booleano de “True” o “False”.

Si la condición es “True”, se ejecutan las “sentencias” y si es “False” esta instrucción no hace nada.

Instrucción condicional compuesta

si <condición> , entonces

<sentencias1>

sino

<sentencias2>

finsi

condición ⟶ Tiene el mismo significado que en la instrucción anterior.

Si la “condición” es “True”, se ejecutan las “sentencias1”, y si es “False” se ejecutan las “sentencias2”

Instrucción bucle mientras

mientras <condición> , hacer

<sentencias>

finmientras

condición ⟶ Tiene el mismo significado que en las instrucciones anteriores.

1º.- Se comprueba si la “condición” es “True” o “False”.

2º.- Si la “condición” es “True” se ejecutan las “sentencias” y se vuelve al paso 1º, y si es “False” se termina la instrucción.

Instrucción bucle hasta

repetir

<sentencias>

hasta <condición>

_____

condición ⟶ Tiene el mismo significado que en las instrucciones anteriores.

1º.- Se ejecutan las “sentencias”.

2º.- Si la condición es “True” se vuelve al paso 1º, y si es “False” se termina la instrucción.

Instrucción bucle acotado

para variable = inicio hasta fin , hacer

<sentencias>

finpara

_____

inicio
fin
} ⟹ Son valores de tipo entero.

Se ejecutan las “sentencias” un número de veces igual a la resta entre los valores “fin” menos “inicio”.

Instrucción comparativa

```
seleccionar  opción  <referencia>

    opción1:    <comparación1>

        <sentencias1>

    opción2:    <comparación2>

        <sentencias2>
    :
    :
    :
    opciónk:    <comparaciónk>

        <sentenciask>

    else:

        <sentencias else>

finseleccionar
```

referencia ⟶ Es una variable o expresión.

comparación ⟶ Variable o expresión que se compara con “referencia” dando un resultado booleano.

sentencias ⟶ Si la comparación “i” es “True”, se ejecutan sus “sentenciasi” asociadas y se termina la instrucción.

else ⟶ Si todas las comparaciones son “False” se ejecutan las “sentencias else” y se termina la instrucción.

2.2.1.- Elementos comunes a los algoritmos de Kruskal y Prim

Elementos para almacenar la información básica:

Utilizaremos la variable global "***n***" para guardar el número de elementos del grafo.

Sea *N* el conjunto de nodos del grafo con *"n"* elementos, *N = { 1 , 2 , ... , n }*, entonces necesitamos una matriz de dos dimensiones a la que llamaremos "**valor_arista**" de *" n× n "* elementos que contenga el valor de las aristas existentes entre cada par de nodos del grafo:

$$\text{valor\_arista} = \begin{array}{c} \\ 1 \\ 2 \\ : \\ : \\ n \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \ldots & n \end{array} \\ \begin{pmatrix} & & & \\ & & & \\ & & & \\ & & & \\ & & & \end{pmatrix} \end{array}$$

Evidentemente, esta matriz es simétrica por ser el grafo no dirigido. Si no existe una arista entre dos nodos, por ejemplo, entre los nodos *"i"* , *"j"*, entonces hacemos:

valor_arista [i , j] = -1
valor_arista [j , i] = -1

Aunque el usuario vea letras en los nombres de los nodos, internamente serán números para hacer más sencilla la programación (los nodos del grafo A, B, C, ... se corresponden con los nodos 1, 2, 3, ... respectivamente).

Comprobar que el grafo es conexo

Después de finalizar el diseño completo del grafo, pasaríamos a realizar los algoritmos deseados. Es al pasar del grafo a los algoritmos cuando debo comprobar que el grafo creado es conexo, ya que no podemos realizar los algoritmos de Kruskal y Prim con un grafo no conexo. En este punto se ejecutará en el lenguaje de programación elegido (en este caso Java) lo que equivale al pseudocódigo que voy a escribir.

La idea es la siguiente:

Inicialmente, nos creamos una componente conexa *B* de un sólo nodo (cualquier nodo del grafo), por ejemplo, el nodo 1. Vamos añadiendo nodos del grafo a *B* con la condición de que *B* siga siendo una componente conexa y si al final todos los nodos del grafo están en *B*, el grafo es conexo. Si no podemos conseguir esto porque hay algún nodo o nodos aislados del resto de nodos del grafo, el grafo no es conexo.

Me creo la matriz booleana de una dimensión "*nodos_de_B*" con un número de elementos igual al número de nodos del grafo. Cada elemento "*i*" de esta matriz tomará los valores:

nodos_de_B [i] → True ⟹ Si el nodo "i" del grafo ∈ B

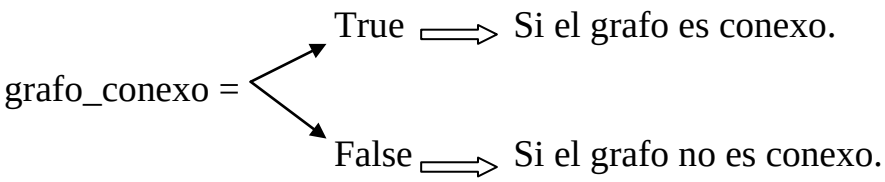
nodos_de_B [i] → False ⟹ Si el nodo "i" del grafo ∉ B

La variable "cont" nos va diciendo, en todo momento, el número de nodos del grafo que están en B. Lógicamente, en el momento en que "cont = n" siendo "n" el número de nodos del grafo, el grafo es conexo.

La implementación en pseudocódigo se muestra en la función "grafo_conexo".

Función **“grafo conexo”**

Función booleana que devuelve los valores:



El pseudocódigo será:

(inicializamos el conjunto B con el nodo 1)

nodos_de_B [1] ⟵ True

para i = 2 hasta n , hacer

nodos_de_B [i] ⟵ False

finpara

grafo_conexo ⟵ True

(como el conjunto B tiene un nodo, inicializamos
el número de nodos de B a 1 en la variable “cont”)

cont ⟵ 1

```
para  k = 1  hasta  n-1  ,  hacer
    si  grafo_conexo  ,  entonces
        grafo_conexo  ←  False
        para  i = 1  hasta  n  ,  hacer
            si  nodos_de_B [ i ]  ,  entonces
                para  j = 1  hasta  n  ,  hacer
                    si  ( no ( nodos_de_B [ j ] ) )  ,  entonces
                        si  valor_arista [ i , j ]  ≥  0  ,  entonces
                            (existe una arista entre un nodo i  B∈
                                y un nodo  j  ∉B)
                            (añadimos el nodo "j" a B y sumamos 1
                                al número de nodos de B)
                            nodos_de_B [ j ]  ←  True
                            cont  ←  cont + 1
                            (como he añadido un nodo más a B, sigue habiendo
                                posibilidades de que el grafo sea conexo)
                            grafo_conexo  ←  True
                            si  cont = n  ,  entonces
                                "El grafo es conexo" ; "Fin de la función"
                            finsi
                        finsi
                    finsi
                finpara
            finsi
        finpara
    si no
        "Grafo no conexo" ; "Fin de la función"
    finsi
finpara
```

2.2.2.- Implementación del algoritmo de Kruskal

En esta parte de la documentación del proyecto no voy a tratar el conjunto de todas las soluciones óptimas posibles, si las hay, (esto lo dejo para más adelante) sino simplemente el poder llegar a una solución óptima cualquiera del conjunto de soluciones óptimas posibles.

Haré dos desarrollos del algoritmo de Kruskal en dos partes diferenciadas:

Primera parte: Lógica y desarrollo de la implementación general del algoritmo de Kruskal de forma automática sin intervención del usuario.

Segunda parte: Implementación del algoritmo de Kruskal a través de la intervención de un usuario seleccionando aristas sobre un grafo.

Me va a ser muy útil la implementación en pseudocódigo de esta primera parte porque voy a desarrollar las ideas, soportes de datos, un algoritmo, y sobre todo, estructurando el pseudocódigo en procedimientos y funciones, puedo reutilizar muchos de estos procedimientos y funciones en el desarrollo del pseudocódigo para resolver la segunda parte y también para calcular todas las soluciones óptimas posibles. La idea es desarrollar muchas explicaciones e implementar pseudocódigo reutilizable con el fin de que en una etapa posterior, en mi caso para hallar todas las soluciones o para que el usuario pueda resolver el algoritmo de Kruskal seleccionando aristas, se pueda simplificar y aclarar mejor el desarrollo sin detenerse en detalles y explicaciones básicas, con lo que se agiliza y entiende el proceso mucho mejor.

En la segunda parte, si en la resolución de nuestro problema sólo existe una única solución óptima, el usuario deberá ir seleccionando aristas hasta encontrarla, pero si existen varias soluciones óptimas se dará por buena cualquier solución óptima que encuentre el usuario.

2.2.2.2.- Lógica y desarrollo de la implementación general del algoritmo de forma automática sin intervención del usuario

Pasos a realizar:

1.- Definir el grafo $G = < N , A >$ conexo, no dirigido, y con valores no negativos en sus aristas.

2.- Calcular el número de nodos *"n"* de *N*.

3.- Clasificar el conjunto de aristas *A* en orden creciente de valores.

4.- Para $i = 1$ hasta $n - 1$, hacer el bucle:

Añadir al conjunto de aristas solución la arista de *G* con menor valor que no haya sido seleccionada previamente y que no forme un ciclo con otras aristas del conjunto de aristas solución.

5.- $G' = < N , Aristas\ solución >$ es el árbol de recubrimiento mínimo, es decir, la solución óptima.

Los pasos 1 y 2 son parte común a todos los procesos relacionados con los algoritmos de Kruskal y Prim y su realización corresponde al desarrollo de la parte gráfica. Cuando se construye el grafo, se controlan sus características, valores de sus aristas y se calcula el número *"n"* de sus nodos.

Luego lo que hay que desarrollar ahora son los pasos del 3 al 5.

Paso 3º: Ordenación de las aristas del grafo de menor a mayor valor

El algoritmo de ordenación actuará sobre una matriz de una dimensión que llamaré "***aristas_ordenandas***". Cada elemento de la matriz consta de cuatro datos:

nodoi	nodoj	valor	fase

El dato *"valor"* es el valor de la arista que esta entre los nodos *"nodoi"* y *"nodoj"*.

El dato *"fase"* puede tomar los valores → disponible, solución, forma_ciclo

Más adelante, cuando estudiemos el paso 4º explicaré la función del dato *"fase"*.

Si quiero referirme a un elemento *k* de esta matriz con toda su información haré *"aristas_ordenedas [k] "* y si quiero referirme sólo a un dato en concreto del elemento *k*, por ejemplo, al valor de la arista, haré *"aristas_ordenedas.valor [k]"*.

En un principio, esta matriz no está ordenada. Cuando finalice el algoritmo de ordenación, que veremos más adelante, los elementos de esta matriz estarán ordenados de menor a mayor por el valor de cada arista (dato *"valor"*).

De la matriz *"valor_arista"* extraigo las aristas y las llevo a la matriz *"aristas_ordenandas"* en cualquier orden. Como la matriz *"valor_arista"* es simétrica por ser el grafo no dirigido, las aristas *"matriz_arista [i , j]"* y *"matriz_arista [j , i]"* son la misma arista, con lo cual sólo llevo una de ellas a la matriz *"aristas_ordenandas"*.

El seudocódigo del paso de las aristas de una matriz a otra es:

```
m ← 0

para i = 1 hasta n , hacer

    para j = i + 1 hasta n , hacer

                                  Condición para que exista la arista.

        si valor_arista [ i , j ] ≥ 0 , entonces

            m ← m + 1

            aristas_ordenadas.nodoi [ m ] ← i

            aristas_ordenadas.nodoj [ m ] ← j

            aristas_ordenadas.valor [ m ] ← valor_arista [ i , j ]

            aristas_ordenadas.fase [ m ] ← "disponible"

        finsi

    finpara

finpara
```

En la variable *"m"* tenemos el número de aristas del grafo que es también el número de elementos de la matriz *"aristas_ordenadas"*.

Cabe destacar también, que *aristas_ordenadas.fase [m] = "disponible"*, para todas las aristas *"m"*, lo que explicaré más adelante.

Utilizaré el método de ordenación por selección:

3.1.- Busco el elemento más pequeño de todos y lo pongo el primero, evidentemente el elemento que estaba en la primera posición pasa a ocupar la posición que tenía el elemento más pequeño que hemos encontrado, es decir, se intercambian.

3.2.- Busco desde la segunda posición en adelante, el elemento más pequeño, y lo pongo en segundo lugar, haciendo también un intercambio, y así sucesivamente hasta que todos los elementos queden ordenados.

Para implementar el paso 3.1 sobre la matriz *"aristas_ordenadas"*:

"minj" Me dará el índice del elemento de menor valor en su dato *"valor"* de la matriz *"aristas_ordenadas"*.

"minx" Me irá dando el menor valor del dato *"valor"* de las aristas que están en la matriz *"aristas_ordenadas"*.

Inicialmente :

```
minj ←— 1

minx ←— aristas_ordenadas.valor [ 1 ]

para  j = 2  hasta  m  ,  hacer

    si  aristas_ordenadas.valor [ j ] <  minx  ,  entonces

        minx ←— aristas_ordenadas.valor [ j ]

        minj ←— j

    finsi

finpara
```

Cuando ya he calculado *minj* y *minx* hago el intercambio:

Sólo necesito dos variables más *"primer_nodo"* y *"segundo_nodo"* para guardar la información que se puede perder al hacer el intercambio:

primer_nodo ⟶ Guarda el *nodoi* de la arista con menor dato *"valor"*.
segundo_nodo ⟶ Guarda el *nodoj* de la arista con menor dato *"valor"*.

El dato *"valor"* de la arista más pequeña ya está guardado en *"minx"* y el dato *"fase"* siempre es *"disponible"* para todas las aristas.

Los nodos de la arista con el menor dato *"valor"* se encuentran accesibles en:

aristas_ordenadas.nodoi [minj]
aristas_ordenadas.nodoj [minj]

Luego para que no los pierda al hacer el intercambio, los guardaré:

primer_nodo ⟵ aristas_ordenadas.nodoi [minj]
segundo_nodo ⟵ aristas_ordenadas.nodoj [minj]

El primer elemento lo llevo a la posición en donde esta el elemento de menor valor

aristas_ordenadas [minj] ⟵ aristas_ordenadas [1]

En la posición del primer elemento pongo el elemento de menor valor:

aristas_ordenadas.nodoi [1] ⟵ primer_nodo

aristas_ordenadas.nodoj [1] ⟵ segundo_nodo

aristas_ordenadas.valor [1] ⟵ minx

Ampliándolo con el paso 3.2, es decir, haciendo el algoritmo completo:

```
para  i = 1  hasta  m-1  ,  hacer
        minj ⟵ i
        minx ⟵ aristas_ordenadas.valor [ i ]
        para  j = i + 1  hasta  m  ,  hacer
            si  aristas_ordenadas.valor [ j ]  <  minx  ,  entonces
                minj ⟵ j
                minx ⟵ aristas_ordenadas.valor [ j ]
            finsi
        finpara
        primer_nodo ⟵ aristas_ordenadas.nodoi [ minj ]
        segundo_nodo ⟵ aristas_ordenadas.nodoj [ minj ]
        aristas_ordenadas [ minj ] ⟵ aristas_ordenadas [ i ]
        aristas_ordenadas.nodoi [ i ] ⟵ primer_nodo
        aristas_ordenadas.nodoj [ i ] ⟵ segundo_nodo
        aristas_ordenadas.valor [ i ] ⟵ minx
finpara
```

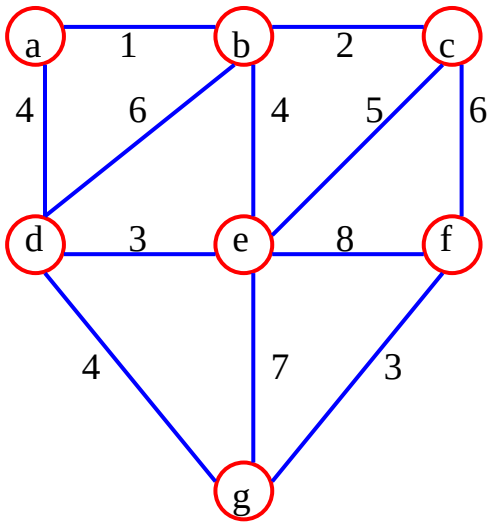
Paso 4º: Realización del bucle que nos dé la solución final

En el paso 4º, cabe destacar qué solución damos al problema de comprobar la no formación de un ciclo en el grafo solución al añadir una arista:

Inicialmente cada nodo del grafo por separado forma una componente conexa (trivial). Partiendo de esta situación inicial, solo añadimos aristas que vayan de una componente conexa a otra. Cuando añadimos una arista entre las componentes conexas B_1 y B_2 desaparecen ambas componentes conexas y se crea una nueva componente conexa $B = B_1 \, U \, B_2$.

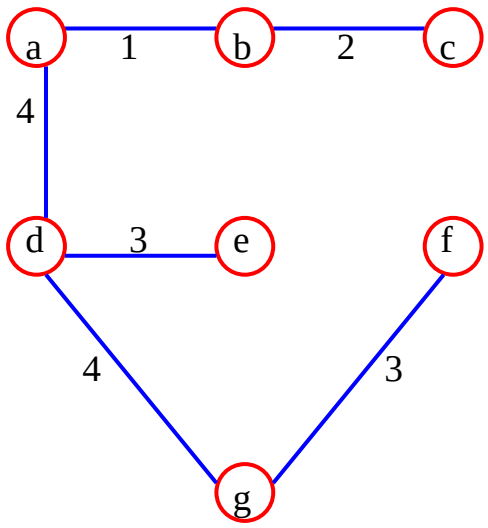
Si la arista que queremos añadir (que será la de menor valor según el algoritmo de Kruskal) la colocásemos entre dos nodos de una misma componente conexa, entonces se nos formaría siempre un ciclo porque antes de añadir la nueva arista todos los nodos de la componente conexa (por definición de componente conexa) están ya unidos entre sí por aristas.

Luego siguiendo el proceso de añadir aristas sólo entre componentes conexas distintas, al final, dichas componentes conexas se irán agrupando hasta que sólo quede una componente conexa y sus nodos sean los mismos que los nodos del grafo. En ese momento ya tendremos la solución óptima. Por ejemplo, consideramos el siguiente grafo:



Paso	Arista considerada	Componentes conexas
Iniciación	——	{a} , {b} , {c} , {d} , {e} , {f} , {g}
1	{a,b}	{a,b} , {c} , {d} , {e} , {f} , {g}
2	{b,c}	{a,b,c} , {d} , {e} , {f} , {g}
3	{d,e}	{a,b,c} , {d,e} , {f} , {g}
4	{f,g}	{a,b,c} , {d,e} , {f,g}
5	{a,d}	{a,b,c,d,e} , {f,g}
6	{b,e}	Rechazada (se forma un ciclo)
7	{d,g}	{a,b,c,d,e,f,g}

El grafo solución es:



Antes de hacer el seudocódigo de este paso 4º definimos:

aristas_ordenadas.fase [k] = “disponible”

Cuando dicha arista *"k"* no ha sido examinada todavía para ver la posibilidad de incluirla en las aristas solución. Inicialmente, como es lógico y ya hemos visto antes, todos los elementos de la matriz *"aristas_ordenadas"* tienen en su dato *"fase"* el valor *"disponible"*.

aristas_ordenadas.fase [k] = "solución"

Cuando dicha arista *"k"* ya ha sido examinada y se ha incluido en la solución óptima.

aristas_ordenadas.fase [k] = "forma_ciclo"

Cuando la arista *"k"* ya ha sido examinada y ha sido descartada de la solución por formar un ciclo.

Al finalizar el algoritmo, las aristas de la solución óptima serán las que tengan el valor *"solución"* en el dato *"fase"* de la matriz *"aristas_ordenadas"*.

Definimos la matriz *"**componente_conexa**"* de dos dimensiones y de $n \times n$ elementos siendo *"n"* el número de nodos del grafo.

Inicialmente, los valores de la matriz son:

$$\begin{pmatrix} 1 & -1 & -1 & \dots & -1 \\ 2 & -1 & -1 & \dots & -1 \\ : & : & & & : \\ : & : & & & : \\ n & -1 & -1 & \dots & -1 \end{pmatrix}$$

Cada fila representa una componente conexa y sólo forman parte de la componente conexa los nodos, que tienen valores de 1 a *n*, descartando los valores –1. Las *"n"* componentes conexas iniciales (*"n"* filas) tienen un sólo nodo en cada componente conexa, situado en la columna 1. El algoritmo parte de *"n"* componentes conexas y cada componente conexa es un nodo distinto del grafo y todos los demás valores son –1.

A medida que avanza el algoritmo y se añaden aristas a la solución óptima, la matriz va cambiando y si una componente conexa desaparece, todos los valores de su fila serán –1, luego para saber si existe una componente conexa en una fila, nos posicionamos en esa fila y comprobamos el valor de su primera columna; si es –1 no existe componente conexa en esa fila. Para las filas que si tengan una componente conexa, avanzando desde la primera columna en adelante, el primer valor –1 que nos encontremos en una fila nos indica que se han terminado los elementos de esa componente conexa y todos los valores que aparecen a partir del primer valor –1 hacia la derecha en esa fila son –1.

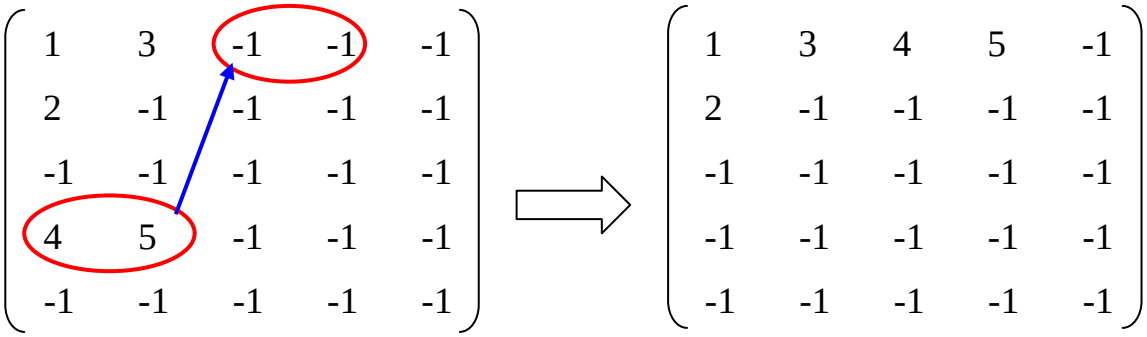
Cuando se unen dos componentes conexas en una, se une la segunda componente conexa a la primera, añadiendo sus nodos en la fila de la primera componente conexa a continuación de los nodos de la primera componente conexa y sustituyendo los valores –1 que tenía la primera componente conexa, y después se ponen valores –1 en toda la fila en donde estaba la segunda componente conexa para hacerla desaparecer.

Al finalizar el algoritmo con el conjunto de aristas solución completado, esta matriz tendrá sólo una componente conexa que es una fila cuyos miembros son todos los nodos del grafo y el resto de filas tendrán como valor –1 en todos sus elementos.

Gráficamente se ve mejor, por ejemplo, si la matriz *"componente_conexa"* en un momento dado es:

$$\begin{pmatrix} 1 & 3 & -1 & -1 & -1 \\ 2 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ 4 & 5 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{pmatrix}$$

Y hemos seleccionado una arista que une los nodos 1 y 4 como arista que pertenece al conjunto solución de aristas, tendremos que unir las componentes conexas de las filas 1 y 4 como sigue:



Cuando se añada la siguiente arista, que estará entre el nodo 2 y alguno de los nodos restantes se llegará a la solución óptima y el valor 2 correspondiente a *componente_conexa [2 , 1]* pasará a la posición *componente_conexa [1 , 5]* quedando *componente_conexa [2 , 1] = -1*.

El algoritmo para construir el seudocódigo es:

4.1.- Nos creamos *"n"* componentes conexas, es decir, nos creamos la matriz *"componente_conexa"* con sus valores iniciales.

Tenemos que encontrar una solución óptima exactamente con *n-1* aristas, siendo *"n"* el número de nodos del grafo.

El seudocódigo para crear la matriz *"componente_conexa"* es:

```
para  i = 1  hasta  n  ,  hacer
      componente_conexa [ i , 1 ]  ⟵  i
      para  j = 2  hasta  n  ,  hacer
            componente_conexa [ i , j ]  ⟵  -1
      finpara
finpara
```

4.2.- Extraigo los datos que me interesan de cada elemento de la matriz *"aristas_ordenadas"* y los guardo en las variables *"nodop"* y *"nodoq"*. Por ejemplo, para la primera arista sería:

índice ⟵ 1

nodop ⟵ aristas_ordenadas.nodoi [índice]

nodoq ⟵ aristas_ordenadas.nodoj [índice]

La variable ***"índice"*** me indica la siguiente arista de la matriz *"aristas_ordenadas"* que tengo que examinar. Las aristas están ordenadas de menor a mayor por el dato *"valor"* de cada elemento en la matriz *"aristas_ordenadas"* y

como el algoritmo de Kruskal me dice que tengo que escoger las aristas de menor a mayor valor, basta con recorrer la matriz *"aristas_ordenadas"* desde su primer elemento en adelante.

4.3.- Investigamos los nodos en los que se inserta la arista seleccionada (estos nodos los guardamos en las variables *"nodop"* y *"nodoq"*).

Miramos la componente conexa de uno de ellos, por ejemplo, *"nodop"* y en los nodos que componen dicha componente conexa buscamos el otro *"nodoq"*:

Si lo encontramos:

Descartamos la arista seleccionada, ya que se formaría un ciclo.

Si no lo encontramos:

Juntamos en la misma componente conexa las componentes conexas de los nodos que inciden en la arista seleccionada (estos nodos son *"nodop"* y *"nodoq"*) ⟹ Ponemos *"solución"* en el dato *"fase"* de la arista seleccionada ⟹ Hacemos *número_aristas_solución* = = *número_aristas_solución + 1* ⟹

⟹ Si número_aristas_solución
- = n-1 ⟹ Fin.
- < n-1 ⟹ Volver a hacer el mismo proceso para la siguiente arista.

Antes de continuar con el seudocódigo del paso 4.3 desarrollaré algunos procedimientos y funciones que me harán falta más tarde:

Procedimiento "**busca_componente_conexa**"

Como la matriz *"componente_conexa"* puede tener filas con todos sus valores –1, es decir, filas que no tienen una componente conexa, este procedimiento calcula el valor de la fila de la matriz *"componente_conexa"* en donde se encuentra una componente conexa a partir de la fila que está contenida en la variable *"fila_componente_conexa"*.

El seudocódigo es:

```
i ←—— fila_componente_conexa

para  j = i  hasta  n  ,  hacer

    si  componente_conexa [ j , 1 ]  ≠  -1  ,  entonces

            salir del para

    sino

            fila_componente_conexa  ←——  fila_componente_conexa  +  1

    finsi

finpara
```

El resultado final (la fila encontrada en donde esta la componente conexa) queda en la variable global *"fila_componente_conexa"*.

Función "**misma_componente (nodo)**"

La función booleana *"misma_componente (nodo)"* nos devolverá los valores:

- True ⟹ Si los dos nodos de la arista seleccionada están en la misma componente conexa.
- False ⟹ En caso contrario.

Examinamos la componente conexa del *"nodop"* que esta en la fila *"fila_componente_conexa"* desde la primera columna para buscar el *"nodo"* que se pasa como parámetro.

Esta función en seudocódigo es:

para j = 1 hasta n , hacer

si componente_conexa [fila_componente_conexa , j] = -1 , entonces

Cuando encontramos un valor –1 en la componente conexa ya no hay más nodos que buscar y terminamos.

misma_componente ← False

fin de la función

finsi

si componente_conexa [fila_componente_conexa , j] = nodo , entonces

Si encontramos el “nodo” en la componente conexa, terminamos.

misma_componente ← True

fin de la función

finsi

finpara

Función "**inicio**"

Nos da la posición de la primera columna con valor –1 de la componente conexa en la que se encuentra el *nodop* de la matriz *"componente_conexa"*. La fila de la componente conexa del *nodop* esta en la variable global *"componentep"*.

```
para  j = 1  hasta  n  ,  hacer
    si  componente_conexa [ componentep , j ] = -1  ,  entonces
            inicio = j
            fin de la función
    finsi
finpara
```

Función "**fin**"

Nos da el número de nodos que irán a añadirse de la componente conexa del *"nodoq"* a la componente conexa del *"nodop"*. La fila de la componente conexa del *"nodoq"* esta en la variable global *"componenteq"*.

```
contador ⟵ 0

para  j = 1  hasta  n  , hacer
    si componente_conexa [ componenteq , j ]  ≠ -1  ,  entonces
            contador ⟵ contador + 1
    si no
            fin ⟵ contador
            fin de la función
    finsi
finpara
```

Procedimiento "**unir componentes_conexas**"

Uno las componentes conexas en donde están los nodos *"nodop"* y *"nodoq"* en una sola componente conexa situada en la fila de la componente conexa del *nodop* y hago desaparecer la componente conexa de la fila en donde esta el *nodoq*.

Llamo a las funciones "inicio" y "fin":

k ← inicio

r ← fin

s ← 1

para j = k hasta k + r , hacer

componente_conexa [componentep , j] ← componente_conexa [componenteq , s]

s ← s + 1

finpara

(ahora, elimino la componente conexa del nodoq)

para j = 1 hasta r , hacer

componente_conexa [componenteq , j] ← -1

finpara

Procedimiento "**componente_nodo (nodo)**"

Encuentra la componente conexa en donde esta un nodo que se pasa como parámetro de entrada del procedimiento. Se empieza a buscar desde la fila contenida en la variable *"fila_componente_conexa"*.

```
mientras  fila_componente_conexa ≤ n  ,  hacer
    para  j = 1  hasta  n  ,  hacer
        si  componente_conexa [fila_componente_conexa , j ] < 0  , entonces
            (cuando halla recorrido todos los valores de la fila no negativos
            sin éxito, salimos del bucle para buscar en otra fila)
            salir del para
        finsi
        si  componente_conexa [fila_componente_conexa , j ] = nodo  ,  entonces
            (En "fila_componente_conexa" tengo la fila
            de la componente conexa del nodo)
            salir del procedimiento
        finsi
    finpara
    fila_componente_conexa  ←  fila_componente_conexa + 1
    busca_componente_conexa
finmientras
```

A partir de aquí, hago un resumen del pseudocódigo visto hasta ahora y continuaré haciendo el pseudocódigo que falta (punto 4.3).

Para pasar las aristas de la matriz *"valor_arista"* a la matriz *"aristas_ordenadas"*:

```
m ← 0
para i = 1 hasta n , hacer
    para j = i + 1 hasta n , hacer
        si valor_arista [ i , j ] ≥ 0 , entonces
            m ← m + 1
            aristas_ordenadas.nodoi [ m ] ← i
            aristas_ordenadas.nodoj [ m ] ← j
            aristas_ordenadas.valor [ m ] ← valor_arista [ i , j ]
            aristas_ordenadas.fase [ m ] ← "disponible"
        finsi
    finpara
finpara
```

Además, he completado el resto de datos de cada elemento de la matriz *"aristas_ordenadas"*.

"m" es el número de elementos de dicha matriz y e inicializado todos los datos *"fase"* de cada elemento de la matriz al valor *"disponible"*.

Ahora , ordeno la matriz *"aristas_ordenadas"* de menor a mayor por el dato *"valor"* de cada elemento:

```
para  i = 1  hasta  m – 1  ,  hacer
    minj ⟵ i
    minx ⟵ aristas_ordenadas.valor [ i ]
    para  j = i + 1  hasta  m  ,  hacer
        si  aristas_ordenadas [ j ].valor  <  minx  ,  entonces
            minj ⟵ j
            minx ⟵ aristas_ordenadas.valor [ j ]
        finsi
    finpara
    primer_nodo ⟵ aristas_ordenadas.nodoi [ minj ]
    segundo_nodo ⟵ aristas_ordenadas.nodoj [ minj ]
    aristas_ordenadas [ minj ] ⟵ aristas_ordenadas [ i ]
    aristas_ordenadas.nodoi [ i ] ⟵ primer_nodo
    aristas_ordenadas.nodoj [ i ] ⟵ segundo_nodo
    aristas_ordenadas.valor [ i ] ⟵ minx
finpara
```

Me creo la matriz “componente_conexa”:

```
para   i = 1   hasta   n   ,   hacer
        componente_conexa [ i , 1 ]  ⟵  i
        para   j = 2   hasta   n   ,   hacer
                componente_conexa [ i , j ]  ⟵  -1
        finpara
finpara
```

El seudocódigo del punto 4.3 es:

Nos creamos la variable “numero_aristas_solucion” y la inicializamos a cero

```
numero_aristas_solucion ⟵  0
```

Cuando esta variable tenga un valor igual a “n-1” ya hemos terminado y el conjunto de aristas óptimo será los elementos de la matriz “aristas_ordenadas” que tengan el valor “solución” en su dato “fase”.

```
índice  ⟵  0

mientras   numero_aristas_solución  <  n – 1    ,   hacer

        índice  ⟵  índice + 1
        nodop  ⟵  aristas_ordenadas.nodoi [ índice ]
        nodoq  ⟵  aristas_ordenadas.nodoj [ índice ]
```

Busco la primera componente conexa a partir de la fila 1 en la matriz “componente_conexa”

fila_componente_conexa ⟵ 1

busca_componente_conexa

Busco la componente conexa, que es una fila de la matriz "componente_conexa", en donde esta el nodop y la guardo en la variable "fila_componente_conexa".

componente_nodo (nodop)

A continuación examino dicha componente conexa desde el principio para saber si el otro nodoq esta también en la misma componente conexa.

si misma_componente (nodoq) , entonces

aristas_ordenadas.fase [índice] ⟵ "forma_ciclo"

si no

componentep ⟵ fila_componente_conexa

Se busca donde esta la fila de la otra componente conexa del nodoq y esa fila la guardaré en la variable "componenteq"

Busco una componente conexa a partir de la fila 1 en la matriz "componente_conexa"

fila_componente_conexa ⟵ 1

busca_componente_conexa

Busco la componente conexa en donde esta el nodoq y la guardo en la variable "fila_componente_conexa".

componente_nodo (nodoq)

componenteq ⟵ fila_componente_conexa

Uno las componentes conexas en donde están los nodos "nodop" y "nodoq" en una sola componente conexa

unir_componentes_conexas

Añado la arista elegida al conjunto de aristas solución

aristas_ordenadas.fase [índice] ⟵ "solución"

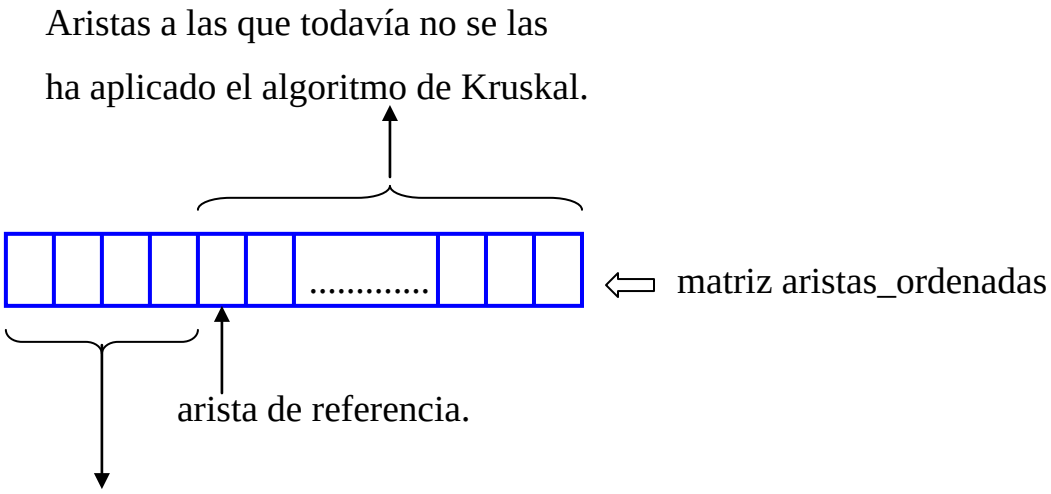
número_aristas_solución ⟵ número_aristas_solución + 1

finsi

finmientras

2.2.2.3.- Implementación del algoritmo a través de la intervención del usuario seleccionando aristas sobre un grafo

Para un caso general, nos podemos encontrar con la siguiente situación:



Denomino *"**arista de referencia"*** a la primera arista, en orden, de la matriz *aristas_ordenadas* a la que todavía no se la ha aplicado el algoritmo de Kruskal.

El usuario puede seleccionar en el grafo una arista cuya posición en la matriz *"aristas_ordenadas"* puede ser:

Caso 1º: Anterior a la arista de referencia.

Caso 2º: Coincide con la arista de referencia.

Caso 3º: Posterior a la arista de referencia.

Dependiendo de esta posición, desarrollaré un proceso u otro, es decir, llamaré:

Proceso1 ⟹ Desarrolla el caso 1º.

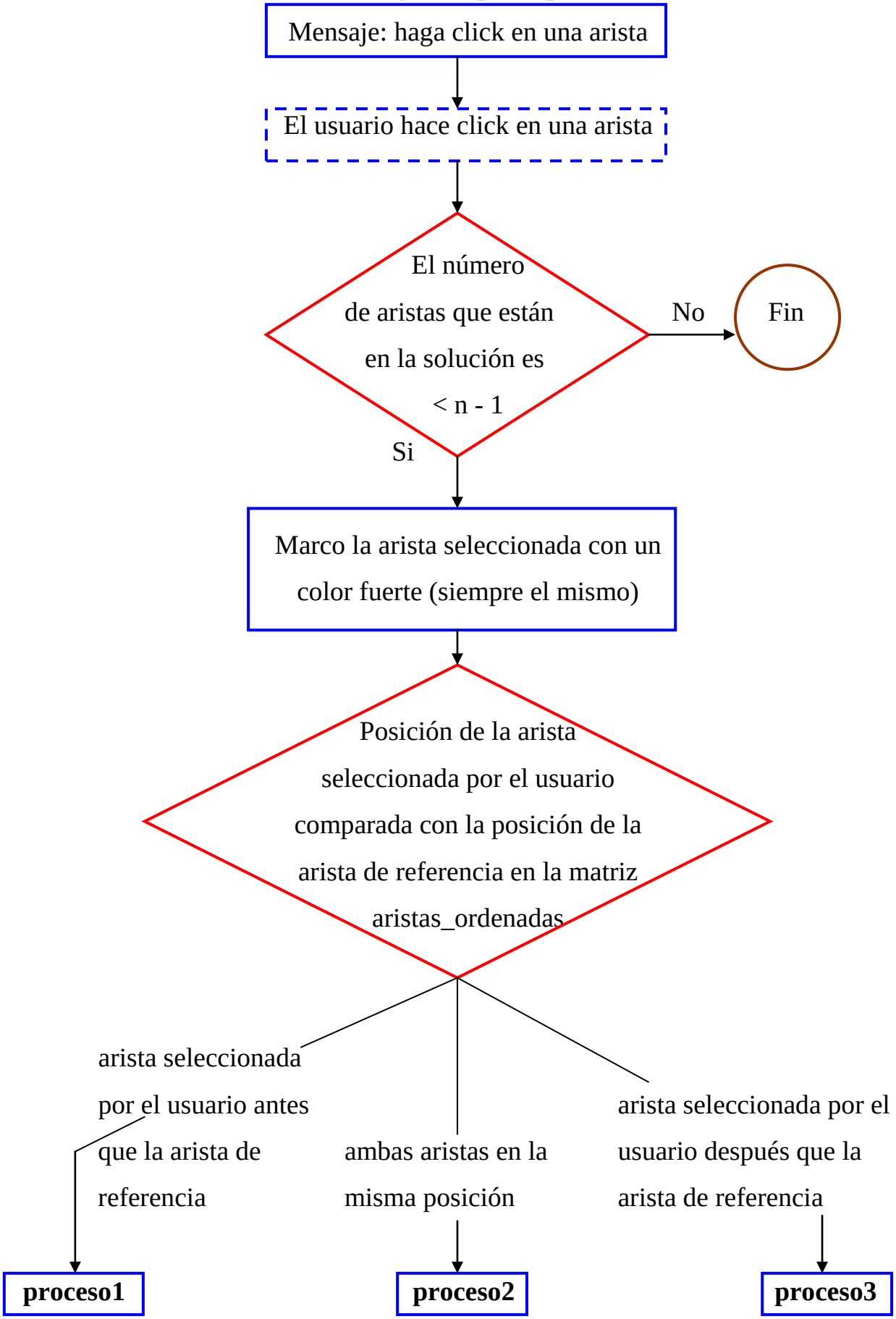
Proceso2 ⟹ Desarrolla el caso 2º.

Proceso3 ⟹ Desarrolla el caso 3º.

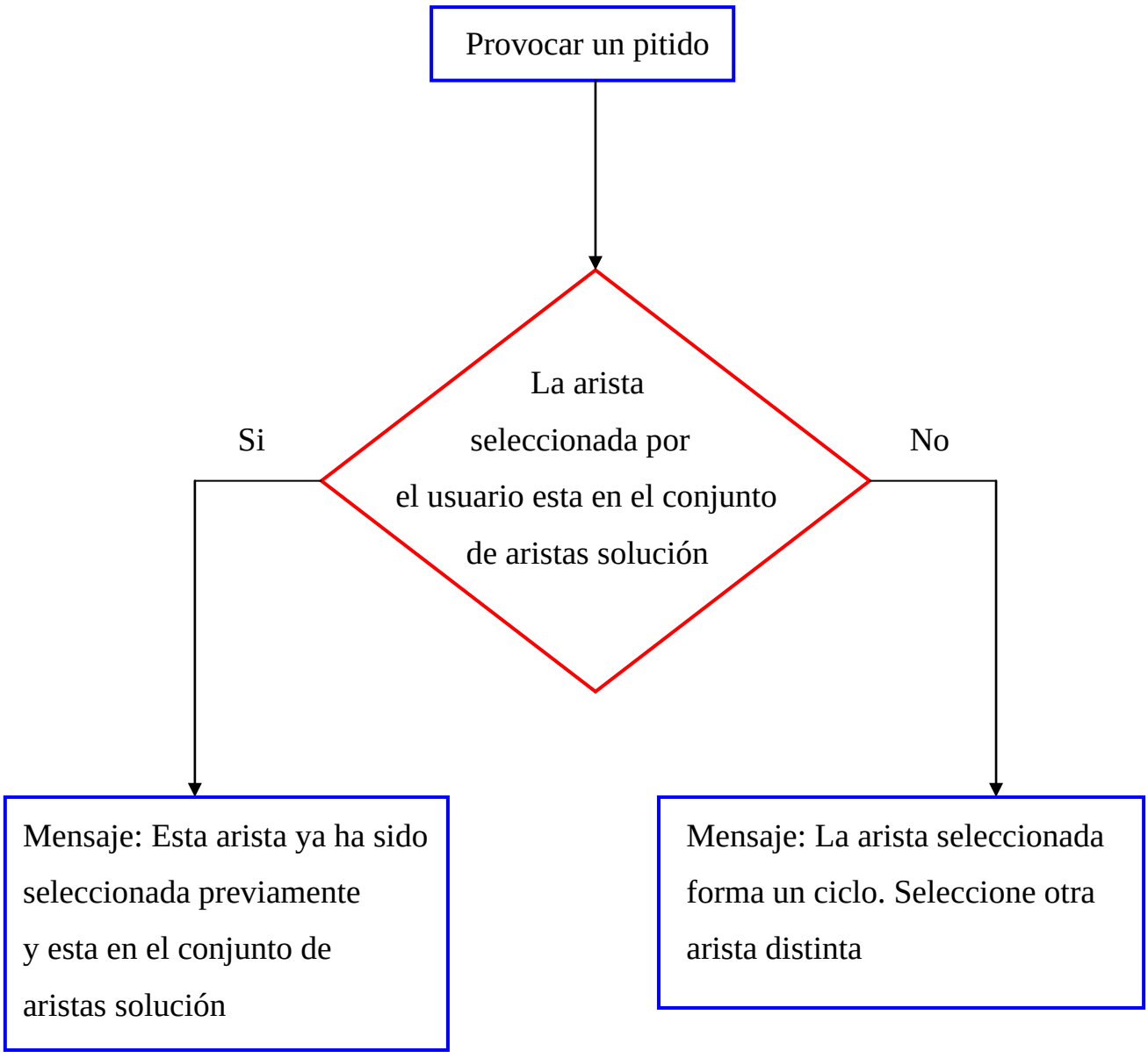
En esta división baso el método que voy a emplear. Primero desarrollaré los organigramas y después el seudocódigo correspondiente a ellos.

Dentro del seudocódigo me van a servir funciones y procedimientos empleados en la primera parte cuando resolví la solución automática.

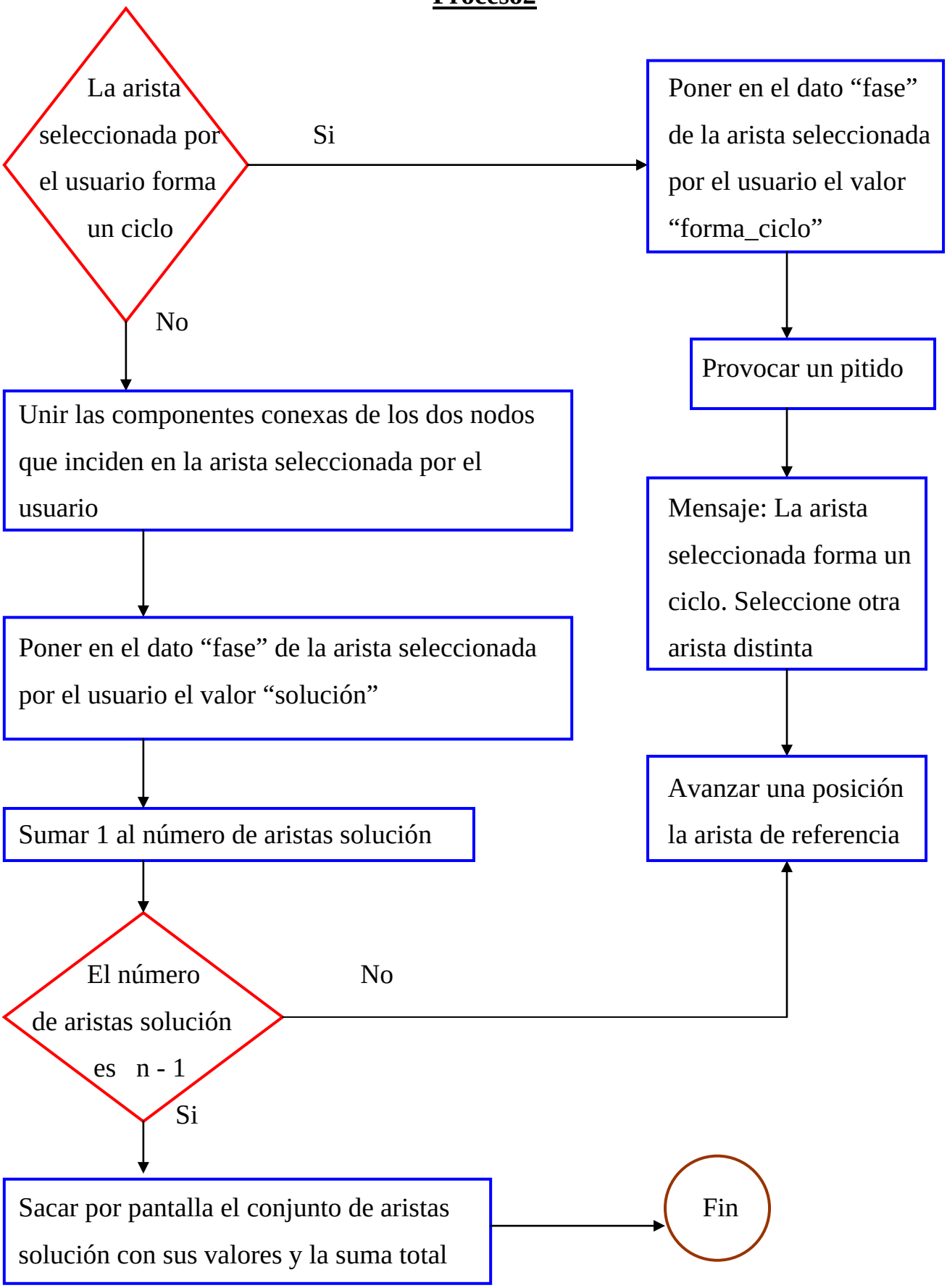
Programa principal



Proceso1


Provocar un pitido
La arista seleccionada por el usuario esta en el conjunto de aristas solución
Si
No
Mensaje: Esta arista ya ha sido seleccionada previamente y esta en el conjunto de aristas solución
Mensaje: La arista seleccionada forma un ciclo. Seleccione otra arista distinta

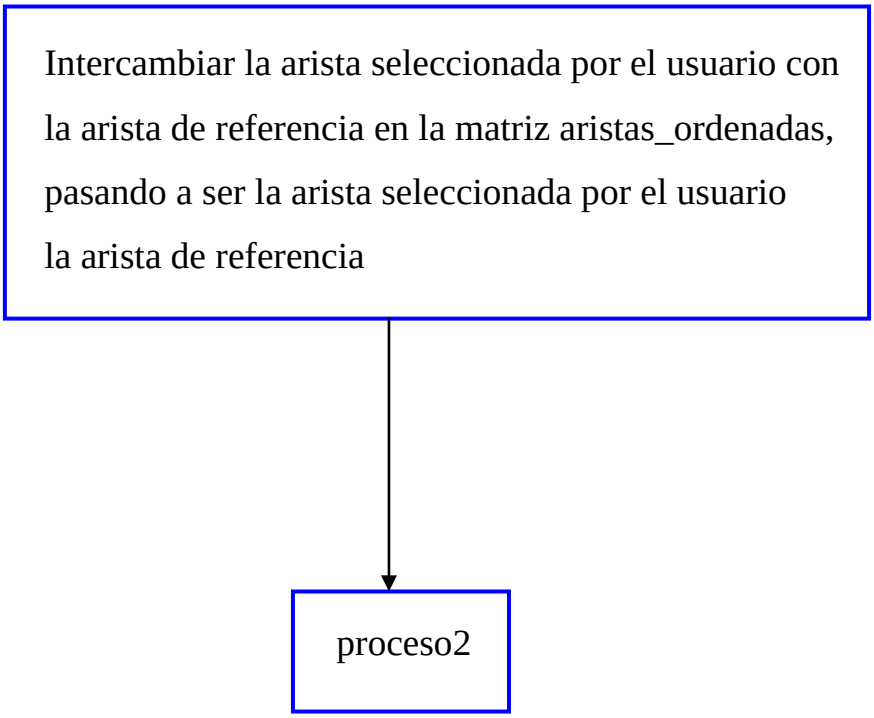
Proceso2


La arista seleccionada por el usuario forma un ciclo
Si
No
Poner en el dato "fase" de la arista seleccionada por el usuario el valor "forma_ciclo"
Provocar un pitido
Mensaje: La arista seleccionada forma un ciclo. Seleccione otra arista distinta
Avanzar una posición la arista de referencia
Unir las componentes conexas de los dos nodos que inciden en la arista seleccionada por el usuario
Poner en el dato "fase" de la arista seleccionada por el usuario el valor "solución"
Sumar 1 al número de aristas solución
El número de aristas solución es n - 1
No
Si
Sacar por pantalla el conjunto de aristas solución con sus valores y la suma total
Fin

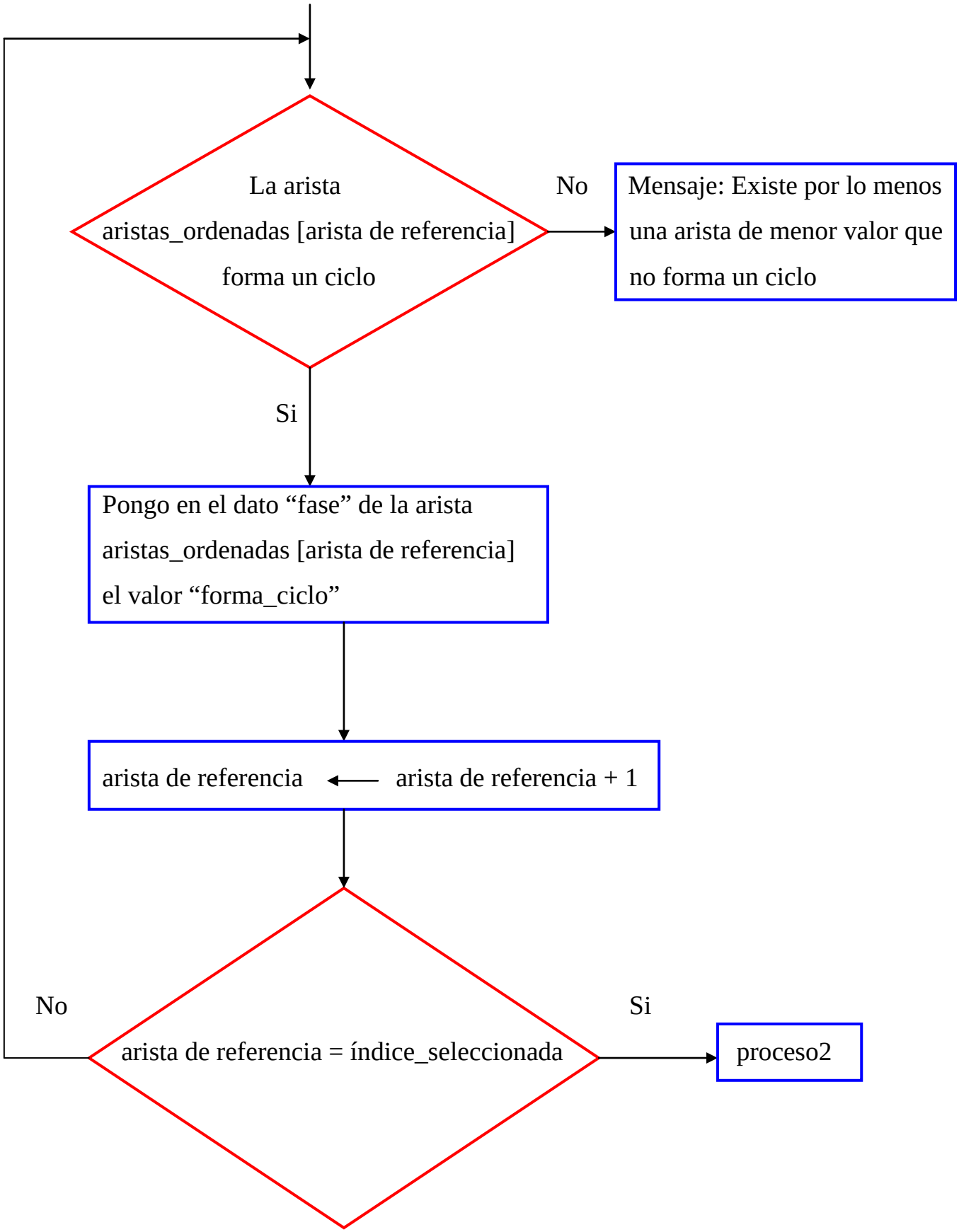
Proceso3


Las aristas que van
desde la arista de referencia
hasta la arista seleccionada
por el usuario tienen el
mismo dato “valor”
Si, todas
Unas si y otras no
No, ninguna
desarrollo1
desarrollo3
desarrollo2

Desarrollo1

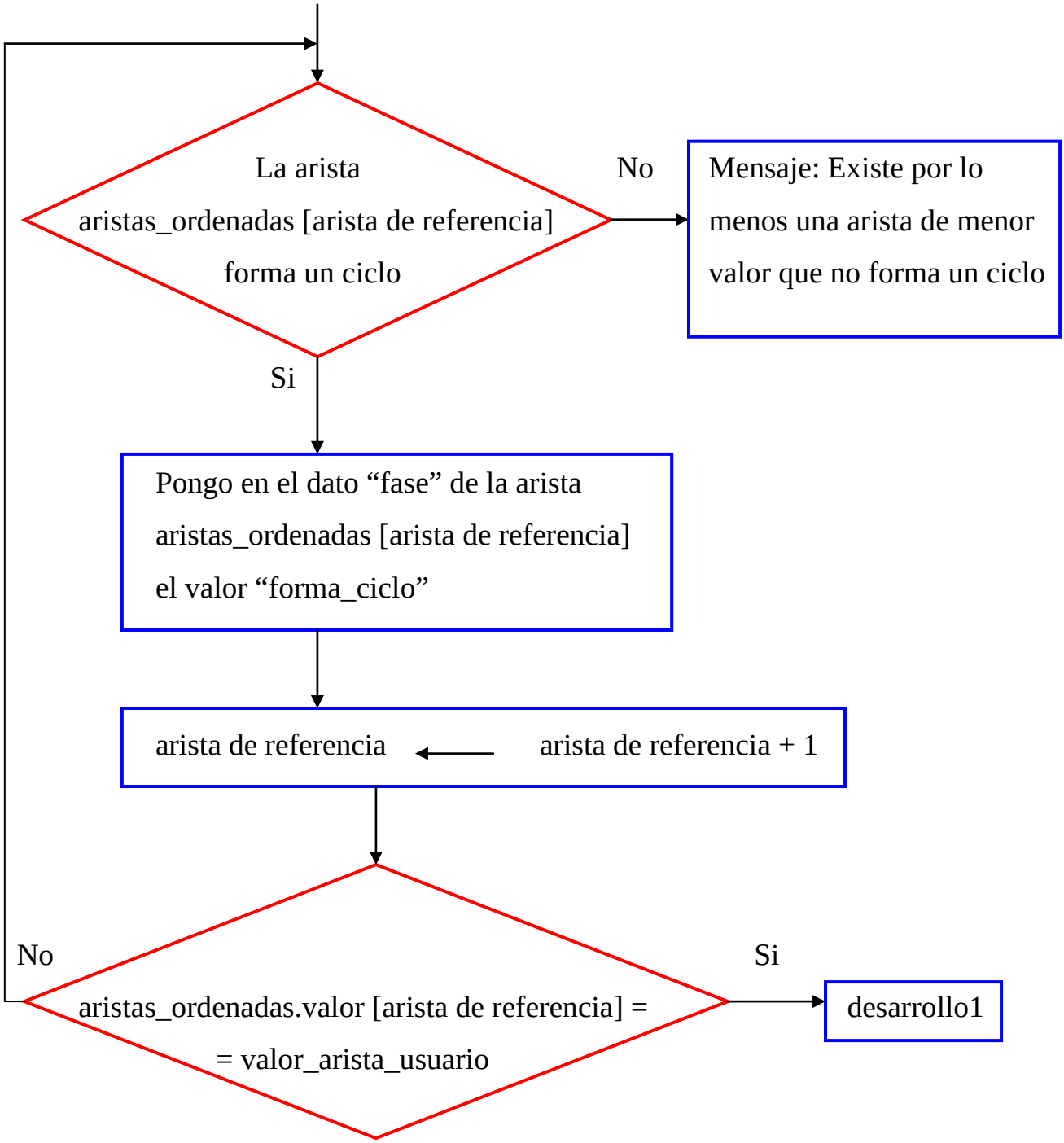


Desarrollo2



Desarrollo3

Las aristas con distinto dato *"valor"* estarán obligatoriamente al principio por tener la matriz *"aristas_ordenadas"* sus elementos colocados de menor a mayor valor, luego primero compruebo si alguna de ellas, como mínimo, pertenece al conjunto de aristas solución.



Programa principal

arista_de_referencia ⟵ 1

numero_aristas_solución ⟵ 0

Mensaje: “Haga click en una arista”

Evento click del usuario en una arista

Si numero_aristas_solución < n – 1 , entonces

(Los datos de la arista seleccionada los introduciré en las variables siguientes)

nodop ⟵ un nodo incidente en la arista

nodoq ⟵ el otro nodo incidente en la arista

valor_arista_usuario ⟵ valor de la arista

“Marco la arista seleccionada con un color fuerte (siempre el mismo)”

(Calculo la posición de la arista seleccionada en la matriz aristas_ordenadas)

calcular_índice_arista_seleccionada

En la variable “índice_seleccionada” tengo dicha posición. Según este situada la arista seleccionada por el usuario respecto a la posición de la arista de referencia en la matriz aristas_ordenadas, realizaré uno de los tres procesos siguientes

```
seleccionar  opción  índice_seleccionada
        opción 1 :  <  arista_de_referencia
                proceso1
        opción 2 :  =  arista_de_referencia
                proceso2
        opción 3 :  >  arista_de_referencia
                proceso3
    finseleccionar
finsi
```

Procedimiento “calcular_índice_arista_seleccionada”

Como es muy probable que el usuario seleccione aristas que no ha seleccionado antes, empezaré a buscar en la matriz *aristas_ordenadas* a partir de la arista de referencia, y si se termina la matriz y no encuentro la arista seleccionada por el usuario, entonces buscaré en el trozo de matriz que queda, es decir, desde la primera posición hasta la posición de la arista de referencia.

Primera parte de la búsqueda:

```
para   i = arista_de_referencia   hasta   m   ,   hacer
        busqueda_índice [ i ]
finpara
```

Si todavía no se ha encontrado la arista seleccionada, hago la segunda parte de la búsqueda:

```
para   i = 1   hasta   arista_de_referencia   - 1   ,   hacer
        busqueda_índice [ i ]
finpara
```

Procedimiento “búsqueda_índice [i]

si ((aristas_ordenadas.nodoi [i] = nodop y aristas_ordenadas.nodoj [i] = nodoq) ó (aristas_ordenadas.nodoi [i] = nodoq y aristas_ordenadas.nodoj [i] = nodop)) , entonces

índice_seleccionada ⟵ i

salir del procedimiento “calcular_índice_arista_seleccionada”

finsi

Proceso1

“provocar un pitido”

si aristas_ordenadas.fase [índice_seleccionada] = “solución” , entonces

Mensaje: ”Esta arista ya ha sido seleccionada previamente y esta en el conjunto de aristas solución”

sino

Mensaje: “La arista seleccionada forma un ciclo. Seleccione otra arista distinta”

finsi

Proceso2

fila_componente_conexa ← 1
busca_componente_conexa
componente_nodo (nodop)

si misma_componente (nodoq) , entonces

aristas_ordenadas.fase [índice_seleccionada] ← "forma_ciclo"
"provocar un pitido"
Mensaje: "La arista seleccionada forma un ciclo. Seleccione otra arista distinta."

sino

componentep ← fila_componente_conexa
fila_componente_conexa ← 1
busca_componente_conexa
componente_nodo (nodoq)
componenteq ← fila_componente_conexa
unir_componentes_conexas

aristas_ordenadas.fase [índice_seleccionada] ← "solución"

numero_aristas_solución ← número_aristas_solución + 1

si número_aristas_solución = n – 1 , entonces
resultados_finales
finsi

finsi
arista_de_referencia ← arista_de_referencia + 1

Procedimiento **“resultados_finales”**

```
suma ← 0

para i = 1 hasta m , hacer

    si aristas_ordenadas.fase [ i ] = “solución” , entonces

        “Sacar por pantalla la arista”:

            aristas_ordenadas.nodoi [ i ]

            aristas_ordenadas.nodoj [ i ]

            aristas_ordenadas.valor [ i ]

            suma ← suma + aristas_ordenadas.valor [ i ]

    finsi

    si ( aristas_ordenadas.fase [ i ] = “disponible”
                        ó
                    ( i = m ) ) , entonces

        “Sacar por pantalla la “suma”

        salir del procedimiento “proceso2”

    finsi

finpara
```

Proceso3

```
todas_mismo_valor ←— "True"
todas_con_valor_distinto ←— "True"
para i = arista_de_referencia hasta índice_seleccionada – 1 , hacer
    si aristas_ordenadas.valor [ i ] ≠ valor_arista_usuario , entonces
        todas_mismo_valor ←— "False"
    sino
        todas_con_valor_distinto ←— "False"
    finsi
finpara
si todas_mismo_valor , entonces
    desarrollo1
sino
    si todas_con_valor_distinto , entonces
        desarrollo2
    sino
        desarrollo3
    finsi
finsi
```

Procedimiento **“desarrollo1”**

(Intercambio las aristas)

aux ⟵ aristas_ordenadas [arista_de_referencia]

aristas_ordenadas [arista_de_referencia] ⟵ aristas_ordenadas [índice_seleccionada]

aristas_ordenadas [índice_seleccionada] ⟵ aux

índice_seleccionada ⟵ arista_de_referencia

(Y después paso al proceso 2)

proceso2

Procedimiento **“desarrollo2”**

```
k ←—— arista_de_referencia
para  i = k  hasta  índice_seleccionada – 1  ,  hacer
        nodopaux ←—— aristas_ordenadas.nodoi [ i ]
        nodoqaux ←—— aristas_ordenadas.nodoj [ i ]
        fila_componente_conexa ←—— 1
        busca_componente_conexa
        componente_nodo (nodopaux)
    si  misma_componente (nodoqaux)  ,  entonces
            aristas_ordenadas.fase [ i ]  ←——  “forma_ciclo”
            arista_de_referencia  ←——  arista_de_referencia + 1
    sino
            “Mensaje: Existe por lo menos una arista de menor valor que no
            forma un ciclo.”
            salir del procedimiento “desarrollo2”
    finsi
finpara
```

(ahora coinciden “arista_de_referencia” e “índice_seleccionada”)

proceso2

Procedimiento **“desarrollo3”**

```
k ←—— arista_de_referencia

para  i = k  hasta  índice_seleccionada – 1  ,  hacer

      nodopaux ←—— aristas_ordenadas.nodoi [ i ]
      nodoqaux ←—— aristas_ordenadas.nodoj [ i ]

      fila_componente_conexa ←—— 1

      busca_componente_conexa

      componente_nodo (nodopaux)

      si  misma_componente (nodoqaux)  ,  entonces

            aristas_ordenadas.fase [ i ]  ←—— “forma_ciclo”

            arista_de_referencia  ←——  arista_de_referencia + 1

            si  [ aristas_ordenadas.valor [arista_de_referencia ] =
                        = valor_arista_usuario ]  ,  entonces

                  desarrollo1

                  salir del procedimiento “desarrollo3”

            finsi

      sino

            Mensaje: “Existe por lo menos una arista de menor valor que
                        no forma un ciclo”

            salir del procedimiento “desarrollo3”

      finsi

finpara
```

2.2.3.- Implementación del algoritmo de Kruskal con todas las soluciones (solución automática)

El desarrollo de esta parte del proyecto lo voy a dividir en cuatro fases:

En la primera fase, haré un estudio de las situaciones que se pueden presentar y de cómo resolverlas. Explicaré que soportes de datos voy a necesitar y el algoritmo general sin dar detalles en profundidad sobre él, sólo para comprender el proceso que voy a seguir y que detallaré mejor en las siguientes fases.

En la segunda fase, haré un estudio sobre las aristas que tienen el mismo dato *"valor"* y después implementaré en pseudocódigo (sólo para este grupo de aristas) la solución dada sin tener en cuenta las demás aristas del grafo, es decir, daré una solución parcial. Antes de escribir este pseudocódigo realizaré sus organigramas funcionales para comprender mejor y tener una visión global de todos los procesos.

En la tercera fase, resolveré la solución completa tratando todas las aristas del grafo. Realizaré un organigrama funcional que explique los pasos que voy a hacer según las posibles situaciones que se puedan presentar, y después implementaré en pseudocódigo todas las soluciones óptimas que existan siguiendo la estructura del organigrama.

En la cuarta y última fase, resolveré los problemas que pueden aparecer cuando se producen un gran número de soluciones. Esto obliga a investigar y cambiar parte del programa que, aunque ya funciona bien, se tiene que adaptar a la posible situación de resolver el algoritmo de Kruskal cuando el grafo genera excesivas soluciones.

Utilizaré funciones y procedimientos ya vistos en el apartado anterior cuando se trató solamente una solución pero volveré a escribir algunos de ellos ya que el programa ahora va a ser más complicado: necesita añadir algunos parámetros de entrada a dichos procedimientos y funciones y la matriz *"aristas_ordenadas"* pasa de una a dos dimensiones.

2.2.3.1.- Situaciones posibles, algoritmo general y estructuras de datos.

En la matriz *"aristas_ordenadas"* las aristas están ordenadas de menor a mayor por el dato *"valor"*, luego siguiendo esta ordenación de aristas y aplicándolas el algoritmo de Kruskal, si nos encontramos más de una arista con el mismo valor debo calcular el número de ellas que tiene ese valor y hacer el siguiente estudio sólo para ese grupo de aristas con el mismo dato *"valor"*.

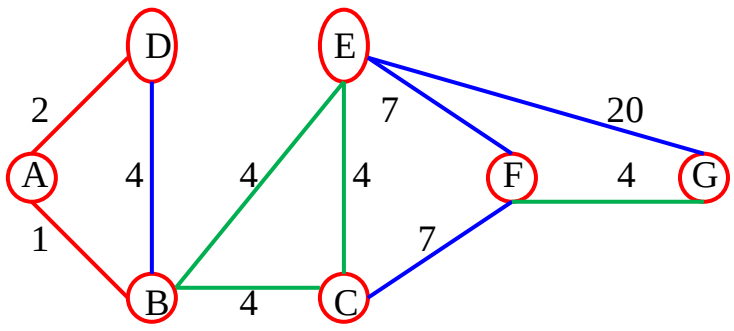
1.- Si una arista aparece en todas las soluciones posibles haciendo todas las posibles combinaciones en orden de elección sobre dicho grupo de aristas, dicha arista estará en todas las soluciones óptimas y no generará más soluciones.

2.- Si una arista cualquiera de este grupo es elegida en primer lugar y forma un ciclo, entonces nunca estará incluida en ninguna solución óptima.

3.- Las aristas que unas veces formen parte de una solución óptima y otras no, según el orden en que son elegidas, serán las que originen varias soluciones óptimas. Tengo que probar todas las posibles combinaciones en orden de elección de estas aristas (las aristas de los puntos 1 y 2 no cuentan) para componer todas las posibles soluciones óptimas.

4.- Para las aristas del punto 3, hay que descartar las posibles soluciones repetidas, es decir soluciones con las mismas aristas pero cambiadas de orden.

Estos cuatro puntos se entenderán muy bien con el siguiente ejemplo. Dado el grafo:



Si el algoritmo de Kruskal se encuentra en un punto en el que el conjunto de aristas solución es *{{A , B} , {A , D}}*, y ahora tengo que considerar el grupo de cinco aristas de valor 4:

La arista *{B , D}* nunca estará en una solución óptima porque siempre formaría un ciclo (caso 2º).

La arista *{F , G}* siempre estará en cualquier solución óptima (caso 1º).

Las aristas *{{B ,C} , {B , E} , {C , E}}* formarán parte de una u otra solución óptima dependiendo del orden en que se seleccionen (caso 3º).

Por último, cuando ya tengamos todas las posibles soluciones, hay que descartar las que tengan las mismas aristas pero en distinto orden, por ejemplo, la solución:

..... , *{B , C} , {B , E},* .....

es la misma solución que

..... , *{B , E} , {B , C},* .....

con lo que tenemos que descartar una de las dos soluciones (caso 4º).

Soporte para almacenar todas las soluciones y algoritmo general

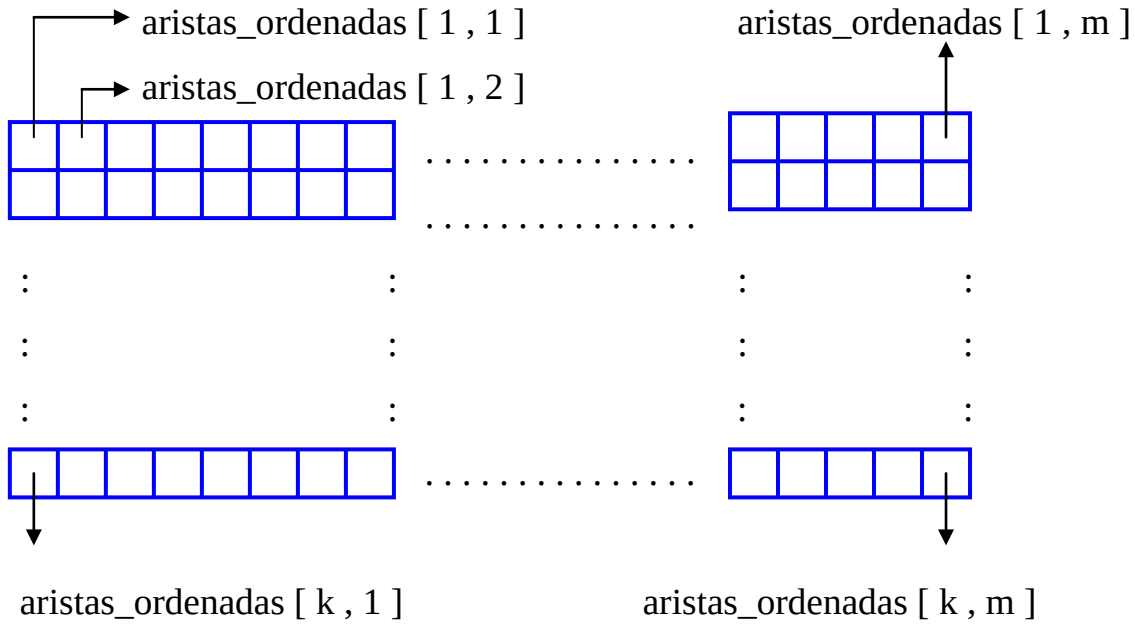
Lo primero que hay que definir es donde voy a almacenar todas las posibles soluciones. Hasta ahora, he utilizado la matriz de una dimensión ***"aristas_ordenadas"*** y voy a seguir utilizando esta matriz como soporte para guardar todas las soluciones simplemente ampliando en uno su número de dimensiones. Es decir, empezaré con una sola fila y al final del proceso tendré una matriz de *"k"* filas y *"m"* columnas, siendo *"k"* el número de soluciones óptimas únicas y *"m"* el número de aristas del grafo.

Llamaré ***"fila activa"*** a la fila de la matriz *"aristas_ordenadas"* sobre la que se está aplicando el algoritmo de Kruskal.

Esta matriz se creará inicialmente, como ya he dicho, con una sola fila y cada vez que se producen nuevas soluciones óptimas únicas, ampliará su número de filas en un

valor igual al número de soluciones óptimas producidas menos una (ya que la fila activa cuenta también como una solución). Es decir, cada fila representa una solución óptima distinta.

Cada fila de la matriz contendrá todas las aristas del grafo y cuando a dichas aristas se les aplique el algoritmo de Kruskal, el valor *"solución"* del dato *"fase"* de cada arista nos dirá que aristas están en la solución óptima representada por esa fila de la matriz *"aristas_ordenadas"*.



La variable ***"fila"*** contendrá el valor de la fila activa a cuyos elementos se les está aplicando el algoritmo de Kruscal y la variable ***"columna"*** marcará la posición de la columna (es decir, de la arista) en la que estamos en la fila activa.

Inicialmente, partimos de la fila uno y si el grafo sólo tiene una solución óptima no se crean más filas y todo el proceso se realiza sobre la fila uno.

Si el grafo tiene más de una solución óptima, de esta primera fila surgirán más filas y a su vez, de cualquier fila nueva creada también podrán surgir más filas, quedando la matriz *"aristas_ordenadas"* al final del proceso, con un número de filas igual al número de soluciones óptimas, y así, basta con recorrer cada fila y mirar si el dato

"solución" esta en el campo *"fase"* de cada arista para obtener las aristas solución de la fila de la matriz *"aristas_ordenadas"* que estamos recorriendo.

Los elementos de una nueva fila recién creada tendrán los siguientes datos:

Desde la primera arista hasta el grupo de aristas con el mismo valor, la información de las aristas es la misma que contiene la fila activa.

Las aristas con el mismo dato *"valor"* tendrán los datos de aplicar el algoritmo de Kruskal sobre ellas en un determinado orden de selección.

Y por último, las aristas que están detrás del grupo de aristas con el mismo valor serán iguales a las aristas de la fila activa que ocupan las mismas posiciones (columnas) con su dato *"disponible"* en el campo *"fase"*.

Al proceso de creación de más filas se llega cuando se produce la situación del punto 3 anteriormente vista y descartando las soluciones repetidas (punto 4) nos quedan varias soluciones.

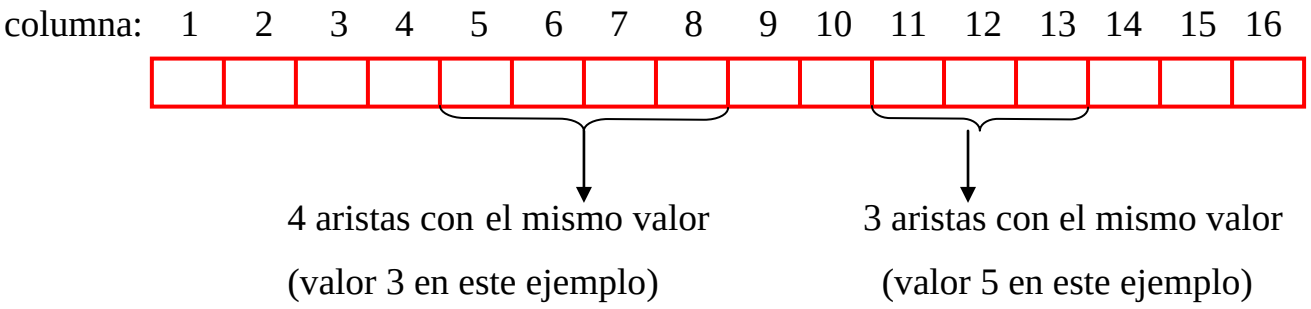
Cuando se calcula la última arista de una solución óptima aplicando el algoritmo de Kruskal sobre las aristas de una fila, se pasa a la siguiente fila, si existe, y si no existe se termina el proceso. En la nueva fila, inicializamos de nuevo la matriz *"componente_conexa"* y vamos recorriendo la fila. Las aristas que tienen en el dato *"fase"* el valor *"solución"* van reconstruyendo dicha matriz y cuando nos encontramos una arista con el dato *"disponible"* en su campo *"fase"* comenzamos a aplicar el algoritmo de Kruskal de esa arista en adelante.

Para ver más claras estas explicaciones, lo mejor es seguir un ejemplo. Supongamos que tenemos inicialmente una matriz *"aristas_ordenadas"* de una sola fila con los siguientes valores en las aristas (columnas):

columna	valor
1	1,5
2	2
3	2,3
4	2,5
5	3
6	3
7	3
8	3

columna	valor
9	3,6
10	4
11	5
12	5
13	5
14	6,2
15	6,5
16	6,7

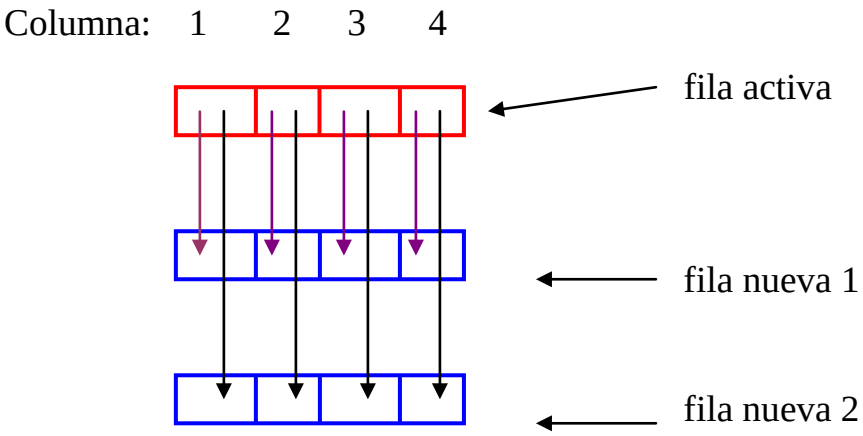
La fila activa (cuando empezamos el proceso sólo existe una fila) tendrá la forma:



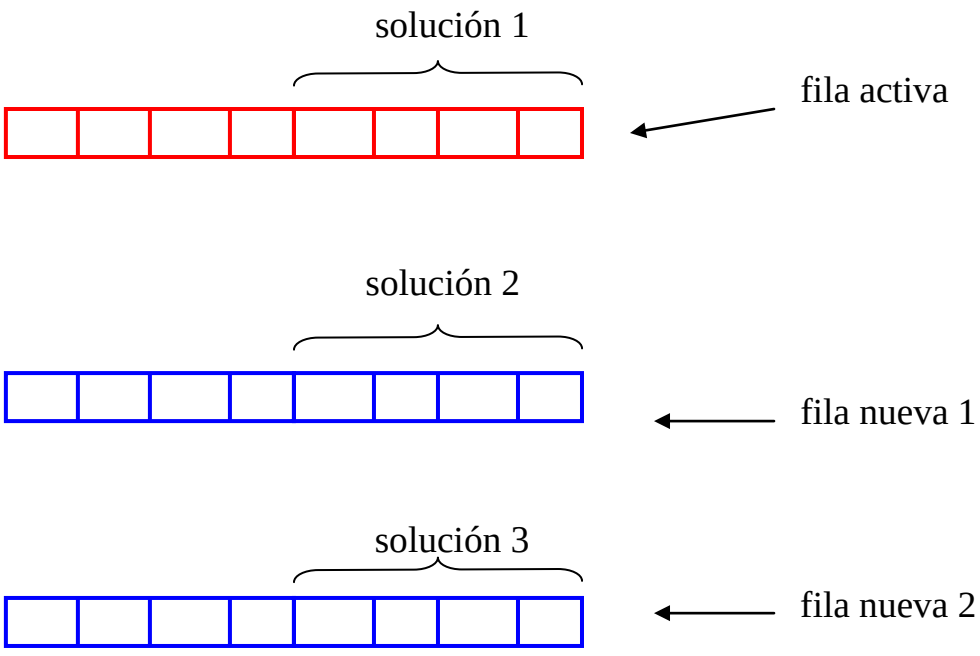
Aplicamos el algoritmo de Kruskal a esta única fila y cuando llagamos al primer grupo de aristas con el mismo valor (de la columna 5 a la 8) descartando las aristas que nunca están en ninguna solución y las que siempre están en todas las soluciones, nos quedan, por ejemplo, tres soluciones únicas posibles, luego tenemos que crear dos nuevas filas.

Se ha aplicado el algoritmo de Kruskal en la fila activa hasta la arista de la columna 8. En las aristas 1, 2, 3, 4, de esta fila activa tenemos *"forma_ciclo"* o

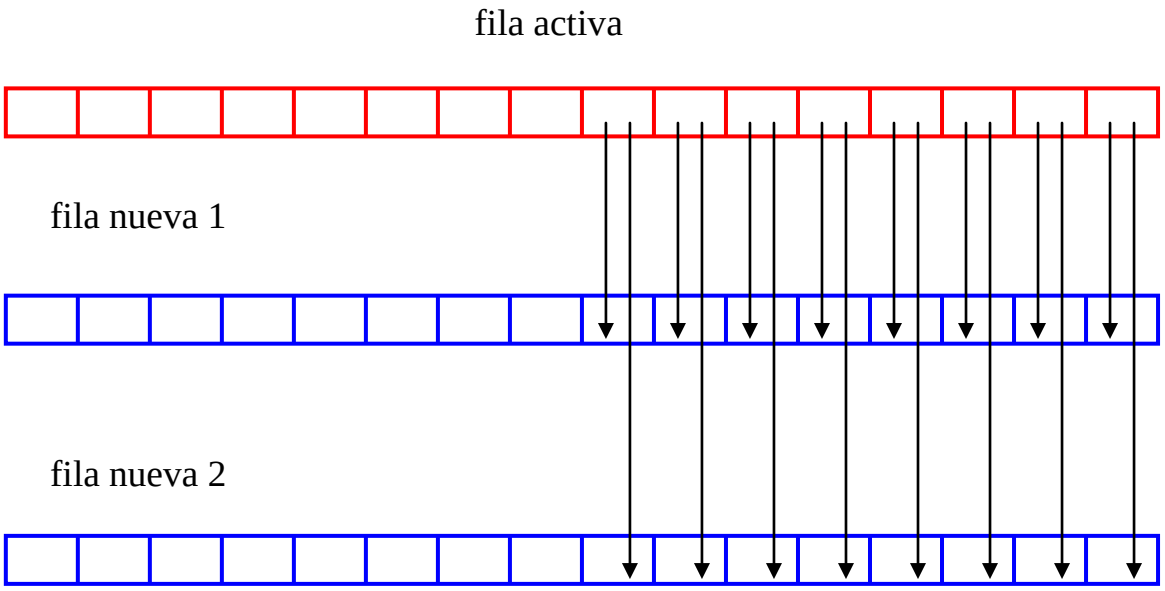
"solución" en sus datos *"fase"*. Estas cuatro aristas se copian idénticas con toda su información en las cuatro primeras aristas de las dos filas nuevas:



De la arista 5 a la 8 tenemos tres posibles soluciones (es decir, los datos *"fase"* de estas aristas son distintos para cada solución), luego en la fila activa ponemos una solución y en las dos filas nuevas ponemos las otras dos soluciones:



Por último, sólo nos queda añadir las aristas de la 9 a la 16 de la fila activa a las dos filas nuevas. A estas aristas no se las ha aplicado el algoritmo de Kruskal y no han sufrido ningún cambio (siguen teniendo *"disponible"* en su dato *"fase"*).

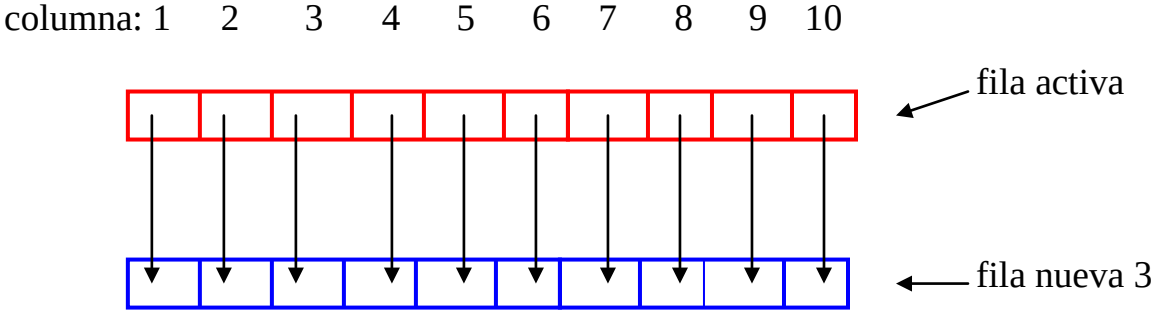


Una vez completado el algoritmo de Kruskal hasta la arista 8, seguimos aplicando dicho algoritmo desde la fila activa (que sigue siendo la misma) y dejamos aparcadas las otras dos filas nuevas. Cuando el algoritmo de Kruskal llega a las aristas 11, 12, 13, se vuelve a repetir el proceso y si, por ejemplo, obtenemos dos soluciones nuevas, hacemos lo siguiente:

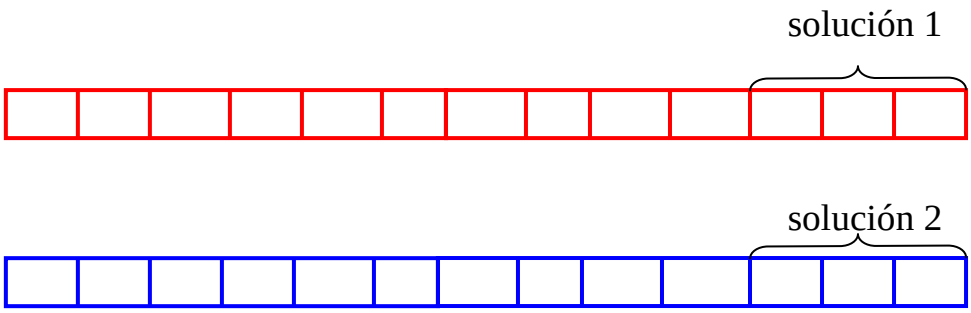
Nos creamos una nueva fila 3.

Las filas nuevas 1 y 2 no se modifican en todo este proceso.

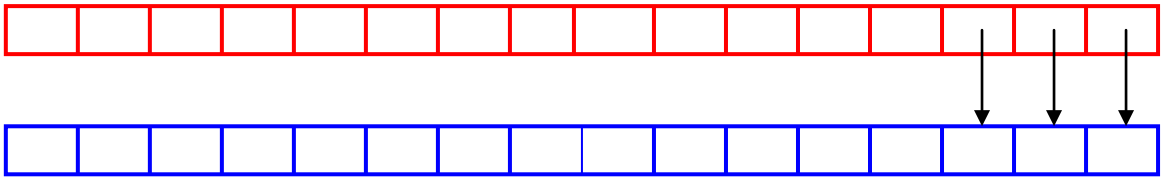
Las aristas de la 1 a la 10 de la fila activa se copian idénticas a la nueva fila 3.



De la arista 11 a la 13 tenemos dos posibles soluciones óptimas; ponemos una solución en la fila activa y la otra en la nueva fila 3:



y ahora, añadimos las aristas de la 14 a la 16 de la fila activa a la nueva fila 3:



Dejamos aparcada la nueva fila 3 y continuamos con el algoritmo de Kruskal sobre las aristas 14, 15, 16 de la fila activa. Cuando se halle la solución óptima (no tiene porque llegarse hasta la última arista) se pasa a la siguiente fila (fila nueva 1) como fila activa. En la nueva fila 1 se inicializa la matriz *"componente_conexa"* y se va recorriendo la fila desde la primera arista hasta la arista 8. en este recorrido, las aristas con valor *"solución"* en su dato *"fase"* reconstruyen la matriz *"componente_conexa"*. A continuación, a partir de la arista 9 se aplica el algoritmo de Kruskal y se empieza otra vez el mismo proceso pudiéndose generar más soluciones óptimas (más filas nuevas) al llegar a las aristas 11, 12, 13, si se da el caso, (las aristas 5, 6, 7, 8, ya no van a generar ninguna solución nueva en ninguna fila). Continuamos este proceso hasta que finalicemos el algoritmo de Kruskal en la

última fila *"k"* creada de la matriz *"aristas_ordenadas"* y ya no exista una siguiente fila después.

Cuando un grupo de aristas con el mismo valor produce varias soluciones, habíamos dicho que poníamos cada solución única en una fila distinta (tantas soluciones como filas nuevas más la fila activa), luego tengo que idear un método para colocar las aristas de cada solución en la fila correspondiente.

Supongamos que el número de aristas con el mismo valor, descartadas las aristas que están siempre en todas las soluciones y las que nunca están en ninguna solución, es *"num"*. El número de posibles combinaciones de dichas aristas en cuanto al orden en que pueden ser seleccionadas para desarrollar el algoritmo de Kruskal es *(num!)*, es decir, *[num * (num – 1) * (num – 2) * ... * 2 * 1]* , luego tenemos que hacer el algoritmo de Kruskal *(num!)* veces y guardar la información de cada resultado (dato *"fase"* de las *"num"* aristas).

Para ello, me creo la matriz ***"solución_parcial"*** con un número de elementos de *(num!)* filas por *(num)* columnas. Cada elemento tiene la misma información de una arista de la matriz *"aristas_ordenadas"*, es decir, los campos *"nodop"*, *"nodoq"*, *"valor"* y *"fase"*.

Para cada resultado (algoritmo de Kruskal sobre una combinación en orden de elección de las *"num"* aristas y partiendo de la matriz *"componente_conexa"* obtenida hasta llegar al grupo de *"num"* aristas) se guardan las aristas en una fila de la matriz *"solución_parcial"* (el campo *"fase"* nos dirá el resultado de aplicar el algoritmo de Kruskal a cada arista).

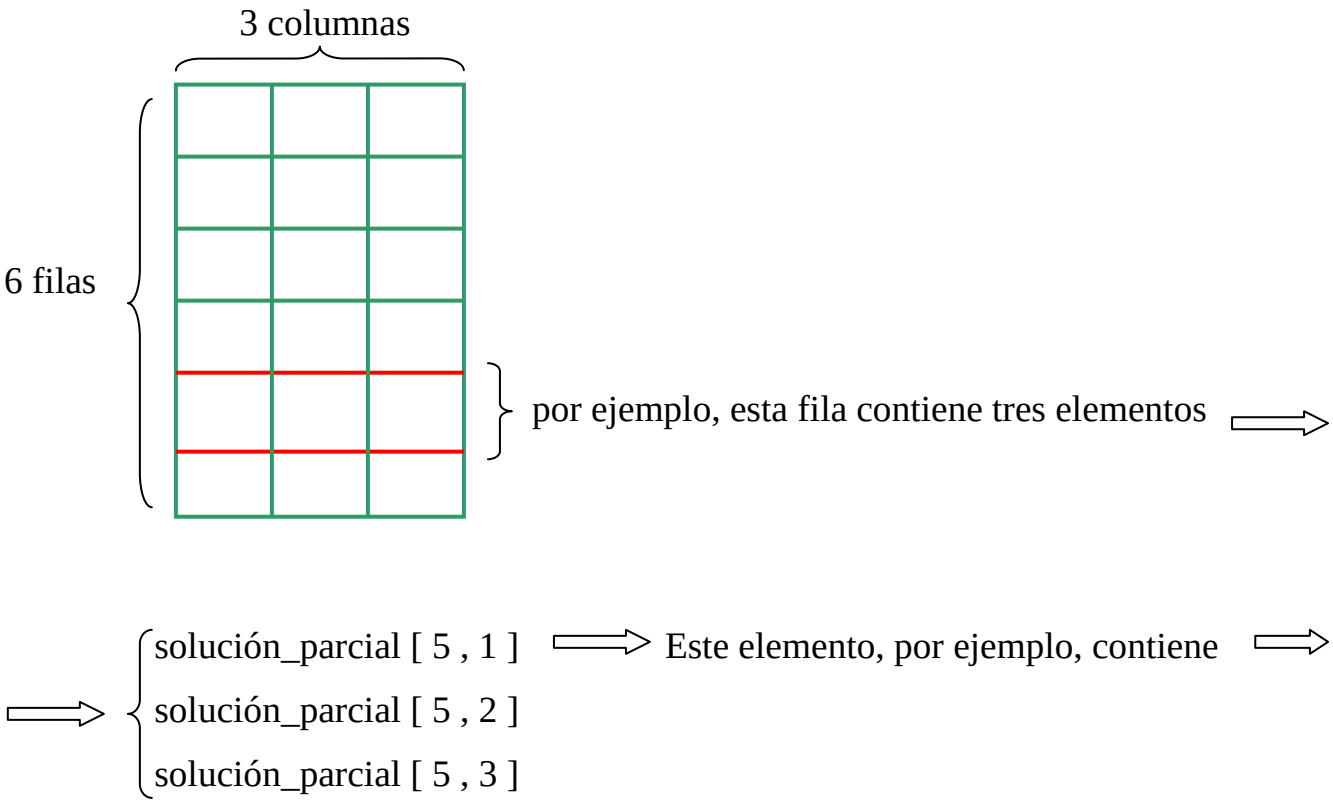
Para descartar las soluciones repetidas, en la matriz *"solución_parcial"* comparo el resultado de cada solución (una fila) con los demás resultados (resto de filas) y voy descartando las soluciones repetidas.

A continuación, se añaden tantas filas a la matriz *"aristas_ordenadas"* como soluciones no repetidas existan menos una, ya que una solución se pone en la fila activa.

Las nuevas filas serán una copia idéntica de la fila activa hasta llegar al grupo de aristas con el mismo valor que originan varias soluciones, diferenciándose unas filas de otras precisamente en los distintos valores del dato *"fase"* de las aristas con el mismo valor por aplicar el algoritmo de Kruskal según el orden de elección de dichas aristas.

Siguiendo nuestro ejemplo anterior, de las aristas 5, 6, 7, 8, hay que descartar las que están en todas las soluciones o en ninguna solución. Supongamos que descartamos la arista 5 con lo que nos quedarían las aristas 6, 7 y 8, como generadoras de soluciones óptimas *(num = 3)*. El número de combinaciones posibles entre estas tres aristas es *3! = 3 * 2 * 1 = 6*.

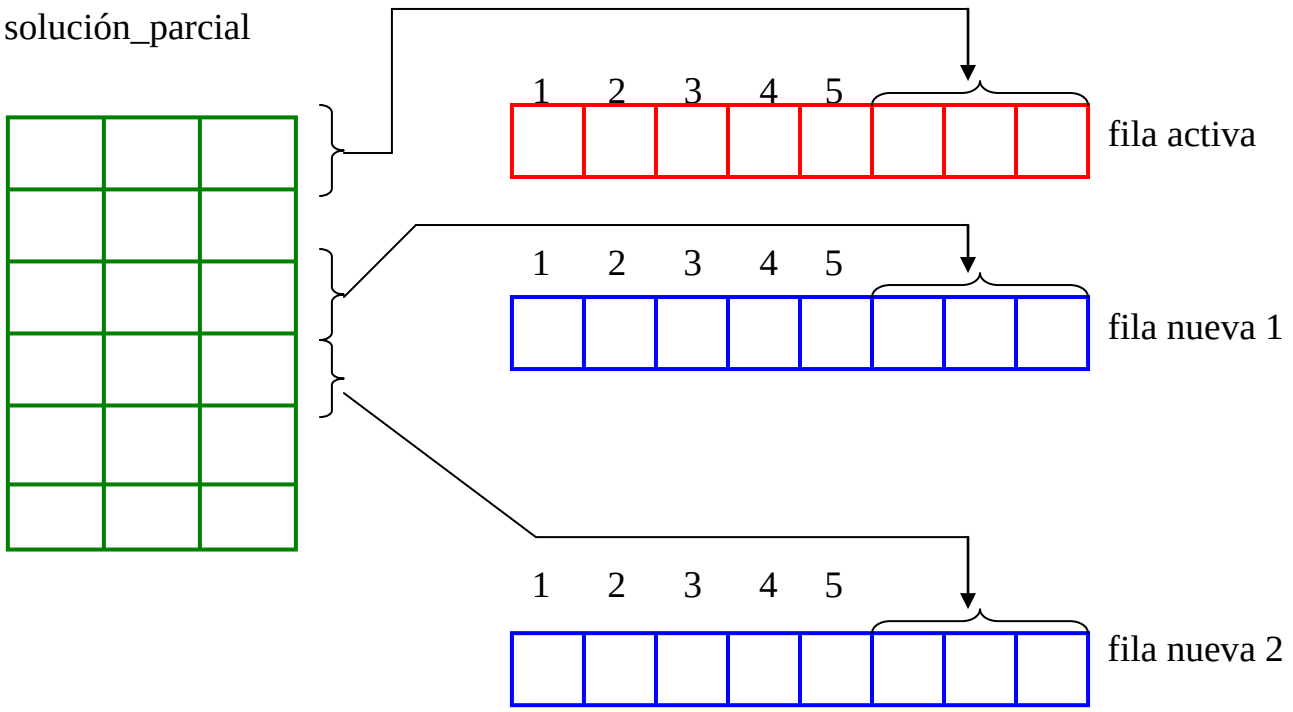
La matriz *"solución_parcial"* será:



solución_parcial.nodop [5 , 1] que es igual a aristas_ordenadas.nodop [1 , 6]
solución_parcial.nodoq [5 , 1] que es igual a aristas_ordenadas.nodoq [1 , 6]
solución_parcial.valor [5 , 1] que es igual a aristas_ordenadas.valor [1 , 6]
solución_parcial.fase [5 , 1] ⇒

⇒ Este valor lo calcula el algoritmo de Kruskal para un cierto orden de selección de estas tres aristas.

Supongamos, para seguir el ejemplo ya visto, que hay tres soluciones repetidas, con lo que nos quedan tres soluciones óptimas únicas *(6 – 3 = 3)*. Si esas soluciones únicas están en las filas 1, 3 y 4, de la matriz *"solución_parcial"*, tenemos que llevar la fila 1 a la fila activa, la fila 3 a la fila nueva 1 y la fila 4 a la fila nueva 3, en las columnas (aristas) 6, 7, 8:



2.2.3.2.- Solución para el grupo de aristas con el mismo valor

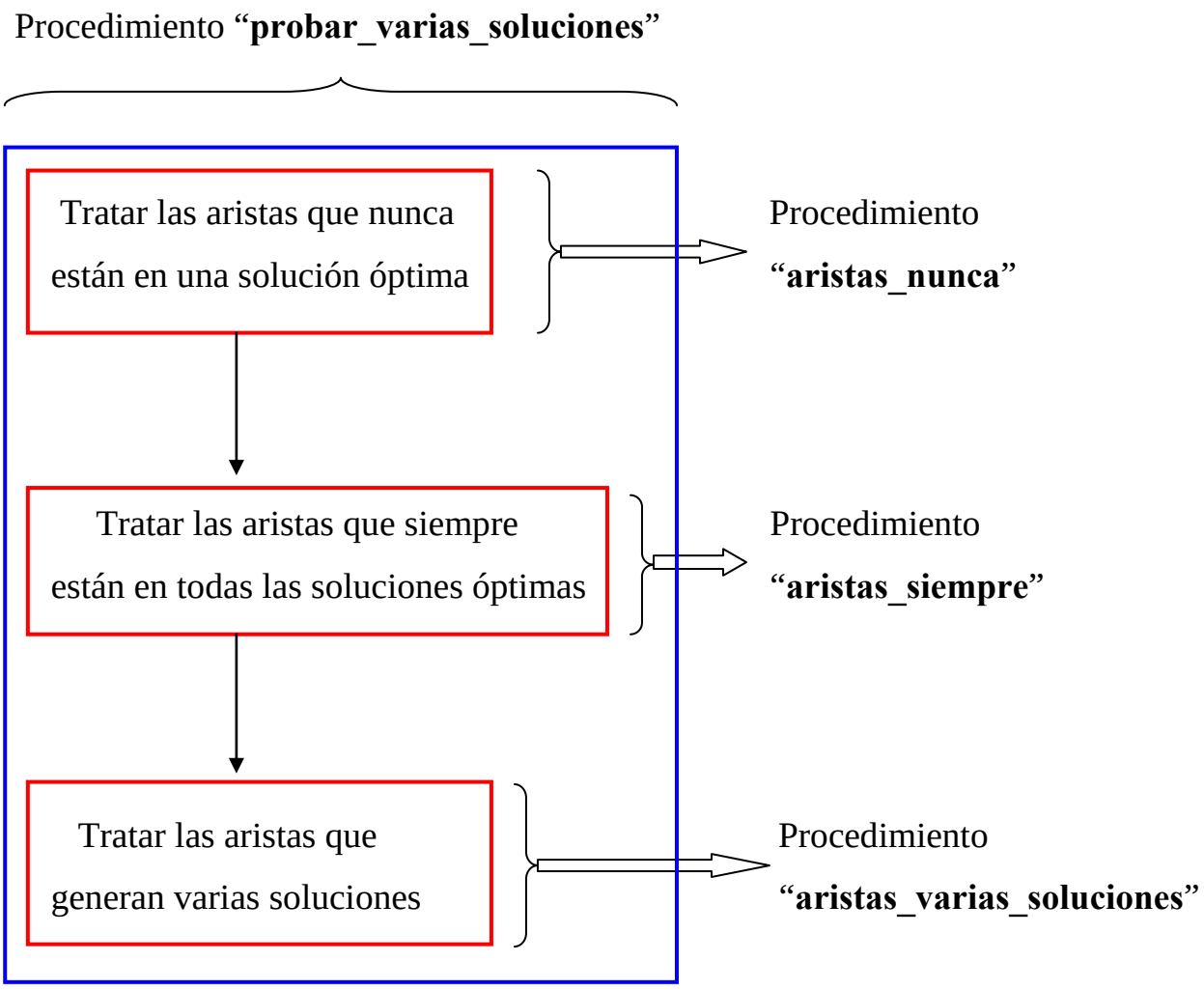
Ahora vamos a analizar en profundidad y a implementar el caso de un grupo de varias aristas con el mismo valor. Esto nos va a simplificar mucho el trabajo posterior cuando tengamos que hacer el algoritmo de Kruskal para todas las aristas del grafo porque nos vamos a ahorrar muchas explicaciones y también por tener ya implementado buena parte de todo el pseudocódigo.

Programa principal

Para el tratamiento de varias aristas con el mismo dato *"valor"*, utilizaré el procedimiento *"probar_varias_soluciones"* que se activará sólo cuando se detecte más de una arista con el mismo dato *"valor"*. Siendo *"m"* el número total de aristas del grafo, *"fila"* la fila activa y *"columna"* la posición de la arista dentro de la fila activa:

```
si ( (columna < m)
     y
   aristas_ordenadas.valor [fila, columna] =
   = aristas_ordenadas.valor [fila, columna +1] ) , entonces
     probar_varias_soluciones
finsi
```

Como se ve en este código, cuando existe más de una arista con el mismo dato en el campo *"valor"*, se activará el procedimiento *"probar_varias_soluciones"* cuyo esquema es:



Es decir, el procedimiento *"probar_varias_soluciones"* lo he dividido en tres procedimientos sucesivos.

Procedimiento "**probar_varias_soluciones**"

Calculo el número de aristas seguidas con el mismo dato "valor" utilizando la función "num_aristas_mismo_valor"

num ⟵ num_aristas_mismo_valor

(Primero trataré las aristas que nunca están en ninguna solución óptima (sí las hay))

aristas_nunca

Después trataré las aristas que están siempre en todas las soluciones óptimas (si las hay)

aristas_siempre

(Y por último, trataré las aristas que generan varias soluciones únicas (sí las hay))

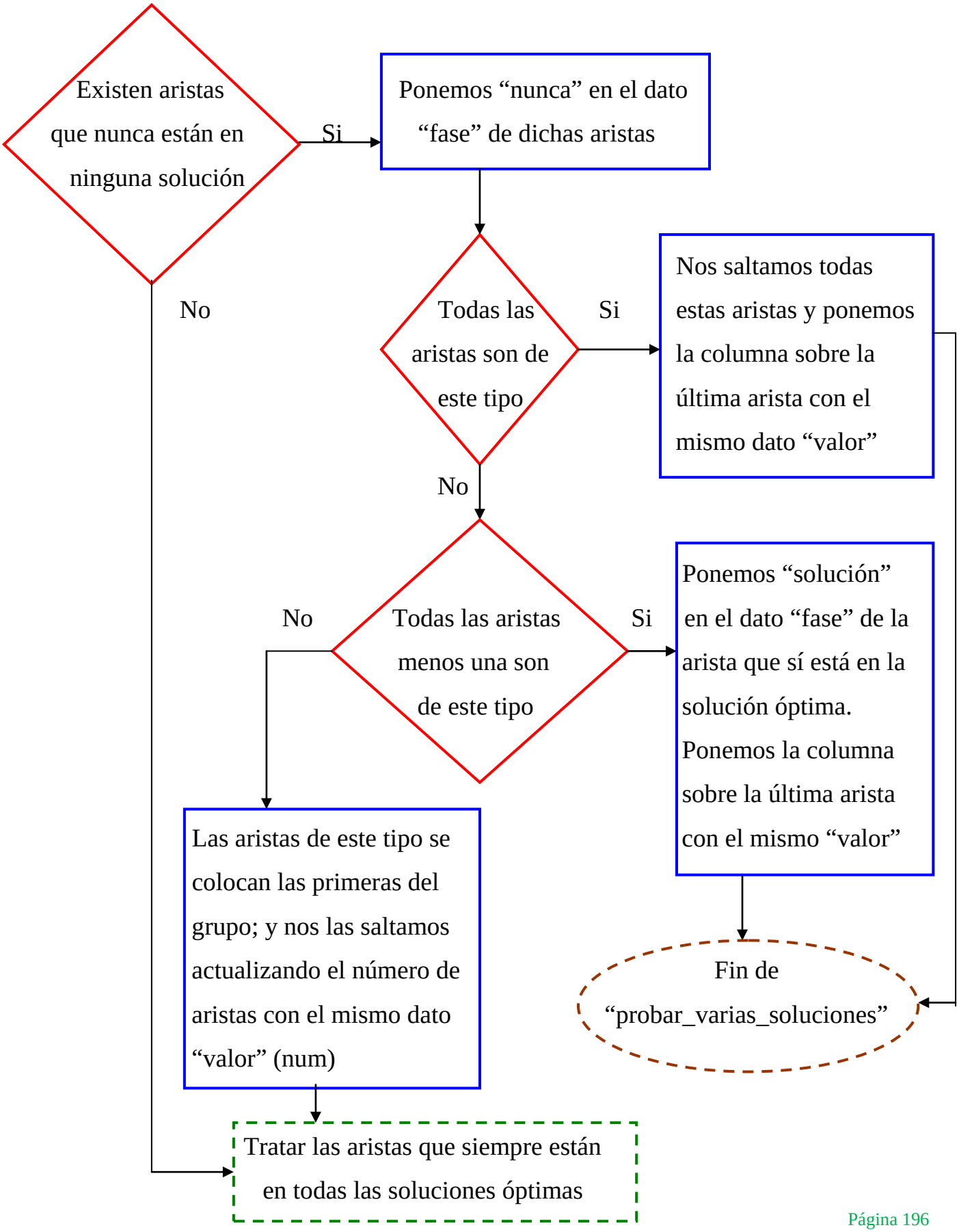
aristas_varias_soluciones

Función "**num_aristas_mismo_valor**"

Calcula el número de aristas que tienen un mismo dato *"valor"* a partir de la arista *"aristas_ordenadas [fila, columna]"* en adelante. Esta función se activa cuando a partir de dicha arista existen al menos dos aristas seguidas con el mismo dato *"valor"*.

```
num_aristas_mismo_valor ⟵ 2

( Trabajo con la variable "copia_ columna"
  para no modificar la variable "columna" )

copia_ columna ⟵ columna + 1

mientras ( (copia_ columna < m)
           y
           ( aristas_ordenadas.valor [fila, copia_ columna] =
           = aristas_ordenadas.valor [fila, copia_ columna + 1] ) ), hacer

    num_aristas_mismo_valor ⟵ num_aristas_mismo_valor + 1

    copia_ columna ⟵ copia_ columna + 1

finmientras
```

Esquema funcional del procedimiento "**aristas_nunca**"



Procedimiento "**aristas_nunca**"

(Para no modificar la variable "columna" trabajo con la variable "copia_ columna")

copia_ columna ⟵ columna

(Inicializo a cero el número de aristas de este tipo)

num_nunca ⟵ 0

para i = 1 hasta num , hacer

(Si existen aristas que no estarán nunca en ninguna solución, calculo su número y pongo "nunca" en su dato "fase")

nodop ⟵ aristas_ordenadas.nodoi [fila, copia_ columna]

nodoq ⟵ aristas_ordenadas.nodoj [fila, copia_ columna]

(Busco la primera componente conexa a partir de la fila 1 en la matriz "componente_conexa")

fila_componente_conexa ⟵ 1

busca_componente_conexa (componente_conexa)

(Busco la componente conexa, que es una fila de la matriz "componente_conexa", en donde esta el nodop y la guardo en la variable "fila_componente_conexa")

componente_nodo (nodop, componente_conexa)

A continuación, examino la componente conexa encontrada desde el principio para saber si el otro nodoq esta también en esta componente conexa

si misma_componente (nodoq, componente_conexa) , entonces

aristas_ordenadas.fase [fila, copia_ columna] ⟵ "nunca"

num_nunca ⟵ num_nunca + 1

finsi

copia_ columna ⟵ copia_ columna + 1

finpara

---

si num_nunca > 0 , entonces

(Existen aristas que nunca estarán en ninguna solución)

si num_nunca = num , entonces

Si todo el grupo de aristas es de este tipo, no se generan varias soluciones y todo el grupo de aristas con él mismo dato "valor" no estarán en ninguna solución

columna ⟵ columna + num – 1

salir del procedimiento "probar_varias_soluciones"

finsi

---

si num_nunca = num – 1 , entonces

(No se generan varias soluciones, pongo la única arista que no tiene el valor “nunca” en el dato “fase” en el conjunto de aristas solución)

copia_ columna ⟵ columna

para i = 1 hasta num , hacer

si aristas_ordenadas.fase [fila, copia_ columna] ≠ “nunca” , entonces

(Actualizo la matriz “componente_conexa”)

actualizar_componentes_conexas (copia_ columna, componente_conexa)

(Pongo “solución” en el dato “fase”)

aristas_ordenadas.fase [fila, copia_ columna] = “solución”

(Aumento en uno el número de aristas de la solución)

numero_aristas_solución ⟵ numero_aristas_solución + 1

salir del para

finsi

copia_ columna ⟵ copia_ columna + 1

finpara

(Pongo la columna en la posición de la última arista con el mismo valor)

columna ← columna + num – 1

salir del procedimiento “probar_varias_soluciones”

finsi

Y por último, la única posibilidad que queda es que existan por lo menos dos aristas que pueden pertenecer a soluciones óptimas. En este caso, pongo al principio las aristas con el dato “nunca” y pongo la “columna” sobre la primera arista que no tenga el dato “nunca”

recolocación

finsi

Procedimiento **recolocación**

Las aristas con el valor *"nunca"* en el dato *"fase"* las coloco al principio del grupo de aristas que tienen el mismo dato *"valor"*. Después, avanzo la columna hasta la primera arista que no tenga el valor *"nunca"* en el dato *"fase"*.

Para que este procedimiento se active, al menos dos aristas con el mismo dato *"valor"* no tienen el término *"nunca"* en su dato *"fase"*.

```
para  i = columna  hasta  columna + num_nunca  ,  hacer
    si  aristas_ordenadas.fase [ fila, i ]  ǂ  "nunca"  ,  entonces
        para  j = i + 1  hasta  columna + num –1  ,  hacer
            si  aristas_ordenadas.fase [ fila, j ] = "nunca"  ,  entonces
                aux  ←  aristas_ordenadas [ fila, j ]
                aristas_ordenadas [ fila, j ]  ←  aristas_ordenadas [ fila, i ]
                aristas_ordenadas [ fila, i ]  ←  aux
                salir del para interno
            finsi
        finpara
    finsi
finpara
```

Ahora, avanzo la columna hasta la primera arista
que no tenga el valor “nunca” en el dato “fase”

columna ⟵ columna + num_nunca

Y ahora actualizo el número de aristas con el mismo dato “valor”
al descartar las aristas que nunca estarán en una solución óptima

num ⟵ num – num_nunca

Procedimiento "**actualizar_componentes_conexas (cop_columna, comp_conexa)**"

Une las componentes conexas en la matriz indicada por *"comp_conexa"* de los nodos de la arista que esta en la posición indicada por la fila *"fila"* y la columna *"cop_ columna"* de la matriz *"aristas_ordenadas"*.

(Primero extraigo los nodos)

nodop ⟵ aristas_ordenadas.nodoi [fila, cop_ columna]

nodoq ⟵ aristas_ordenadas.nodoj [fila, cop_ columna]

(Pongo la fila de la componente conexa del "nodop" en la variable "componenetp")

fila_componente_conexa ⟵ 1

busca_componente_conexa (comp_conexa)

componente_nodo (nodop, comp_conexa)

componentep ⟵ fila_componente_conexa

(Pongo la fila de la componente conexa del "nodoq" en la variable "componenetq")

fila_componente_conexa ⟵ 1

busca_componente_conexa (comp_conexa)

componente_nodo (nodoq, comp_conexa)

componenteq ⟵ fila_componente_conexa

(Y por fin, uno las dos componentes conexas)

unir_componentes_conexas (comp_conexa)

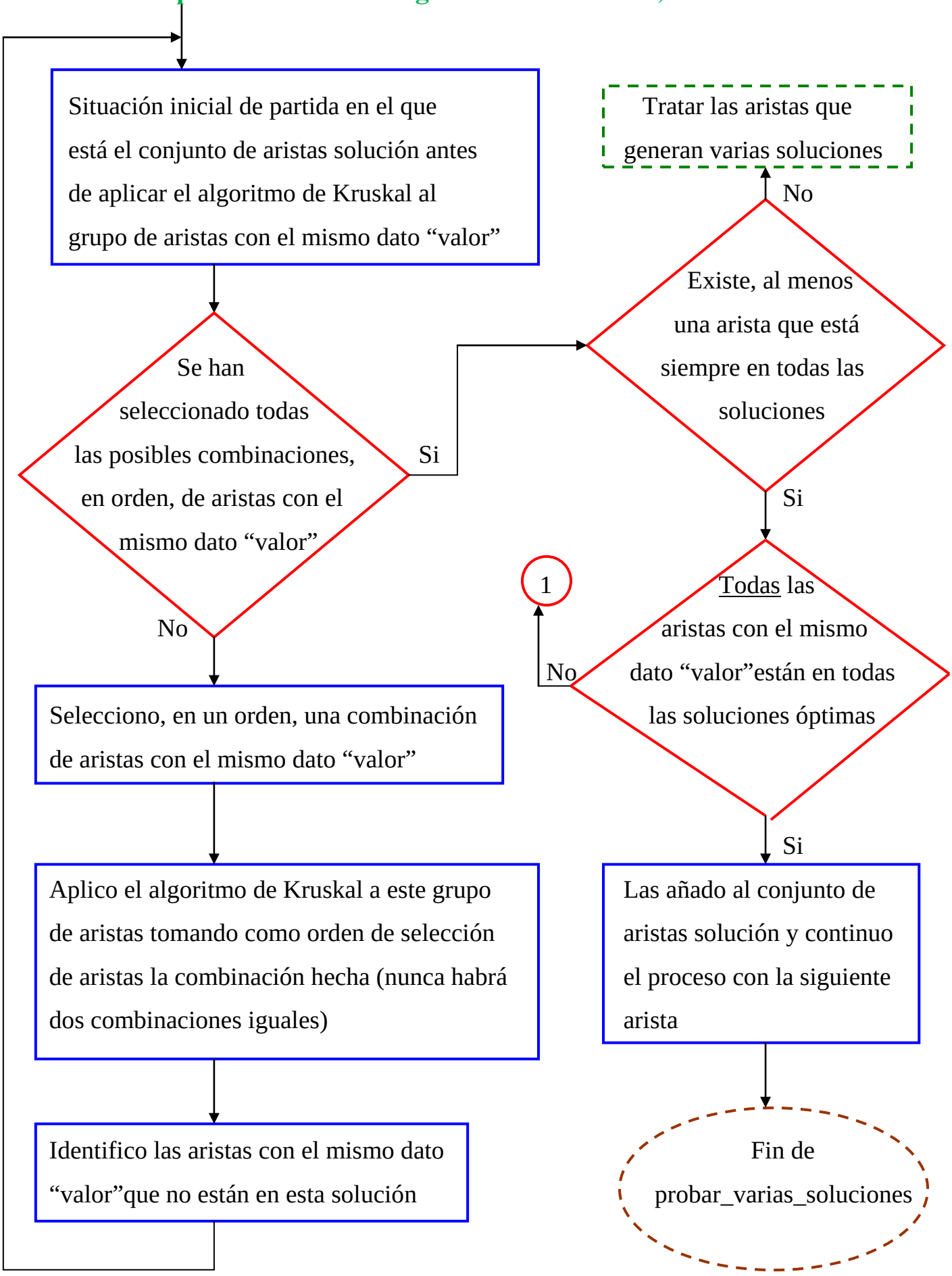
Explicación del procedimiento “aristas_siempre”

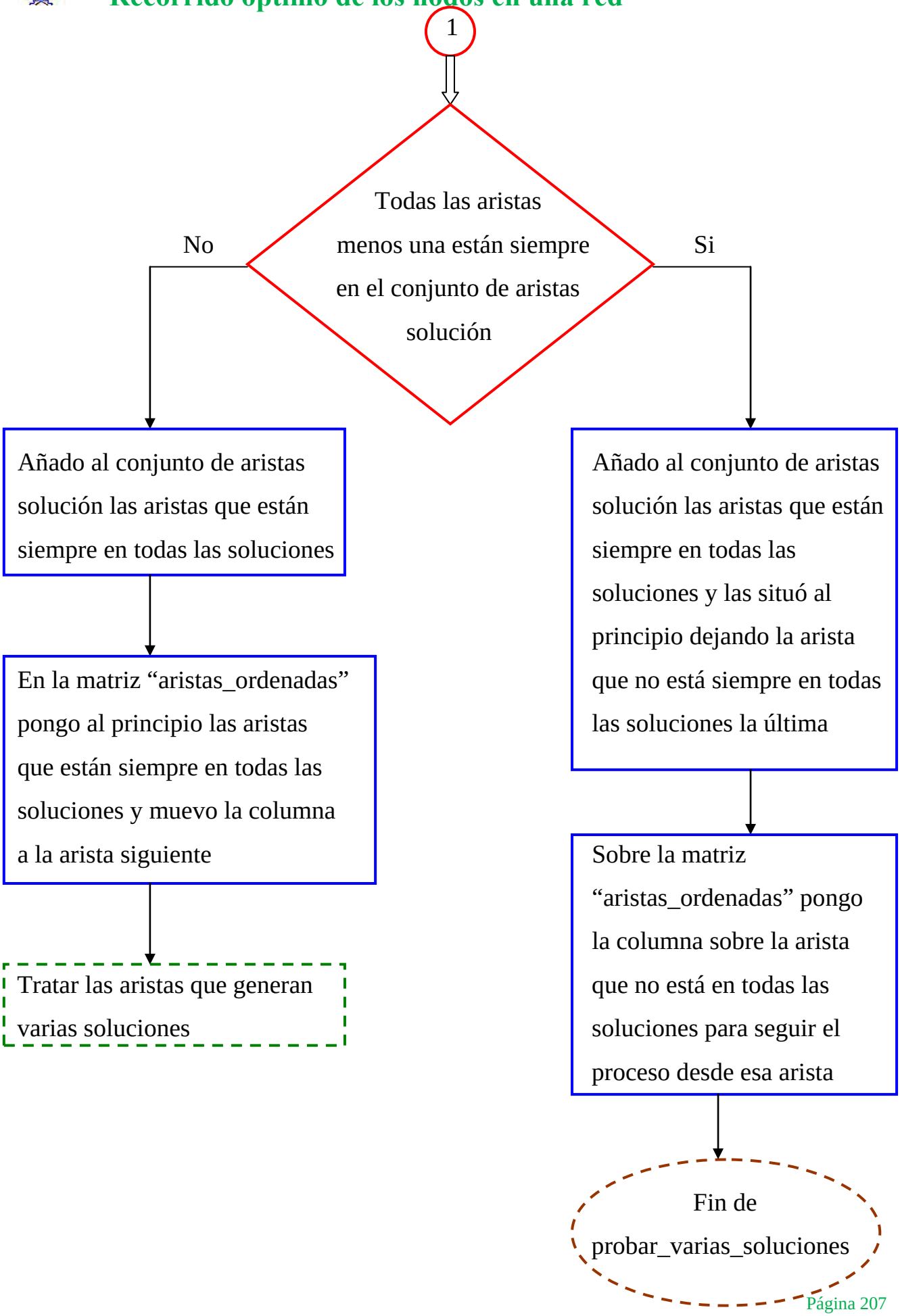
Cuando llegamos a este procedimiento, el número de aristas *“num”* con el mismo dato *“valor”* es siempre mayor o igual a dos y están descartadas las aristas que aunque también tengan el mismo dato *“valor”* nunca estarán en ninguna solución óptima.

Sólo para este grupo de aristas y teniendo en cuenta lo avanzado que pueda estar el algoritmo de Kruskal al llegar a este grupo de aristas, la idea es seguir aplicando dicho algoritmo a todas las combinaciones posibles en orden de selección de las aristas con el mismo dato *“valor”* y ver que aristas están siempre en la solución óptima sea cual sea su orden de elección. Después, si he encontrado aristas que están siempre en todas las soluciones óptimas, las situamos en los primeros lugares del grupo, las incluimos en el conjunto de aristas solución y corremos la *“columna”* a la primera arista que venga a continuación.

Para entender de una forma simple las situaciones que se producen en este proceso, se muestra a continuación el organigrama del procedimiento *“aristas_siempre”*.

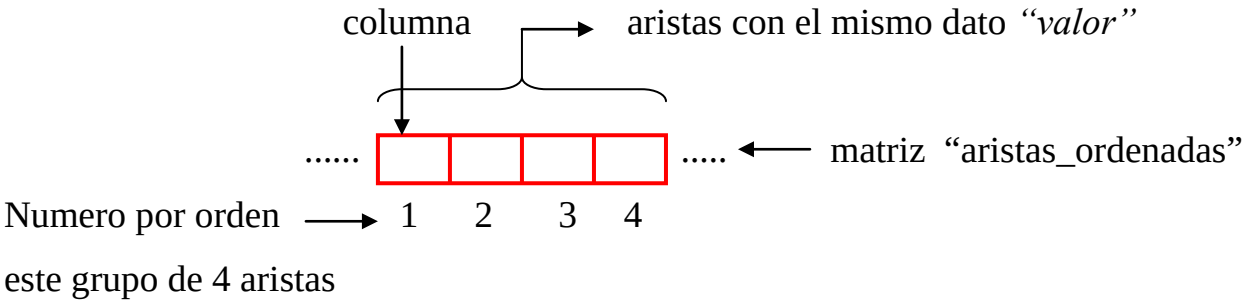
Esquema funcional del procedimiento **“aristas_siempre”**


Situación inicial de partida en el que está el conjunto de aristas solución antes de aplicar el algoritmo de Kruskal al grupo de aristas con el mismo dato "valor"
Se han seleccionado todas las posibles combinaciones, en orden, de aristas con el mismo dato "valor"
Si
No
Selecciono, en un orden, una combinación de aristas con el mismo dato "valor"
Aplico el algoritmo de Kruskal a este grupo de aristas tomando como orden de selección de aristas la combinación hecha (nunca habrá dos combinaciones iguales)
Identifico las aristas con el mismo dato "valor"que no están en esta solución
Existe, al menos una arista que está siempre en todas las soluciones
No
Tratar las aristas que generan varias soluciones
Si
Todas las aristas con el mismo dato "valor"están en todas las soluciones óptimas
No
1
Si
Las añado al conjunto de aristas solución y continuo el proceso con la siguiente arista
Fin de probar_varias_soluciones


1
Todas las aristas menos una están siempre en el conjunto de aristas solución
No
Si
Añado al conjunto de aristas solución las aristas que están siempre en todas las soluciones
En la matriz “aristas_ordenadas” pongo al principio las aristas que están siempre en todas las soluciones y muevo la columna a la arista siguiente
Tratar las aristas que generan varias soluciones
Añado al conjunto de aristas solución las aristas que están siempre en todas las soluciones y las situó al principio dejando la arista que no está siempre en todas las soluciones la última
Sobre la matriz “aristas_ordenadas” pongo la columna sobre la arista que no está en todas las soluciones para seguir el proceso desde esa arista
Fin de probar_varias_soluciones

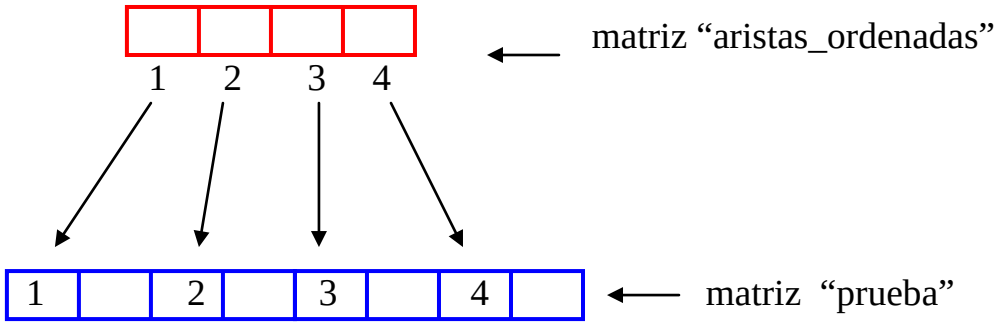
Para aclarar mejor el método que voy a utilizar, sigo un ejemplo concreto:

Descartadas las aristas que nunca estarán en ninguna solución óptima, supongamos que nos quedan 4 aristas con el mismo dato *"valor" (num = 4)* en la fila de la matriz *"aristas_ordenadas"* que estamos tratando:



Me creo una nueva matriz que llamaré *"**prueba**"*, de *"num"* elementos en la que cada elemento consta de dos datos.

Hago una correspondencia entre el número de orden dado en la matriz *"aristas_ordenadas"* y el primer dato de cada elemento de la matriz *"prueba"*:

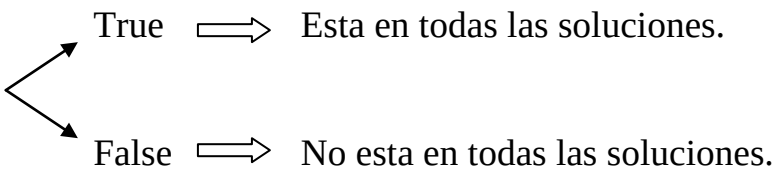


Cada elemento de la matriz *"prueba"* constará de:

1º dato: Le corresponde un valor de la numeración hecha a las aristas con el mismo dato *"valor"* de la matriz *"aristas_ordenadas"*, como se ve en la figura

anterior. Este valor siempre es el mismo y representa a la arista de la matriz *"aristas_ordenadas"* con la misma numeración.

2º dato: Es un valor booleano que al finalizar el proceso que voy a realizar nos indica, para la arista representada por el dato 1º, si dicha arista esta siempre en todas las soluciones óptimas:



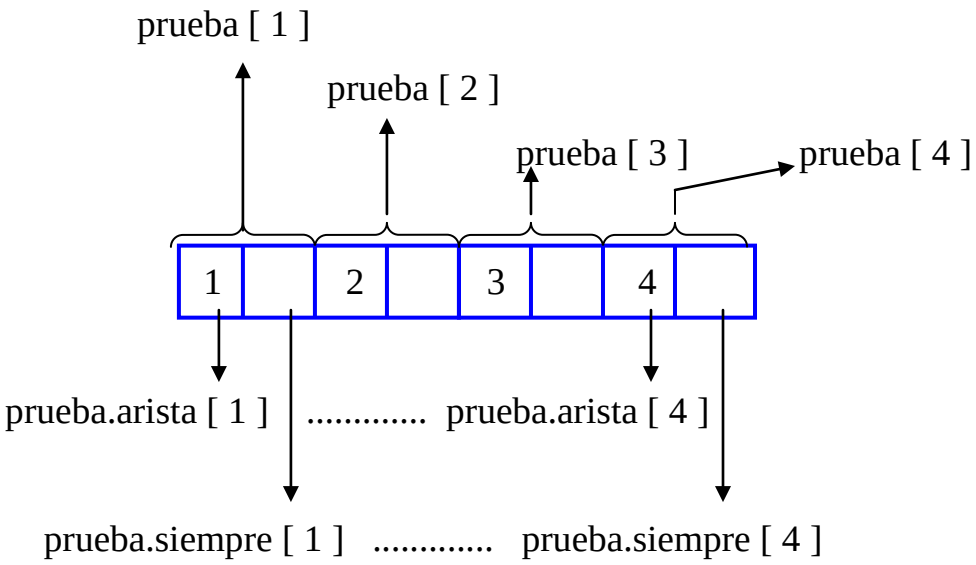
Inicialmente el valor es siempre *"True"* y después el procedimiento que estoy explicando se encargará de poner *"False"* en el número de orden de las aristas que no estén en todas las soluciones.

Cada elemento de la matriz *"prueba"* tendrá, como ya he dicho, dos partes (*num* = 4 elementos en nuestro ejemplo), y la forma de referenciarlos será:

prueba.arista [i]

prueba.siempre [i]

Quedará:



Los elementos "prueba.arista" son valores enteros, constantes y correlativos a partir del valor 1, es decir:

prueba.arista [1] = 1 siempre

prueba.arista [2] = 2 siempre

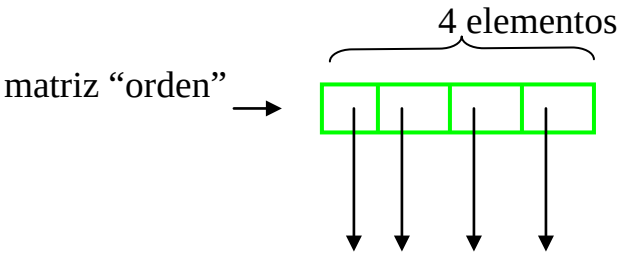
: :

: :

prueba.arista [num] = num siempre

El procedimiento que voy a seguir para probar todas las posibles soluciones que pueda generar este grupo de aristas con el mismo dato *"valor"* y comprobar que aristas del grupo están siempre en todas las soluciones, es el siguiente:

Me creo otra matriz *"**orden**"* de *"num"* elementos y cada elemento puede tomar los valores que he utilizado para numerar las aristas de la matriz *"aristas_ordenadas"* sin que se puedan repetir, es decir, en nuestro ejemplo, *num = 4*:



Cada elemento puede tomar los valores 1, 2, 3, 4 pero estos valores no se pueden repetir.

El valor de un elemento de la matriz *“orden”* representa la arista de la matriz *“aristas_ordenadas”* con el mismo número de orden que dicho valor.

La idea es hacer todas las posibles permutaciones de los elementos de la matriz *“orden”* y para cada permutación aplicar el algoritmo de Kruskal con el orden de selección de las aristas con el mismo valor dado por la matriz *“orden”* y ver que aristas no están en la solución.

Si una arista con número de orden *“i”*, es decir, *“orden [k] = i”*, no esta en la solución, hago *“prueba.siempre [i] = False”*.

Para verlo mejor seguiré un ejemplo concreto:

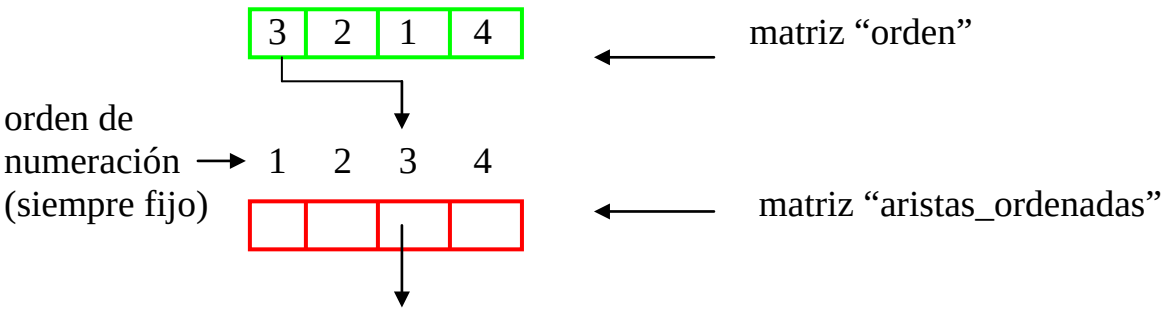
Con num = 4, los valores que tomará la matriz *“orden”* serán:

1 2 3 4	2 1 3 4	3 1 2 4	4 1 2 3
1 2 4 3	2 1 4 3	3 1 4 2	4 1 3 2
1 3 2 4	2 3 1 4	3 2 1 4	4 2 1 3
1 3 4 2	2 3 4 1	3 2 4 1	4 2 3 1

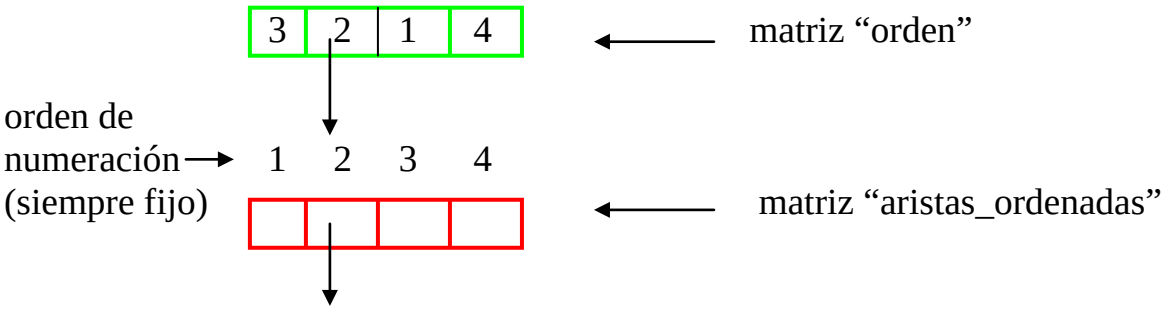
1 4 2 3 2 4 1 3 3 4 1 2 4 3 1 2

1 4 3 2 2 4 3 1 3 4 2 1 4 3 2 1

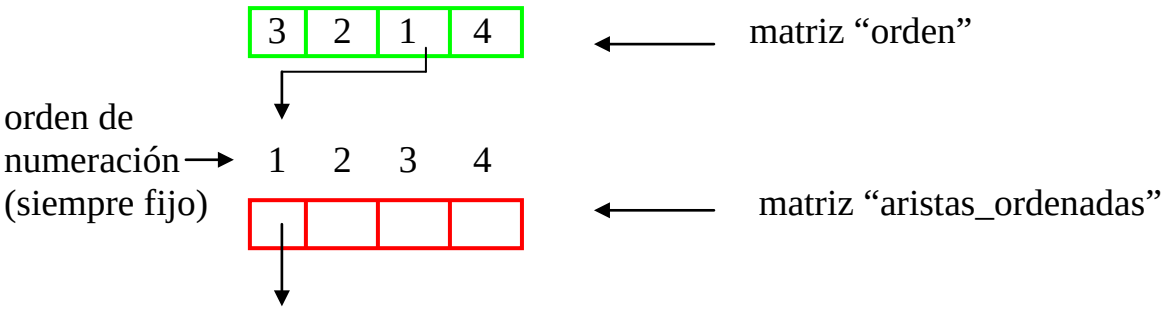
Si en un momento dado la matriz *"orden"* tiene los valores 3 2 1 4 el algoritmo de Kruskal continuará eligiendo en el siguiente orden las aristas:



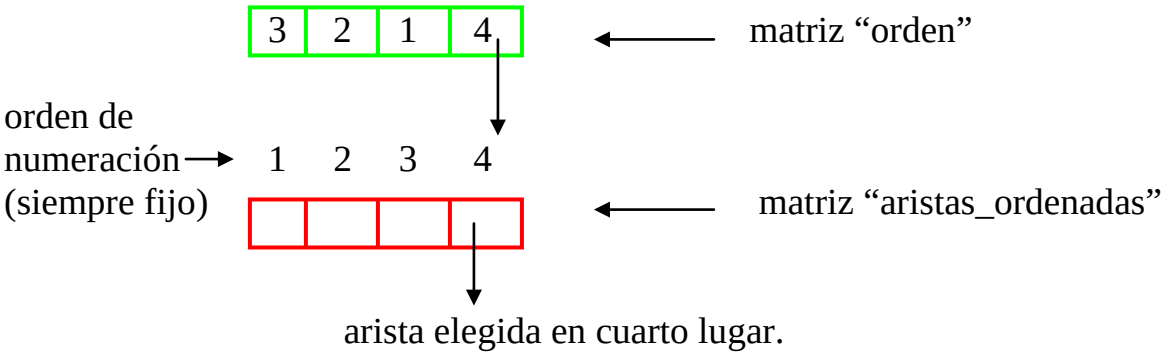
arista elegida en primer lugar.



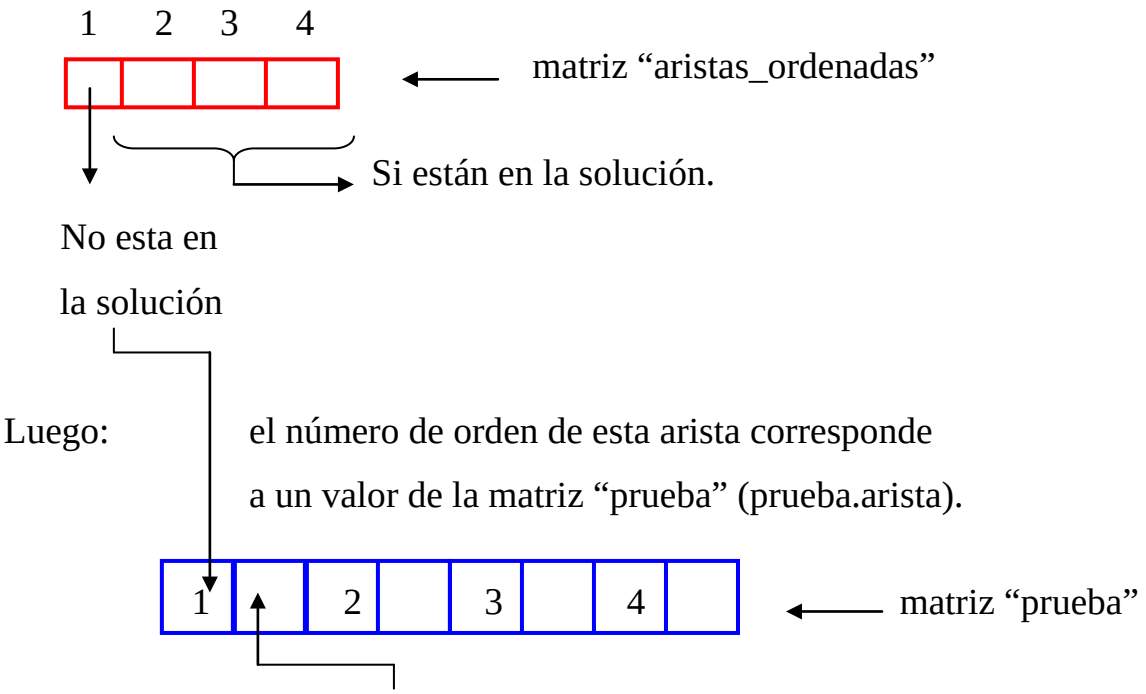
arista elegida en segundo lugar.



arista elegida en tercer lugar.

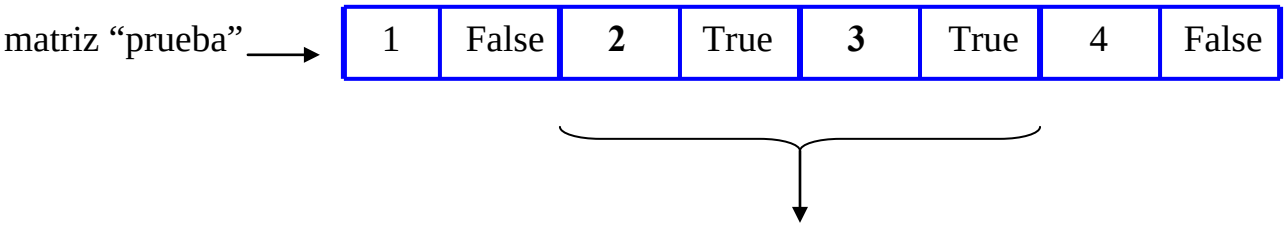


Vamos a suponer que cuando el algoritmo de Kruskal termina para estas 4 aristas el resultado ha sido:

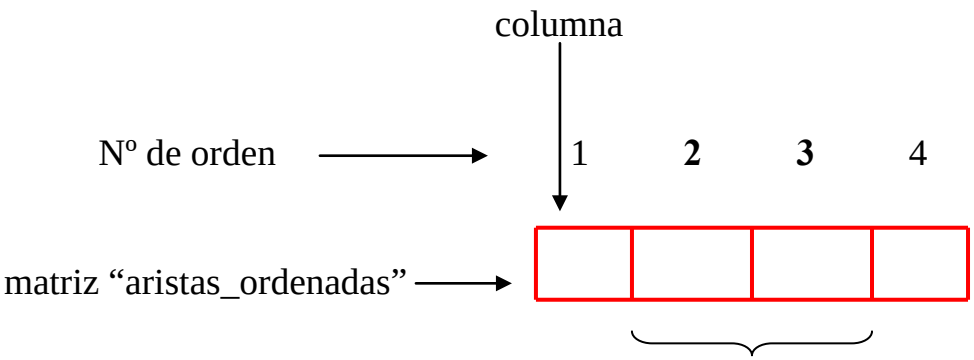


Aquí hay que poner *“False”* para saber que la arista asociada al número de orden 1 en la matriz *“aristas_ordenadas”* no va a estar siempre en todas las soluciones óptimas.

Siguiendo el mismo proceso para cada una de las 24 posibles combinaciones de valores que puede tomar la matriz *“orden”* con “*num* = 4” (4! = 4 * 3 * 2 * 1 = 24) en la matriz *“prueba”* tendremos las aristas que estarán siempre en todas las soluciones. Si, por ejemplo, el resultado final es:



Las aristas en la matriz *“aristas_ordenadas”* cuyos números de orden correspondan al 2 y al 3 estarán siempre en todas las soluciones óptimas, es decir, haciendo coincidir los números de orden:



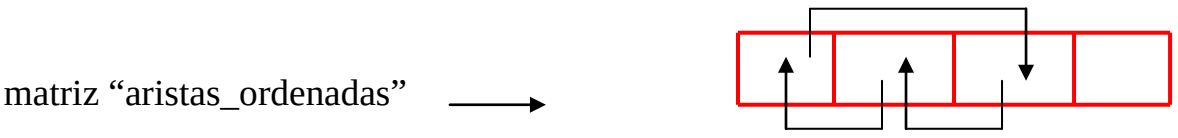
Las aristas:

aristas_ordenadas [fila, columna + 1]

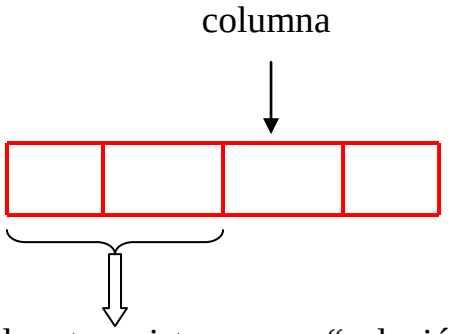
aristas_ordenadas [fila, columna + 2]

están siempre en todas las soluciones.

A continuación, paso las aristas que están siempre en todas las soluciones al principio del grupo de aristas con el mismo dato *“valor”*:

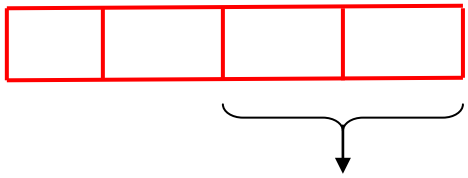


y muevo la *"columna"* hasta la primera arista que no está siempre en todas las soluciones:



En el dato "fase" de estas aristas pongo "solución"

A continuación, si el número de aristas que no están siempre en todas las soluciones es mayor o igual que dos, paso al procedimiento *"aristas_varias_soluciones"*. En nuestro ejemplo:



Estas dos aristas se consideran en el procedimiento *"aristas_varias_soluciones"*.

Hasta aquí la explicación teórica del método que voy a utilizar con las aristas que tienen el mismo dato *"valor"* y están siempre en todas las soluciones.

A continuación escribiré el pseudocódigo.

Procedimiento **"aristas_siempre"**

```
(Inicializo todos los campos de la matriz "prueba")
para  i = 1  hasta  num  ,  hacer
        prueba.siempre [ i ]  ←  True
        prueba.arista [ i ]  ←  i
finpara

(Inicializo a cero los elementos de la matriz "orden")
para  i = 1  hasta  num  ,  hacer
        orden [ i ]  ←  0
finpara

(Pongo "False" en las correspondientes aristas de la matriz "prueba" que no estarán en todas las soluciones)

k  ←  num

solución (num)

(En la matriz "prueba" tengo las aristas que están siempre en todas las soluciones y las que no lo están. Según esta información haré unas cosas u otras)

examinar_resultados
```

Procedimiento "**solución (num)**"

Se trata de un procedimiento recursivo que da como resultado las aristas que están siempre en todas las soluciones. Este resultado queda reflejado en los valores *"True"* de la matriz *"prueba"*.

Este procedimiento aplica el algoritmo de Kruskal al grupo de aristas con el mismo dato *"valor"* en el orden de selección dado por la matriz *"orden"* asociando el valor *"False"* en la matriz *"prueba"* a las aristas que no están en la solución óptima. Cuando la matriz *"orden"* ya ha tenido todas las posibles combinaciones, se termina el proceso. Se hace un poco complicado, a simple vista, el seguimiento del pseudocódigo de un procedimiento recursivo, por lo que he agrupado sentencias, en la medida de lo posible, en forma de funciones y procedimientos para simplificarlo, después, comentaré con detalle dichos procedimientos y funciones.

Cuando se llega a los procedimientos *"inicializo_componente_conexa"* y *"lanzar_solución"* ya tengo una combinación de aristas disponible en la matriz *"orden"*. Todo el pseudocódigo excepto estos dos procedimientos mencionados tienen como única misión crear las distintas combinaciones en la matriz *"orden"*.

```
para  i = 1  hasta  k  ,  hacer
    si  no  iguales ( i )  ,  entonces
        orden [ k – num + 1]  ⟵  i
        si  num > 1  ,  entonces
            solución (num – 1)
        sino
            inicializo_componente_conexa
            lanzar_solución
            orden [ k – num + 1 ]  ⟵  0
        finsi
    finsi
finpara
orden [ k – num +1 ]  ⟵  0
si  num < k  ,  entonces
    orden [ k – num ]  ⟵  0
finsi
```

Procedimiento "**inicializo_componente_conexa**"

Para no modificar la matriz *"componente_conexa"* en las pruebas que tengo que hacer para saber que aristas están en todas las soluciones, trabajaré con una copia suya que llamaré *"copia_componente_conexa"*.

Siendo *"n"* el número de nodos del grafo, la matriz *"copia_componente_conexa"* la inicializo a los valores que tiene la matriz *"componente_conexa"* justo antes de aplicar el algoritmo de Kruskal al grupo de aristas con el mismo dato *"valor"*.

```
para  i = 1  hasta  n  ,  hacer
    para  j = 1  hasta  n  ,  hacer
        copia_componente_conexa [ i , j ]  ←  componente_conexa [ i , j ]
    finpara
finpara
```

Función "**iguales (i)**"

Compara el valor entero del parámetro de entrada *"i"* con los valores mayores que cero de la matriz *"orden"*, y si el valor de *"i"* coincide con alguno de estos valores, la función devuelve *"True"*, devolviendo *"False"* en caso contrario.

El pseudocódigo es:

```
iguales = false
para  j = 1  hasta  num  ,  hacer
    si  i = orden [ j ]  ,  entonces
            iguales = "True"
            salir de la función
    sino
        si  orden [ j ] = 0  ,  entonces
                salir de la función
        finsi
    finsi
finpara
```

Explicación del procedimiento "lanzar_solución"

Este procedimiento reemprende el algoritmo de Kruskal desde el punto en donde se quedó (justo antes del grupo de aristas con el mismo valor) para probar sólo con las aristas con el mismo dato *"valor"* y cuyo orden de selección esta dado en la matriz *"orden"*.

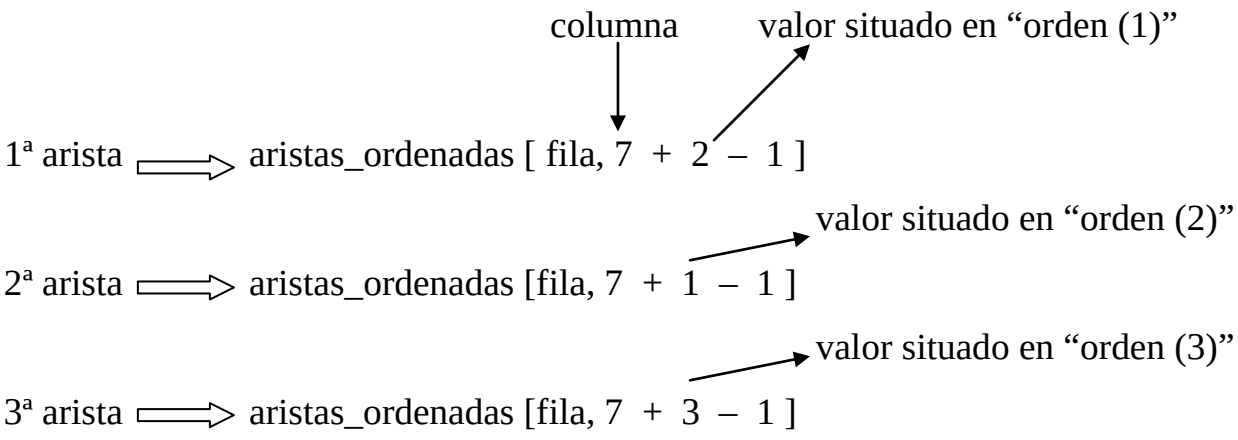
En la práctica, para hacer el pseudocódigo hago una correspondencia entre los valores de la matriz *"orden"* y las aristas de la matriz *"aristas_ordenadas"*:

Valor	corresponde a la arista
1	aristas_ordenadas [fila, columna + 1 –1]
2	aristas_ordenadas [fila, columna + 2 –1]
3	aristas_ordenadas [fila, columna + 3 –1]
4	aristas_ordenadas [fila, columna + 4 –1]

Si en un momento dado la matriz *"orden"* tiene los valores:

2	1	3	4

y el valor de *"columna"* es 7, quiere decir que siguiendo el algoritmo de Kruskal seleccionaremos las aristas en el orden siguiente:



valor situado en "orden (4)"

4ª arista ⟹ aristas_ordenadas [fila, 7 + 4 – 1]

La matriz *"prueba"* tiene la misma correspondencia en la matriz *"aristas_ordenadas"* que tiene la matriz *"orden"*, es decir, por ejemplo, el elemento *prueba.arista [2]* que tendrá siempre el valor 2, se corresponde con la arista *"aristas_ordenadas [fila, columna + 2 – 1]"*.

Por ejemplo, si la matriz *"orden"* tiene los valores:

2	1	3	4

El algoritmo de Kruskal probaría las aristas en este orden:

1ª ⟶ aristas_ordenadas [fila, columna + 2 – 1]
2ª ⟶ aristas_ordenadas [fila, columna + 1 – 1]
3ª ⟶ aristas_ordenadas [fila, columna + 3 – 1]
4ª ⟶ aristas_ordenadas [fila, columna + 4 – 1]

y para las aristas que no estén en la solución óptima:

Si por ejemplo, suponemos que la 2ª arista probada no esta en la solución, entonces la arista a la que corresponde, que es "aristas_ordenadas [fila, columna + (1) – 1]" se la debe indicar que no estará en todas las soluciones óptimas de la forma

prueba.siempre [1] ⟵ False

A continuación escribiré el pseudocódigo.

Procedimiento **"lanzar_solución"**

```
para  i = 1  hasta  num  ,  hacer
        valor_columna  ←  columna + orden [ i ] – 1
        nodop  ←  aristas_ordenadas.nodoi [fila, valor_ columna]
        nodoq  ←  aristas_ordenadas.nodoj [fila, valor_ columna]
        fila_componente_conexa  ←  1
        busca_componente_conexa (copia_ componente_conexa)
        componente_nodo (nodop, copia_ componente_conexa)
    si  misma_componente (nodoq, copia_ componente_conexa)  ,  entonces
            prueba.siempre [ orden [ i ] ]  ←  False
    sino
            componentep  ←  fila_componente_conexa
            fila_componente_conexa  ←  1
            busca_componente_conexa (copia_componente_conexa)
            componente_nodo (nodoq, copia_componente_conexa)
            componenteq  ←  fila_componente_conexa
            unir_componentes_conexas (copia_componente_conexa)
    finsi
finpara
```

Procedimiento "**examinar_resultados**"

```
copia_ columna ←—— columna

num_aristas_true ←—— 0

( Averiguo si existe alguna arista, como mínimo,
  que esta siempre en todas las soluciones )

para  i = 1  hasta  num  ,  hacer

    si  prueba.siempre [ i ]  ,  entonces
    ( Si existen aristas que están en todas las soluciones, calculo su número, pongo
      "solución" en su dato "fase" y sumo su número a "núnero_aristas_solución" )

        aristas_ordenadas.fase [fila, copia_ columna]  ←——  "solución"

        num_aristas_true  ←——  num_aristas_true + 1

        número_aristas_solución  ←——  número_aristas_solución + 1

    finsi

    copia_ columna  ←——  copia_ columna + 1

finpara

si  num_aristas_true > 0  ,  entonces

  si  (num = num_aristas_true)  ,  entonces
```

actualizo la matriz “componente_conexa” ya que todas las aristas en número de “num” están en la solución.

copia_ columna ← columna

para i = 1 hasta num , hacer

actualizar_componentes_conexas (copia_ columna, componente_conexa)

copia_ columna ← copia_ columna + 1

finpara

columna ← columna + num – 1

salir del procedimiento “probar_varias_soluciones”

finsi

si num = num_aristas_true + 1 , entonces

Para todas las aristas menos una en todas las soluciones óptimas:
Pongo al principio las aristas que están siempre en todas las soluciones. Actualizo la matriz de componentes conexas al añadir aristas nuevas al conjunto de aristas solución. Avanzo la “columna” hasta la arista que no este en el conjunto de aristas solución.

actualización

columna ← columna + num_aristas_true – 1

salir del procedimiento “probar_varias_soluciones”

finsi

(Nos queda num >= num_aristas_true + 2 , como única posibilidad)

actualización

columna ⟵ columna + num_aristas_true

finsi

Procedimiento **"actualización"**

(Pongo al principio las aristas que están siempre en todas las soluciones.)

```
para  i = columna  hasta  columna + num_aristas_true  ,  hacer

  si  aristas_ordenadas.fase [ fila , i ]  ≠  "solución"  ,  entonces

    para  j = i + 1  hasta  columna + num – 1  ,  hacer

      si  aristas_ordenadas.fase [ fila , j ] = "solución"  ,  entonces

            aux  ←  aristas_ordenadas [ fila , j ]

            aristas_ordenadas [ fila , j ]  ←  aristas_ordenadas [ fila , i ]

            aristas_ordenadas [ fila , i ]  ←  aux

            salir del para interno

      finsi

    finpara

   finsi

finpara
```

(actualizo la matriz de componentes_conexas)

copia_columna ⟵ columna

para i = 1 hasta num_aristas_true , hacer

actualizar_componentes_conexas (copia_ columna, componente_conexa)

copia_ columna ⟵ copia_ columna + 1

finpara

y ahora actualizo el número de aristas con el mismo dato “valor” al descartar las aristas que siempre estarán en todas las soluciones

num ⟵ num _ num_aristas_true

Explicación del procedimiento “aristas_varias_soluciones”

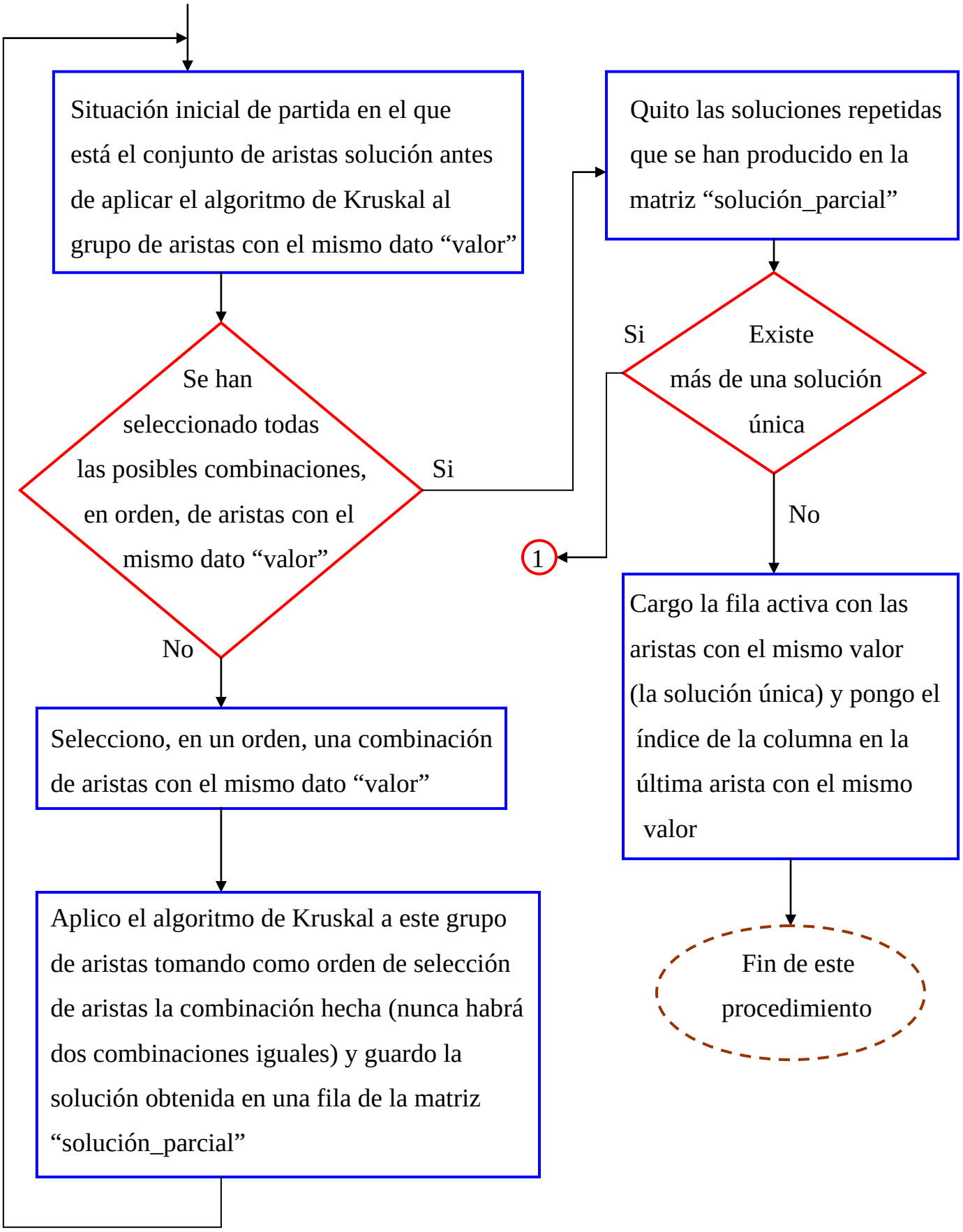
Una vez descartadas las aristas que están siempre en todas las soluciones y las que nunca están en ninguna solución, si todavía existe más de una arista con el mismo dato *“valor”*, aplicaré el algoritmo de Kruskal a cada combinación, en orden de selección, de dichas aristas y según el número de soluciones óptimas únicas que se produzcan crearé nuevas filas en la matriz *“aristas_ordenadas”*.

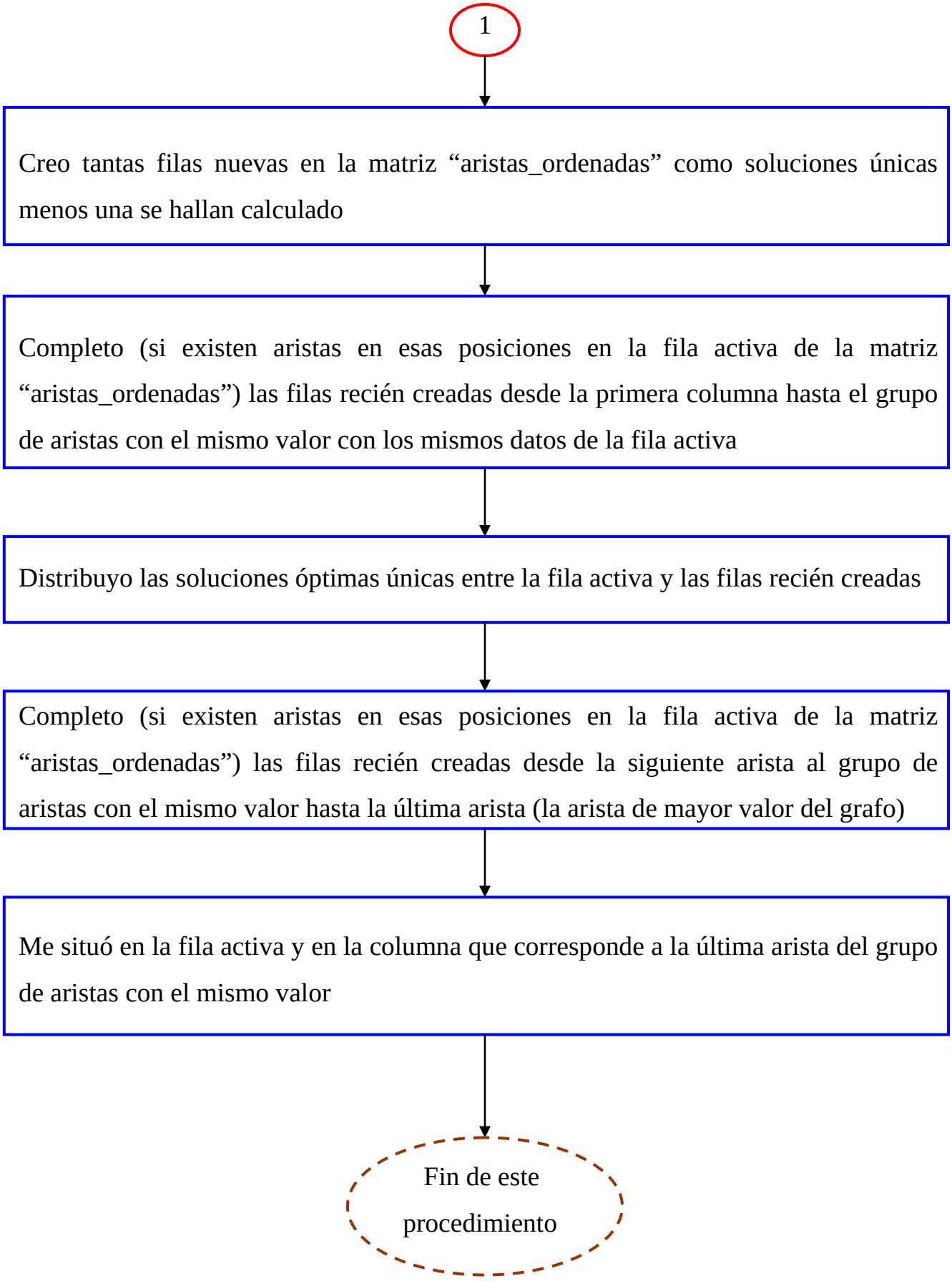
Este procedimiento realiza el algoritmo de Kruskal para las aristas con el mismo valor y distribuye cada combinación única de soluciones de estas aristas, según el orden de selección de las mismas, entre todas las filas recién creadas y la fila activa de la matriz *“aristas_ordenadas”*.

Después, no en este procedimiento, seguiré aplicando el algoritmo de Kruskal a la fila activa y cuando calcule todas las aristas de la solución almacenada en la fila activa empezaré a aplicar el algoritmo de Kruskal a cada fila nueva (cada fila contendrá una solución óptima única) para calcular todas las soluciones óptimas posibles.

Primero haré el organigrama funcional y después la implementación en pseudocódigo.

Esquema funcional del procedimiento **"aristas_varias_soluciones"**




1
Creo tantas filas nuevas en la matriz “aristas_ordenadas” como soluciones únicas menos una se hallan calculado
Completo (si existen aristas en esas posiciones en la fila activa de la matriz “aristas_ordenadas”) las filas recién creadas desde la primera columna hasta el grupo de aristas con el mismo valor con los mismos datos de la fila activa
Distribuyo las soluciones óptimas únicas entre la fila activa y las filas recién creadas
Completo (si existen aristas en esas posiciones en la fila activa de la matriz “aristas_ordenadas”) las filas recién creadas desde la siguiente arista al grupo de aristas con el mismo valor hasta la última arista (la arista de mayor valor del grafo)
Me situó en la fila activa y en la columna que corresponde a la última arista del grupo de aristas con el mismo valor
Fin de este procedimiento

Procedimiento "**aristas_varias_soluciones**"

(Inicializo a cero los elementos de la matriz "orden")

para i = 1 hasta num , hacer

orden [i] ⟵ 0

finpara

Creo la matriz "solución_parcial" de "num!" filas por "num" columnas.

fila_solución_parcial ⟵ 0

num_soluciones_no_repetidas ⟵ 0

k ⟵ num

Cargo todas las soluciones posibles de las aristas con el mismo valor (incluso las soluciones que están repetidas) sobre las filas de la matriz "solución_parcial"

generar_soluciones (num)

Después de ejecutarse el procedimiento "generar_soluciones", en la variable "fila_solución_parcial" tenemos el total de las filas de la matriz "solución_"parcial"

total_filas_soluciòn_parcial ⟵ fila_solución_parcial

(Marco las soluciones repetidas de la matriz "solución_parcial")
marcar_soluciones_repetidas

Después de ejecutarse el procedimiento "marcar_soluciones_repetidas", en la variable "num_soluciones_no_repetidas" tenemos el número de soluciones únicas obtenido.

```
si  num_soluciones_no_repetidas = 1  ,  entonces
    (Cargo la fila activa con la única solución)
    col ←—— 0
    para  j = columna  hasta  (columna + num – 1)  ,  hacer
        col ←—— col + 1
        aristas_ordenadas [ fila , j ]  ←——  solución_parcial [ 1 , col ]
        si  solución_parcial.fase [ 1 , col ] = "solución"  ,  entonces
            Actualizo la matriz "componente_conexa"
            con la nueva arista de la solución.
            actualizar_componentes_conexas ( j , componente_conexa )
            (Sumo uno al número de aristas de la solución)
            número_aristas_solución ←—— número_aristas_solución + 1
        finsi
    finpara
sino
    Aumento el número de filas de la matriz "aristas_ordenadas" según el número
    de soluciones únicas calculado. Reparto las distintas soluciones entre las filas
    recién creadas y la fila activa
    ampliar_matriz
finsi

(coloco la columna en la última arista del grupo de aristas con el mismo valor)
```

columna ⟵ columna + num – 1

Procedimiento "**generar_soluciones (num)**"

Este procedimiento recursivo es muy difícil de seguir a simple vista y tiene básicamente una estructura muy parecida al procedimiento *"solución (num)"* que ya hemos visto porque también calcula todas las posibles combinaciones en orden de selección del grupo de aristas con el mismo dato *"valor"*.

Este procedimiento, aplica el algoritmo de Kruskal a cada posible combinación en el orden de seleccionar las aristas del grupo de aristas con el mismo valor.

Para ello, genera una por una todas las posibles combinaciones en orden de selección de las aristas con el mismo valor y con ayuda del procedimiento incluido *"cargar_combinación"* aplica el algoritmo de Kruskal a cada combinación creada y almacena cada resultado en las filas de la matriz *"solución_parcial"*. En realidad, es el procedimiento incluido *"cargar_combinación"* el que hace todas estas tareas y el resto del pseudocódigo proporciona el orden de selección de las aristas con el mismo valor.

Cada resultado (campo *"fase"* de dichas aristas) y el resto de la información de estas aristas lo almaceno en una fila distinta de la matriz *"solución_parcial"*.

```
para  i = 1  hasta  k  ,  hacer

    si  no  iguales ( i ) ,  entonces

        orden [k – num + 1]  ←——  i

        si  num > 1  ,  entonces

            generar_soluciones (num – 1)

        sino

            Parto de una copia de la matriz "componente_conexa"
            obtenida justo antes de aplicar el algoritmo de Kruskal a
            las aristas con el mismo valor.
            inicializo_componente_conexa

            Pongo una solución obtenida con el algoritmo de
            Kruskal en una fila de la matriz "solución_parcial"

            cargar_combinación

            orden [k – num + 1]  ←——  0

        finsi

    finsi

finpara

orden [k – num + 1]  ←——  0

si  num < k  ,  entonces

    orden [k – num]  ←——  0

finsi
```

Procedimiento **"cargar_combinación"**

Aplica el algoritmo de Kruskal al grupo de aristas con el mismo valor, en un orden de selección de dichas aristas dado por la matriz *"orden"*, teniendo en cuenta la aplicación del algoritmo de Kruskal a las anteriores aristas con menor valor (es decir, la matriz *"componente_conexa"* de partida será la obtenida al aplicar el algoritmo de Kruskal a las aristas que están antes del grupo de aristas con el mismo valor.

Las aristas en las filas de la matriz *"solución_parcial"* están colocadas en el mismo orden que en la fila activa de la matriz *"aristas_ordenadas"*, por ejemplo, si en la fila activa hay tres aristas con el mismo valor (las aristas situadas en las columnas 27, 28 y 29), entonces todas las filas de la matriz *"solución_parcial"* tendrán en la columna 1 la arista 27, en la columna 2 la arista 28 y en la columna 3 la arista 29. lo único que puede variar de unas filas a otras es el dato *"fase"* de cada arista al aplicar el algoritmo de Kruskal a una determinada combinación según el orden indicado por la matriz *"orden"*.

fila_solución_parcial ⟵ fila_solución_parcial + 1

para i = 1 hasta num , hacer

Paso la información de la arista de la matriz "aristas_ordenadas" dada por la matriz "orden" a la matriz "solución_parcial", excepto el dato "fase" que lo tengo que calcular con el algoritmo de Kruskal.

valor_columna ⟵ columna + orden [i] – 1

nodop ⟵ aristas_ordenadas.nodoi [fila , valor_columna]

nodoq ⟵ aristas_ordenadas.nodoj [fila , valor_columna]

solución_parcial.nodop [fila_solución_parcial , orden [i]] ←— nodop

solución_parcial.nodoq [fila_solución_parcial , orden [i]] ←— nodoq

solución_parcial.valor [fila_solución_parcial , orden [i]] ←—

←— aristas_ordenadas.valor [fila , valor_columna]

Ahora aplico el algoritmo de Kruskal a esta arista para obtener el dato que falta ("fase")

fila_componente_conexa ←— 1

busca_componente_conexa (copia_componente_conexa)

componente_nodo (nodop, copia_componente_conexa)

si misma_componente (nodoq, copia_componente_conexa) , entonces

solución_parcial.fase [fila_solución_parcial , orden [i]] ←— "forma_ciclo"

sino

componentep ←— fila_componente_conexa

fila_componente_conexa ←— 1

busca_componente_conexa (copia_componente_conexa)

componente_nodo (nodoq, copia_componente_conexa)

componenteq ←— fila_componente_conexa

unir_componentes_conexas (copia_componente_conexa)

solución_parcial.fase [fila_solución_parcial , orden [i]] ←— "solución"

finsi

finpara

Procedimiento **"marcar_soluciones_repetidas"**

Si nos encontramos más de una fila de la matriz *"solución_parcial"* con el dato *"solución"* en las mismas aristas, se nos han producido soluciones repetidas que no hay que tenerlas en cuenta.

Para saltarnos estas filas repetidas de la matriz *"solución_parcial"*, marcamos cada fila repetida de la siguiente manera: en la primera arista (columna) de una fila que contiene una solución repetida ponemos *"repetido"* en el dato *"fase"*.

Importante: La variable global *"varias_soluciones"* se tiene que inicializar a *"False"* en el programa principal, y nos indica si existe más de una solución óptima.

Implementación del algoritmo de Kruskal, todas las soluciones

(Primero marcaré las soluciones repetidas)

```
para  i = 1  hasta  total_filas_solución_parcial  –  1  ,  hacer

    si  solución_parcial.fase [ i , 1 ]  ≠  “repetido”  ,  entonces

        para  j = i + 1  hasta  total_filas_solución_parcial  ,  hacer

            si  solución_parcial.fase [ j , 1 ]  ǂ  “repetido”  ,  entonces

                para  k = 1  hasta  num  ,  hacer

                    si  [ solución_parcial.fase [ i , k ]  ≠
                        ≠ solución_parcial.fase [ j , k ]  ]  entonces

                        salir del para interno

                    finsi

                    si  k = num  ,  entonces

                        solución_parcial.fase [ j , 1 ]  ⟵  “repetido”

                    finsi

                finpara

            finsi

        finpara

    finsi

finpara
```

Cuando ya tenga las soluciones repetidas marcadas, tengo que contar en la variable "num_soluciones_no_repetidas" el número de soluciones únicas obtenido

```
para  i = 1  hasta  total_filas_solución_parcial  ,  hacer

    si  solución_parcial.fase [ i , 1 ]  ≠  "repetido"  ,  entonces

        num_soluciones_no_repetidas  ←  num_soluciones_no_repetidas  +  1

        varias_soluciones  ←  True

    finsi

finpara
```

Procedimiento **"ampliar_matriz"**

Debo ampliar el número de filas de la matriz *"aristas_ordenadas"* en un valor igual al de soluciones únicas que se hallan producido menos una, ya que tengo que distribuir las soluciones: una solución va a la fila activa y las otras soluciones a las filas que tengo que crear.

Una vez creadas las nuevas filas, el orden de los pasos a seguir es el siguiente:

1º.- Siempre que las aristas con el mismo dato *"valor"* que estamos tratando no sean las primeras aristas de la matriz *"aristas_ordenadas"*, tengo que cargar toda la información de la fila activa desde la primera arista hasta la arista anterior al grupo de aristas con el mismo valor (arista situada en la variable *"columna" – 1*) en todas las filas recién creadas en las mismas posiciones (de la columna 1 a la columna contenida en la variable *"columna" – 1*).

2º.- Cargo las distintas soluciones (cada grupo de aristas con el mismo dato *"valor"* que forme una solución) en la fila activa y en las filas recién creadas, en las columnas que van de *"columna"* a *("columna" + "num " – 1)*, es decir, en las columnas que van a continuación de las columnas del paso 1º hasta la última columna con el mismo dato valor.

3º.- Siempre que las aristas con el mismo dato *"valor"* que estamos tratando no sean las últimas aristas de la matriz *"aristas_ordenadas"*, tengo que completar las filas recién creadas desde la arista siguiente al grupo de aristas con el mismo dato *"valor"* hasta la última arista *"m"* (arista de mayor valor del grafo), es decir, las columnas que van a continuación de las columnas del paso 2º. Estas aristas se pasan desde la fila activa (posiciones de columna después de la última arista con el

mismo valor en adelante) a cada fila recién creada en las mismas posiciones de columna respectivamente.

Después, en el programa completo que veremos más adelante, me posiciono dentro de la fila activa en la siguiente columna al grupo de aristas con el mismo valor, y si ya no existe dicha columna, paso a la fila siguiente, y si no existe ninguna fila más, se ha terminado el programa y se sacan por pantalla todas las soluciones.

(Creo filas nuevas en la matriz “aristas_ordenadas”)

crear_filas_nuevas

Cargo las filas recién creadas con las aristas (si existen) de la fila activa desde la primera arista hasta la arista anterior al grupo de aristas con el mismo valor

si columna > 1 , entonces

cargar_aristas_anteriores

finsi

Cargo las distintas soluciones únicas (grupos de aristas con el mismo valor) en la fila activa y las filas recién creadas

cargar_aristas_mismo_valor

Y por último, cargo las aristas (si existen) de la fila activa que están después del grupo de aristas con el mismo dato “valor” que estamos tratando en las filas recién creadas

si (columna + num – 1) < m , entonces

(Siendo “m” el número total de aristas del grafo)

cargar_aristas_posteriores

finsi

Procedimiento **"crear_filas_nuevas"**

Creo tantas filas nuevas en la matriz *"aristas_ordenadas"* como soluciones únicas calculadas menos una. Estas filas nuevas se sitúan después de la última fila (fila de mayor valor) de la matriz.

Importante : La variable global *"total_filas_aristas_ordenadas"* tiene que estar inicializada con el valor uno en el programa principal fuera del procedimiento *"probar_varias_soluciones"* y además sólo se inicializa una vez para que cuando se vuelva a entrar en el procedimiento *"crear_filas_nuevas"* conserve el valor que tenía cuando se salió de dicho procedimiento por última vez.

El pseudocódigo es:

Crear un número de filas ((num_soluciones_no_repetidas) – 1) añadidas a la matriz *"aristas_ordenadas"*.

total_filas_aristas_ordenadas ←

← (total_filas_aristas_ordenadas) + ((num_soluciones_no_repetidas) – 1)

Procedimiento **"cargar_aristas_anteriores"**

```
para  i = (total_filas_aristas_ordenadas) – (num_soluciones_no_repetidas) + 2
hasta  total_filas_aristas_ordenadas  ,  hacer
        para  j = 1  hasta  (columna – 1)  ,  hacer
                aristas_ordenadas [ i, j ]  ←  aristas_ordenadas [ fila, j ]
        finpara
finpara
```

Para comprender mejor este procedimiento, voy a hacer hincapié en un detalle significativo:

En el bucle *"para"* de la variable *"i"* (*"i"* representa los valores de las filas recién creadas), si queremos empezar por la primera fila recién creada, el primer valor de *"i"* será:

- El número de filas recién creadas es :

  (num_soluciones_no_repetidas) – 1

- Tenemos que restar del total de filas de la matriz *"aristas_ordenadas"* el valor anterior y después sumar uno para empezar por la primera fila recién creada.

Luego la primera fila recién creada tendrá un valor de:

(total_filas_aristas_ordenadas) – ((num_soluciones_no_repetidas) – 1) + 1 =

= (total_filas_aristas_ordenadas) – (num_soluciones_no_repetidas) + 2

y la última fila recién creada tendrá, lógicamente, el valor de *"total_filas_aristas_ordenadas"*.

Procedimiento **"cargar_aristas_mismo_valor"**

(Primero cargaré las aristas en la fila activa)

col ⟵ 0

para i = 1 hasta total_filas_solución_parcial , hacer

  si solución_parcial.fase [i , 1] ≠ "repetido" , entonces

    para j = columna hasta (columna + num – 1) , hacer

      col ⟵ col + 1

      aristas_ordenadas [fila , j] ⟵ solución_parcial [i , col]

      si solución_parcial.fase [i , col] = "solución" , entonces

        Actualizo la matriz "componente_conexa"
        con la nueva arista de la solución.

        actualizar_componentes_conexas (j , componente_conexa)

        (Sumo uno al número de aristas de la solución)
        número_aristas_solución ⟵ número_aristas_solución + 1

      finsi

    finpara

    solución_parcial.fase [i , 1] ⟵ "repetido"

    salir del para

  finsi

finpara

(Ahora cargaré las aristas en las filas recién creadas)

Antes de seguir con el pseudocódigo, daré una explicación de lo que representan algunas variables para entender mejor lo que queda de este procedimiento.

Para las filas recién creadas de la matriz *"aristas_ordenadas"*:

La variable *"i"* me indica el índice de las filas recién creadas de la matriz *"aristas_ordenadas"*. Inicialmente, dicha variable se inicializa a la primera fila recién creada:

("total_filas_aristas_ordenadas") – ("num_soluciones_no_repetidas") + 2

y al finalizar el proceso debe valer *"total_filas_aristas_ordenadas"*.

La variable *"p"* representa las columnas de la matriz *"aristas_ordenadas"* desde la primera a la última arista con el mismo dato *"valor"*, es decir, desde la posición indicada por la variable *"columna"* hasta *("columna" + "num" – 1)*.

Para la matriz *"solución_parcial"*:

La variable *"j"* representa las filas de dicha matriz y la variable *"col"* las columnas. La variable *"j"* se inicializa con el valor dos porque para el valor uno, si la fila uno no estaba repetida, ya ha sido cargada en la fila activa de la matriz *"aristas_ordenadas"* y hay que descartarla.

El pseudocódigo se muestra a continuación:

```
i ←— (total_filas_aristas_ordenadas) - (num_soluciones_no_repetidas) + 2
col ←— 0
para j = 2 hasta total_filas_solución_parcial , hacer
    si solución_parcial.fase [ j , 1 ] ≠ "repetido" , entonces
        para p = columna hasta (columna + num - 1) , hacer
            col ←— col + 1
            aristas_ordenadas [ i , p ] ←— solución_parcial [ j , col ]
        finpara
        col ←— 0
        i ←— i + 1
    finsi
finpara
```

La matriz "solución_parcial" ya no nos hace falta y la quito de la memoria para aprovechar mejor los recursos del ordenador.

Borrar de memoria la matriz solución_parcial

Procedimiento **“cargar_aristas_posteriores”**

Y por último, sólo nos queda cargar las aristas de la fila activa hacia las filas recién creadas a partir de la arista *“columna + num”* en adelante.

```
para   i = (total_filas_aristas_ordenadas)  –  (num_soluciones_no_repetidas)  +  2

hasta   total_filas_aristas_ordenadas   ,   hacer


    para   j = columna + num   hasta   m   ,   hacer

            aristas_ordenadas [ i , j ]  ←——  aristas_ordenadas [ fila , j ]

    finpara


finpara
```

Procedimiento **"busca_componente_conexa (comp_conexa)"**

Este procedimiento calcula el valor de la fila de la matriz *"comp_conexa"* en donde se encuentra una componente conexa a partir de la fila *"fila_componente_conexa"*. Se pasa como parámetro la matriz *"comp_conexa"*.

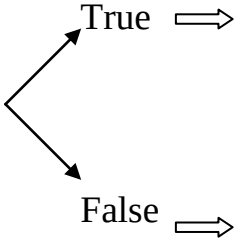
El pseudocódigo es:

```
i ←—— fila_componente_conexa
para  j = i  hasta  n  ,  hacer
    si  comp_conexa [ j , 1 ] ≠ - 1  ,  entonces
        salir del para
    sino
        fila_componente_conexa ←—— fila_componente_conexa + 1
    finsi
finpara
```

El resultado final (la fila encontrada en donde esta la componente conexa) queda en la variable global *"fila_componente_conexa"*.

Función "**misma_componente (nodo, comp_conexa)**"

La función booleana *"misma_componente"* nos devolverá los valores:



True ⟹ Si los dos nodos de la arista seleccionada están en la misma componente conexa.

False ⟹ En caso contrario.

Examinamos la componente conexa del *"nodop"* que esta en la fila *"fila_componente_conexa"* desde la primera columna para buscar el *"nodo"* que se pasa como parámetro.

Esta función en pseudocódigo es:

para j = 1 hasta n , hacer

si comp_conexa [fila_componente_conexa , j] = - 1 , entonces

Cuando encontramos un valor –1 en la componente conexa ya no hay más nodos que buscar y terminamos.

misma_componente ⟵ False

fin de la función

finsi

si comp_conexa [fila_componente_conexa , j] = nodo , entonces

Si he encontrado el “nodo” en la componente conexa, terminamos.

misma_componente ⟵ True

fin de la función

finsi

finpara

Función **inicio (comp_conexa)**

Nos da la posición de la columna del primer valor –1 de la componente conexa del *nodop* de la matriz *"comp_conexa"*. La fila de la componente conexa del *nodop* esta en la variable global *"componentep"*.

```
para  j = 1  hasta  n  ,  hacer
    si  comp_conexa [ componentep , j ] = -1  ,  entonces
            inicio = j
            fin de la función
    finsi
finpara
```

Función **fin (comp_conexa)**

Nos da el número de nodos que irán a añadirse de la componente conexa del *"nodoq"* a la componente conexa del *"nodop"*. La fila de la componente conexa del *"nodoq"* esta en la variable global *"componenteq"*.

```
contador ⟵ 0
para  j = 1  hasta  n  , hacer
    si  comp_conexa [ componenteq , j ]  ≠  -1  ,  entonces
            contador ⟵ contador + 1
    sino
            fin ⟵ contador
            fin de la función
    finsi
finpara
```

Procedimiento **unir componentes_conexas (comp_conexa)**

Uno las componentes conexas en donde están los nodos *"nodop"* y *"nodoq"* en una sola componente conexa situada en la fila de la componente conexa del *nodop* y hago desaparecer la componente conexa de la fila en donde esta el *nodoq*.

(Llamo a las funciones "inicio" y "fin")

k ⟵ inicio (comp_conexa)

r ⟵ fin (comp_conexa)

s ⟵ 1

para j = k hasta k + r , hacer

comp_conexa [componentep , j] ⟵ comp_conexa [componenteq , s]

s ⟵ s + 1

finpara

(ahora, elimino la componente conexa del nodoq)

para j = 1 hasta r , hacer

comp_conexa [componenteq , j] ⟵ -1

finpara

Procedimiento "**componente_nodo (nodo, comp_conexa)**"

Encuentra la componente conexa en donde esta un *"nodo"* que se pasa como parámetro de entrada del procedimiento. Se empieza a buscar desde la fila contenida en la variable *"fila_componente_conexa"*.

```
mientras  fila_componente_conexa ≤ n  ,  hacer
    para  j = 1  hasta  n  ,  hacer
        si  comp_conexa [fila_componente_conexa , j ]  <  0  , entonces
            cuando halla recorrido todos los valores de la fila no negativos
            sin éxito, salimos del bucle para buscar en otra fila
                salir del para
        finsi
        si  comp_conexa [fila_componente_conexa , j ]  =  nodo  ,  entonces
                (en "fila_componente_conexa" tengo la
                 fila de la componente conexa del nodo)
                salir del procedimiento
        finsi
    finpara
    fila_componente_conexa = fila_componente_conexa + 1
    busca_componente_conexa (comp_conexa)
finmientras
```

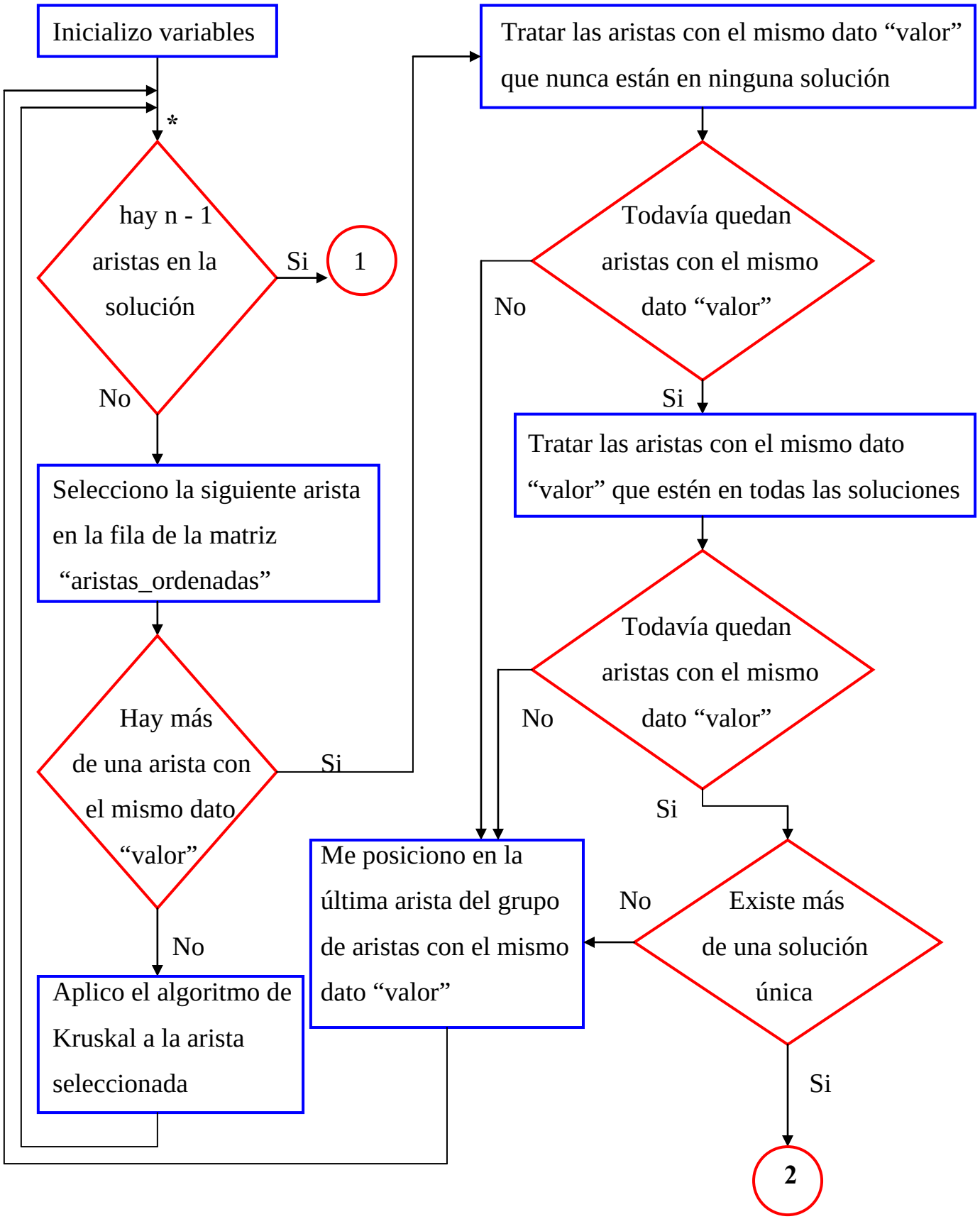
2.2.3.3.- Solución completa tratando todas las aristas del grafo

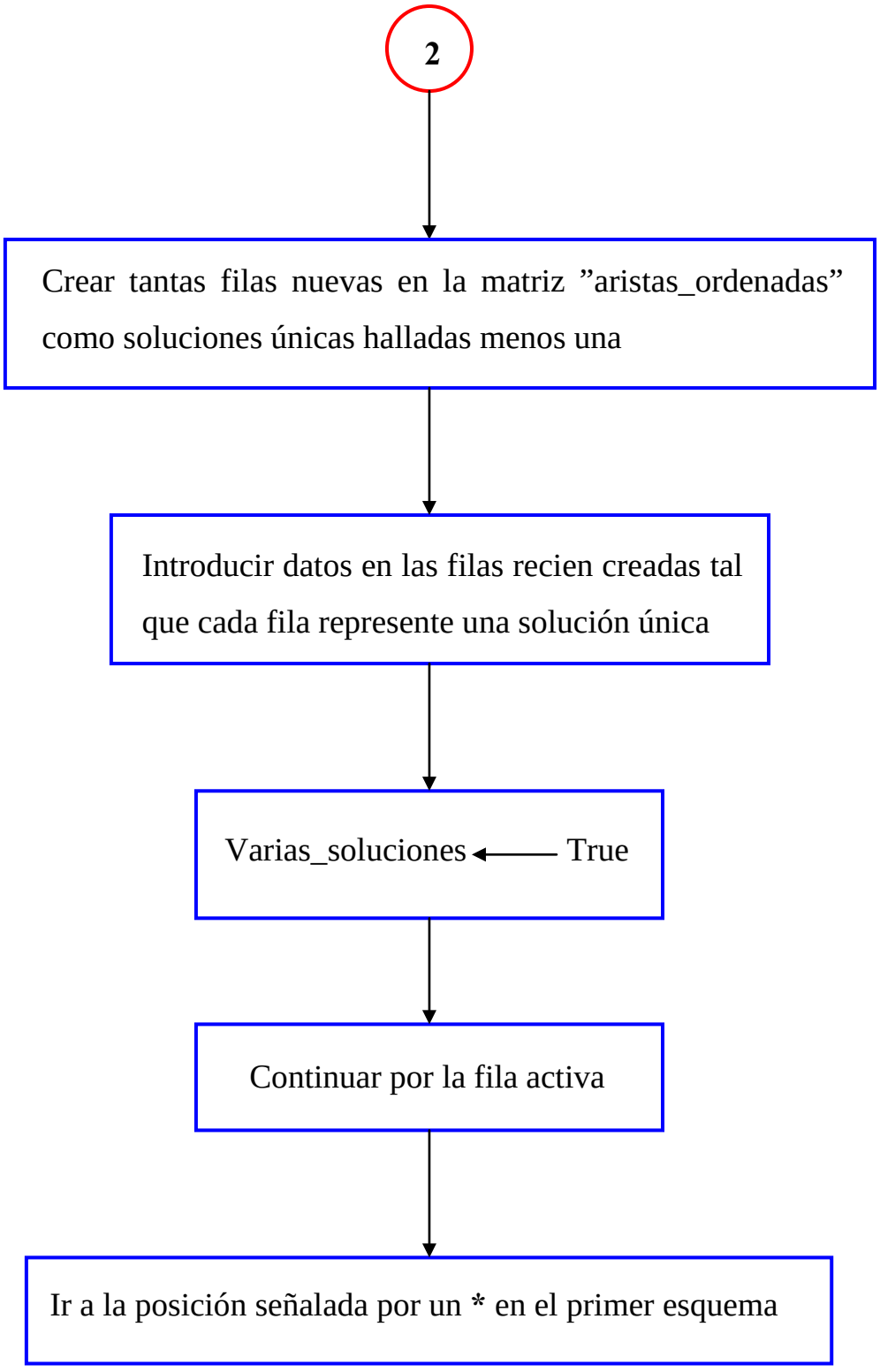
Desarrollaré el organigrama funcional completo que hay que seguir pasando directamente al pseudocódigo que implemente dicho organigrama completando todas las posibles soluciones.

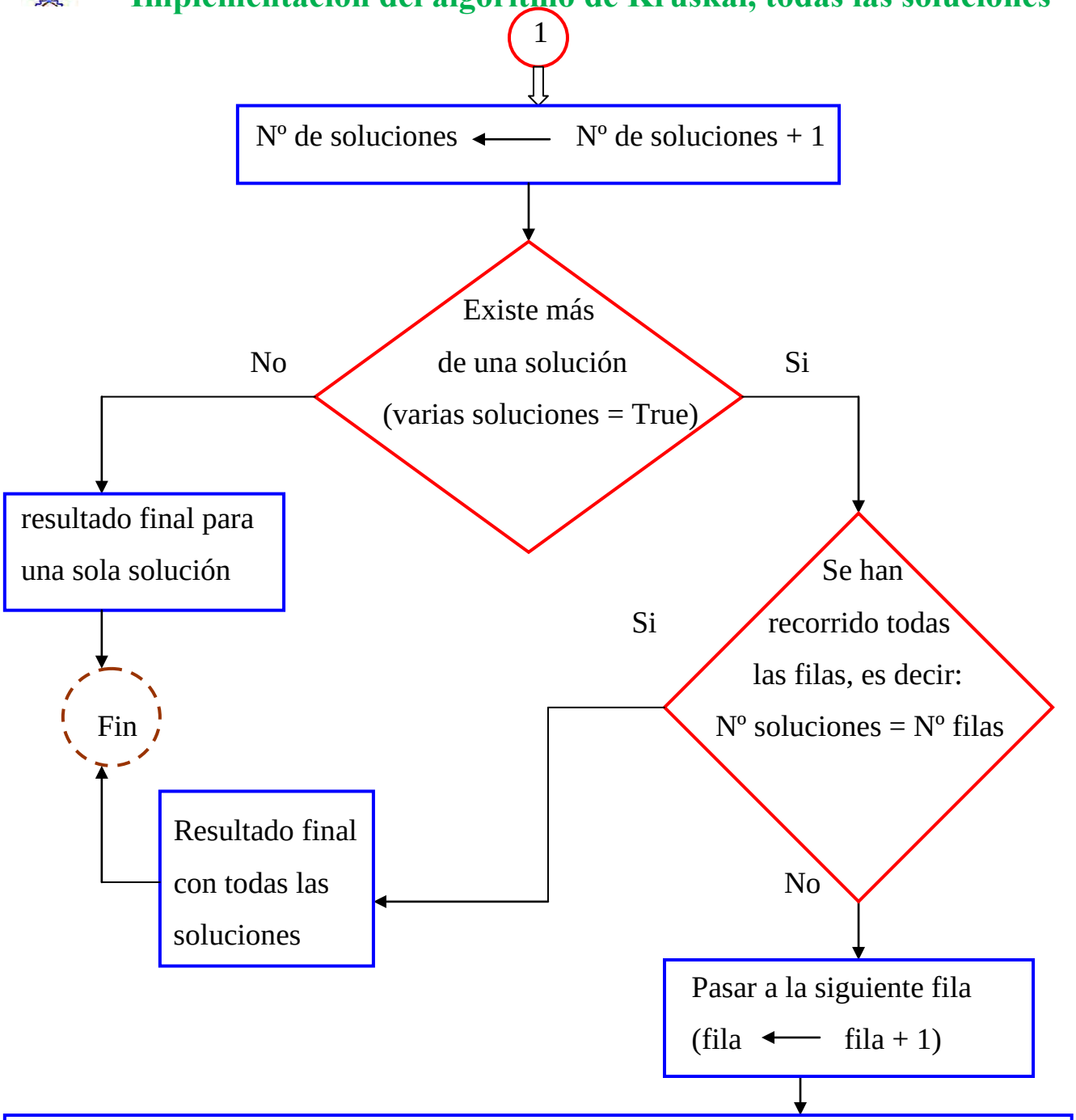
El planteamiento es ahora totalmente distinto al proceso que se realizaba cuando el usuario seleccionaba las aristas en el grafo. Al ser una solución automática independiente del usuario, se van escogiendo siempre las aristas correctas (cuando actuaba el usuario, sin embargo, había que tener en cuenta que arista había elegido y obrar en consecuencia), esto supone una ventaja pero se tienen que calcular todas las soluciones posibles que puedan darse (el usuario sólo hallaba, como es lógico, una solución óptima cualquiera a la vez interactuando con el grafo).

Esto nos plantea un algoritmo muy distinto aunque algunas funciones y procedimientos realizados cuando el usuario resuelve el problema de calcular una solución óptima se pueden aprovechar si se adaptan a las dos dimensiones que tiene ahora la matriz *"aristas_ordenadas"*.

Esquema funcional del desarrollo de **todas las soluciones**




2
Crear tantas filas nuevas en la matriz ”aristas_ordenadas” como soluciones únicas halladas menos una
Introducir datos en las filas recien creadas tal que cada fila represente una solución única
Varias_soluciones ← True
Continuar por la fila activa
Ir a la posición señalada por un * en el primer esquema



Me posiciono en la 1ª columna de la nueva fila. Avanzo por sus elementos actualizando la matriz "componente_conexa" para cada arista que esté en la solución hasta la 1ª arista en donde comienza el trozo de fila no explorada (dato "fase" = "disponible"). También he calculado el número de aristas de la solución hasta la primera arista no explorada.

Ir a la posición señalada por un * en el primer esquema para seguir con el proceso a partir de la primera arista del trozo de fila no explorada en adelante

Variables

La explicación de estas variables dará una idea clara del flujo de control del programa lo que aclarará mucho su seguimiento.

fin_del_programa ⟶ El programa se compone de un bucle *"mientras"* que se ejecuta hasta que se produce una situación que debe terminar la ejecución del programa para sacar los resultados finales. Para controlar esta finalización, es decir, salirse del bucle *"mientras"* he creado esta variable booleana que inicialmente es *"False"* y cuando hay que sacar la o las soluciones óptimas finales se pone a *"True"*.

varias_soluciones ⟶ Variable booleana que nos indica si se ha añadido alguna fila a la matriz *"aristas_ordenadas"*, es decir, si existen varias soluciones. Es *"False"* inicialmente y sólo se pone a *"True"* cuando existe más de una fila (número de soluciones óptimas igual al número de filas).

número_de_soluciones ⟶ Nos da el número total de soluciones óptimas.

número_aristas_solución ⟶ Es el contador del número de aristas de una solución óptima en concreto. Se hace muy significativo cuando toma el valor de *"n – 1"* siendo *"n"* el número de nodos del grafo. Con este valor se completa una determinada solución óptima.

fila ⟶ Su valor nos da la fila activa, es decir, la fila de la matriz *"aristas_ordenadas"* en la que se está aplicando el algoritmo de Kruskal.

fila_solución_parcial ⟶ Nos dice la fila de la matriz *"solución_parcial"* en la que estamos.

total_filas_solución_parcial ⟶ Nos indica el total de filas de la matriz *"solución_parcial"* incluidas las filas con soluciones repetidas para las aristas con el mismo valor.

num_soluciones_no_repetidas ⟶ Nos dice el total de filas de la matriz *"solución_parcial"* excluidas las filas con soluciones repetidas.

total_filas_aristas_ordenadas ⟶ Nos indica en todo momento el número total de filas de la matriz *"aristas_ordenadas"*.

Programa principal

(Inicializo variables)

número_de_soluciones ← 0

número_aristas_solución ← 0

fila ← 1

total_filas_aristas_ordenadas ← 1

columna ← 0

varias_soluciones ← False

fin_del_programa ← False

mientras no fin_del _programa , hacer

si número_aristas_solución < n – 1 , entonces

(Selecciono la siguiente arista)

columna ← columna + 1

nodop ← aristas_ordenadas.nodoi [fila , columna]

nodoq ← aristas_ordenadas.nodoj [fila , columna]

si ((columna < m) y aristas_ordenadas.valor [fila , columna] = = aristas_ordenadas.valor [fila , columna + 1]) , entonces

(Pueden existir varias soluciones)

probar_varias_soluciones

sino

(Aplico el algoritmo de Kruskal a la arista seleccionada)
sigo_una_solución

finsi

```
sino
        número_de_soluciones ←— número_de_soluciones + 1
    si  varias_soluciones  ,  entonces
        (Si existen varias soluciones, se pasa a la fila siguiente de la matriz (si existe
        dicha fila) para continuar con otra solución. Si ya no quedan más soluciones
        (más filas), se sacan los resultados y se termina el programa.)
        si  número_de_soluciones = total_filas_aristas_ordenadas  ,  entonces
                sacar_todas_las_soluciones
                fin_del_programa ←— True
        sino
                recorrer_matriz
        finsi
    sino
            (Si sólo existe una solución, se muestra y se termina el programa)
            resultado_una_solución
            fin_del_programa ←— True
    finsi
  finsi
finmientras
```

Procedimiento **"sigo_una_solución"**

Tengo que aplicar el algoritmo de Kruskal a la arista seleccionada sabiendo que no hay más aristas con el mismo dato *"valor"* que son candidatas a entrar en el conjunto de aristas de la solución, es decir, no se pueden producir varias soluciones.

Busco la primera componente conexa a partir de la fila 1 en la matriz "componente_conexa"

fila_componente_conexa ← 1

busca_componente_conexa (componente_conexa)

Busco la componente conexa, que es una fila de la matriz "componente_conexa", en donde esta el "nodop" y la guardo en la variable "fila_componente_conexa".

componente_nodo (nodop, componente_conexa)

A continuación examino dicha componente conexa desde el principio para saber si el otro "nodoq" esta también en la misma componente conexa.

si misma_componente (nodoq, componente_conexa) , entonces

aristas_ordenadas.fase [fila , columna] ← "forma_ciclo"

sino

componentep ← fila_componente_conexa

Se busca donde esta la fila de la otra componente conexa del "nodoq" y esa fila la guardaré en la variable "componenteq"

Busco una componente conexa a partir de la fila 1 en la matriz “componente_conexa”

fila_componente_conexa ← 1

busca_componente_conexa (componente_conexa)

Busco la componente conexa en donde esta el “nodoq” y la guardo en la variable “fila_componente_conexa”.

componente_nodo (nodoq, componente_conexa)

componenteq ← fila_componente_conexa

Uno las componentes conexas en donde están los nodos “nodop” y “nodoq” en una sola componente conexa

unir_componentes_conexas (componente_conexa)

(Añado la arista elegida al conjunto de aristas solución)

aristas_ordenadas.fase [fila , columna] ←“solución”

número_aristas_solución ← número_aristas_solución + 1

finsi

Procedimiento **“probar_varias soluciones”**

Este procedimiento ya está hecho en la segunda fase de estos apuntes y al no producirse ninguna variación cuando se integra en el programa general no lo voy a escribir de nuevo, con lo cual, se simplifica enormemente la implementación del programa general que estoy desarrollando ahora.

Para dar una orientación, si voy a mostrar el pseudocódigo del procedimiento *“probar_varias_soluciones”* pero no voy a desarrollar nada más después.

Calculo el número de aristas seguidas con el mismo dato “valor” utilizando la función “num_aristas_mismo_valor”

num ⟵ num_aristas_mismo_valor

primero trataré las aristas que nunca están en ninguna solución óptima (sí las hay)

aristas_nunca

Después trataré las aristas que están siempre en todas las soluciones òptimas (si las hay)

aristas_siempre

Y por último, trataré las aristas que generan varias soluciones ùnicas (sí las hay)

aristas_varias_soluciones

Procedimiento **"recorrer_matriz"**

Me posiciono en la siguiente fila y parto de la matriz *"componente_conexa"* inicial.

Empiezo a recorrer la nueva fila desde la primera columna en adelante y voy mirando el dato *"fase"*, si es *"solución"*, uno las componentes conexas de los nodos de la arista que estamos tratando. El proceso se termina cuando encontramos un dato *"fase" = "disponible"*, o bien, cuando el número de aristas que tiene la solución es *"n – 1"*, siendo *"n"* el número de nodos del grafo, ya que se puede producir la situación de que la última arista de la solución sea la última arista de la matriz *"aristas_ordenadas"*.

```
número_aristas_solución ←— 0
fila ←— fila + 1
columna ←— 1
(Inicializo la matriz "componente_conexa")
para i = 1 hasta n , hacer
    componente_conexa [ i , 1 ] ←— i
    para j = 2 hasta n , hacer
        componente_conexa [ i , j ] ←— - 1
    finpara
finpara
```

mientras [(columna ≤ m) y (aristas_ordenadas.fase [fila, columna] ≠ "disponible")] hacer

    si aristas_ordenadas.fase [fila, columna] = "solución" , entonces

        actualizar_componentes_conexas (columna , componente_conexa)

        número_aristas_solución ← número_aristas_solución + 1

    finsi

    columna ← columna + 1

    si numero_aristas_solución = n – 1 , entonces

        salir del bucle mientras interno

    finsi

fin mientras

columna ← columna - 1

Procedimiento **"resultado_una_solución"**

Las aristas de la solución se ponen en color rojo y en una ventana se muestran las aristas con sus valores y la suma total.

Los datos se obtienen según el siguiente procedimiento:

```
Mensaje: "Sólo existe una solución óptima"
suma ⟵ 0
i ⟵ 1
mientras i ≤ m , hacer
    si aristas_ordenadas.fase [ fila , i ] = "disponible" , entonces
        salir del mientras
    finsi
    si aristas_ordenadas.fase [ fila , i ] = "solución" , entonces
        nodop ⟵ aristas_ordenadas.nodop [ fila , i ]
        nodoq ⟵ aristas_ordenadas.nodoq [ fila , i ]
        valor ⟵ aristas_ordenadas.valor [ fila , i ]
        Mostrar "nodop" , "nodoq" , "valor"
        suma ⟵ suma + valor
        poner en rojo la arista
    finsi
    i ⟵ i + 1
finmientras
Mostrar "suma"
```

Procedimiento **"sacar_todas_las_soluciones"**

Cada fila de la matriz *"aristas_ordenadas"* contiene una solución óptima. Para cada fila, las aristas con el dato *"fase" = "solución"* estarán en la solución óptima correspondiente a su fila, luego tengo que recorrer fila por fila desde la primera columna e ir seleccionando las aristas solución de cada fila para mostrar todas las soluciones.

Cuando se muestre una determinada solución, las aristas solución se marcarán en rojo en el grafo y saldrá una ventana con dichas aristas y su suma.

El procedimiento para obtener todas las soluciones es el siguiente:

```
para  i = 1  hasta  total_filas_aristas_ordenadas  ,  hacer

    número_aristas_solución  ←——  0

    suma  ←——  0

    para  j = 1  hasta  m  ,  hacer

        si  aristas_ordenadas.fase [ i , j ]  =  "solución"  ,  entonces

            Poner la arista "aristas_ordenadas [ i , j ]" en color rojo

            Incluir la arista con el resto de aristas solución que van
            en la ventana de salida

            número_aristas_solución  ←——  número_aristas_solución  +  1

            suma  ←——  suma  +  aristas_ordenadas.valor [ i , j ]

        finsi

        si  número_aristas_solución = n – 1  ,  entonces

            salir del para interno

        finsi

    finpara

finpara
```

2.2.3.4.- Modificaciones del programa para resolver situaciones con demasiadas soluciones

Se pueden hacer dos mejoras muy importantes a la vez que necesarias dentro del procedimiento *"probar_varias_soluciones"*:

- Pasar los algoritmos *"solución"* y *"generar_soluciones"* de recursivos a iterativos.

- Cambiar la programación en el procedimiento *"aristas_varias_soluciones"* de tal forma que la matriz *"solución_parcial"* sólo almacene soluciones únicas desde el principio.

No es necesario hacer modificaciones en el programa principal puesto que al tener una estructura modular, sólo se ha modificado el procedimiento (módulo) *"probar_varias_soluciones"* y el resto queda igual. Por tanto, el organigrama del programa general queda intacto y el código también ya que toma el procedimiento *"probar_varias_soluciones"* como un único bloque.

Paso de los procedimientos recursivos "solución" y "generar_soluciones" a procedimientos iterativos

Para optimizar el rendimiento de la aplicación en cuanto a tiempo de ejecución y consumo de memoria, tengo que pasar los procedimientos recursivos del algoritmo de Kruskal que calcula todas las soluciones llamados *"solución"* y *"generar_soluciones"* a un modelo iterativo, en el que el resultado de cada procedimiento es hallado realizando repeticiones de tareas mediante bucles, en lugar de llamadas del procedimiento a sí mismo, lo que generalmente lleva más tiempo, consume más memoria (la pila ha de almacenar en cada llamada los parámetros que el procedimiento se pasa a sí mismo junto con nuevas copias de todas las variables locales y la dirección de retorno del procedimiento) y es más inseguro (esa pila de llamadas se puede desbordar). La descripción del paso del modelo recursivo al iterativo para los procedimientos *"solución"* y *"generar_soluciones"* coincide, ya que ambos tienen la misma estructura y se diferencian en muy pocas instrucciones.

Para hacer todo esto, siempre es necesaria una estructura de datos que supla a la pila de llamadas; esta se puede implementar mediante una matriz local del procedimiento que se llene en un orden y se vacíe en orden inverso, comportándose en la práctica como una pila. La posición de la cima de esta pila en forma de índice de la matriz se almacena en una variable entera. Se sigue utilizando, por tanto, una estructura de pila, pero ésta ya sólo almacena los mínimos datos necesarios (no todas las variables locales del procedimiento) y no los almacena en la misma pila que guarda las direcciones de retorno de los procedimientos sucesivamente llamados, motivo por el cual el desbordamiento de esta pila recursiva es mucho más crítico para el programa que el desbordamiento de una pila en forma de matriz cuyo contenido sea tan sólo de datos (como la que se implementa en la versión iterativa), que por otro lado no desbordará, ya que en nuestro caso sabemos exactamente cuánta memoria va a necesitar (*"num"* elementos de tamaño entero) y se la reservamos al comienzo, por lo que daría como mucho un error de asignación dinámica de memoria, y no de

desbordamiento. Este posible, aunque poco probable, error de desbordamiento esta controlado mediante un mensaje de error que se lanza al usuario y éste puede continuar con el programa normalmente.

A continuación se muestran los procedimientos iterativos y después una descripción de los elementos que componen el pseudocódigo para poder entenderlos mejor.

Procedimiento iterativo **"solución"**

```
cima ⟵ 2
retorno ⟵ False
pila [ 1 ] ⟵ 1

repetir
        cima ⟵ cima – 1

        para   i = pila [ cima ]    hasta   k   ,   hacer

                si   (retorno)   o   (no  iguales ( i ))   ,   entonces

                        si   no   retorno

                                orden [ k – num + 1]    ⟵     i

                        finsi

                        si   (retorno)  o  (num > 1)   ,   entonces

                                si no retorno
                                        num ⟵    num  –  1
                                        pila [ cima ]    ⟵    i
                                        cima   ⟵   cima  +  1
                                        i  ⟵    0
                                        continuar siguiente iteración
                                sino
                                        retorno  ⟵   False
                                        num  ⟵   num  +  1
                                finsi
```

sino

Para no modificar la matriz ”componente_conexa” trabajaré con una copia de ella con los valores que tiene justo antes de aplicar el algoritmo de Kruskal al grupo de aristas con el mismo valor.

inicializo_componente_conexa

Se reemprende el algoritmo de Kruskal desde el punto en donde se quedó (justo antes del grupo de aristas con el mismo valor) para probar sólo con las aristas con el mismo dato “valor” y cuyo orden de selección está dado por la matriz “orden”.

lanzar_solución

orden [k – num + 1] ⟵ 0

finsi

finsi

finpara

orden [k – num +1] ⟵ 0

si num < k , entonces

orden [k – num] ⟵ 0

finsi

retorno ⟵ True

hasta k = num

Procedimiento iterativo **"generar_soluciones"**

```
cima ⟵ 2
retorno ⟵ False
pila [ 1 ] ⟵ 1

repetir
        cima ⟵  cima – 1

        para  i = pila [ cima ]   hasta  k  ,  hacer

                si  (retorno)  o  (no  iguales ( i ))  ,  entonces

                        si no retorno

                                orden [ k – num + 1]  ⟵  i

                        finsi

                        si  (retorno)  o  (num > 1)  ,  entonces

                                si  no  retorno
                                        num ⟵  num  –  1

                                        pila [ cima ]  ⟵ i

                                        cima  ⟵ cima  +  1

                                        i ⟵  0

                                        continuar siguiente iteración
                                sino
                                        retorno  ⟵ False
                                        num ⟵  num  +  1
                                finsi
```

```
                sino

                    Para no modificar la matriz ”componente_conexa” trabajaré
                    con una copia de ella con los valores que tiene justo antes de
                    aplicar el algoritmo de Kruskal al grupo de aristas con el
                    mismo valor.

                    inicializo_componente_conexa

                    Pongo una solución única obtenida con el algoritmo de
                    Kruskal en una fila de la matriz “solución_parcial”.

                    cargar_combinación

                    orden [ k – num + 1 ]  ⟵  0

                finsi

            finsi

        finpara

        orden [ k – num +1 ]  ⟵  0

    ┌── si  num < k  ,  entonces
    │
    │       orden [ k – num ]  ⟵  0
    │
    └── finsi

        retorno  ⟵ True

hasta  k = num
```

Descripción de los elementos más importantes para explicar el pseudocódigo

- **pila**: una matriz que se comportará como una pila para guardar el valor de *"i"* (el contador del bucle dentro del cual se hace la llamada recursiva). Al realizarse una llamada en cada iteración dentro de la cual se cumplan, además, las condiciones (no iguales (*i*)) y (*num* > 1), la variable *"i"* tiene un valor cualquiera entre 0 y *"k - 1"* cada vez que se realiza la llamada. Si se va a emular una llamada recursiva, habrá que volver al principio del procedimiento, que es el principio del bucle, ejecutándolo de nuevo desde *i = 0*, pero al retorno de la llamada recursiva, o en el momento en que se emula el retorno, la variable *"i"* ha de recuperar el valor que tenía previamente para no empezar a ejecutar el bucle desde el principio, sino desde la iteración en la que se hubiese quedado, por lo que el valor debió ser almacenado; como esto ha de ocurrir con el valor de *"i"* cada vez que se emula una llamada recursiva, y se emularán *"n"* llamadas de hasta *"num"* niveles de profundidad, se almacenará cada valor de *"i"* en esta matriz de *"num"* elementos, y se recuperará del mismo.

- **cima**: un entero que indicará qué elemento de la matriz *"pila"* representa la cima: cuando vale cero, la cima es el primer elemento (índice 0) y sólo hay un elemento en la pila; cuando vale uno, la cima es el segundo elemento (índice 1) y hay dos elementos en la pila, y así sucesivamente (en Java, el 0 es el índice del primer elemento de una matriz; en el pseudocódigo que he seguido en la documentación del proyecto es el 1, y así aparecerá en este documento).

- **num**: indicador de profundidad de llamada, que sirve para caracterizar cada llamada particular y para distinguir cualquier caso recursivo del caso trivial

(el caso recursivo se da para valores de *"num"* mayores de uno, y el trivial para *"num"* igual a uno). Cada vez que se emula una nueva llamada se decrementa *"num"*, y se incrementa cada vez que emulamos un retorno.

- **orden**: Matriz en la que se almacena una permutación única de los valores entre 0 y *"num - 1"* (1 y *"num"* en pseudocódigo); cuando esté completamente lleno, el nivel de profundidad de llamada recursiva o, en el caso iterativo, el valor de la cima de la pila, han alcanzado sus valores máximos, y *"num"* ha alcanzado su valor mínimo (1), con lo que se ejecuta el caso trivial (que genera una nueva solución parcial). Su uso no cambia en la versión iterativa.

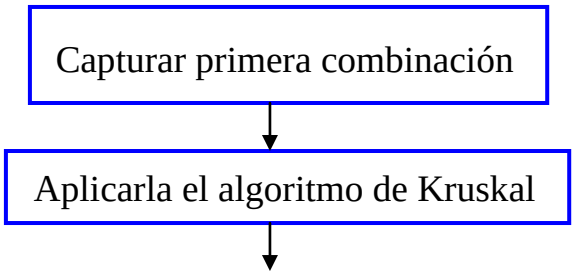
- **retorno**: es una variable booleana responsable de la *"emulación"* de la recursividad mediante un diseño iterativo; en el lugar en el que antes se hacía una llamada recursiva pasando el parámetro *"num - 1"*, ahora se decrementa *"num"*, se almacena en la pila el valor del contador *"i"* y se le da como nuevo valor el 0, de manera que al continuar la ejecución del bucle (no llegando a ejecutar el resto del código del mismo, como pasaría al realizar una llamada recursiva), se haría primero el incremento, por lo que la *"i"* valdría uno, como pasaría al entrar de nuevo en el procedimiento al realizar una llamada recursiva. En el momento en que termine de ejecutarse el bucle *"para"* interno, se produciría un retorno en la función recursiva, lo que emulamos poniendo *"retorno"* a *True* (verdadero) y volviendo mediante un nuevo bucle, externo al *"para"*; la *"i"* recupera, en ese momento, el valor que tuviese anteriormente desde la pila, y se ha de llagar al punto dentro del bucle inmediatamente posterior a aquél en el que se realizaban las acciones de emulación de llamada recursiva, por lo que se ha de obviar todo el código dentro del bucle hasta ese punto condicionándolo a que se ejecute sólo si *"retorno"* vale falso; llegados al punto posterior a

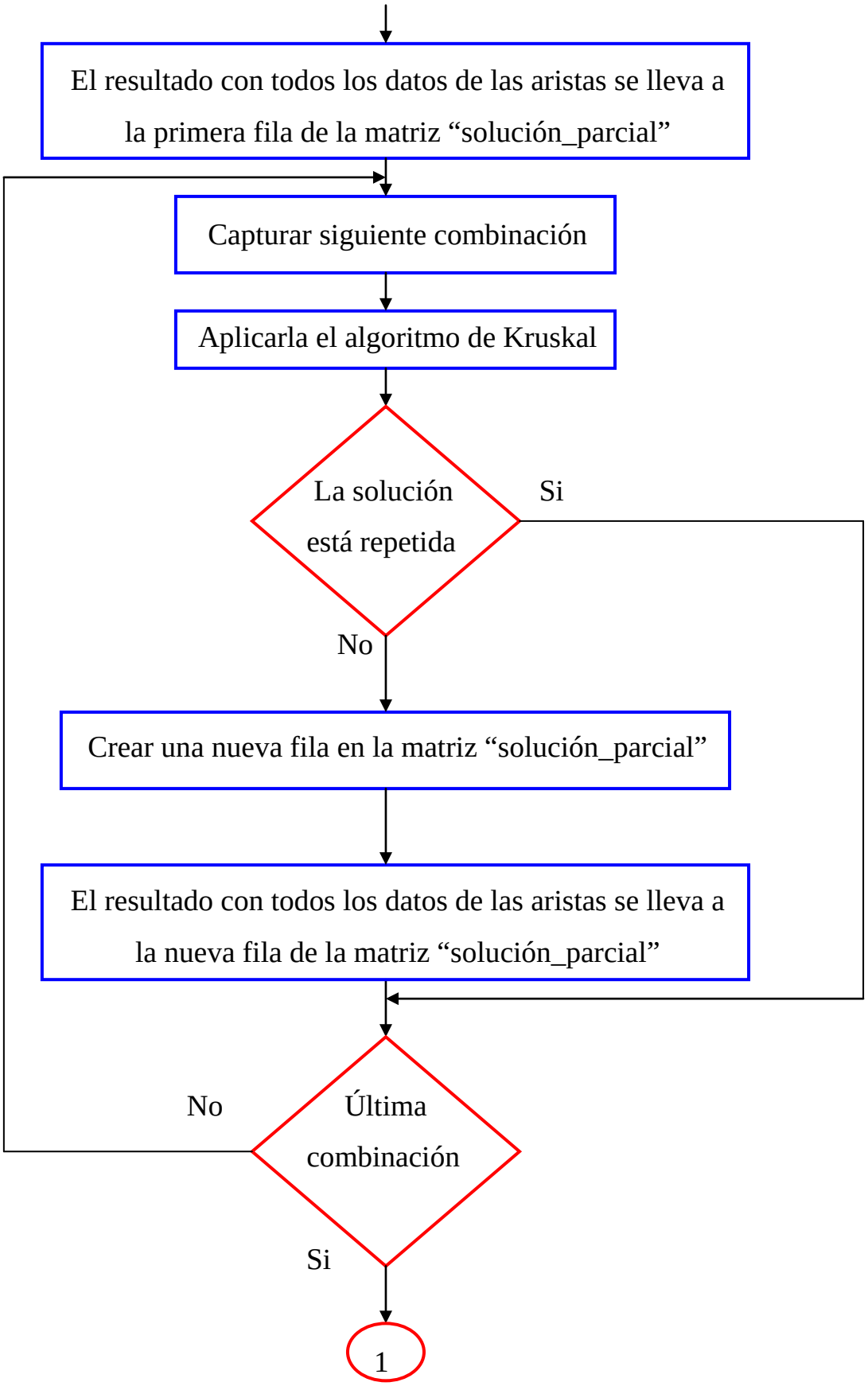
aquél en el que se emulaba la llamada, se vuelve a poner *"retorno"* a falso y se incrementa *"num"*.

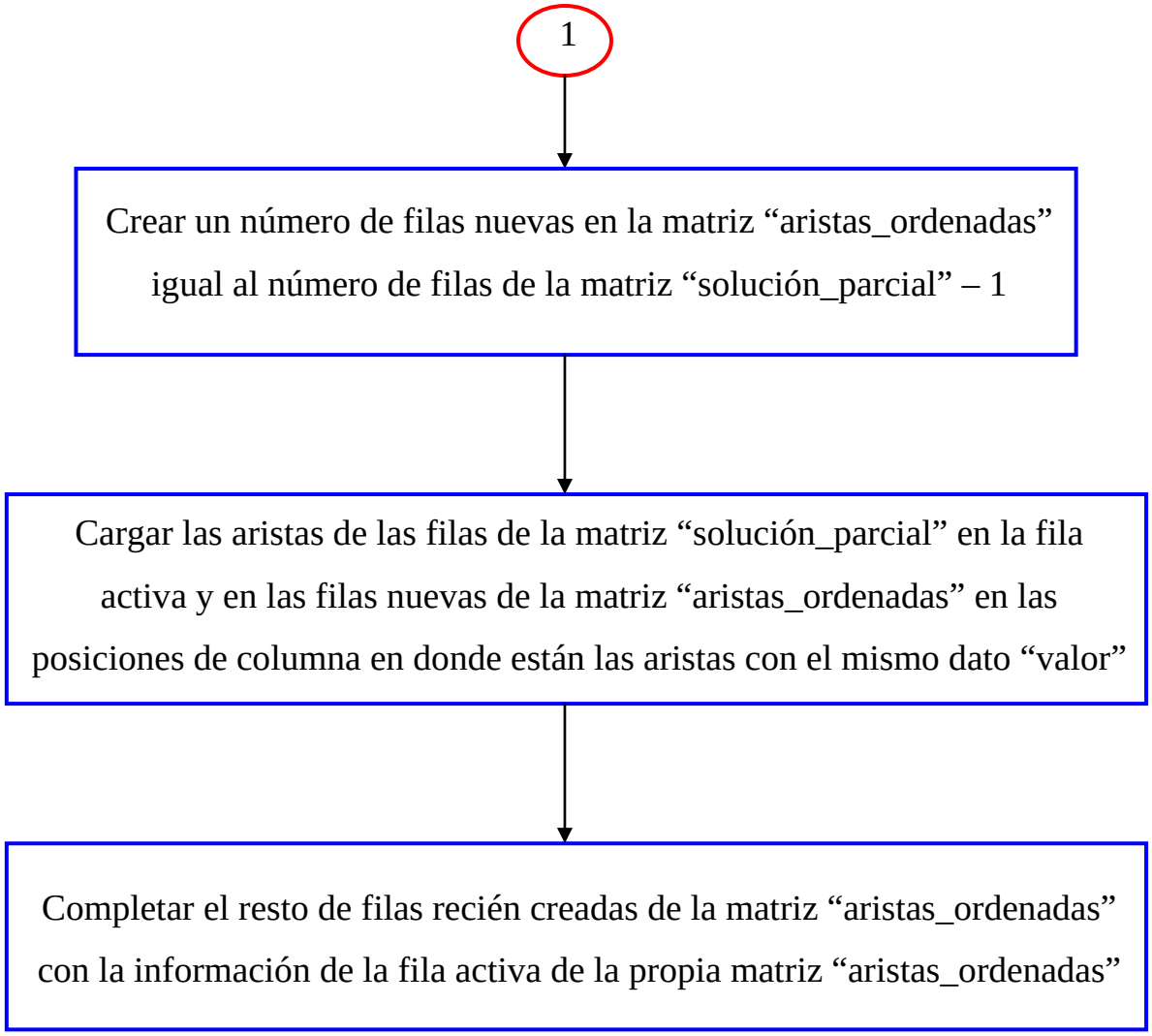
Cambiar el procedimiento "aristas_varias_soluciones" para que la matriz "solución_parcial" sólo almacene soluciones únicas

Las distintas combinaciones en orden de selección de aristas sobre el grupo de aristas con el mismo valor nos la da el procedimiento iterativo *"generar_soluciones"*. Se examinan todas las posibles combinaciones y para cada una de ellas se tienen que realizar los pasos mostrados en el organigrama siguiente con el fin de conseguir al finalizar el proceso que cada fila de la matriz *"aristas_ordenadas"* tenga una solución óptima única.

A continuación del organigrama, se desarrollan los procedimientos en pseudocódigo.




El resultado con todos los datos de las aristas se lleva a la primera fila de la matriz “solución_parcial”
Capturar siguiente combinación
Aplicarla el algoritmo de Kruskal
La solución está repetida
Si
No
Crear una nueva fila en la matriz “solución_parcial”
El resultado con todos los datos de las aristas se lleva a la nueva fila de la matriz “solución_parcial”
Última combinación
No
Si
1


1
Crear un número de filas nuevas en la matriz “aristas_ordenadas” igual al número de filas de la matriz “solución_parcial” – 1
Cargar las aristas de las filas de la matriz “solución_parcial” en la fila activa y en las filas nuevas de la matriz “aristas_ordenadas” en las posiciones de columna en donde están las aristas con el mismo dato “valor”
Completar el resto de filas recién creadas de la matriz “aristas_ordenadas” con la información de la fila activa de la propia matriz “aristas_ordenadas”

Procedimiento "**aristas_varias_soluciones**"

(Inicializo a cero los elementos de la matriz "orden")

para i = 1 hasta num , hacer

    orden [i] ⟵ 0

finpara

La variable "fila_solución_parcial" cambia de nombre y pasa a ser "última_fila_solución_parcial" para expresar mejor su significado

última_fila_solución_parcial ⟵ 0

Cargo todas las soluciones únicas posibles del grupo de aristas con el mismo valor sobre las filas de la matriz "solución_parcial".

k ⟵ num

(Utilizo el procedimiento iterativo para generar estas soluciones.)

generar_soluciones

Después de ejecutarse el procedimiento "generar_soluciones" en la variable "última_fila_solución_parcial" se encuentra el total de las filas de la matriz "solución_parcial"

si última_fila_solución_parcial = 1 , entonces

(Cargo la fila activa con la única solución)

col ← 0

para j = columna hasta (columna + num – 1) , hacer

col ← col + 1

aristas_ordenadas [fila , j] ← solución_parcial [1 , col]

si solución_parcial.fase [1 , col] = "solución" , entonces

Actualizo la matriz "componente_conexa" con la nueva arista de la solución.

actualizar_componentes_conexas (j , componente_conexa)

(Sumo uno al número de aristas de la solución)

número_aristas_solución ← número_aristas_solución + 1

finsi

finpara

sino

Aumento el número de filas de la matriz "aristas_ordenadas" según el número de soluciones únicas calculado. Reparto las distintas soluciones entre las filas recién creadas y la fila activa

ampliar_matriz

finsi

(Coloco la columna en la última arista del grupo de aristas con el mismo valor)

columna ← columna + num – 1

Procedimiento iterativo **"generar_soluciones"**

```
cima ⟵ 2

retorno ⟵ False

pila [ 1 ] ⟵ 1

repetir

    cima ⟵ cima – 1

    para  i = pila [ cima ]   hasta  k  ,  hacer


    si  (retorno)  o  (no  iguales ( i ))  ,  entonces

        si  no  retorno

        orden [ k  –  num  +  1]  ⟵  i

        finsi

        si  (retorno)  ó  (num  >  1)  ,  entonces
```

si no retorno

num ⟵ num – 1

pila [cima] ⟵ i

cima ⟵ cima + 1

i ⟵ 0

continuar siguiente iteración

sino

retorno ⟵ False

num ⟵ num + 1

finsi

sino

Parto de una copia de la matriz “componente_conexa” obtenida justo antes de aplicar el algoritmo de Kruskal a las aristas con el mismo valor

inicializo_componente_conexa

Pongo una solución obtenida con el algoritmo de Kruskal en la matriz “solución_parcial”, sólo si esta solución es única.

```
                cargar_combinación_válida

                orden [ k – num + 1 ]  ⟵  0

            finsi

        finsi

    finpara

    orden [ k – num + 1 ]  ⟵  0

    si  num < k  ,  entonces

            orden [ k – num ]  ⟵  0

    finsi

    retorno  ⟵  True

hasta  k = num
```

Procedimiento **"cargar_combinación_válida"**

si última_fila_solución_parcial = 0 , entonces

Calculo el algoritmo de Kruskal para la primera combinación de aristas dada por la matriz "orden" y cargo la primera fila de la matriz "solución_parcial" con los datos de dichas aristas, incluido el dato "fase" obtenido de aplicarlas el algoritmo de Kruskal.

cargar_primera_fila

sino

Aplico el algoritmo de Kruskal a la combinación de aristas dada por la matriz "orden". Después, compruebo si este resultado se ha obtenido ya para otra combinación de aristas distinta y si no es así, he obtenido un resultado único y lo cargo en la matriz "solución_parcial".

comprobar_repetida

finsi

Procedimiento **"cargar_primera_fila"**

última_fila_solución_parcial ⟵ 1

para i = 1 hasta num , hacer

Paso la información de la arista de la matriz "aristas_ordenadas" dada por la matriz "orden" a la matriz "solución_parcial", excepto el dato "fase" que lo tengo que calcular con el algoritmo de Kruskal.

valor_columna ⟵ columna + orden [i] – 1

Tengo que saber que nodos son "nodop" y "nodoq" porque al aplicar después el algoritmo de Kruskal sobre esta arista, se utilizan en otros procedimientos para calcular el dato "fase"

nodop ⟵ aristas_ordenadas.nodoi [fila , valor_columna]

nodoq ⟵ aristas_ordenadas.nodoj [fila , valor_columna]

solución_parcial.nodop [última_fila_solución_parcial , orden [i]] ⟵ nodop

solución_parcial.nodoq [última_fila_solución_parcial , orden [i]] ⟵ nodoq

solución_parcial.valor [última_fila_solución_parcial , orden [i]] ⟵ aristas_ordenadas.valor [fila , valor_columna]

Ahora aplico el algoritmo de Kruskal a esta arista para obtener el dato que falta (“fase”)

fila_componente_conexa ← 1

busca_componente_conexa (copia_componente_conexa)

componente_nodo (nodop, copia_componente_conexa)

si misma_componente (nodoq, copia_componente_conexa) , entonces

(Cargo el dato “fase” en la arista correspondiente a la primera fila de la matriz “solución_parcial”)

solución_parcial.fase [última_fila_solución_parcial , orden [i]] ← “forma_ciclo”

sino

componentep ← fila_componente_conexa

fila_componente_conexa ← 1

busca_componente_conexa (copia_componente_conexa)
componente_nodo (nodoq, copia_componente_conexa)

componenteq ← fila_componente_conexa

unir_componentes_conexas (copia_componente_conexa)

(Cargo el dato “fase” en la arista correspondiente a la primera fila de la matriz “solución_parcial”)

solución_parcial.fase [última_fila_solución_parcial , orden [i]] ← “solución”

finsi

finpara

Procedimiento **"comprobar_repetida"**

En primer lugar, tengo que cargar las aristas dadas por la matriz "orden" en la matriz auxiliar "aux". Para cada arista, primero cargaré los datos de los dos nodos y el valor de la arista y después tengo que aplicar el algoritmo de Kruskal a dicha arista para saber el dato "fase".

para i = 1 hasta num , hacer

valor_columna ⟵ columna + orden [i] – 1

Tengo que obtener los datos "nodop" y "nodoq" ya que se utilizan en otros procedimientos cuando apliquemos el algoritmo de Kruskal a la arista

nodop ⟵ aristas_ordenadas.nodoi [fila , valor_columna]

nodoq ⟵ aristas_ordenadas.nodoj [fila , valor_columna]

aux.nodop [i] ⟵ nodop

aux.nodoq [i] ⟵ nodoq

aux.valor [i] ⟵ aristas_ordenadas.valor [fila , valor_columna]

Ahora aplico el algoritmo de Kruskal a esta arista para obtener el dato que falta ("fase")

fila_componente_conexa ⟵ 1

busca_componente_conexa (copia_componente_conexa)

componente_nodo (nodop, copia_componente_conexa)

si misma_componente (nodoq, copia_componente_conexa) , entonces

(Cargo el dato “fase” en la matriz “aux”)

aux.fase [i] ⟵ “forma_ciclo”

sino

componentep ⟵ fila_componente_conexa

fila_componente_conexa ⟵ 1

busca_componente_conexa (copia_componente_conexa)

componente_nodo (nodoq, copia_componente_conexa)

componenteq ⟵ fila_componente_conexa

unir_componentes_conexas (copia_componente_conexa)

(Cargo el dato “fase” en la matriz “aux”)

aux.fase [i] ⟵ “solución”

finsi

finpara

Ahora tengo que comprobar si después de aplicar el algoritmo de Kruskal a esta combinación de aristas dada por la matriz “orden” el resultado es el mismo que aparece en alguna fila de la matriz “solución_parcial”.

```
fila_repetida ⟵ False

para i = 1 hasta última_fila_solución_parcial , hacer
    para j = 1 hasta num , hacer
        si aux.fase [ j ] ≠ solución_parcial.fase [ i , j ] , entonces
            salir del para interno
        finsi
    finpara
    si j = num , entonces
        fila_repetida ⟵ True
        salir del para interno
    finsi
finpara
```

Y por último, si el resultado es único, añado una fila más a la matriz "solución_parcial" y pongo en ella la nueva solución única.

si no fila_repetida , entonces

crear una fila nueva en la matriz "solución_parcial"

Incremento el índice que nos indica cual es la última fila de la matriz "solución_parcial"

última_fila_solución_parcial ⟵ última_fila_solución_parcial + 1

para j = 1 hasta num , hacer

Cargo la matriz "aux" en la fila recién creada de la matriz "solución_parcial"

solución_parcial [última_fila_solución_parcial , j] ⟵ aux [j]

finpara

finsi

Procedimiento **"ampliar_matriz"**

(Creo filas nuevas en la matriz "aristas_ordenadas")

crear_filas_nuevas

Cargo las filas recién creadas con las aristas (si existen) de la fila activa desde la primera arista hasta la arista anterior al grupo de aristas con el mismo valor

si columna > 1 , entonces

cargar_aristas_anteriores

finsi

Cargo las distintas soluciones únicas (grupos de aristas con el mismo valor) en la fila activa y las filas recién creadas

cargar_aristas_mismo_valor

Y por último, cargo las aristas (si existen) de la fila activa que están después del grupo de aristas con el mismo dato "valor" que estamos tratando en las filas recién creadas

si (columna + num – 1) < m , entonces

(Siendo "m" el número total de aristas del grafo)

cargar_aristas_posteriores

finsi

Procedimiento **"crear_filas_nuevas"**

Creo tantas filas nuevas en la matriz *"aristas_ordenadas"* como soluciones únicas calculadas menos una. Estas filas nuevas se sitúan después de la última fila (fila de mayor valor) de la matriz.

Importante : La variable global *"total_filas_aristas_ordenadas"* tiene que estar inicializada con el valor uno en el programa principal fuera del procedimiento *"probar_varias_soluciones"* y además sólo se inicializa una vez para que cuando se vuelva a entrar en el procedimiento *"crear_filas_nuevas"* conserve el valor que tenía cuando se salió de dicho procedimiento por última vez.

El pseudocódigo es:

Crear un número de filas ((última_fila_solución_parcial) – 1) añadidas a la matriz "aristas_ordenadas".

total_filas_aristas_ordenadas ⟵

⟵ (total_filas_aristas_ordenadas) + ((última_fila_solución_parcial) – 1)

Procedimiento **"cargar_aristas_anteriores"**

```
para  i = (total_filas_aristas_ordenadas) – (última_fila_solución_parcial) + 2
hasta  total_filas_aristas_ordenadas  ,  hacer

    para  j = 1  hasta  (columna – 1)  ,  hacer

        aristas_ordenadas [ i, j ]  ←  aristas_ordenadas [ fila, j ]

    finpara

finpara
```

Procedimiento **"cargar_aristas_mismo_valor"**

(Primero cargaré las aristas en la fila activa)

col ⟵ 0

para j = columna hasta (columna + num – 1) , hacer

col ⟵ col + 1

aristas_ordenadas [fila , j] ⟵ solución_parcial [1 , col]

si solución_parcial.fase [1 , col] = "solución" , entonces

Actualizo la matriz "componente_conexa"
con la nueva arista de la solución.

actualizar_componentes_conexas (j , componente_conexa)

(Sumo uno al número de aristas de la solución)

número_aristas_solución ⟵ número_aristas_solución + 1

finsi

finpara

(Ahora cargaré las aristas en las filas recién creadas)

i ⟵ (total_filas_aristas_ordenadas) - (última_fila_solución_parcial) + 2

```
col ← 0
para j = 2 hasta última_fila_solución_parcial , hacer
    para p = columna hasta (columna + num - 1) , hacer
        col ← col + 1
        aristas_ordenadas [ i , p ] ← solución_parcial [ j , col ]
    finpara
    col ← 0
    i ← i + 1
finpara
```

La matriz “solución_parcial” ya no nos hace falta y la quito de la memoria para aprovechar mejor los recursos del ordenador.

Borrar de memoria la matriz solución_parcial

Procedimiento **"cargar_aristas_posteriores"**

Y por último, sólo nos queda cargar las aristas de la fila activa hacia las filas recién creadas a partir de la arista *"columna + num"* en adelante.

```
para   i = (total_filas_aristas_ordenadas)  –  (última_fila_solución_parcial)  +  2
hasta   total_filas_aristas_ordenadas   ,   hacer

    para   j = columna + num   hasta   m   ,   hacer

        aristas_ordenadas [ i , j ]  ←——  aristas_ordenadas [ fila , j ]

    finpara

finpara
```

Por último, sólo quiero hacer notar que la variable *"última_fila_solución_parcial"* ha sustituido a las variables *"fila_solución_parcial"* y *"total_filas_solución_parcial"*.

2.2.4.- Implementación del algoritmo de Prim

En esta parte de la documentación del proyecto no voy a tratar el conjunto de todas las soluciones óptimas posibles, si las hay, (esto lo dejo para más adelante) sino simplemente el poder llegar a una solución óptima cualquiera del conjunto de soluciones óptimas posibles.

Haré dos desarrollos del algoritmo de Prim en dos partes diferenciadas:

Primera parte: Lógica y desarrollo de la implementación general del algoritmo de Prim de forma automática sin intervención del usuario.

Segunda parte: Implementación del algoritmo de Prim a través de la intervención del usuario seleccionando aristas sobre un grafo.

La primera parte no la voy a implementar en el código del lenguaje de programación que voy a utilizar (java) ya que, mas adelante, implementaré como solución automática aquella que me dé todas las soluciones óptimas posibles. Sin embargo, me va a ser muy útil la implementación en seudocódigo de esta primera parte porque voy a desarrollar las ideas, soportes de datos, un algoritmo, y sobre todo, estructurando el seudocódigo en procedimientos y funciones, puedo reutilizar muchos de estos procedimientos y funciones en el desarrollo del seudocódigo para resolver la segunda parte y también para calcular todas las soluciones óptimas posibles. La idea es desarrollar muchas explicaciones e implementar seudocódigo reutilizable con el fin de que en una etapa posterior, en mi caso para hallar todas las soluciones o para que el usuario pueda resolver el algoritmo de Prim seleccionando aristas, se pueda simplificar y aclarar mejor el desarrollo sin detenerse en detalles y explicaciones básicas, con lo que se agiliza y entiende el proceso mucho mejor.

En la segunda parte, si en la resolución de nuestro problema sólo existe una única solución óptima, el usuario deberá ir seleccionando aristas hasta encontrarla, pero si existen varias soluciones óptimas se dará por buena cualquier solución óptima que encuentre el usuario.

2.2.4.1.- Lógica y desarrollo de la implementación general del algoritmo de forma automática sin intervención del usuario

Pasos a realizar:

1.- Definir el grafo $G = < N , A >$ conexo, no dirigido y con valores no negativos en sus aristas.

2.- Calcular el número de nodos *"n"* de *N*.

3.- B = { un nodo arbitrario de *N* }.

4.- $T = \{ \emptyset \}$ (*T* contendrá las aristas del árbol de recubrimiento mínimo, es decir, las aristas de la solución).

5.- Mientras $B \neq N$, hacer

buscar una arista $e = \{ i , j \}$ de valor mínimo,

tal que el nodo $i \in B$ y el nodo $j \notin B$

$T \leftarrow T \cup \{ e \}$

$B \leftarrow B \cup \{ j \}$

6.- Cuando $B = N$, $G' = < N , T >$ es el árbol de recubrimiento mínimo.

Los pasos 1 y 2 son parte común a todos los procesos relacionados con los algoritmos de Kruskal y Prim y su realización corresponde al desarrollo de la parte gráfica. Cuando se construye el grafo, se controlan sus características, valores de sus aristas y se calcula el número *"n"* de sus nodos.

Luego lo que hay que desarrollar ahora son los pasos del 3 al 6.

Elementos para almacenar la información básica del programa

El algoritmo de Prim va creando un subconjunto *B* de nodos del grafo que es conexo y cuyas aristas *T* pertenecen a la solución que queremos encontrar. *B* va creciendo y cuando *B* = *N*, sus aristas *T* nos dan la solución.

Entre los nodos de *B* no se pueden añadir aristas porque se formarían ciclos con lo que sólo se pueden añadir aristas entre un nodo de *B* y un nodo del grafo que no pertenezca a *B*.

Me creo la variable *"número_nodos_en_B"* que me dice en todo momento el número de nodos que tiene *B*, y así, cuando el valor de dicha variable coincida con el número de nodos *"n"* del grafo, habré hallado la solución y terminaré el programa.

Para saber en todo momento que nodos no están en *B*, me crearé una matriz de una dimensión y *"n"* elementos, siendo *"n"* el número de nodos del grafo, llamada *"para_nodos_fuera_de_B"* que será booleana y tomará los valores:

para_nodos_fuera_de_B [i] (para i = 1, 2, ... , n)

- True ⇨ Si el nodo "i" esta fuera del conjunto B.
- False ⇨ Si el nodo "i" esta dentro del conjunto B.

También, para saber en todo momento si un nodo $i \notin B$ tiene alguna arista que incida en algún nodo de *B*, me creo una matriz booleana de una dimensión y *"n"* elementos, siendo *"n"* el número de nodos del grafo, llamada *"con_aristas_hacia_B"* que tomará los valores:

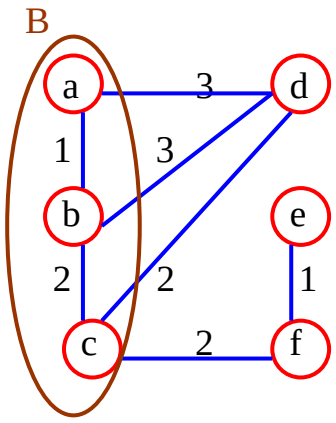
con_aristas_hacia_B [i] (para i = 1, 2, ... , n)

- True ⇨ Si el nodo *"i"* tiene al menos una arista que incida en algún nodo de *B*.
- False ⇨ En caso contrario.

En un momento determinado del algoritmo, para cada nodo $i \in N$ y que no pertenezca a *B* me interesa saber cual es el valor de la arista (si existe, es decir, *"con_aristas_hacia_B [i] = True"*) de menor valor que sale de *i* y que incida en cualquier nodo de *B*, luego me creo una matriz de una dimensión y *"n"* elementos, siendo *"n"* el número de nodos del grafo, llamada *"valor_min [i]"* que almacene este valor mínimo para todo nodo $i \notin N$ e $i \notin B$.

Por ejemplo:

Si el proceso de ejecución del algoritmo está en la fase que se muestra en la figura:



valor_min [e] = no hay ninguna arista del nodo *"e"* que incida en los nodos del conjunto *B*, valor_arista [e , i] = -1 para todo $i \notin B$; y el programa se va a saltar el nodo *"e"*.

valor_min [i] , para i = a , b , c, no lo va a considerar el programa por ser nodos de *B*.

valor_min [d] = 2 (arista cd); valor_min [f] = 2 (arista cf);

Luego si conozco el valor mínimo de la arista que va de un nodo $i \in N$ tal que $i \notin B$ a un nodo de *B*, y conozco lógicamente el nodo *i*, sólo me queda saber qué nodo en concreto de *B* es en el que incide dicha arista de menor valor y que parte del nodo *i*.

Para ello, me creo la matriz de una dimensión y *"n"* elementos, siendo *"n"* el número de nodos del grafo, llamada *"mas_proximo_en _B"*. Es decir, para cada nodo $i \in N$ tal que $i \notin B$:

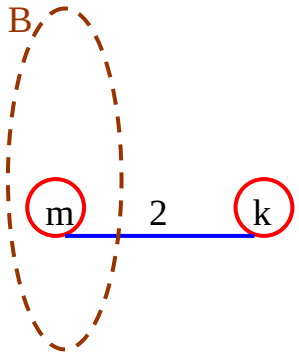
valor_min [i] = "valor más pequeño de la arista que va del nodo *" i "* y que incide en cualquier nodo de *B*".

mas_proximo_en _B [i] = "nodo en concreto de *B* en el que incide la arista que sale del nodo *" i "* y cuyo valor está en *"valor_min [i]" "*.

Para guardar el nodo *" i "* que no pertenece a *B*, del que parte la arista solución, necesitaremos una variable que llamaré *"aux"*.

Ejemplo:

Si la arista *"mk"* es la de menor valor del nodo *"k"* a cualquier nodo de *B*, la información completa es:



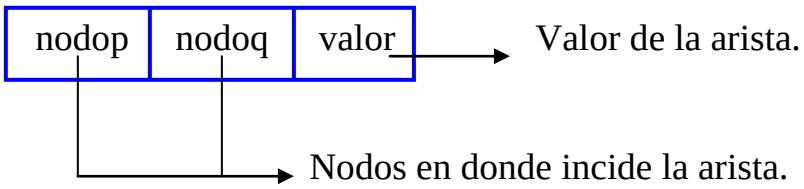
valor_min [k] ⟵ 2

aux ⟵ k

mas_proximo_en _B [k] ⟵ m

Por último, para guardar las aristas *T* de la solución me creo la matriz de una dimensión y de *"n – 1"* elementos, siendo *"n"* el número de nodos del grafo, llamada *"aristas_de_la_solución"*.

Cada elemento de esta matriz constará de los siguientes campos:



Para referirme a un campo en particular de un elemento *"i"* usaré la notación:

"aristas_de_la_solución.nodop [i]"
"aristas_de_la_solución.nodoq [i]"
"aristas_de_la_solución.valor [i]"

y para referirme a un elemento *"i"* de la matriz con todos sus campos:

"aristas_de_la_solución [i]"

Para saber en todo momento la posición en la matriz *"aristas_de_la_solución"* en la que se debe insertar la siguiente arista, me creo la variable *"índice"*. Inicialmente, esta variable vale cero y cuando se calcula una nueva arista de la solución se incrementa en uno, insertándose la arista en *"aristas_de_la_solución [índice]"*.

Resolución del algoritmo de Prin

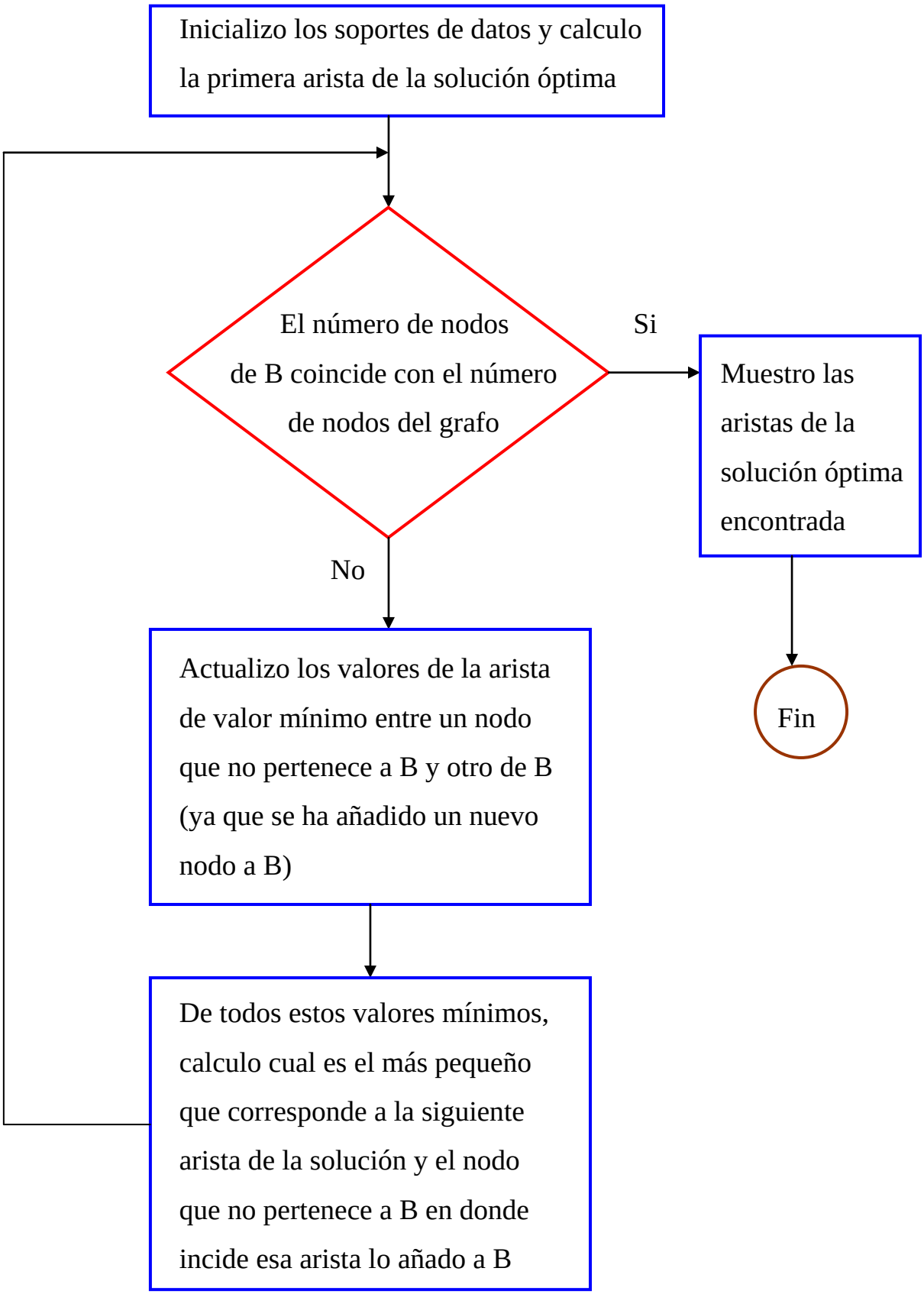
Una vez que tenemos los soportes de datos que voy a necesitar, primero desarrollaré el algoritmo a través de esquemas en forma de organigramas para tener una visión global y simplificada de los pasos que tengo que hacer.

A continuación, resolveré en seudocódigo dichos organigramas. El seudocódigo esta dividido en procedimientos para un mejor seguimiento y una posterior

reutilización cuando resuelva el algoritmo de Prim a través de un usuario y cuando calcule todas las posibles soluciones.

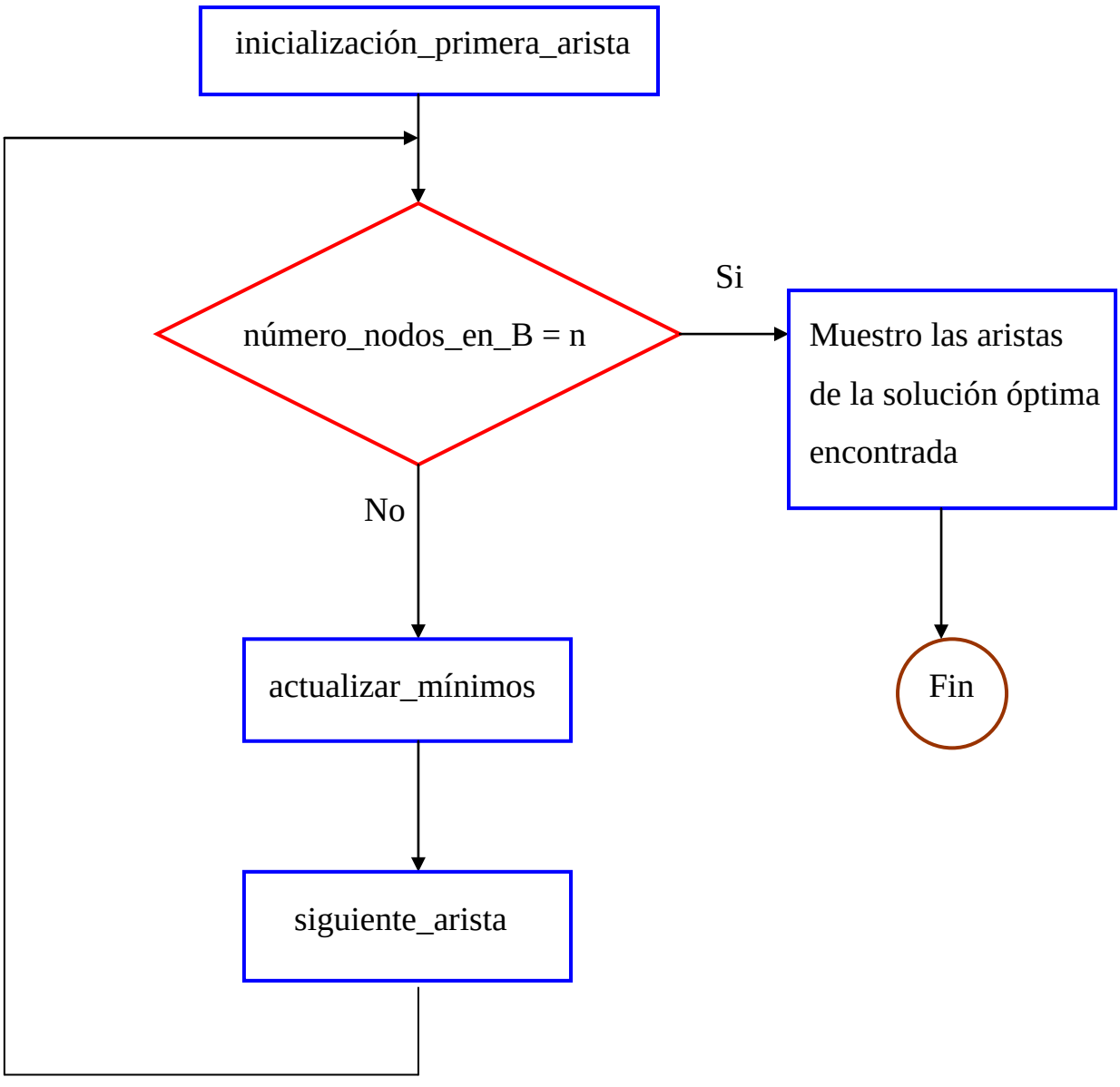
Los organigramas son los siguientes:

Esquema funcional del **programa principal**

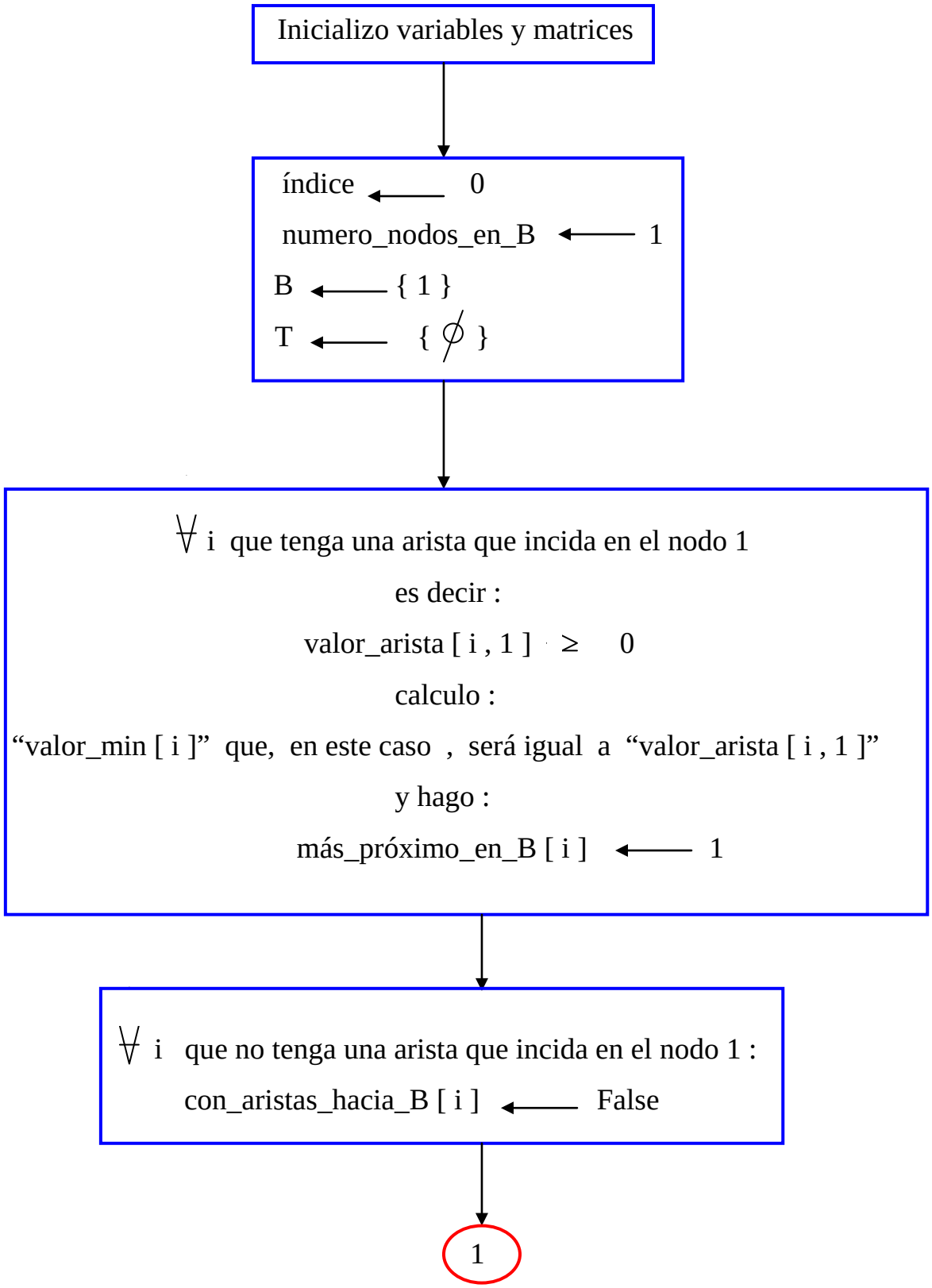


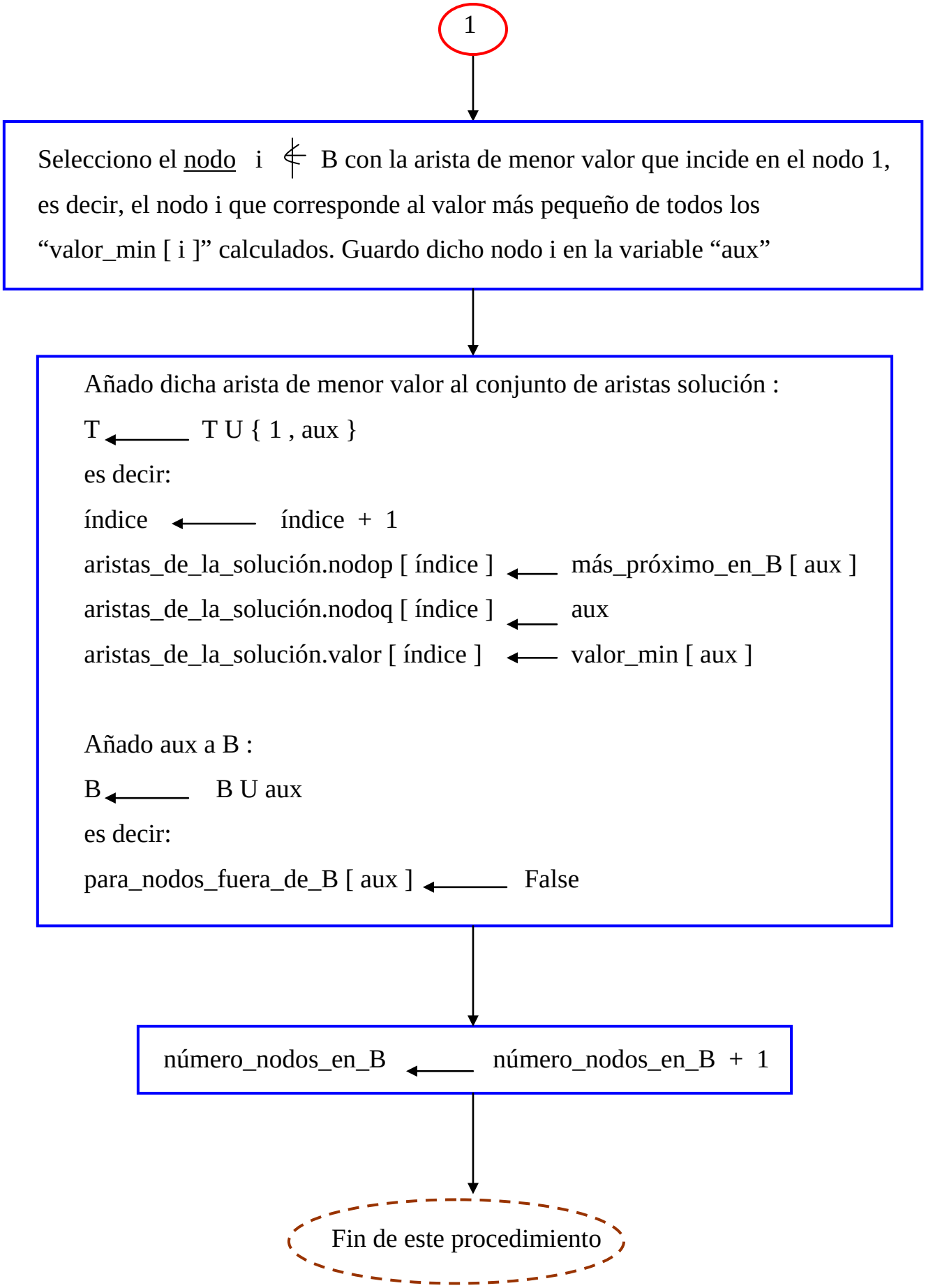
Esquema a nivel de procedimientos del **programa principal**

El anterior esquema se puede traducir a nivel de procedimientos en el siguiente:

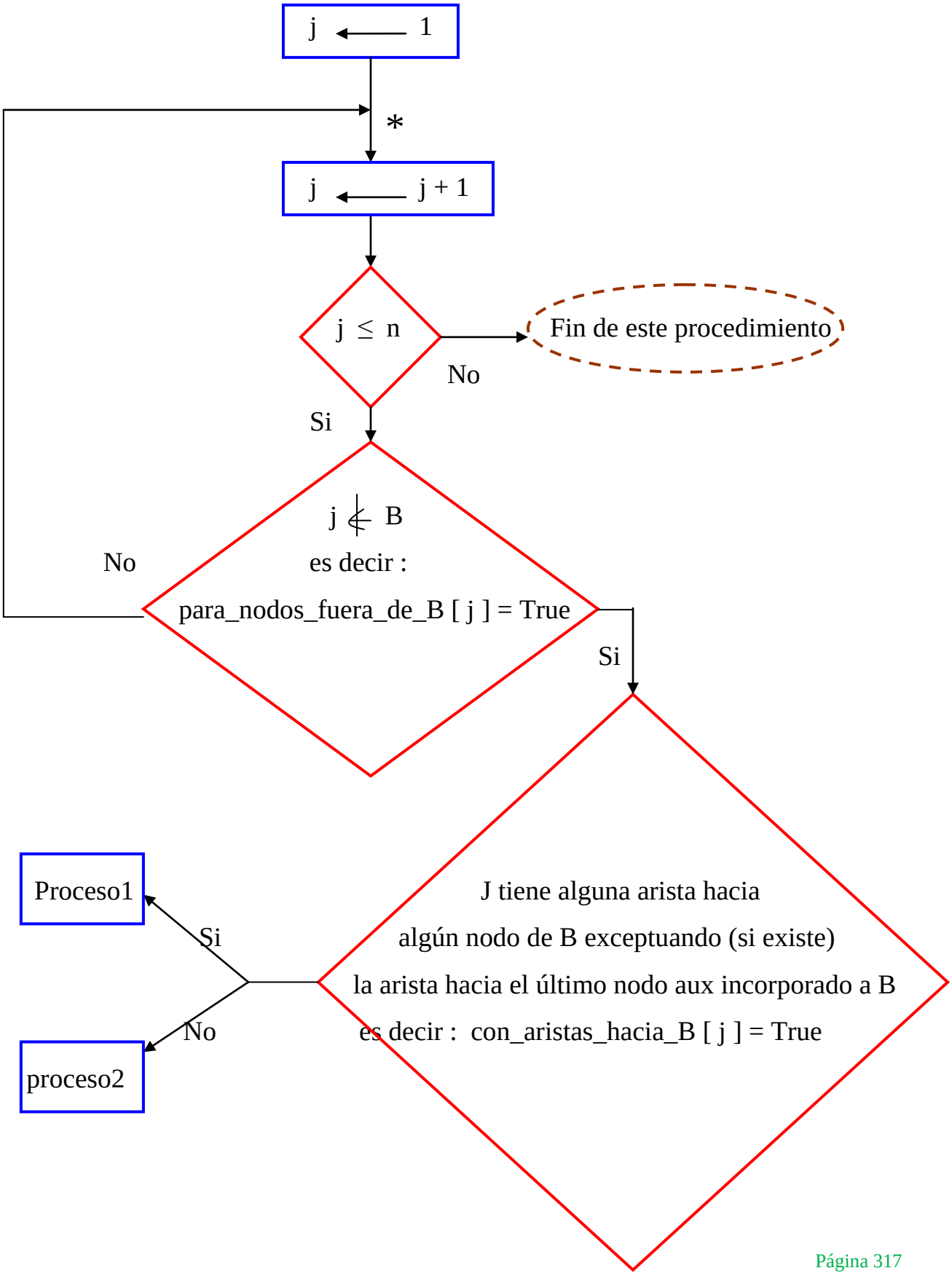


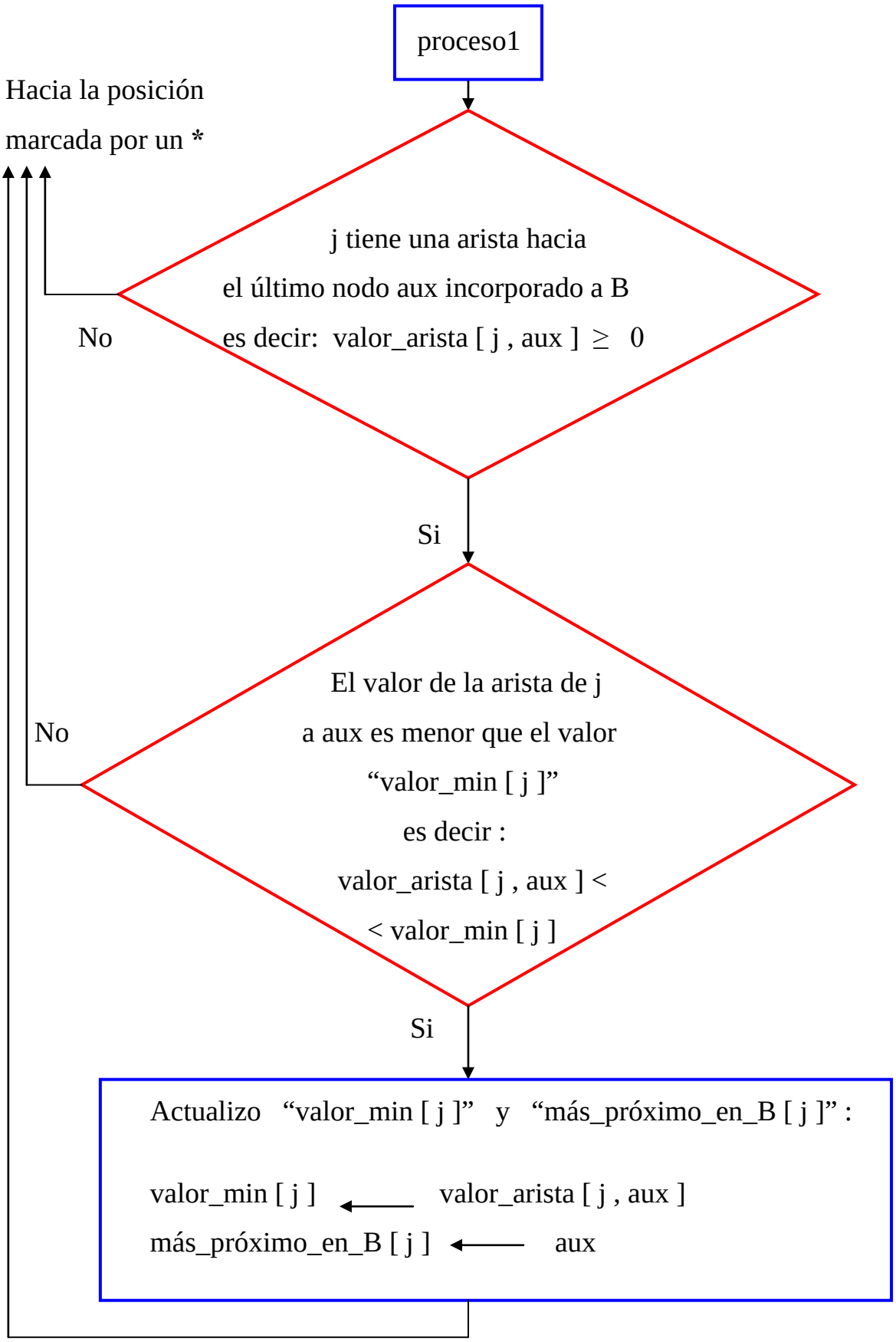
Procedimiento **"inicialización_primera_arista"**

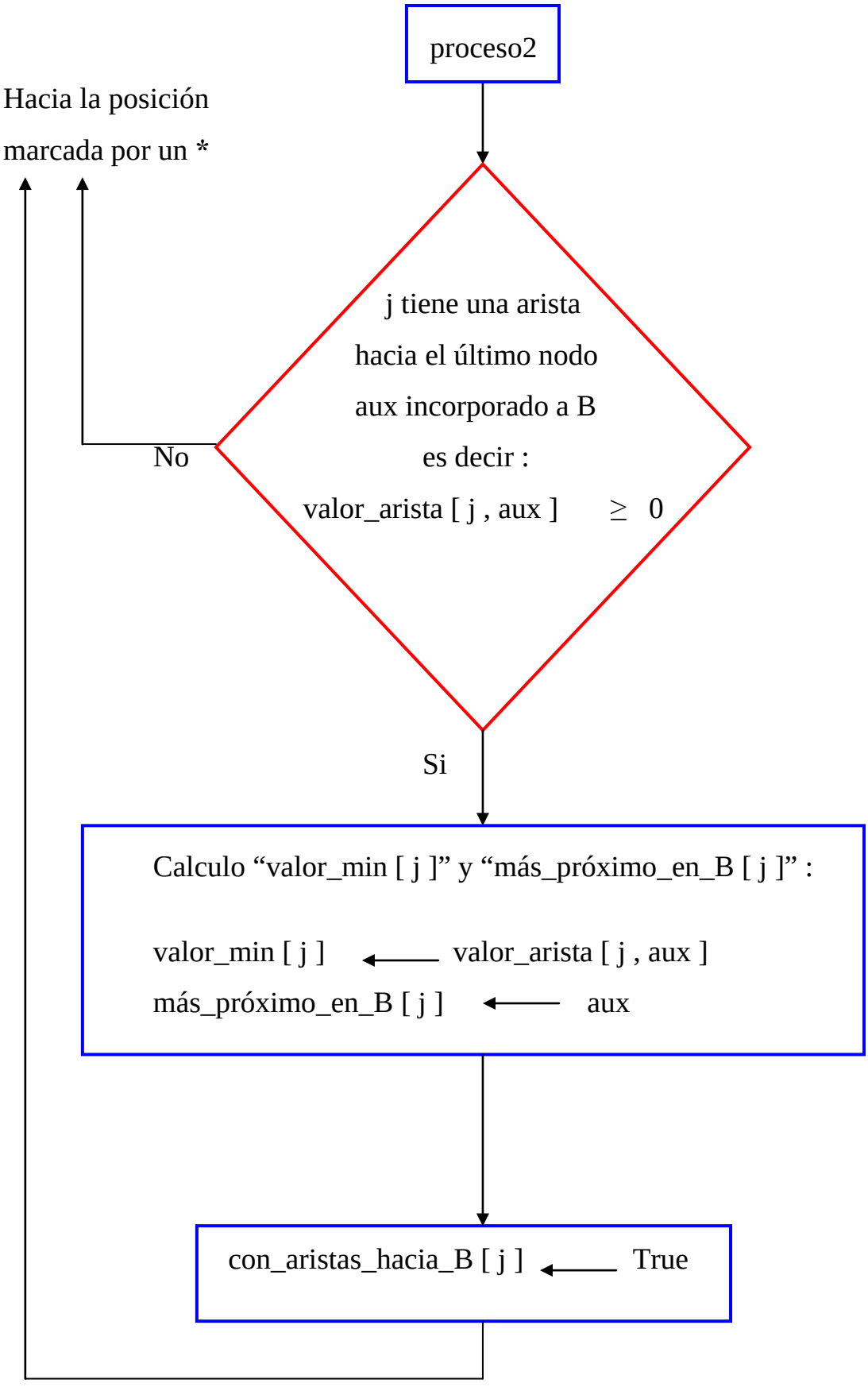



1
Selecciono el nodo i ∉ B con la arista de menor valor que incide en el nodo 1, es decir, el nodo i que corresponde al valor más pequeño de todos los "valor_min [i]" calculados. Guardo dicho nodo i en la variable "aux"
Añado dicha arista de menor valor al conjunto de aristas solución :
T ← T U { 1 , aux }
es decir:
índice ← índice + 1
aristas_de_la_solución.nodop [índice] ← más_próximo_en_B [aux]
aristas_de_la_solución.nodoq [índice] ← aux
aristas_de_la_solución.valor [índice] ← valor_min [aux]
Añado aux a B :
B ← B U aux
es decir:
para_nodos_fuera_de_B [aux] ← False
número_nodos_en_B ← número_nodos_en_B + 1
Fin de este procedimiento

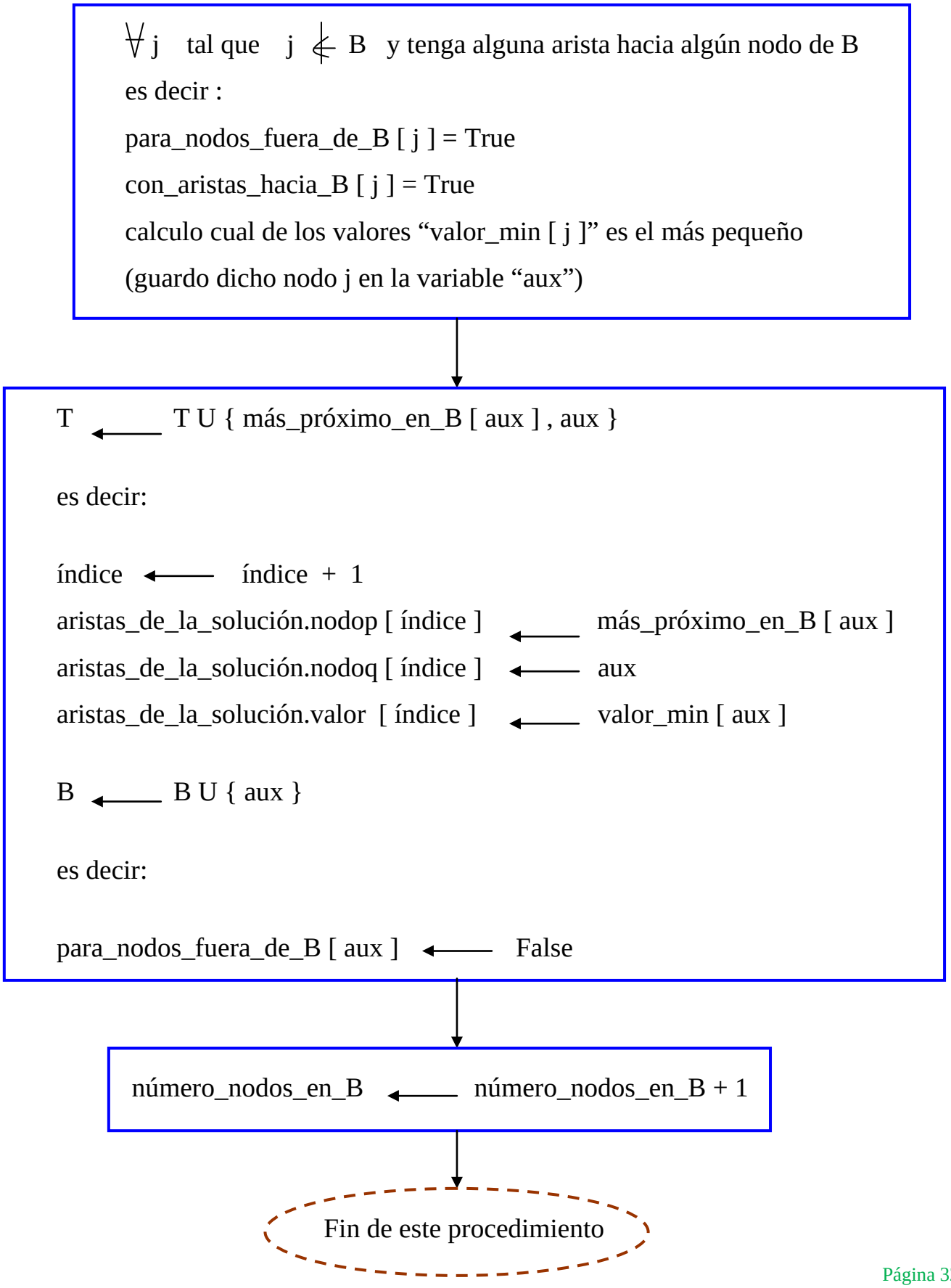
Procedimiento **"actualizar_mínimos"**




proceso1
Hacia la posición
marcada por un *
j tiene una arista hacia
el último nodo aux incorporado a B
es decir: valor_arista [j , aux] ≥ 0
No
Si
El valor de la arista de j
a aux es menor que el valor
"valor_min [j]"
es decir :
valor_arista [j , aux] <
< valor_min [j]
No
Si
Actualizo "valor_min [j]" y "más_próximo_en_B [j]" :
valor_min [j] ← valor_arista [j , aux]
más_próximo_en_B [j] ← aux


proceso2
Hacia la posición
marcada por un *
j tiene una arista
hacia el último nodo
aux incorporado a B
es decir :
valor_arista [j , aux] ≥ 0
No
Si
Calculo “valor_min [j]” y “más_próximo_en_B [j]” :
valor_min [j] ← valor_arista [j , aux]
más_próximo_en_B [j] ← aux
con_aristas_hacia_B [j] ← True

Procedimiento **siguiente_arista**



Implementación en seudocódigo

Programa principal

```
inicialización_primera_arista

mientras  número_nodos_en_B  <  n  ,  hacer
        actualizar_mínimos
        siguiente_arista
finmientras

muestro las aristas de la solución óptima
```

Procedimiento **"inicialización_primera_arista"**

inicializo la variable "índice" que me indicará la primera posición libre de la matriz "aristas_de_la_solución" para insertar la siguiente arista de la solución

índice ⟵ 0

Matriz para_nodos_fuera_de_B [j] , j = 1 , 2 , ... , n

Inicialmente todos sus elementos tienen valor "True" excepto para el nodo 1, con lo que pongo el nodo 1 en B como nodo de inicio (B ⟵ { 1 }).

para i = 2 hasta n , hacer

    para_nodos_fuera_de_B [i] ⟵ True

finpara

para_nodos_fuera_de_B [1] ⟵ False

(ahora B pasa de tener cero nodos a tener un nodo)

número_nodos_en_B ⟵ 1

Matriz con_aristas_hacia_B [j] , j = 1 , 2 , ... , n

Inicialmente tiene todos sus elementos con valor "True" excepto para el nodo 1 que está en B.

para i = 2 hasta n , hacer

    con_aristas_hacia_B [i] ⟵ True

finpara

con_aristas_hacia_B [1] ⟵ False

Cálculo del contenido inicial de las matrices *"valor_min"* y *"mas_proximo_en_B"*:

Inicialmente, el valor de *"valor_min [i]"* para cualquier nodo *i* que tenga una arista con el nodo 1 será precisamente el valor de dicha arista puesto que *B* sólo tiene el nodo 1. Luego, si hay nodos *"i"* con aristas que inciden en el nodo 1 ponemos el valor de dichas aristas en *"valor_min [i]"* , y si no existen aristas que inciden en el nodo 1 , ponemos "False" en *"con_aristas_hacia_B [i]"*:

para i = 2 hasta n , hacer

Condición para que exista la arista entre el nodo *"i"* y el nodo 1.

  si valor_arista [i , 1] ≥ 0 , entonces

    valor_min [i] ⟵ valor_arista [i , 1]

    mas_proximo_en _B [i] ⟵ 1

  si no

    con_aristas_hacia_B [i] ⟵ False

  finsi

finpara

Inicialmente, como *B = { 1 }*, todos los nodos del grafo que tienen una arista con el nodo 1 tienen como valor dicho nodo 1 en su matriz *"mas_proximo_en_B"*, es decir:

mas_proximo_en_B [i] ⟵ 1 , para todo *i* con una arista que incide en el nodo 1, y además , la distancia más corta entre el nodo 1 y el resto de nodos del grafo nos la dará la arista (si existe) que incide de cada nodo en el nodo 1.

Ahora tengo que seleccionar el valor mas pequeño de todos los

"valor_min [i]" ; $\forall\ i \notin B$

calculados.

Después, tengo que añadir al conjunto de aristas solución la arista que corresponde a este valor más pequeño hallado, y también tengo que incluir en *B* el nodo que incide en dicha arista y que no esta en *B*. Por último, añadiré un nodo más al número de nodos de *B*. Todos estos procesos están agrupados en el procedimiento *"siguiente_arista"*.

Resumiendo, el código completo de este procedimiento será:

```
índice ⟵ 0
número_nodos_en_B ⟵ 1

para  i = 2  hasta  n  ,  hacer
        para_nodos_fuera_de_B [ i ]  ⟵  True
finpara

para_nodos_fuera_de_B [ 1 ]  ⟵  False

para  i = 2  hasta  n  ,  hacer
        con_aristas_hacia_B [ i ]  ⟵  True
finpara
```

```
con_aristas_hacia_B [ 1 ]  ←  False

para  i =  2  hasta  n  ,  hacer
    si  valor_arista [ i , 1]  ≥  0  ,  entonces
        valor_min [ i ]  ←valor_arista [ i , 1 ]
        mas_proximo_en _B [ i ]  ←1
    si no
        con_aristas_hacia_B [ i ]  ←False
    finsi
finpara

siguiente_arista
```

Procedimiento **siguiente_arista**

Primero calculo el valor más pequeño de todos los *"valor_min"* ya que será el valor de la arista solución.

min ⟵ ponemos en esta variable el valor mas alto posible.

para j = 2 hasta n , hacer

  si para_nodos_fuera_de_B [j] , entonces

    si con_aristas_hacia_B [j] , entonces

      si valor_min [j] ≤ min , entonces

        min ⟵ valor_min [j]

        aux ⟵ j (guardamos el nodo "j" en la variable "aux")

      finsi

    finsi

  finsi

finpara

La siguiente arista de la solución irá del nodo contenido en la variable *"aux"* al nodo *"más_próximo_en_B [aux]"* y su valor será *"valor_min [aux]"*.

Ahora, añado la arista solución a T:

T ⟵ T U { mas_proximo_en_B [aux] , aux }

mas_proximo_en_B [aux] ∈ B

aux ∉ B

índice ⟵ índice + 1

arristas_de_la_solución.nodop [índice] ⟵ más_próximo_en_B [aux]

arristas_de_la_solución.nodoq [índice] ⟵ aux

arristas_de_la_solución.valor [índice] ⟵ valor_min [aux]

(Añado el nodo almacenado en la variable “aux” a B)

para_nodos_fuera_de_B [aux] ⟵ False

(Sumo 1 al número de nodos de B)

numero_nodos_en_B ⟵ numero_nodos_en_B + 1

Procedimiento **actualizar_mínimos**

Al añadir un nodo más a *B* contenido en la variable *"aux"*, (ahora el nodo contenido en *"aux"* $\in B$) tenemos que hacer lo siguiente:

$\forall\ j \notin B$, actualizo o calculo (según los casos) *"valor_min [j] "* y *"más_próximo_en_B [j]"*.

Para ver todos los casos posibles, lo mejor es seguir el organigrama gráfico hecho anteriormente para este procedimiento.

El bucle empieza desde el nodo 2 ya que el nodo 1 esta en B desde el principio y nunca se va a considerar.

```
para  j = 2  hasta  n  ,  hacer
    si  para_nodos_fuera_de_B [ j ]  ,  entonces
        si  con_aristas_hacia_B [ j ]  ,  entonces
                proceso1
        sino
                proceso2
        finsi
    finsi
finpara
```

Procedimiento **proceso1**

```
si  valor_arista [ j , aux ]  ≥  0  ,  entonces
    si  valor_arista [ j , aux ]  <  valor_min [ j ]  ,  entonces
        valor_min [ j ]  ⟵  valor_arista [ j , aux ]
        más_próximo_en_B [ j ]  ⟵  aux
    finsi
finsi
```

Procedimiento **proceso2**

```
si  valor_arista [ j , aux ]  ≥  0  ,  entonces
    valor_min [ j ]  ⟵  valor_arista [ j , aux ]
    más_próximo_en_B [ j ]  ⟵  aux
    con_aristas_hacia_B [ j ]  ⟵  True
finsi
```

2.2.4.2.- Implementación del algoritmo de Prim a través de la intervención del usuario seleccionando aristas sobre un grafo

Primero explicaré algunas variables para seguir mejor el programa, después continuaré con los organigramas funcionales para ver el programa en su conjunto y por último escribiré el seudocódigo.

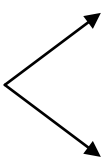
El usuario , primero debe seleccionar con el ratón el nodo de inicio y después debe ir seleccionando aristas hasta completar la solución óptima.

Si existen varias soluciones, no se van a calcular todas a la vez de forma automática sino que el usuario puede llegar a una solución óptima de las posibles soluciones que existan iterando con el grafo.

Variables

- La variable ***"índice"*** nos dice en todo momento la situación en la matriz "aristas_de_la_solución" de la última arista introducida. Se inicializa en el evento click en un nodo y se va incrementando cada vez que se añade una arista al conjunto de aristas solución en el procedimiento *"nueva_situación"*.

- La variable booleana ***"sólo_aristas"*** controla la activación o desactivación de los eventos click sobre un nodo o una arista:



True ⟹ Está activo el evento click en las aristas y no en los nodos.

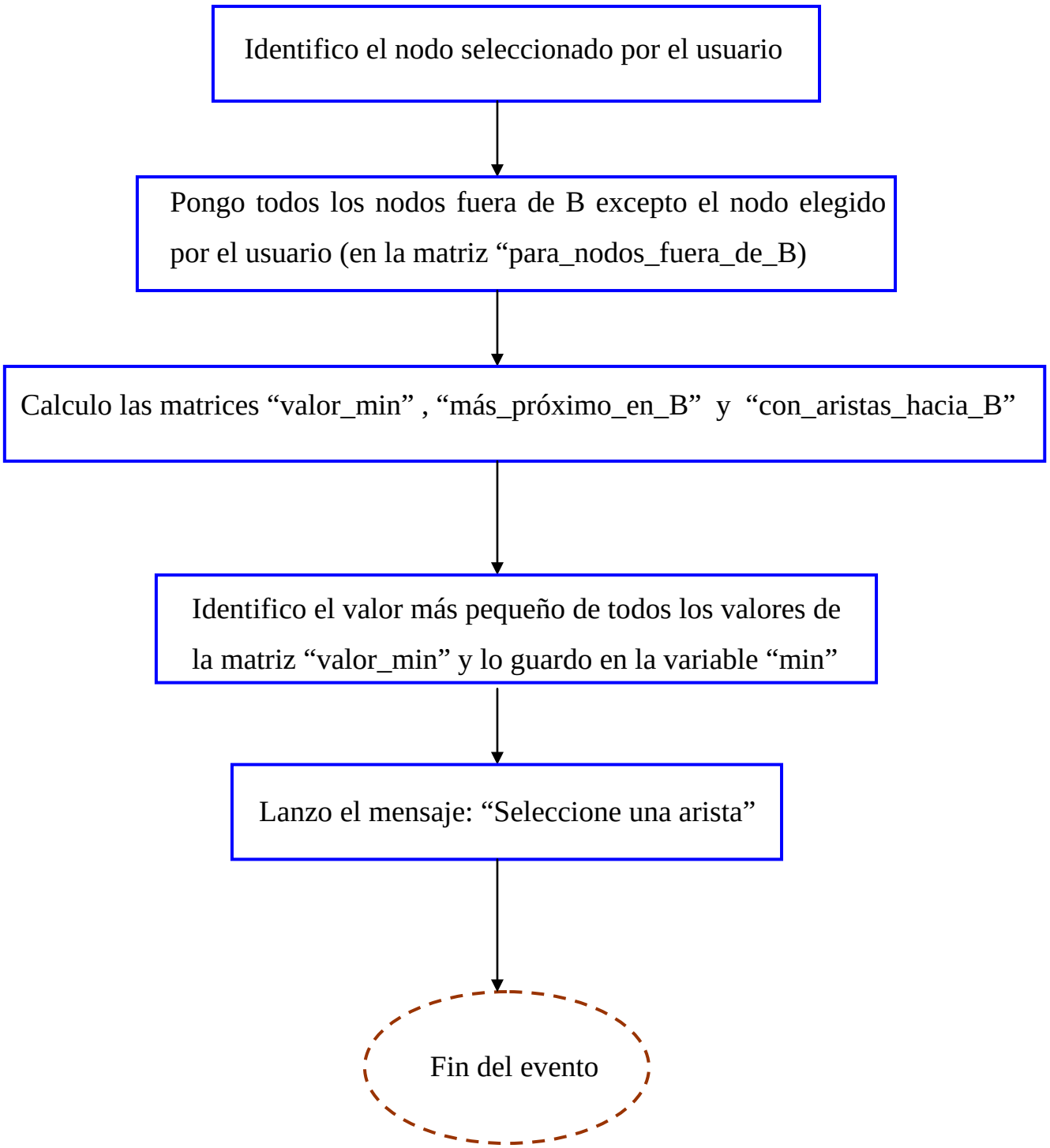
False ⟹ Está activo el evento click en los nodos y no en las aristas.

El programa, a través de esta variable obliga a que la primera acción del usuario sea seleccionar un nodo y cuando el usuario lo haga, sólo podrá seleccionar aristas.

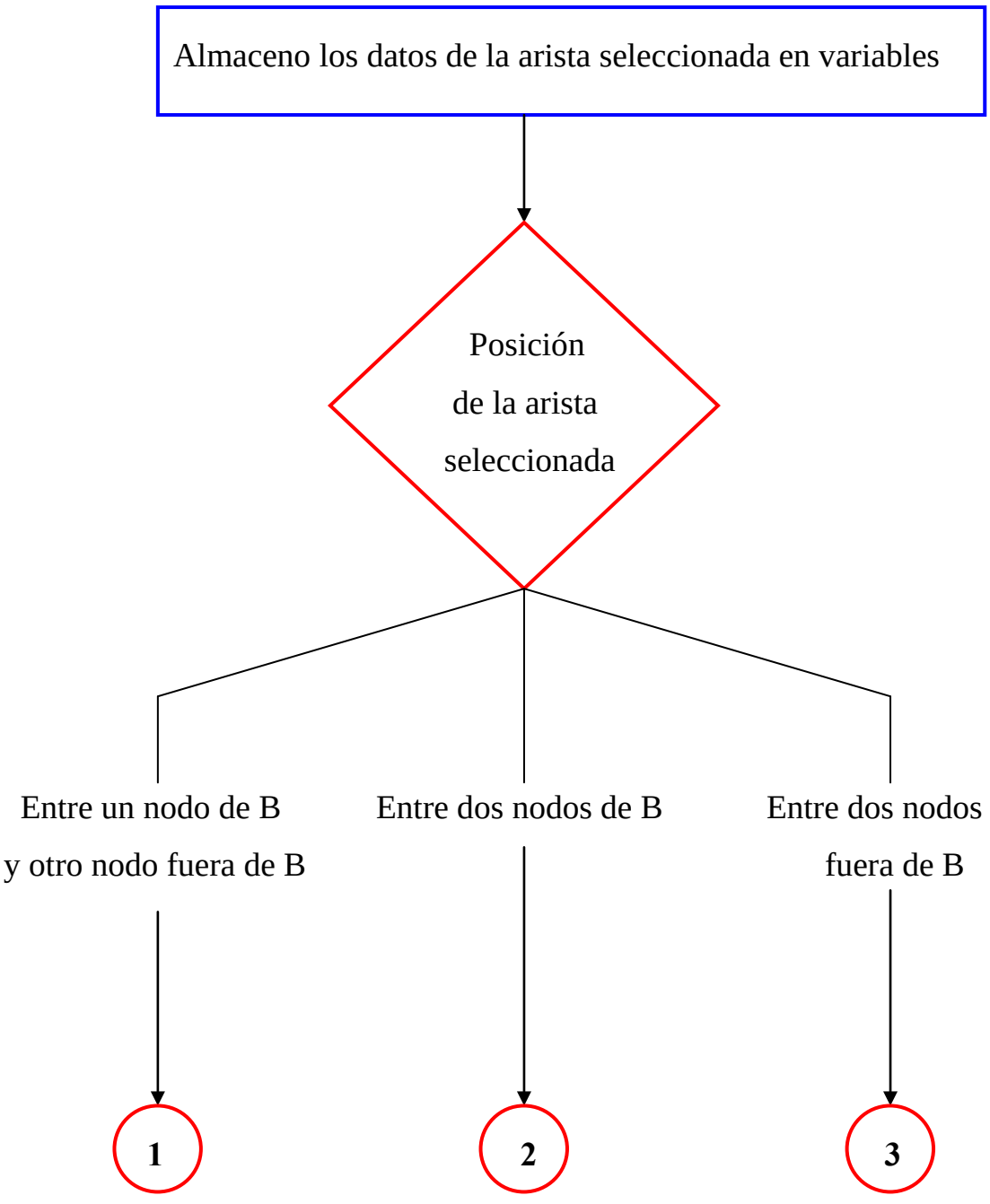
- La variable *"min"* almacena el valor más pequeño de todos los valores de la matriz *"valor_min"*. Esto, nos permite saber en todo momento el valor de la siguiente arista que debe incluirse en el conjunto de las aristas de la solución y será la referencia a comparar con el valor de la arista escogida por el usuario.

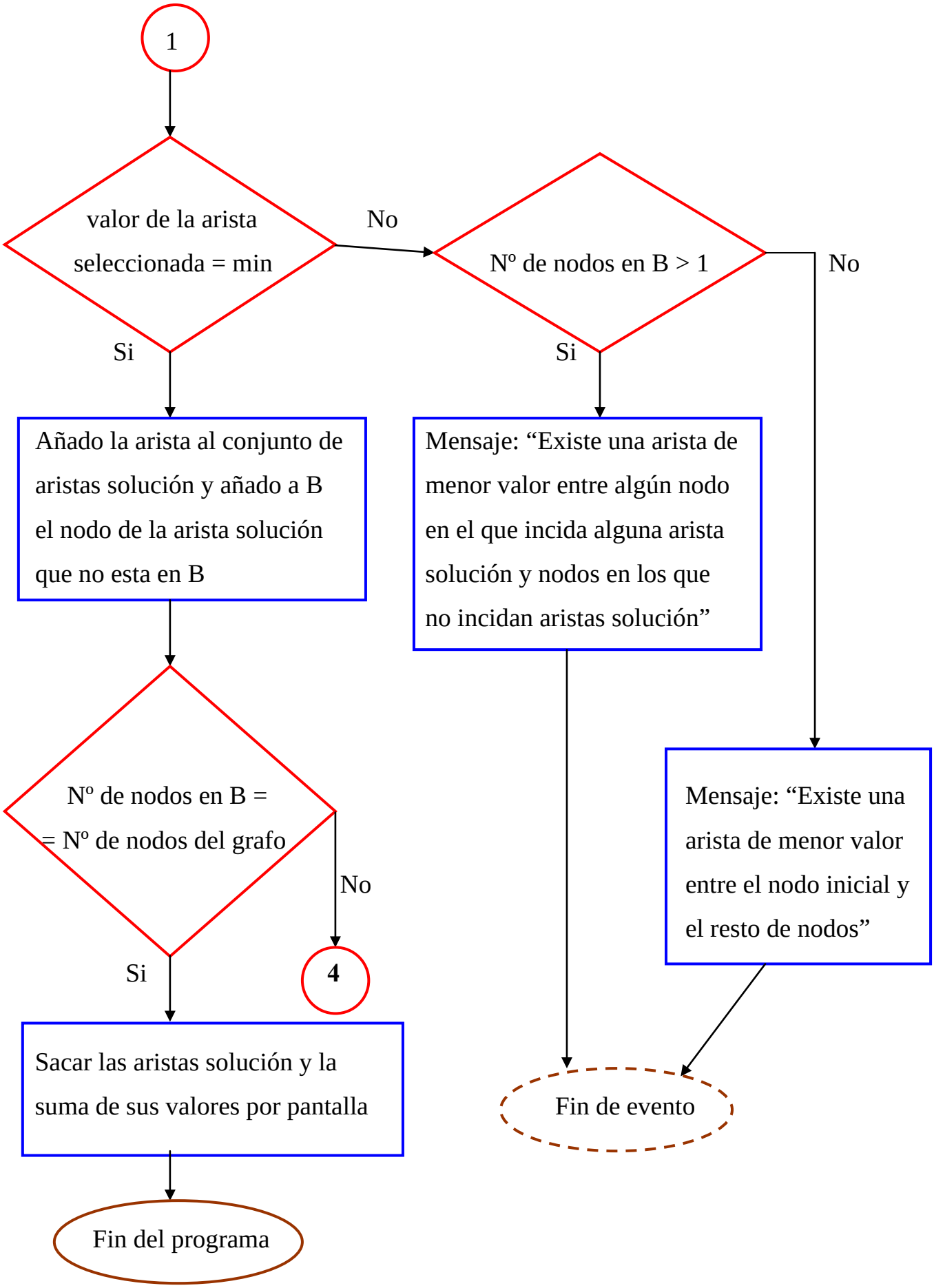
- La variable ***"número_nodos_en_B"*** nos dice el número de nodos que tiene el conjunto *B* en todo momento, lo cual es importante para la finalización del programa cuando dicha variable tenga un número de nodos igual al número de nodos del grafo. Se inicializa a uno cuando el usuario selecciona el nodo de origen (evento click de un nodo) y se incrementa cada vez que se añade un nuevo nodo al conjunto *B* (procedimiento "nueva_situación").

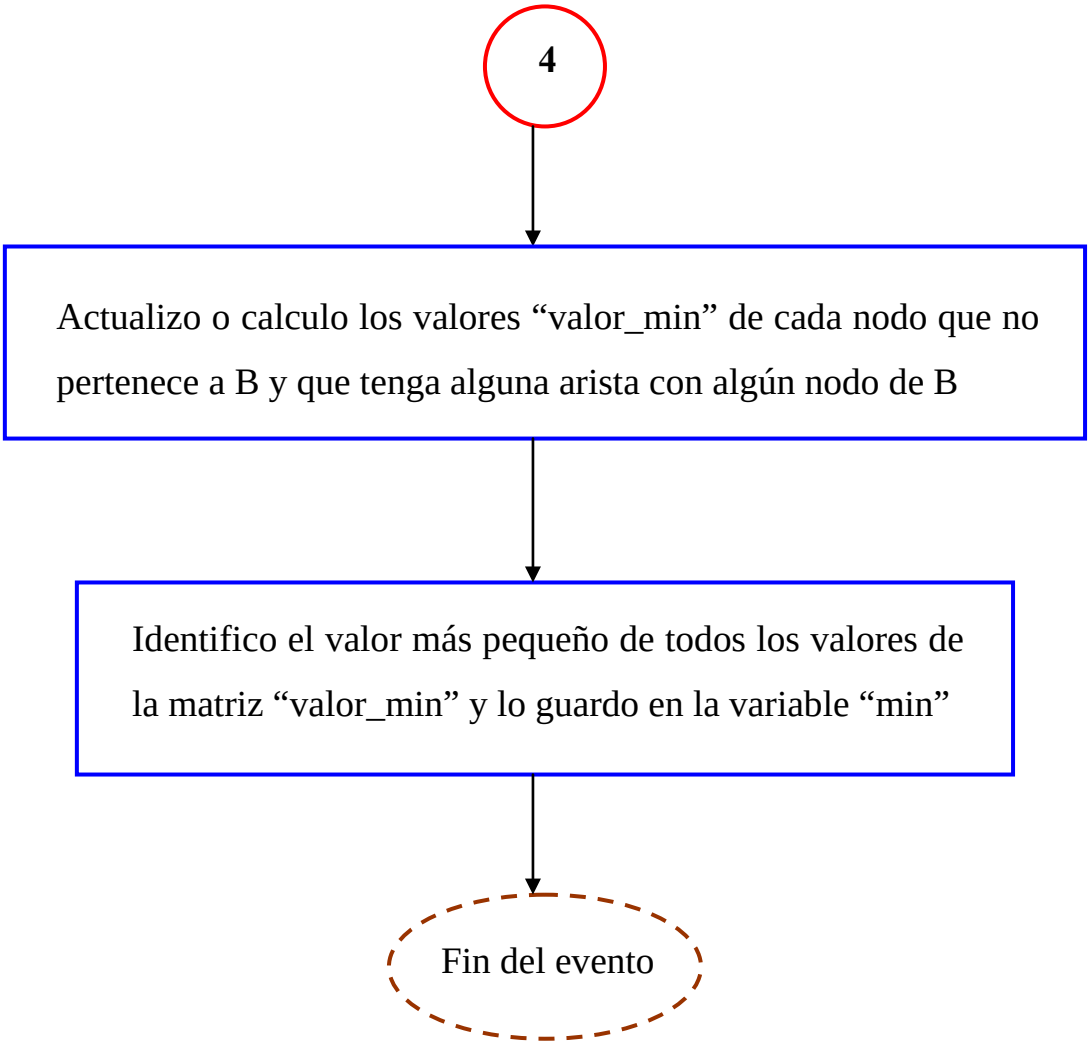
Organigrama funcional del **"evento click en un nodo"**

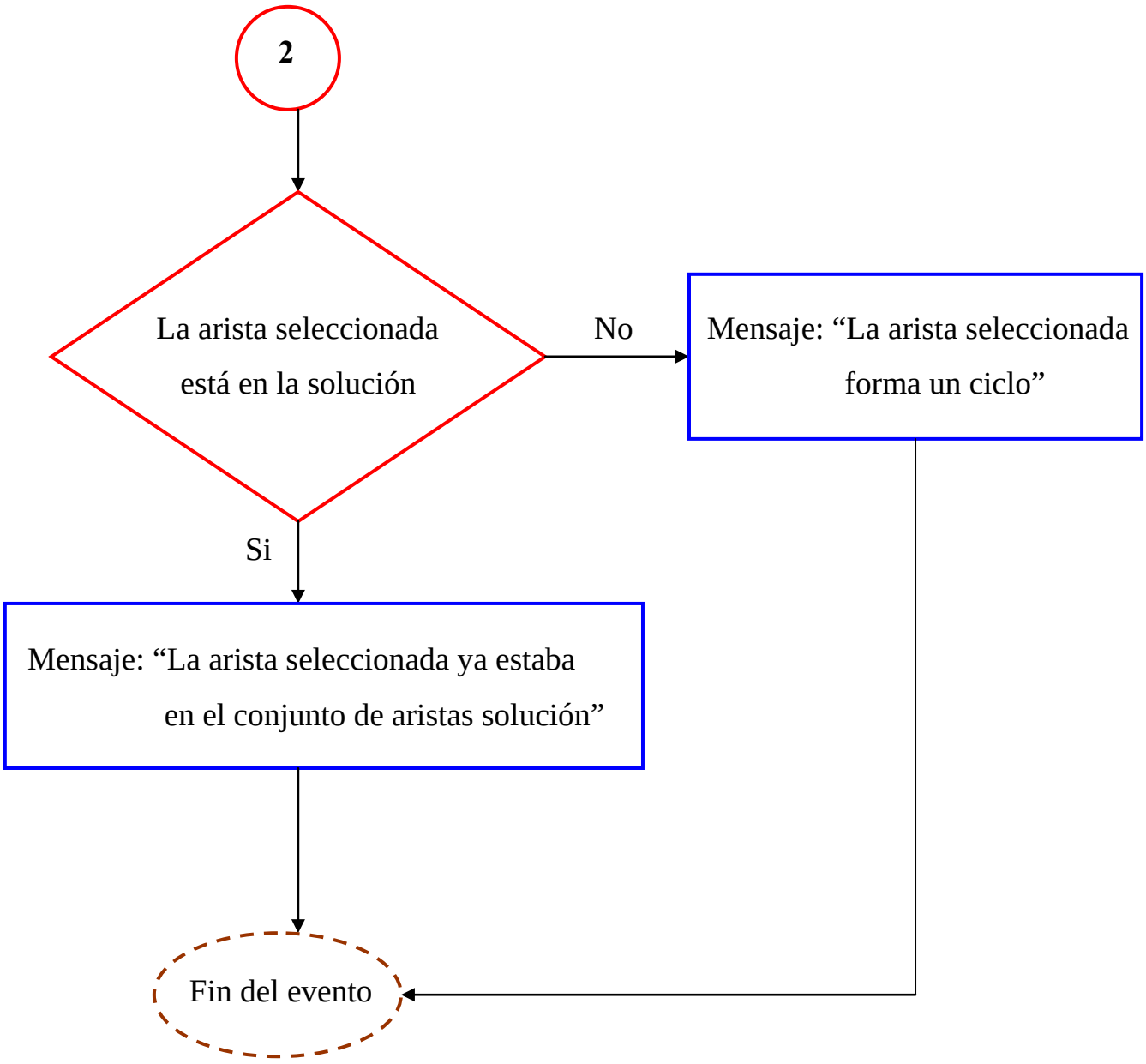


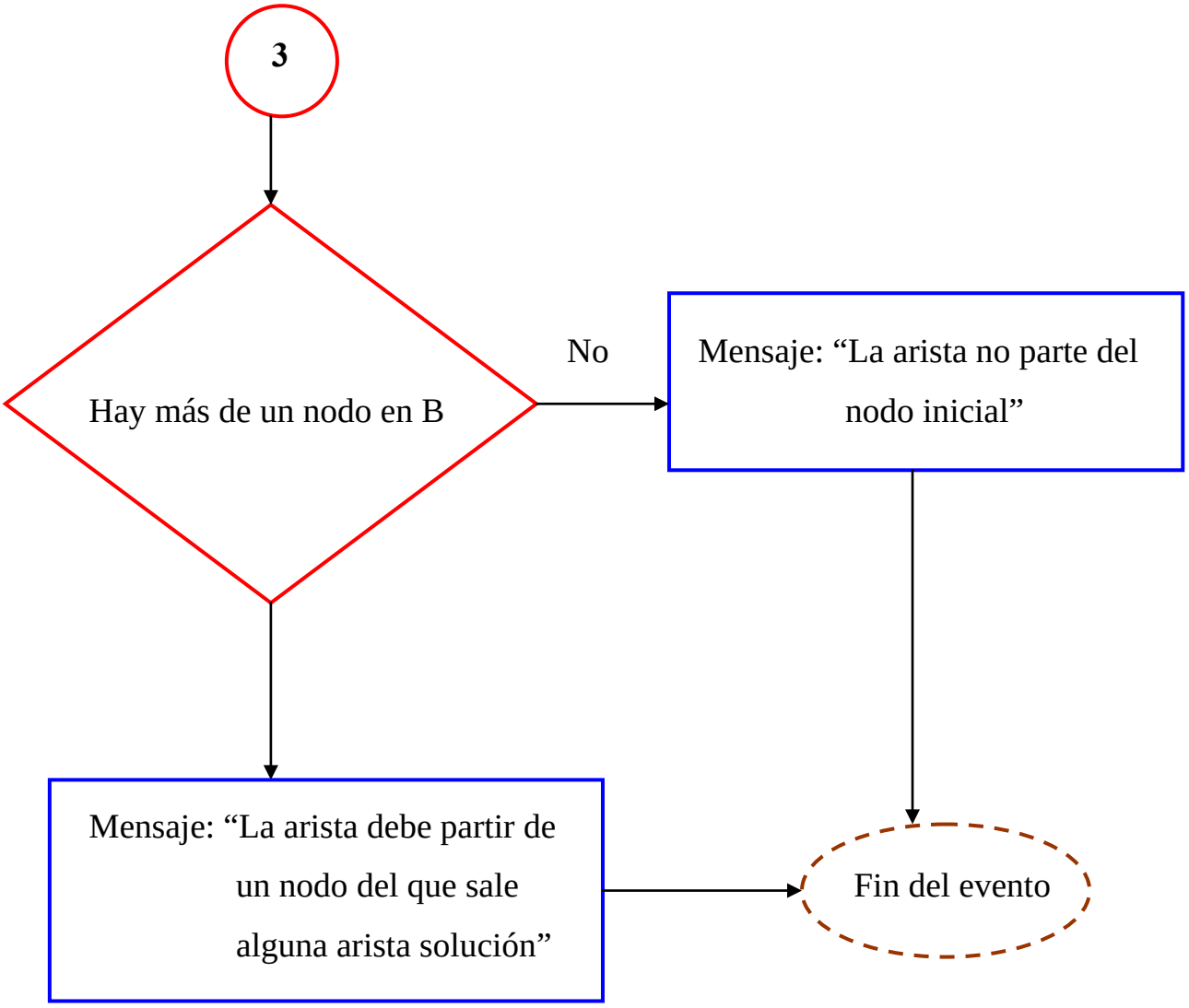
Organigrama funcional del **“evento click en una arista”**




1
valor de la arista seleccionada = min
No
Nº de nodos en B > 1
No
Si
Si
Añado la arista al conjunto de aristas solución y añado a B el nodo de la arista solución que no esta en B
Mensaje: "Existe una arista de menor valor entre algún nodo en el que incida alguna arista solución y nodos en los que no incidan aristas solución"
Nº de nodos en B = = Nº de nodos del grafo
No
Mensaje: "Existe una arista de menor valor entre el nodo inicial y el resto de nodos"
Si
4
Sacar las aristas solución y la suma de sus valores por pantalla
Fin de evento
Fin del programa


4
Actualizo o calculo los valores “valor_min” de cada nodo que no pertenece a B y que tenga alguna arista con algún nodo de B
Identifico el valor más pequeño de todos los valores de la matriz “valor_min” y lo guardo en la variable “min”
Fin del evento


2
La arista seleccionada está en la solución
No
Mensaje: “La arista seleccionada forma un ciclo”
Si
Mensaje: “La arista seleccionada ya estaba en el conjunto de aristas solución”
Fin del evento


3
Hay más de un nodo en B
No
Mensaje: “La arista no parte del nodo inicial”
Mensaje: “La arista debe partir de un nodo del que sale alguna arista solución”
Fin del evento

Programa principal

Obligo a que sólo esté activo el evento click en un nodo cuando se empieza la ejecución del programa

sólo_aristas ← False

Mensaje: "Seleccione el nodo de inicio"

Evento click en un nodo

Evento click en una arista

EVENTO CLICK EN UN NODO

si no sólo_aristas , entonces

nodo_inicio ⟵ nodo seleccionado por el usuario

índice ⟵ 0

(Pongo todos los nodos fuera de B excepto el nodo seleccionado por el usuario)

para i = 1 hasta n , hacer

para_nodos_fuera_de_B [i] ⟵ True

finpara

para_nodos_fuera_de_B [nodo_inicio] ⟵ False

(El número de nodos contenidos en B es uno)

número_nodos_en_B ⟵ 1

(Inicializo provisionalmente la matriz “con_aristas_hacia_B”)

para i = 1 hasta n , hacer

con_aristas_hacia_B [i] ⟵ True

finpara

con_aristas_hacia_B [nodo_inicio] ⟵ False

Implementación del algoritmo de Prim

(Calculo el contenido inicial de las matrices “valor_min” y “más_próximo_en_B”)

para i = 1 hasta n , hacer

si para_nodos_fuera_de_B [i] , entonces

si valor_arista [i , nodo_inicio] ≥ 0 , entonces

valor_min [i] ⟵ valor_arista [i , nodo_inicio]

más_próximo_en_B [i] ⟵ nodo_inicio

sino

con_aristas_hacia_B [i] ⟵ False

finsi

finsi

finpara

Calculo el valor más pequeño de todos los
“valor_min” y lo guardo en la variable “min”

calcular_min

(Obligo a que sólo esté activo el evento click en una arista)

sólo_aristas ⟵ True

Mensaje: “Seleccione una arista”

sino

Mensaje: “Debe seleccionar una arista”

finsi

Procedimiento **"calcular_min"**

De todos los valores existentes de la matriz *"valor_min"* calculo el más pequeño y lo guardo en la variable *"min"*.

```
min ⟵ ponemos en esta variable el valor más alto posible
para  j = 1  hasta  n  ,  hacer
    si  para_nodos_fuera_de_B [ j ]  ,  entonces
        si  con_aristas_hacia_B [ j ]  ,  entonces
            si  valor_min [ j ]  ≤  min  ,  entonces
                min ⟵ valor_min [ j ]
            finsi
        finsi
    finsi
finpara
```

EVENTO CLICK EN LA ARISTA

si sólo_aristas , entonces

(Los datos de la arista seleccionada los introduciré en las siguientes variables)

nodop ⟵ un nodo incidente en la arista
nodoq ⟵ el otro nodo incidente en la arista

valor_arista_usuario ⟵ valor de la arista seleccionada

La arista seleccionada debe estar entre un nodo de B y otro nodo fuera de B, para comprobarlo utilizo el procedimiento “lugar_arista”

lugar_arista

Después de ejecutarse el procedimiento “lugar_arista”, si la variable booleana “arista_posible” es True, la arista seleccionada está entre un nodo de B y otro nodo fuera de B, y si la variable booleana “los_dos_nodos_en_B” es True, la arista seleccionada está entre dos nodos de B

si arista_posible , entonces

si valor_arista_usuario = min , entonces

(El usuario ha elegido bien)
arista_correcta

sino

(El usuario ha elegido mal)
existe_arista_de_menor_valor

finsi

sino

(El usuario ha elegido mal)
situación_arista_erronea

finsi

sino

Mensaje: “Debe seleccionar el nodo origen”

finsi

Procedimiento **"lugar_arista"**

```
los_dos_nodos_en_B  ←——  False

arista_posible  ←——  False

si  para_nodos_fuera_de_B [ nodop ]  ,  entonces
    si  no  para_nodos_fuera_de_B [ nodoq ]  ,  entonces
        arista_posible  ←——  True
    finsi
sino
    si  para_nodos_fuera_de_B [ nodoq ]  ,  entonces
        arista_posible  ←——  True
    sino
        los_dos_nodos_en_B  ←——  True
    finsi
finsi
```

Procedimiento **“existe_arista_de_menor_valor”**

La arista seleccionada está entre un nodo de *B* y otro nodo que no pertenece a *B* pero no es la arista de menor valor.

si número_nodos_en_B > 1 , entonces

Mensaje: “Existe una arista de menor valor entre algún nodo en el que incida alguna arista solución y nodos en los que no incidan aristas solución”

sino

Mensaje: “Existe una arista de menor valor entre el nodo inicial y el resto de nodos”

finsi

Procedimiento **"situación_arista_erronea"**

La arista seleccionada esta mal elegida porque no está entre un nodo de *B* y otro nodo que no pertenece a *B*.

Nos quedan dos posibilidades en cuanto a la situación de la arista:

Que esté entre dos nodos de *B*.

Que esté entre dos nodos que no pertenecen a *B*.

```
si   los_dos_nodos_en_B   ,   entonces
        Ahora tengo que saber si la arista seleccionada es una arista que ya está
        en el conjunto de aristas solución o en caso contrario formará un ciclo
     donde_está_en_B
sino
     si   numero_nodos_en_B > 1   ,   entonces
          Mensaje: "La arista debe partir de un nodo del que sale alguna
                    arista solución"
     sino
          Mensaje: "La arista no parte del nodo inicial"
     finsi
finsi
```

Procedimiento **“donde_está_en_B”**

Si los dos nodos pertenecen a *B*, lógicamente, existe por lo menos una arista en el conjunto de aristas solución y más de un nodo en el conjunto *B*.

La función booleana *“está_en_la_solución”* me dirá si la arista seleccionada ya ha sido incluida con anterioridad en el conjunto de aristas solución.

si esta_en_la_solución , entonces

(Si la arista seleccionada está en el conjunto de aristas solución)

Mensaje: “La arista seleccionada ya estaba en el conjunto de aristas solución”

sino

(Si la arista seleccionada no está entre las aristas de la solución)

Mensaje: “La arista seleccionada forma un ciclo”

finsi

Función **"está_en_la_solución"**

está_en_la_solución = False

para i = 1 hasta índice , hacer

si ((nodop = aristas_de_la_solución.nodop [i] y nodoq = aristas_de_la_solución.nodoq [i]) ó (nodop = aristas_de_la_solución.nodoq [i] y nodoq = aristas_de_la_solución.nodop [i])) , entonces

está_en_la_solución = True

finsi

finpara

Procedimiento **“arista_correcta”**

Mensaje: “Arista de la solución”

Poner la arista de color rojo

Añado la arista al conjunto de aristas solución, y añado a B el nodo de la arista solución que no esta en B

nueva_situación

(Actualizo los valores “valor_min” de cada nodo que no pertenece a B)

actualizar_mínimos

(Ahora tengo que calcular el menor de todos los valores “valor_min”)

calcular_min

Procedimiento **"nueva_situación"**

(Añado la arista seleccionada por el usuario al conjunto de aristas solución)

índice ← índice + 1

aristas_de_la_solución.nodop [índice] ← nodop

aristas_de_la_solución.nodoq [índice] ← nodoq

aristas_de_la_solución.valor [índice] ← valor_arista_usuario

(Añado a B el nodo de la arista solución que no esta en B)

si para_nodos_fuera_de_B [nodop] , entonces

    para_nodos_fuera_de_B [nodop] ← False

    aux ← nodop

sino

    para_nodos_fuera_de_B [nodoq] ← False

    aux ← nodoq

finsi

(Incremento en uno el número de nodos en B)

número_nodos_en_B ← número_nodos_en_B + 1

(Tengo que comprobar si se han completado todas las aristas de la solución)

(Siendo “n” el número de nodos del grafo)

si número_nodos_en_B = n , entonces

    sacar_la_solución

    Fin del programa

finsi

Procedimiento **"actualizar_mínimos"**

```
para  j = 1  hasta  n  ,  hacer
    si  para_nodos_fuera_de_B [ j ]  ,  entonces
        si  con_aristas_hacia_B [ j ]  ,  entonces
                proceso1
        sino
                proceso2
        finsi
    finsi
finpara
```

Procedimiento **"proceso1"**

```
si  valor_arista [ j , aux ]  ≥  0  ,  entonces
    si  valor_arista [ j , aux ]  <  valor_min [ j ]  ,  entonces
        valor_min [ j ]  ←  valor_arista [ j , aux ]
        más_próximo_en_B [ j ]  ←  aux
    finsi
finsi
```

Procedimiento **"proceso2"**

```
si  valor_arista [ j , aux ]  ≥  0  ,  entonces
    valor_min [ j ]  ←  valor_arista [ j , aux ]
    más_próximo_en_B [ j ]  ←  aux
    con_aristas_hacia_B [ j ]  ←  True
finsi
```

Procedimiento **"sacar_la_solución"**

Las aristas solución junto con la suma total de los valores de las aristas solución se muestran en una ventana.

Los datos se obtienen según el siguiente procedimiento:

```
suma ← 0

para  i = 1  hasta  índice  ,  hacer

        nodop ← aristas_de_la_solución.nodop [ i ]

        nodoq ← aristas_de_la_solución.nodoq [ i ]

        valor ← aristas_de_la_solución.valor [ i ]

        Mostrar  "nodop" , "nodoq" , "valor"

        suma ← suma + valor

finpara

Mostrar  "suma"
```

2.2.5.- Implementación del algoritmo de Prim con todas las soluciones (solución automática)

Para explicar la resolución del algoritmo de Prim con todas las soluciones óptimas seguiré las siguientes fases:

Primera fase:

- Primero explicaré el proceso general que voy a seguir para obtener todas las soluciones óptimas únicas.
- A continuación, explicaré los soportes de datos (matrices) sobre los que se va a apoyar el programa.

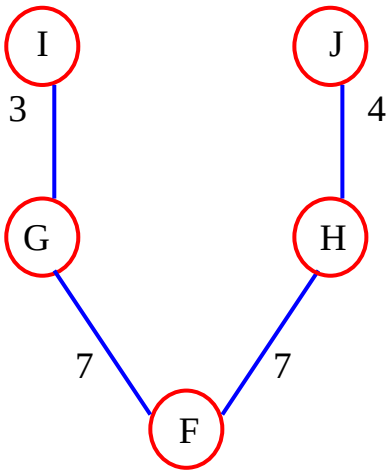
Segunda fase:

- Desarrollo del organigrama funcional para ver todo el proceso de forma global.
- Explicación de las variables del programa para controlar y seguir mejor el flujo de instrucciones.
- Implementación del programa en pseudocódigo con comentarios explicativos.

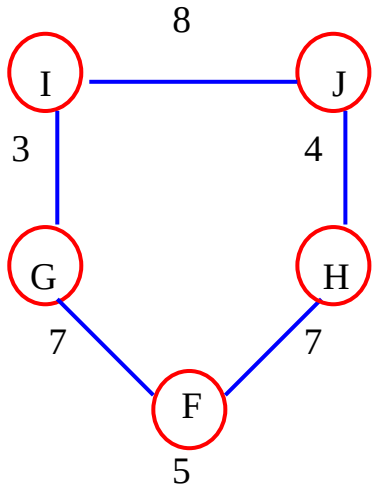
2.2.5.1.- Proceso general y soporte de datos

El programa sigue el algoritmo de Prim dividiendo el grafo en dos grupos de nodos: nodos en el conjunto *B* y nodos fuera de *B* y actuando sobre las aristas que unen estos dos grupos de nodos.

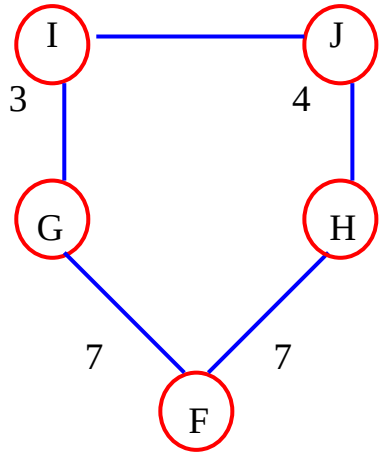
Las aristas que pueden producir varias soluciones son las aristas que tienen el mismo valor. Si existen aristas con el mismo valor entre nodos de *B* y nodos fuera de *B* de tal forma que esas aristas son las apropiadas, según el algoritmo de Prim, para estar en el conjunto de aristas de la solución (tienen el valor más pequeño entre nodos de *B* y nodos fuera de *B*), hay que crear tantas soluciones óptimas como aristas con el mismo valor se produzcan. Después, sólo cuando las *(n –1)* aristas de cada solución óptima completa sean conocidas, hay que asegurarse de que las soluciones halladas no están repetidas. Para explicar el porque de este proceder seguiremos varios ejemplos. En todos los ejemplos empezamos el algoritmo de Prim a partir del nodo *F*:



En este ejemplo, a pesar de que existen dos aristas con el mismo valor (7) , sólo existe una solución òptima al no poderse descartar ninguna arista (el grafo no sería conexo).



En este ejemplo sigue habiendo una única solución óptima al ser la arista de valor ocho mayor que las aristas de valor siete.



Sin embargo, en este ejemplo existen dos soluciones óptimas al ser la arista de valor cinco menor que las aristas de valor siete:

{ I , J } , { G , I } , { H , J } , { F , G }

———————

{ I , J } , { G , I } , { H , J } , { F , H }

A diferencia del algoritmo de Kruskal que tiene una visión global de todo el grafo seleccionando todas las aristas del grafo con el mismo valor, el algoritmo de Prim comienza en un nodo y no sabe las aristas que va a encontrar salvo las aristas que salen del conjunto *B* hacia nodos fuera de *B*, es decir, tiene una visión parcial del grafo.

Si el nodo de inicio es el nodo *F*, al empezar, no sabemos que valor tiene la arista $\{ I , J \}$, es más, no sabemos si existe dicha arista, lo único que conocemos es que de *F* salen dos aristas de valor siete y no podemos saber si estas dos aristas del mismo valor van a generar una o dos soluciones. Por estas razones, nos creamos dos

soluciones y en cada solución irá una arista distinta de valor siete. Al final, sólo cuando completemos las dos soluciones, sabremos si están repetidas o no.

Si nos encontramos varias aristas con el mismo valor que puedan estar en la solución óptima, creamos tantas soluciones óptimas como aristas con el mismo valor y cuando se complete cada solución con todas sus aristas (*n – 1* aristas, siendo *n* el número de nodos del grafo), se compara con las soluciones óptimas completas ya existentes para ver si está repetida. Eliminando todas las soluciones repetidas, al final nos quedará una matriz *"aristas_de_la_solución"* con un número de filas igual al número de soluciones óptimas únicas y en cada fila, por tanto, estarán las *(n – 1)* aristas de una solución óptima, única y completa.

Por tanto, mientras no se presenten varias aristas con el mismo valor como candidatas válidas a ser la próxima arista que se incluirá en el conjunto de aristas de la solución, aplicaré el algoritmo de Prim con normalidad arista por arista y cuando surjan varias aristas a la vez como posibles aristas de la solución se hará lo siguiente:

Cada arista estará en una fila distinta de la matriz *"aristas_de_la_solución"*, es decir, una arista se incorporará normalmente a la fila activa, entendiendo por ***fila activa*** la fila en donde estamos guardando, en el momento actual, las aristas de la solución según el algoritmo de Prim (la variable *"fila"* nos dice en todo momento cual es la fila activa) y para el resto de aristas con el mismo valor se crean filas nuevas copiando todas las aristas menos la última desde la fila activa al resto de filas recién creadas y poniendo como última arista de cada fila recién creada una arista distinta con el mismo valor.

Evidentemente, el número de filas nuevas creadas será igual al número de aristas con el mismo valor menos una, ya que una de dichas aristas se incorpora a la fila activa.

A continuación, seguimos el proceso normalmente en la fila activa hasta completar las *(n – 1)* aristas.

El proceso de crear nuevas filas se produce siempre que nos encontramos un grupo de aristas con el mismo valor.

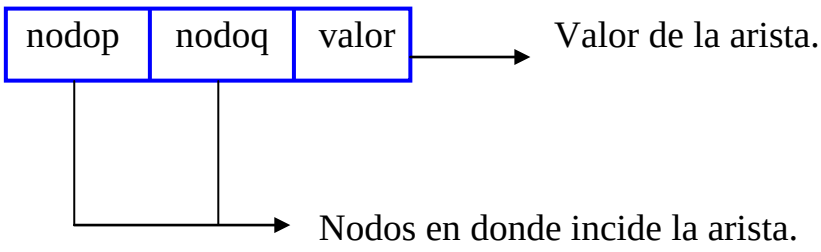
Cuando se completa con *(n – 1)* aristas una determinada solución óptima, comprobamos que la solución no esté repetida y si lo esta, eliminamos la fila activa que la almacena. A continuación examinamos el resto de filas de la matriz *"aristas_de_la_solución"* y si todas sus filas están completas (tienen *(n – 1)* aristas), se termina el algoritmo de Prim. Si queda alguna fila que aun no tiene todas las aristas *(n – 1)*, dicha fila pasa a ser la fila activa y continuamos aplicándola el algoritmo de Prim para completar las aristas de la solución que faltan.

Matriz "aristas_de_la_solución"

Es una matriz que al terminar el programa tiene un número de filas igual al número total de soluciones óptimas únicas posible, y *"n – 1"* columnas, siendo *"n"* el número de nodos del grafo.

Cada fila representa una única solución óptima y cada solución óptima tiene exactamente *"n – 1"* aristas.

Cada elemento de esta matriz contiene la información de una arista solución:



Para referirme a un campo en particular de una fila *"i"* y de una columna (arista) *"j"* usaré la notación:

aristas_de_la_solución.nodop [i , j]

aristas_de_la_solución.nodoq [i , j]

aristas_de_la_solución.valor [i , j]

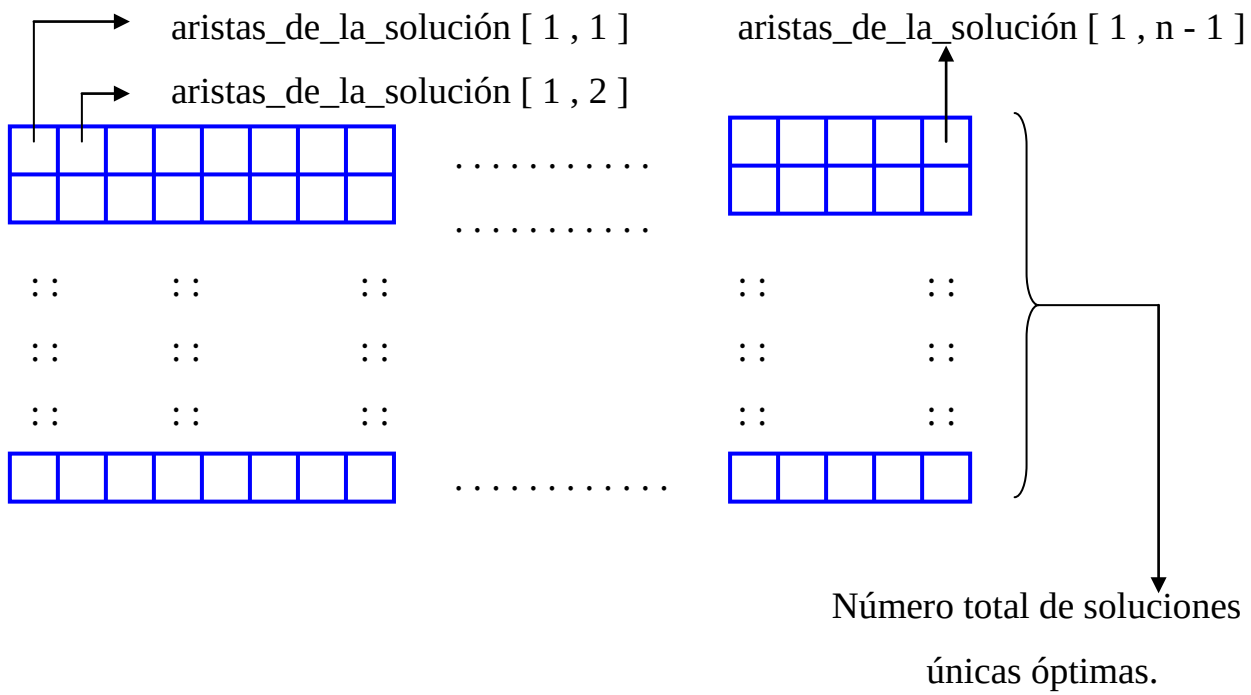
y para referirme a una arista *"j"* de una fila *"i"* de la matriz con todos sus campos:

aristas_de_la_solución [i , j]

Para saber en todo momento donde tengo que almacenar la siguiente arista de la solución, utilizaré las variables *"fila"* y *"columna"*. Estas variables, están inicializadas a uno y cuando se quiere almacenar una nueva arista se incrementa *"columna"* en uno, a no ser que la fila en donde estemos ya tenga *"n – 1"* aristas con lo cual incrementaríamos *"fila"* en uno y ponemos *"columna"* a uno. En definitiva, insertaríamos la nueva arista solución en:

aristas_de_la_solución [fila , columna]

La matriz completa *"aristas_de_la_solución"* cuando finaliza el algoritmo de Prim con todas las soluciones será:



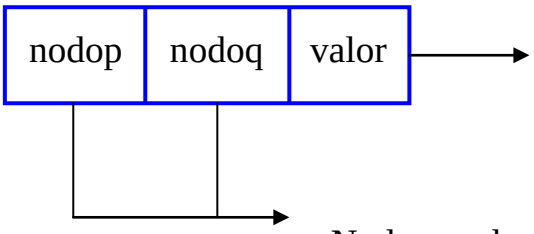
Matriz ”solución parcial”

Tiene un número de filas igual al número de aristas con el mismo valor, menos una, que pueden estar en el conjunto de aristas de la solución, es decir, el número de soluciones óptimas menos una que se pueden producir al intentar añadir una arista nueva al conjunto de aristas solución (menos una porque una de las aristas con el mismo valor se queda en la fila activa).

Cada vez que intentamos añadir una nueva arista al conjunto de aristas solución y se producen varias soluciones, se inicializa el número de filas de esta matriz al valor del número de soluciones óptimas menos una.

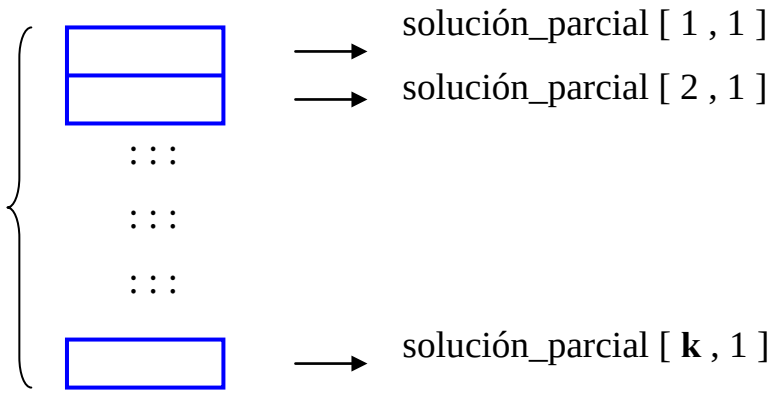
Sólo tendrá una columna y corresponderá a la arista con el mismo valor que deriva en una nueva solución.

Cada elemento tendrá la misma información que un elemento de la matriz *“aristas_de_la_solución”*:



La matriz completa tiene la estructura siguiente:

Suponemos que al intentar añadir una nueva arista a la matriz aristas_de_la_solución se han producido k + 1 posibles soluciones.



Cada único elemento de una fila es una arista que nos abre un camino hacia una solución óptima (que puede que no sea única). Más tarde, cuando se completa una solución óptima (*" n – 1 "* aristas en una fila de la matriz *"aristas_de_la_solución"*) se comprueba si dicha solución está repetida.

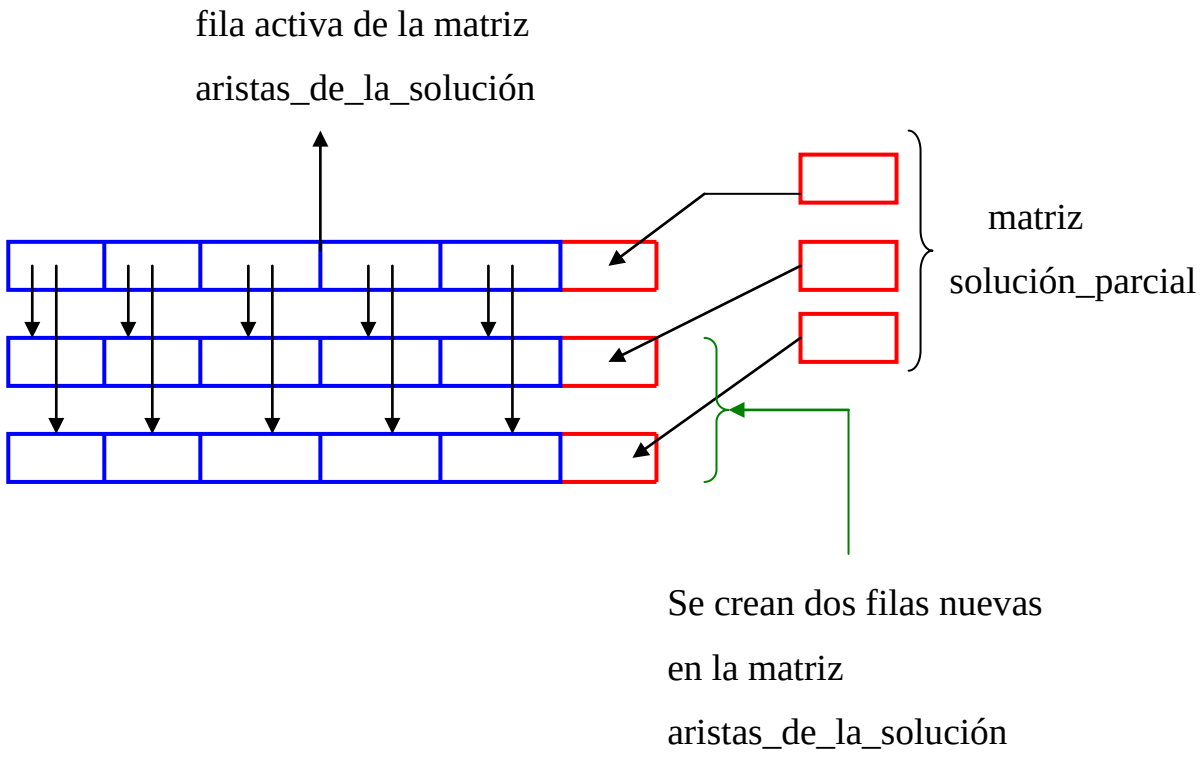
Por ejemplo, con un grafo de 10 nodos y una fila activa con 5 aristas de la solución, nos encontramos con que hay tres aristas con el mismo valor que pueden estar en la solución óptima de la fila activa. Los pasos a seguir son:

1.- Creamos dos filas nuevas en la matriz *"aristas_de_la_solución"* (número de aristas con el mismo valor – 1).

2.- Copiamos la fila activa en las dos filas nuevas.

3.- Una arista con el mismo valor va a la fila activa y las otras dos aristas con el mismo valor a las dos filas nuevas.

Gráficamente:



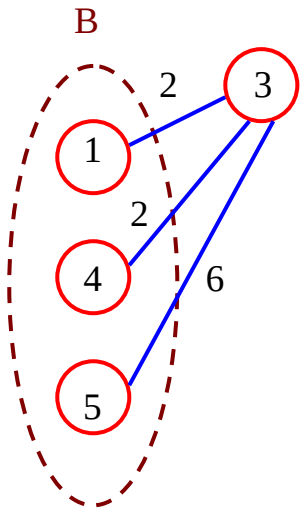
Cada vez que nos encontremos con varias aristas con el mismo valor candidatas a estar en el conjunto de aristas solución (fila activa) seguiremos el mismo proceso.

Cuando la fila activa se complete (tenga $n - 1$ aristas) pasamos la fila activa a la siguiente fila de la matriz *"aristas_de_la_solución"* para aplicando el algoritmo de Prim completar las aristas solución que faltan.

Matriz "más_próximo_en_B"

La matriz *"más_próximo_en_B"* es una matriz de $n \times n$ elementos (siendo n el número de nodos del grafo) que nos dice los nodos *"j"* que están en *B* para un solo nodo *"i"* que no está en *B*, tal que *valor_min [i]* es precisamente el valor de las aristas entre el nodo *"i"* y cada nodo *"j"*.

"i" representa la fila y los nodos *"j"* las columnas. Por ejemplo:



Si como se muestra en la figura el nodo *"i"* es el nodo 3 y el valor más pequeño de la arista que une el nodo 3 que no pertenece a B con cualquier nodo de *B* es 2 (aristas *{ 1 , 3 }* y *{ 3 , 4 }*) , es decir, *" valor_min [3] = 2 "* en las columnas 1 y 4 pondremos los valores 1 y 4 respectivamente y la matriz *"más_próximo_en_B"* para la fila 3 quedará:

más_próximo_en_B [3 , 1] = 1
más_próximo_en_B [3 , 4] = 4

Inicialmente, todos los elementos de la matriz *"más_próximo_en_B"* están puestos al valor - 1 y se van actualizando a medida que avanza el programa. Si se añade un nuevo nodo *"k"* al conjunto *B* y además la arista que une los nodos *"i"* y *"k"* es la de menor valor entre *"i"* y cualquier nodo de *B*, *"valor_min"* cambia y hacemos lo siguiente:

1.- Inicializamos al valor – 1 la fila *"i"* de la matriz *"más_próximo_en_B"* :

más_próximo_en_B [i , j] = - 1 ; j = 1 , 2 , ... , n

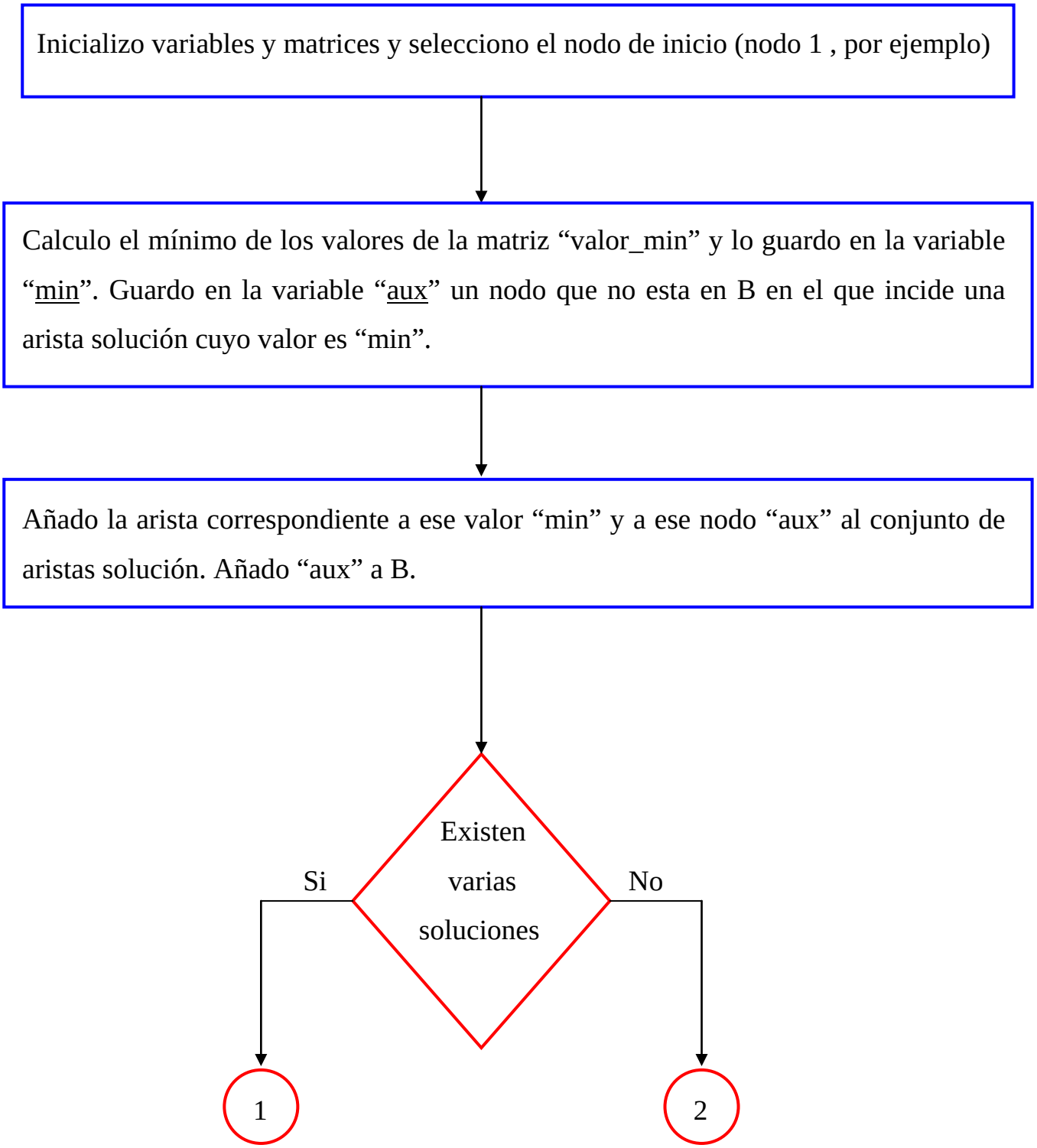
2.- Ponemos los nodos *"j"* en las columnas *"j"*

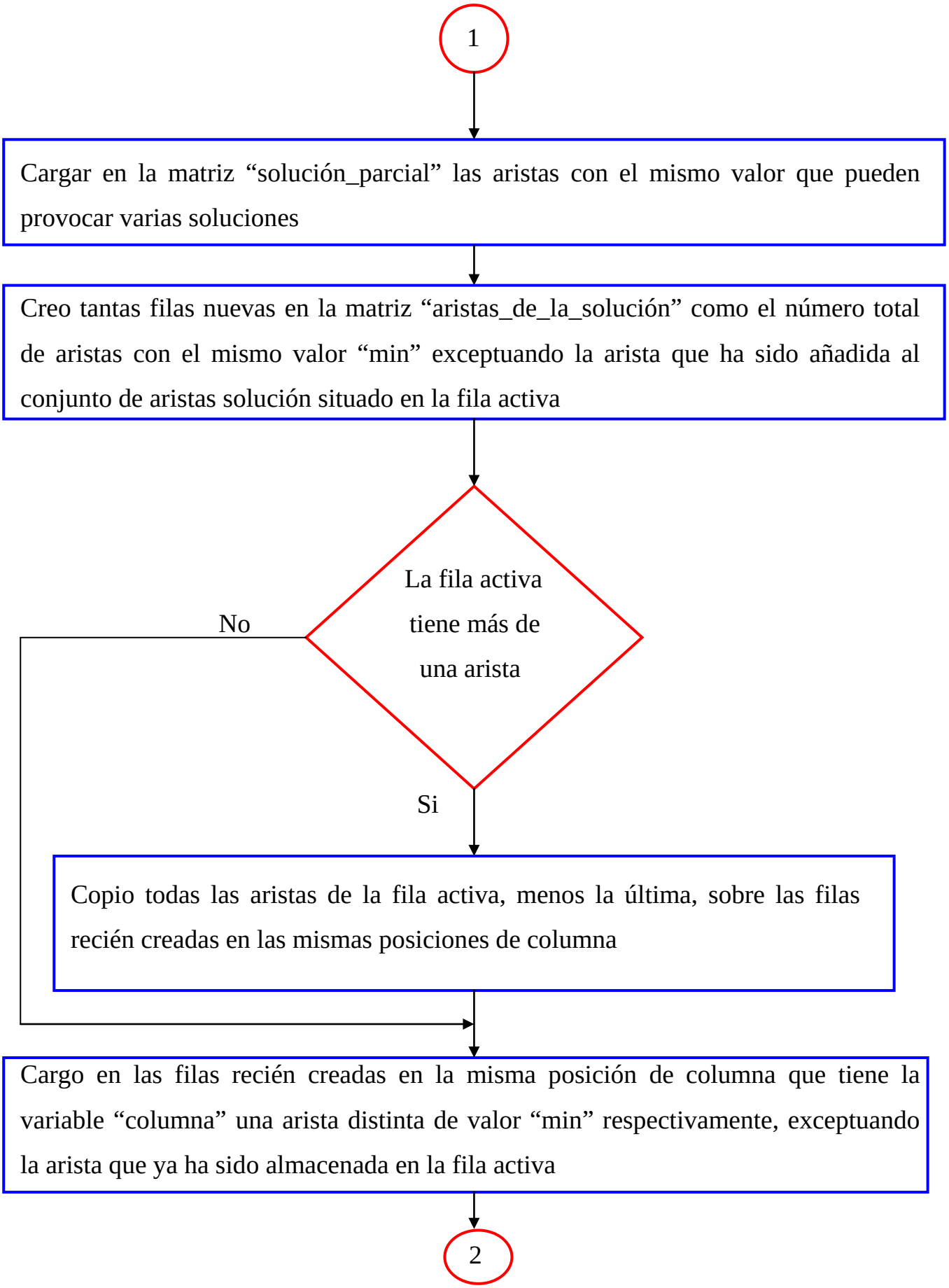
más_próximo_en_B [i , j] = j tal que la arista de *"valor_min [i]"* tenga el mismo valor que las aristas *{ i , j }*.

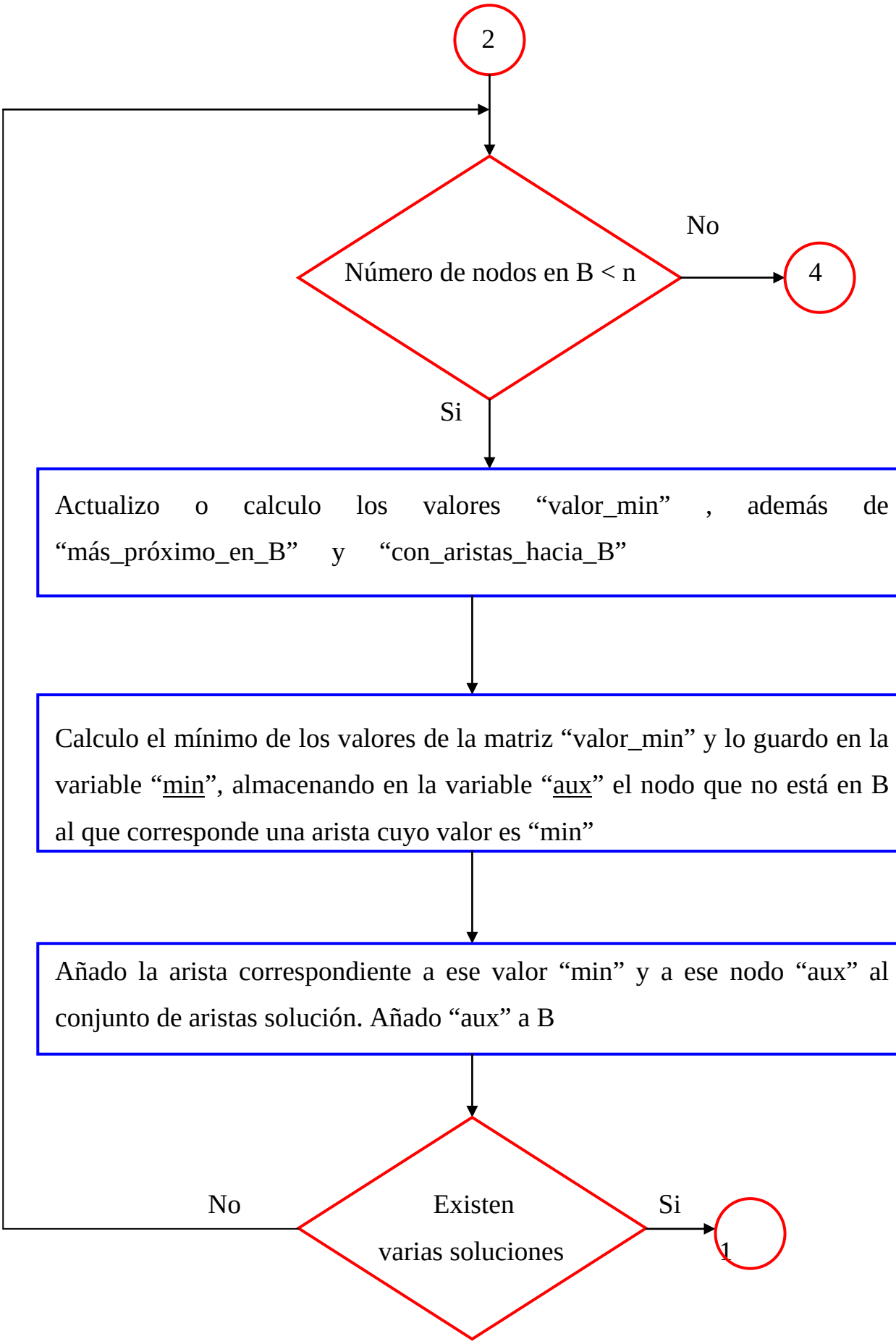
Las matrices *"para_nodos_fuera_de_B"*, *"con_aristas_hacia_B"* y *"valor_min"* permanecen tal y como se definieron en la implementación automática del algoritmo de Prim para calcular solamente una solución.

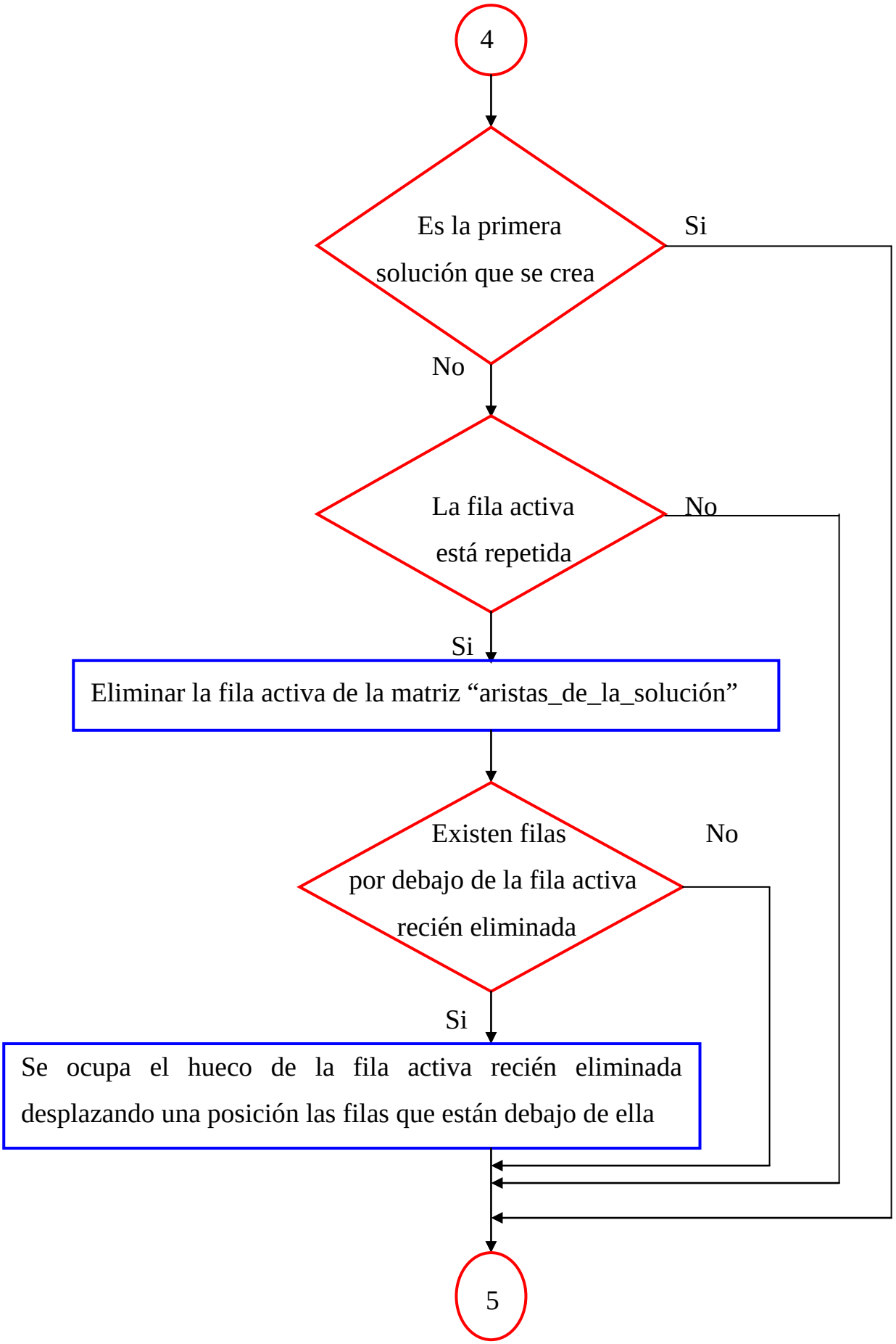
2.2.5.2.- Organigrama, variables y pseudocódigo

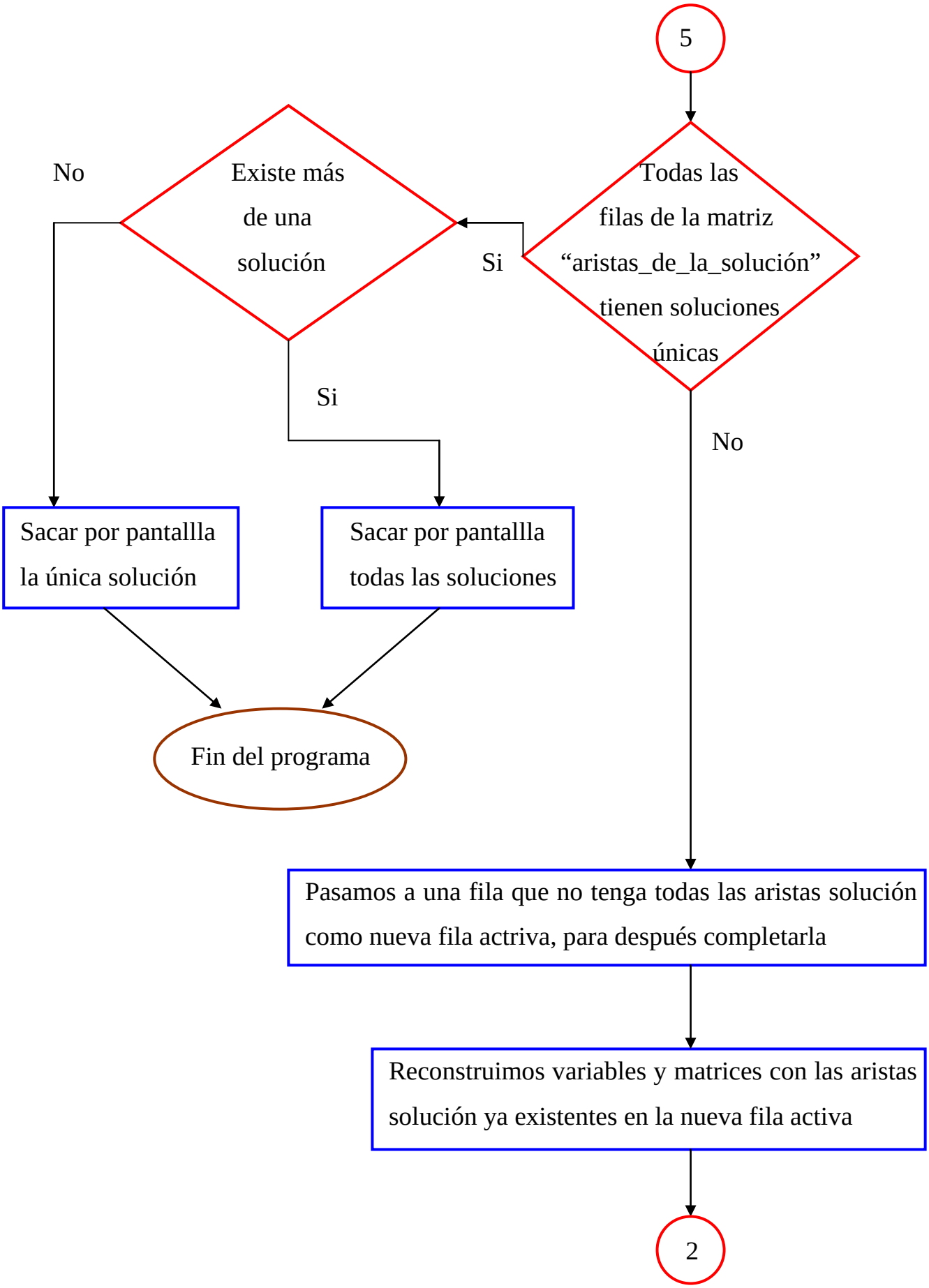
Organigrama funcional




1
Cargar en la matriz “solución_parcial” las aristas con el mismo valor que pueden provocar varias soluciones
Creo tantas filas nuevas en la matriz “aristas_de_la_solución” como el número total de aristas con el mismo valor “min” exceptuando la arista que ha sido añadida al conjunto de aristas solución situado en la fila activa
La fila activa tiene más de una arista
No
Si
Copio todas las aristas de la fila activa, menos la última, sobre las filas recién creadas en las mismas posiciones de columna
Cargo en las filas recién creadas en la misma posición de columna que tiene la variable “columna” una arista distinta de valor “min” respectivamente, exceptuando la arista que ya ha sido almacenada en la fila activa
2


2
Número de nodos en B < n
No
4
Si
Actualizo o calculo los valores “valor_min” , además de “más_próximo_en_B” y “con_aristas_hacia_B”
Calculo el mínimo de los valores de la matriz “valor_min” y lo guardo en la variable “min”, almacenando en la variable “aux” el nodo que no está en B al que corresponde una arista cuyo valor es “min”
Añado la arista correspondiente a ese valor “min” y a ese nodo “aux” al conjunto de aristas solución. Añado “aux” a B
Existen varias soluciones
No
Si
1


4
Es la primera solución que se crea
Si
No
La fila activa está repetida
No
Si
Eliminar la fila activa de la matriz “aristas_de_la_solución”
Existen filas por debajo de la fila activa recién eliminada
No
Si
Se ocupa el hueco de la fila activa recién eliminada desplazando una posición las filas que están debajo de ella
5


5
Todas las filas de la matriz "aristas_de_la_solución" tienen soluciones únicas
Si
Existe más de una solución
No
Si
No
Sacar por pantallla la única solución
Sacar por pantallla todas las soluciones
Fin del programa
Pasamos a una fila que no tenga todas las aristas solución como nueva fila actriva, para después completarla
Reconstruimos variables y matrices con las aristas solución ya existentes en la nueva fila activa
2

Variables

- La variable *"**fila**"* nos indica en todo momento cual es la fila activa.

- En la variable *"**columna**"* tenemos el índice de la columna de la última arista almacenada de la solución en la fila activa de la matriz *"aristas_de_la_solución"*.

- La variable *"**total_filas_aristas_de_la_solución**"* nos indica en todo momento el número de filas de la matriz *"aristas_de_la_solución"*. Se inicializa, lógicamente, al valor uno, se incrementa cuando creamos filas nuevas que tienen soluciones óptimas (una solución óptima por fila), y se decrementa cuando alguna de estas soluciones está repetida (se elimina la fila correspondiente a la solución repetida). Con este proceso de eliminar soluciones repetidas, conseguimos que al finalizar el programa, cada fila de la matriz *"aristas_de_la_solución"* tenga una solución óptima única.

- La variable *"**número_de_soluciones**"* representa el número de soluciones únicas que existen en todo momento. Lógicamente, cuando *"número_de_soluciones"* y *"total_filas_aristas_de_la_solución"* son iguales, terminamos el programa.

- *"**número_nodos_en_B**"* nos informa del número de nodos que están en el conjunto *B*. Con esta variable controlamos el hecho de completar una solución óptima en una fila de la matriz *"aristas_de_la_solución"* con todas las aristas de dicha solución y esto se produce cuando esta variable toma el valor *"n"* (siendo *"n"* el número de nodos del grafo).

- La variable *"**min**"* nos informa del valor más pequeño de todos los valores de la matriz *"valor_min"* existentes para saber que arista es la próxima arista de la solución.

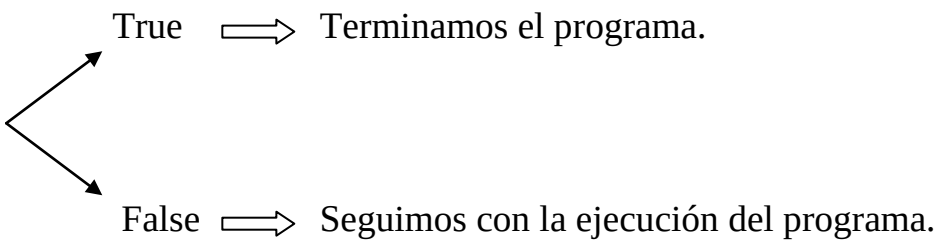
- *"**aux**"* es una variable auxiliar de ayuda que en un principio nos dice el nodo que no está en el conjunto *B* del que sale la siguiente arista solución. Después, dicho nodo guardado en la variable *"aux"* se introduce en *B*.

- *"**aux_en_B**"* nos dice que nodo de *B* es en el que incide la próxima arista de la solución.

- *"**total_filas_nuevas**"* nos indica las filas que se han creado cada vez que nos encontramos varias aristas con el mismo valor como candidatas a estar en el conjunto de aristas de la solución. Este valor es igual al número de aristas con el mismo valor menos una.

- *"**repetida**"* es una variable booleana que nos dice si la solución almacenada y completada en la fila actual está repetida o no.

- La variable *"**fin_del_programa**"* nos dice, lógicamente, cuando se termina el programa:



  Para ello, esta variable se pone como condición en el bucle *"mientras"* del programa principal.

Programa principal

(Inicializo variables)

fila ⟵ 1

columna ⟵ 0

total_filas_aristas_de_la_solución ⟵ 1

número_de_soluciones ⟵ 0

fin_del_programa ⟵ False

(Selecciono como nodo de partida el nodo 1 e inicializo variables y matrices)

nodo_inicial

(Calculo las variables “min” y “aux”)

calcular_min_aux

Añado al conjunto de aristas solución la arista de valor “min” que incide en el nodo “aux” y guardo “aux” en B

añadir_arista

Si existen varias aristas con el mismo valor “min” anteriormente calculado, se crean nuevas filas para tratar las posibles nuevas soluciones producidas

comprobar_varias_soluciones

mientras no fin_del_programa , hacer

si número_nodos_en_B < n , entonces

Actualizo o calculo los posibles nuevos valores de “valor_min” “más_próximo_en_B” y “con_aristas_hacia_B” ya que ahora B tiene un nodo más (el nodo que está en la variable “aux”)

actualizar_mínimos

(Calculo las variables “min” y “aux”)

calcular_min_aux

Añado al conjunto de aristas solución la arista de valor “min” que incide en el nodo “aux” y guardo “aux” en B

añadir_arista

Si existen varias aristas con el mismo valor “min” anteriormente calculado, se crean nuevas filas para tratar las posibles nuevas soluciones

comprobar_varias_soluciones

sino

fila_completada

finsi

finmientras

Procedimiento **"nodo_inicial"**

Selecciono como nodo de partida el nodo 1 , por ejemplo, e inicializo variables y matrices a partir de esta elección, es decir, calculo *"para_nodos_fuera_de_B"*, *"con_aristas_hacia_B"* , *"más_próximo_en_B"* y *"valor_min"*.

(Inicializo la matriz "más_próximo_en_B")

```
para  i = 1  hasta  n  ,  hacer
    para  j = 1  hasta  n  ,  hacer
        más_próximo_en_B [ i , j ]  ←  - 1
    finpara
finpara
```

(Inicializo la matriz "para_nodos_fuera_de_B")

```
para  i = 2  hasta  n  ,  hacer
        para_nodos_fuera_de_B [ i ]  ←  True
finpara
para_nodos_fuera_de_B [ 1 ]  ←  False
```

(Inicializo la variable que nos dice el número de nodos en B)

número_nodos_en_B ← 1

```
(Inicializo, en parte, la matriz  "con_aristas_hacia_B")
para  i = 2  hasta  n  ,  hacer
        con_aristas_hacia_B [ i ]  ←  True
finpara
con_aristas_hacia_B [ 1 ]  ←  False

Calculo los valores "valor_min" y "más_próximo_en_B"
además de inicializar la matriz  "con_aristas_hacia_B"

para  i =  2  hasta  n  ,  hacer

    si  valor_arista [ i , 1]  ≥ 0  ,  entonces
            valor_min [ i ]  ←  valor_arista [ i , 1 ]
            mas_proximo_en _B [ i , 1 ]  ←  1
    si no
            (Termino de inicializar la matriz  "con_aristas_hacia_B")
            con_aristas_hacia_B [ i ] ←  False
    finsi

finpara
```

Procedimiento **"calcular_min_aux"**

Calculo el valor más pequeño de todos los *"valor_min"* y lo guardo en la variable *"min"*, también guardo el nodo de la arista correspondiente al valor *"min"* que no está en *B* en la variable *"aux"*.

Si existen varias aristas con el mismo valor *"min"*, es decir, varios *"valor_min"* iguales a *"min"*, el nodo que guarda la variable *"aux"* pertenece a una cualquiera de dichas aristas.

```
min ← ponemos en esta variable el valor mas alto posible.

para  j = 2  hasta  n  ,  hacer
    si  para_nodos_fuera_de_B [ j ]  ,  entonces
        si  con_aristas_hacia_B [ j ]  ,  entonces
            si  valor_min [ j ]  ≤  min  ,  entonces

                min ← valor_min [ j ]

                (Guardamos el nodo "j" en la variable "aux")
                aux ← j

            finsi
        finsi
    finsi
finpara
```

Procedimiento **"añadir_arista"**

La siguiente arista de la solución para la fila activa irá del nodo contenido en la variable "aux" al nodo "aux_en_B" que tengo que calcular a continuación y su valor será "valor_min [aux]".

Ahora, añado la arista solución a T:

$T \leftarrow T \cup \{ \underbrace{\text{mas\_proximo\_en\_B [aux , aux\_en\_B]}}_{\in B} , \underbrace{\text{aux}}_{\notin B} \}$

columna ⟵ columna + 1

```
para j = 1 hasta n , hacer
    si no para_nodos_fuera_de_B [ j ]
        si más_próximo_en_B [ aux , j ] ≠ - 1 , entonces
            aux_en_B ⟵ j
            más_próximo_en_B [ aux , j ] ⟵ - 1
            salir del para
        finsi
    finsi
finpara
```

aristas_de_la_solución.nodop [fila , columna] ⟵ aux_en_B

arristas_de_la_solución.nodoq [fila , columna] ⟵ aux

arristas_de_la_solución.valor [fila , columna] ⟵valor_min [aux]

(Añado el nodo almacenado en la variable “aux” a B)

para_nodos_fuera_de_B [aux] ⟵False

(Sumo 1 al número de nodos de B)

numero_nodos_en_B ⟵ numero_nodos_en_B + 1

Procedimiento **"comprobar_varias_soluciones"**

Tengo que buscar las aristas que tienen el último valor *"min"* calculado entre un nodo de *B* y otro nodo que no pertenece a *B* incluido *"aux"*. Como la arista que va del nodo *"aux"* al nodo *"aux_en_B"* ya ha sido incluida en el conjunto de aristas solución de la fila activa, hay que descartarla.

```
fila_solución_parcial ←——— 0

para  j = 2  ,  hasta  n  ,  hacer

    si  para_nodos_fuera_de_B [ j ]  ,  entonces

        si  con_aristas_hacia_B [ j ]  ,  entonces

            si  valor_min [ j ]  =  min  ,  entonces

                (Cargo la matriz "solución_parcial")

                almacenar_en_solución_parcial

            finsi

        finsi

    finsi

finpara
```

para i = 1 hasta n , hacer

si más_próximo_en_B [aux , i] > 0 , entonces

fila_solución_parcial ⟵ fila_solución_parcial + 1

Creo una fila de la matriz “solución_parcial”

solución_parcial.nodop [fila_solución_parcial , 1] ⟵ aux

solución_parcial.nodoq [fila_solución_parcial , 1] ⟵ i

solución_parcial.valor [fila_solución_parcial , 1] ⟵ min

finsi

finpara

Al finalizar el proceso de almacenar información en la matriz “solución_parcial” (si es que se llega a almacenar algo) en la variable “fila_solución_parcial” tenemos el total de filas nuevas que después tendremos que añadir a la matriz “aristas_de_la_solución”

total_filas_nuevas ⟵ fila_solución_parcial

si total_filas_nuevas > 0 , entonces

(Se han producido varias posibles soluciones)

tratar_varias_soluciones

finsi

Procedimiento **"almacenar_en_solución_pacial"**

```
para  i = 1  ,  hasta  n  ,  hacer

    si  más_próximo_en_B [ j , i ]  >  0  ,  entonces

        fila_solución_parcial  ←——  fila_solución_parcial  +  1

        Creo una fila de la matriz "solución_parcial"

        solución_parcial.nodop [ fila_solución_parcial , 1 ]  ←—  j


        solución_parcial.nodoq [ fila_solución_parcial , 1 ]  ←—
        ←——  más_próximo_en_B [ j , i ]


        solución_parcial.valor [ fila_solución_parcial , 1 ]  ←—  min

    finsi

finpara
```

Procedimiento **"tratar_varias_soluciones"**

Creamos tantas filas nuevas en la matriz "aristas_de_la_solución" como aristas con el mismo valor menos una

crear_filas_nuevas

si columna > 1 , entonces

Si en la fila activa se ha cargado más de una arista, hay que copiar todas las aristas menos la última de la fila activa al resto de filas recién creadas, en las mismas posiciones de columna

cargar_aristas_anteriores

finsi

Ponemos en cada fila recién creada y en la posición indicada por la variable "columna" una arista distinta con el mismo dato "valor"
(número total de filas recién creadas = número total de aristas con el mismo valor – 1)

cargar_aristas_mismo_valor

Procedimiento **"crear_filas_nuevas"**

"total_filas_nuevas" nos dice el número de soluciones producidas menos una, ya que la arista solución que incluye el nodo "aux" no está incluida

Crear un número de filas nuevas después de la última fila en la matriz "aristas_de_la_solución" igual al valor de la variable "total_filas_nuevas"

Poner en el campo "valor" de todas las aristas de todas las filas nuevas el valor – 1.

total_filas_aristas_de_la_solución ⟵ total_filas_aristas_de_la_solución +
+ total_filas_nuevas

Procedimiento **"cargar_aristas_anteriores"**

para i = (total_filas_aristas_de_la_solución) – (total_filas_nuevas) + 1

hasta

(total_filas_aristas_de_la_solución) , hacer

para j = 1 hasta columna – 1 , hacer

aristas_de_la_solución [i , j] ← aristas_de_la_solución [fila , j]

finpara

finpara

Para comprender mejor este procedimiento, voy a hacer hincapié en un detalle significativo:

En el bucle *"para"* de la variable *"i"* (*"i"* representa los valores de las filas recién creadas), si queremos empezar por la primera fila recién creada, el primer valor de *"i"* será:

- El número de filas recién creadas es :

  total_filas_nuevas

- Tenemos que restar del total de filas de la matriz *"aristas_de_la_solución"* el valor anterior y después sumar uno para empezar por la primera fila recién creada.

Luego la primera fila recién creada tendrá un valor de:

(total_filas_aristas_de_la_solución) – (total_filas_nuevas) + 1

y la última fila recién creada tendrá, lógicamente, el valor de *"total_filas_aristas_de_la_solución"*.

El bucle *"para"* de la variable *"j"* representa el número de aristas que hay que copiar desde la fila activa hacia las filas recién creadas a partir de la columna 1.

Como ya hemos almacenado en la fila activa la arista de la solución que incide en el nodo contenido en la variable *"aux"*, hemos sumado 1 a la variable *"columna"* y lo que queremos es trasladar todas las aristas anteriores a la última que se ha añadido, es decir, desde 1 hasta *(columna – 1)*.

Procedimiento **"cargar_aristas_mismo_valor"**

fil ← 0

para i = (total_filas_aristas_de_la_solución) – (total_filas_nuevas) + 1
hasta
(total_filas_aristas_de_la_solución)
hacer

fil ← fil + 1

aristas_de_la_solución [i , columna] ← solución_parcial [fil , 1]

finpara

Eliminar la matriz "solución_parcial" para ahorrar espacio en la memoria.

Procedimiento **"actualizar_mínimos"**

Al añadir un nodo más a *B* contenido en la variable *"aux"*, (ahora el nodo contenido en *"aux"* $\in B$) tenemos que hacer lo siguiente:

$\forall\ j \notin B$, actualizo o calculo (según los casos) *"valor_min [j] "* y *"más_próximo_en_B [j]"*.

Para ver todos los casos posibles, lo mejor es seguir el organigrama gráfico hecho anteriormente para este procedimiento.

El bucle empieza desde el nodo 2 ya que el nodo 1 esta en *B* desde el principio y nunca se va a considerar.

```
para  j = 2  hasta  n  ,  hacer
    si  para_nodos_fuera_de_B [ j ]  ,  entonces
        si  con_aristas_hacia_B [ j ]  ,  entonces
                proceso1
        sino
                proceso2
        finsi
    finsi
finpara
```

Procedimiento **"proceso1"**

```
si  valor_arista [ j , aux ]  ≥  0  ,  entonces
    si  valor_arista [ j , aux ]  ≤  valor_min [ j ]  ,  entonces
        valor_min [ j ]  ←  valor_arista [ j , aux ]
        para  k = 1  hasta  n  ,  hacer
            más_proximo_en_B [ j , k ]  ←  - 1
        finpara
        para  i = 1  hasta  n  ,  hacer
            si  no  para_nodos_fuera_de_B [ i ]  ,  entonces
                si  valor_arista [ j , i ]  =  valor_min [ j ]  ,  entonces
                    más_próximo_en_B [ j , i ]  ←  i
                finsi
            finsi
        finpara
    finsi
finsi
```

Procedimiento **"proceso2"**

si valor_arista [j , aux] ≥ 0 , entonces

valor_min [j] ⟵ valor_arista [j , aux]

Sólo existe una arista entre el nodo "j" y cualquier nodo de B, y esa arista incide precisamente en los nodos "j" y "aux"

más_próximo_en_B [j , aux] ⟵ aux

con_aristas_hacia_B [j] ⟵ True

finsi

Procedimiento **"fila_completada"**

Actualizo "número_de_soluciones" y obtengo en la variable "repetida" el valor booleano que me indica si las aristas de la solución de la fila activa son las mismas aristas de otra fila ya existente

comprobar_repetida

si repetida , entonces

Eliminar la fila activa "fila" de la matriz "aristas_de_la_solución".

total_filas_aristas_de_la_solución ← total_filas_aristas_de_la_solución –1

si total_filas_aristas_de_la_solución < fila , entonces

(No existen filas por debajo de la fila activa recién eliminada)

No hay que ocupar el hueco de la fila activa recién suprimida

(En este momento total_filas_aristas_de_la_solución = número_de_soluciones)

sino

(Si existen filas por debajo de la fila activa recién eliminada)

Si hay que ocupar el hueco de la fila activa recién suprimida, desplazando una posición las filas que están debajo de ella.

la variable "fila" no se modifica porque la nueva fila activa quedará en la misma posición que la fila activa eliminada

finsi

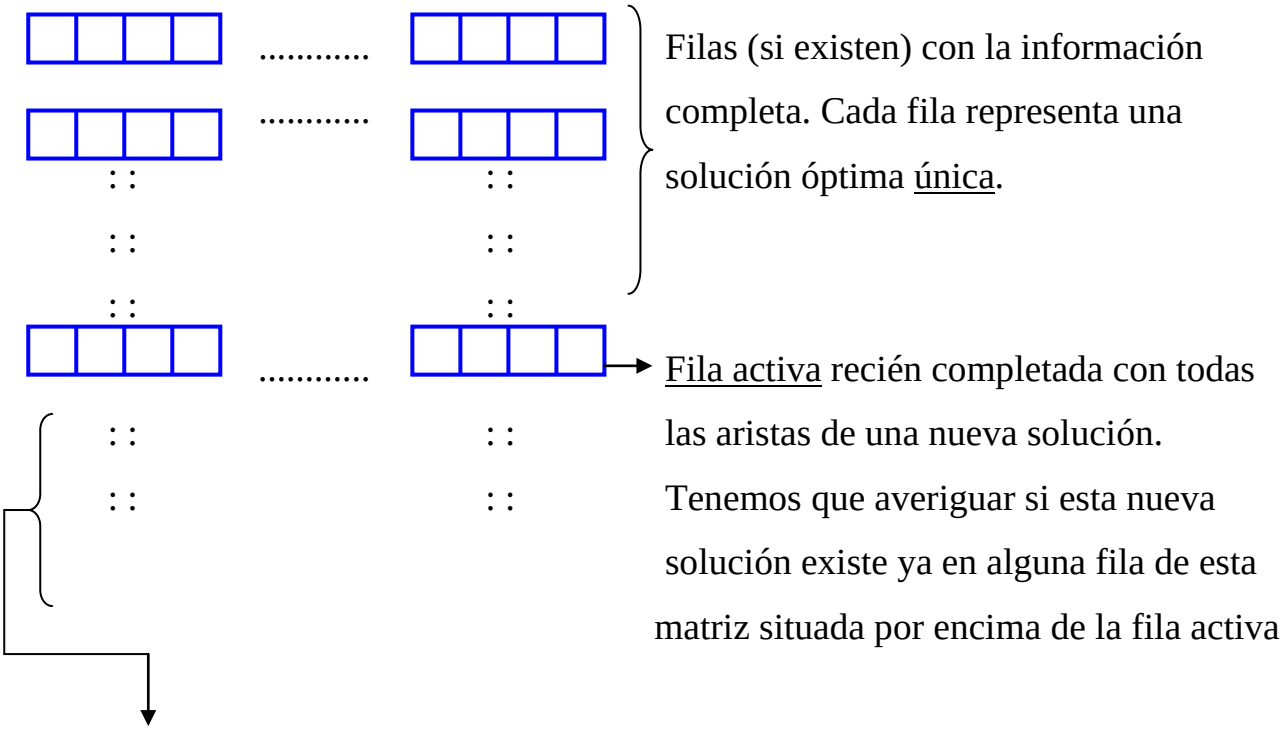
finsi

```
si  total_filas_aristas_de_la_solución = número_de_soluciones  ,  entonces
    si  número_de_soluciones = 1  ,  entonces
        Sacar por pantalla la única solución
    sino
        Sacar por pantalla todas las soluciones
    finsi
    fin_del_programa ⟵ True
sino
    si  no  repetida  ,  entonces
    (Es seguro que existe una fila debajo de la fila activa ya que
    "total_filas_aristas_de_la_solución" > "número_de_soluciones")
        (Si la solución no está repetida, paso a la siguiente fila activa)
        fila ⟵ fila + 1
    finsi
    continuar_siguiente_fila
finsi
```

Explicación del procedimiento "comprobar_repetida"

Tenemos que mirar si la fila activa recién completada con una solución óptima está repetida, es decir, si anteriormente hemos creado otra fila con las mismas *(n – 1)* aristas solución.

Para un caso general, la matriz *"aristas_de_la_solución"* mostrará este aspecto:



Si existen filas con soluciones únicas ya creadas, corresponderán a filas que están situadas en la matriz antes que la fila activa.

La variable *"fila"* nos indica el valor de la fila activa.

El procedimiento *"comprobar_repetida"* nos devuelve una variable booleana *"repetida"* que nos dice si la nueva solución óptima está repetida o no:

repetida → True ⟹ Sí está repetida.

repetida → False ⟹ No está repetida.

Procedimiento **"comprobar_repetida"**

Lo primero que tenemos que saber es si existe alguna solución única ya creada, por lo que este procedimiento se divide en dos partes principales.

si número_de_soluciones = 0 , entonces

(Es la primera solución que se crea)

repetida ⟵ False

número_de_soluciones ⟵ número_de_soluciones + 1

sino

(Ya existe alguna solución única creada)

La función "fila_repetida" nos dice si la fila actual recién completada contiene una solución única o no

repetida ⟵ fila_repetida

finsi

Función **"fila_repetida"**

Condiciones teóricas que nos ayudarán a entender el seguimiento de esta función:

- Con que la fila activa esté repetida en tan sólo una fila cualquiera se devuelve la función *"fila_repetida = True"*.
  Luego cuando se localice la primera fila con las mismas aristas solución que están en la fila activa, terminamos la ejecución de la función con valor *"True"*.
  Esto sucede cuando todas las aristas *"k"* están en alguna columna *"j"* de la fila *"i"* que estamos comparando (ver pseudocódigo).

- Si concluida la inspección de todas las filas anteriores en la matriz a la fila activa no hemos encontrado ninguna fila con las mismas aristas que la fila activa, se termina la ejecución de la función con valor *"False"*.

- Si una arista *"k"* de la fila activa no está en toda la fila completa con la que estamos comparando, las filas son distintas y pasamos a comparar con la siguiente fila.

En el caso en que la fila activa que se está tratando no esté repetida, hay que actualizar la variable *"número_de_soluciones"* sumándola uno.

El pseudocódigo de esta función se muestra a continuación:

para i = 1 hasta (fila – 1) , hacer

para k = 1 hasta (n – 1) , hacer

encontrado ⟵ False

para j = 1 hasta (n – 1) , hacer

si ((aristas_de_la_solución.nodop [fila , k] = aristas_de_la_solución.nodop [i , j]) y (aristas_de_la_solución.nodoq [fila , k] = aristas_de_la_solución.nodoq [i , j]) o (aristas_de_la_solución.nodop [fila , k] = aristas_de_la_solución.nodoq [i , j]) y (aristas_de_la_solución.nodoq [fila , k] = aristas_de_la_solución.nodop [i , j])) , entonces

encontrado ⟵ True

salir del para más interno, si (k = n – 1) la fila está repetida y si (k < n – 1) se pasa al siguiente valor de k

finsi

finpara

si no encontrado , entonces

Si una arista k no se ha encontrado en toda la fila j pasamos a la siguiente fila i porque la fila con la que comparamos la fila actual y dicha fila actual son distintas

salir del segundo para interno (el que tiene la variable k)
para seleccionar otra fila i

finsi

si (k = n – 1) , entonces

Si se han encontrado todas las aristas k en la fila con la que estamos comparando, la fila activa está repetida

fila_repetida ⟵ True

salir de la función

finsi

finpara

finpara

Si no se ha podido llegar nunca al valor (k = n – 1) significa que nunca se ha podido encontrar una fila repetida

fila_repetida ⟵ False

número_de_soluciones ⟵ número_de_soluciones + 1

Procedimiento **"continuar_siguiente_fila"**

En este punto, ya tenemos actualizadas, entre otras, las variables "fila" y "número_de_soluciones"

columna ← 0

En cualquier fila de la matriz "aristas_de_la_solución", la primera arista ("columna" = 1) siempre va a tener el nodo inicial (en nuestro caso el nodo 1), luego para todas las filas sin excepción se inicializará siempre a partir del mismo nodo inicial

nodo_inicial

mientras (columna < (n – 1) y aristas_de_la_solución.valor [fila , columna + 1] ≠ - 1) , hacer

columna ← columna + 1

nodop ← aristas_de_la_solución.nodop [fila , columna]

nodoq ← aristas_de_la_solución.nodoq [fila , columna]

si para_nodos_fuera_de_B [nodop] , entonces

aux ← nodop

sino

aux ← nodoq

finsi

para_nodos_fuera_de_B [aux] ⟵ False

La variable "número_nodos_en_B" se volvió a inicializar a uno en el procedimiento "nodo_inicial"

número_nodos_en_B ⟵ número_nodos_en_B + 1

actualizar_mínimos

finmientras

Al terminar este procedimiento, se volvería otra vez al bucle *"mientras"* del programa principal y lo que suceda a continuación va a depender del valor con el que halla terminado la variable *"columna"* ya que:

columna
- = (n – 1) ⟹ número_nodos_en_B = n , es decir , la fila activa ha sido completada totalmente.
- < (n – 1) ⟹ número_nodos_en_B < n , es decir, la fila activa no tiene todavía todas las aristas solución.

Con lo cual, en el programa principal, si *"número_nodos_en_B = n"* se vuelve otra vez al procedimiento *"fila_completada"* y si *"número_nodos_en_B < n"* seguimos completando con aristas solución, según el algoritmo de Prim, la nueva solución que estará en la fila actual.

2.3- Optimización de la aplicación para los algoritmos que calculan todas las soluciones

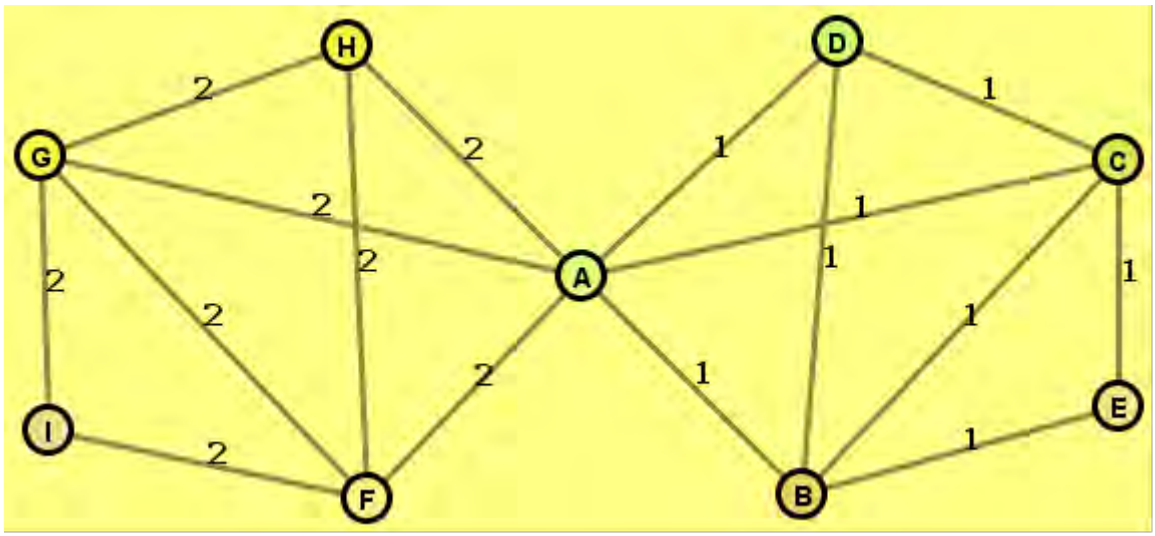
Una vez finalizada la codificación de la aplicación y la posterior depuración, es decir, el proceso de prueba del funcionamiento de la aplicación, localización de comportamientos anómalos, imprevistos y no deseados, la búsqueda de sus causas (errores en la concepción del algoritmo o en su codificación) y omisión de las mismas para la resolución del problema, se tiene una aplicación estrictamente correcta y completamente funcional.

Si bien, llegado a este punto, se puede introducir una serie de mejoras que optimicen el rendimiento para una mayor **eficiencia** en la utilización de los recursos del sistema, economizando tiempo y memoria necesarios para la resolución de los algoritmos dados ciertos grafos.

Asimismo, estas mejoras pueden aumentar la facilidad de utilización de la aplicación por parte del usuario, como se explicará después.

Descripción de los problemas que se pueden presentar

La aplicación puede tardar hasta varios minutos en la resolución de un algoritmo automático sobre un grafo con uno o varios conjuntos de gran cantidad de aristas con el mismo valor (como el que se muestra en la siguiente figura), dado que el número de posibles soluciones se multiplica en tales casos y además hay que tener en cuenta las soluciones repetidas, es decir, localizarlas y descartarlas.



Durante ese tiempo, la aplicación queda bloqueada, no pudiendo realizarse ninguna acción sobre la misma ni cancelar la operación en curso, ni desbloquearla de ningún modo, siendo la única solución esperar a que termine el algoritmo (sin ninguna información acerca de cuánto tiempo puede tardar) o reiniciar la aplicación, perdiendo el grafo sobre el que se estuviese trabajando. Además, el rendimiento del sistema decae mucho, pues los algoritmos requieren mucha potencia de cálculo, ralentizando notablemente cualquier otra aplicación que se esté ejecutando al mismo tiempo.

Búsqueda del método de optimizar la aplicación

Utilizando el buscador "google", busqué información en Internet acerca de la existencia de algún tipo de herramienta de optimización para aplicaciones Java, que me consta que existen para aplicaciones escritas en otros lenguajes: los *profilers*. Estas herramientas generan información estadística sobre el tiempo empleado en la ejecución de cada una de las partes de una aplicación, de manera que resulta relativamente sencillo optimizar el rendimiento de la misma centrando los esfuerzos de análisis de mejora del algoritmo tan sólo en las secciones que resulten críticas para su comportamiento.

Investigación de la opción de optimización de código

Encontré documentación en diversas páginas web que indicaba que el propio intérprete de java, "java.exe", puede generar esta información de optimización tan sólo utilizando la opción "-prof" como uno de sus parámetros, produciendo un archivo denominado "java.prof" en el directorio de trabajo. Esta opción no es válida, en cambio, para el visor de aplicativos "appletviewer.exe", que es el único programa que puede ejecutar applets de Java ("java.exe" no puede) junto con los navegadores web pero, a diferencia de estos últimos, el appletviewer además puede recibir parámetros en la línea de comandos para seleccionar ciertas opciones; parámetros entre los que no se encontraba "-prof". Sin embargo, descubrí que es posible pasar al appletviewer los mismos parámetros que al intérprete si se preceden de la opción "-J". De esta manera, ejecuté en línea de comandos la llamada al intérprete de esta forma: "appletviewer -J-prof Proyecto.htm". Para cada ejecución, hacía un grafo de prueba y ejecutaba uno de los algoritmos automáticos, **generando un archivo de información de rendimiento para cada algoritmo y tipo de grafo**.

Análisis de los resultados

Una vez generados los archivos de depuración, encontré que era demasiado costoso analizar su contenido, pues a pesar de que mostraba una estructura muy bien definida, con el formato "cantidad de llamadas", "función invocada", "función invocadora" y "tiempo total empleado", esta información, concerniente a cada método invocado, se mostraba para **todos** los métodos de la aplicación, incluso de otras clases definidas en la misma y de clases estándar de las que se heredaba o que se utilizaban en algún objeto. El resultado era un listado enorme, de varios megabytes de extensión, que daba esta información útil de los métodos de los algoritmos automáticos mezclada con mucha más información inútil para mis propósitos sobre métodos pertenecientes a clases del lenguaje y otras específicas de mi aplicación que

nada tenían que ver con los algoritmos automáticos. Me encontraba en la necesidad de **filtrar** esa información.

Aprovechando que la información de cada llamada a un método "*B*" desde otro "*A*" (sin importar cuántas veces "*A*" llame a "*B*") se encuentra en una única línea del archivo "java.prof", y que los métodos se identifican porque aparece no sólo el nombre de cada uno, sino también el nombre de las clases a las que pertenecen, utilicé una herramienta de filtrado, "grep.exe" (propia de sistemas UNIX/Linux), que toma un archivo de texto como entrada y genera una salida en la que sólo figuran las líneas del archivo de entrada que contengan un determinado patrón (un conjunto de caracteres, una palabra, ...); en mi caso, el archivo era "java.prof", generado por el appletviewer, y el patrón era todo o parte de el nombre de la clase cuyos métodos quería analizar ("KruskalAutom" o "PrimAutom"). Dado que la salida se vuelca por pantalla, sólo queda redireccionarla a un archivo para que se vuelque en el mismo con el carácter especial " > " , a cuya continuación se pondrá el nombre que tendrá el archivo que contendrá la salida del filtro "grep.exe"; ese archivo contendría la información anteriormente citada (cantidad de llamadas, función invocada, función invocadora y tiempo total empleado en ejecutar todas las llamadas, en milisegundos), pero tan sólo acerca de los métodos pertenecientes a las clases que nos interesa analizar. Por tanto, la llamada al comando de sistema "grep" para obtener uno de estos archivos con información específica sería, por ejemplo, "grep KruskalAutom java.pro > Kruskal.pro" para generar un archivo llamado "Kruskal.pro" con información sobre los métodos del algoritmo automático de Kruskal, y "grep PrimAutom java.pro > Prim.pro" para hacer lo propio con métodos específicos de la clase del algoritmo automático de Prim.

Optimizaciones en los métodos más costosos encontrados

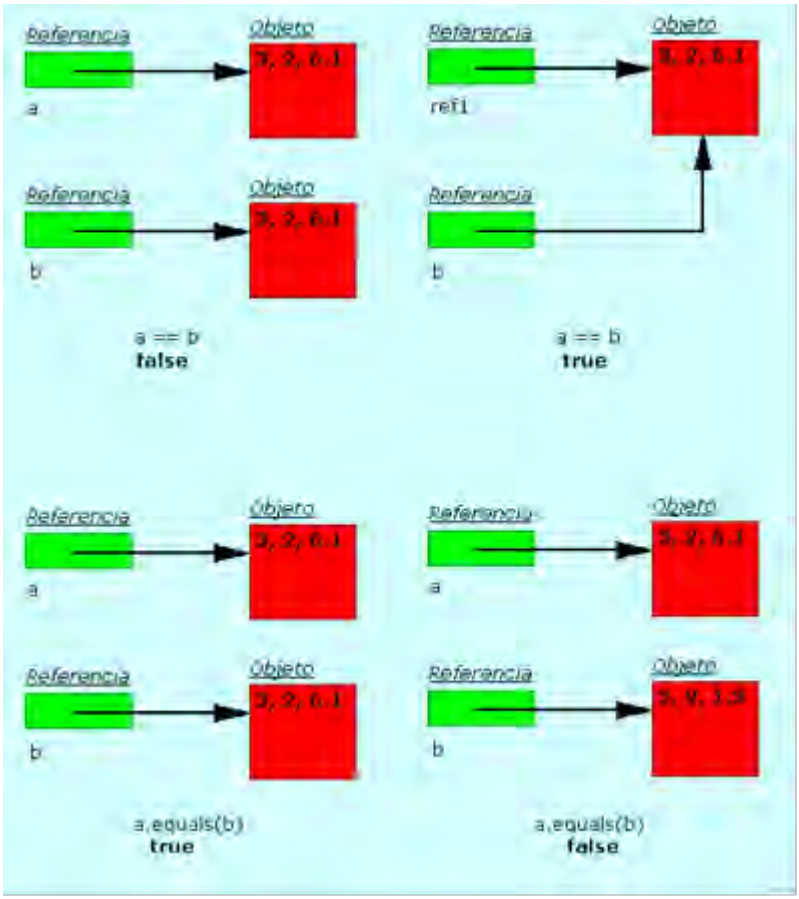
"KruskalAutomático": en esta clase, los que más tiempo llevaban eran *"BuscaComponenteConexa(...)"* cuando era invocado desde *"ComponenteNodo(...)"*, *"CargarCombinación(...)"* y *"LanzarSolución(...)"*; el método *"iguales(...)"* desde *"Solución(...)"* y *"GenerarSoluciones(...)"*; el método *"ComponenteNodo(...)"* cuando era invocado desde *"CargarCombinación(...)"* y los métodos *"Inicio(...)"* y *"Fin(...)"* al ser invocados desde *"UnirComponentesConexas(...)"*.

Comprobé que estas funciones resultaban de complejidad algorítmica irreductible, a pesar de lo cual inserté algunas pequeñas optimizaciones prácticas que no resultaron en una mejora apreciable, por lo que las deshice; sólo prevalece el precálculo, en *"BuscaComponenteConexa(...)"*, de el límite superior del bucle en una variable intermedia, lo que resulta ligeramente más rápido que invocar a la función que devuelve la cifra que se usa como límite (el número de nodos del grafo) en cada iteración.

Otras optimizaciones de esta clase se centran en reducir el consumo de memoria, y se tratarán más adelante.

"PrimAutomático": en esta clase, el método más costoso en tiempo era *"equals(...)"*, perteneciente a la clase *"DatosAristaPrim"* e invocado desde el método de *"PrimAutomático"* llamado *"FilaRepetida(...)"*. Este método devolvía "*true"* si el objeto sobre el que se invocaba y el objeto que se le pasaba como parámetro, también de tipo *"DatosAristaPrim"*, tenían variables de instancia (campos o atributos del objeto) con valores iguales. De esta manera, para comparar un objeto llamado *"a"* con otro llamado *"b"*, el código sería de la forma *"if(a.equals(b))"*, y no *"if(a= =b)"*, porque el operador de comparación de igualdad *"=="* aplicado sobre referencias (nombres de objetos) devuelve *"true"* si hacen referencia al mismo objeto, y *"false"* si se refieren a objetos distintos, por más que los atributos de ambos tengan los mismos valores, que es lo que nos interesa comprobar. De ahí que,

típicamente, cuando se quiere dar a una clase la capacidad de que sus objetos se puedan comparar entre sí, se redefina el método *"equals(...)"* heredado de la clase base de toda clase en Java: *"Object"*.

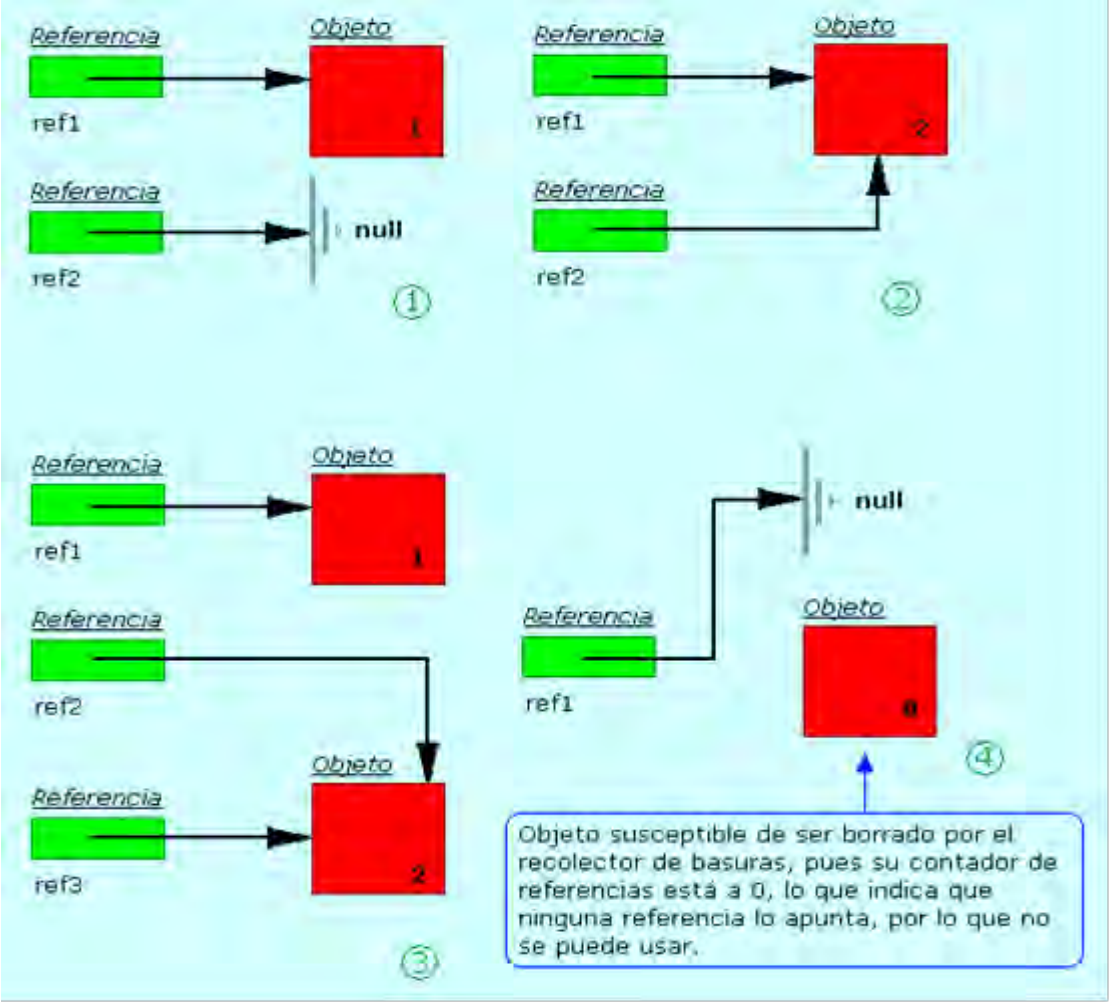


Lo que sucedía en este caso era que, a pesar de que era la manera correcta de comparar dos objetos de esta clase y esto funcionaba correctamente, la función *"equals(...)"* era la que se invocaba más veces (varios millones en un grafo con diez aristas iguales), cada una de las cuales tardaba casi dos microsegundos en ejecutarse (en un AMD K7 a 800 MHz), suponiendo un coste total de varios segundos.

Descubrí que, integrando la parte principal de la función *"equals(...)"* en el único lugar en el que antes era invocada (el interior de una condición de la función *"FilaRepetida(...)"*), la ejecución de ese código era casi instantánea (pues aumentaba

de manera apenas perceptible el coste de *"FilaRepetida(...)"*), y el algoritmo ganaba en velocidad.

Otras optimizaciones del tiempo de ejecución de este algoritmo pasaban por la gestión de memoria. Después de utilizar objetos o matrices de objetos, se reduce la memoria consumida si se libera. La liberación de memoria en Java no es llevada a cabo por los programas (al menos de manera explícita), sino por una herramienta del lenguaje denominada *recolector de basuras*, que puede ejecutarse mediante la llamada a un método (en la forma *"Runtime.getRuntime().gc();"*), de manera que se libera la memoria ocupada por los objetos que ya no están siendo utilizados. Se sabe que un objeto no está siendo utilizado porque todo objeto tiene un contador interno de referencias; cuando se crea un objeto y se apunta con una referencia ("*ref1 = new Clase();"*), el contador de referencias del nuevo objeto se pone a uno, y cada vez que se apunta a un objeto con una nueva referencia (*"ref2 = ref1;"*), se incrementa este contador. Si una referencia se redirecciona (a otro objeto o a *"null"*, como por ejemplo *"ref1 = ref3;"*), el contador de referencias se decrementa; siempre que valga uno o más, el objeto podrá ser aún utilizado (en nuestro caso, *"ref2.método();"*, por ejemplo), pero si se queda a cero (*"ref2 = null;"*), ya no podrá ser utilizado desde ninguna referencia (*"**?**.método();"*) ni se podrá volver a incrementar su contador de referencias ("*ref1 = **?**;"*), luego el objeto queda totalmente inservible, y se considera por tanto un objeto no utilizado susceptible de ser destruido para liberar la memoria que ocupa mediante el recolector de basuras.



El recolector de basuras, no obstante, está implementado mediante un algoritmo muy costoso en términos de tiempo, por lo que no conviene liberar la memoria en métodos invocados con mucha frecuencia, o al menos no conviene liberarla siempre en estos métodos, sino tan sólo en las ocasiones en las que se cumplan ciertas condiciones (como cantidad de memoria no utilizable en cada momento); ésta es la mejora que se ha realizado en las funciones que se describen a continuación:

"ComprobarVariasSoluciones(...)": crea la matriz *"soluciónParcial"*; después de utilizarla, la destruía borrando mediante un bucle las referencias a los objetos de tipo *"DatosAristaPrim"* para que, a continuación, la llamada al recolector de basuras de Java (*"Runtime.getRuntime().gc();"*) liberase la memoria ocupada por esos objetos de la matriz no utilizados y de otros objetos no usados que pudiesen estar ocupándola

aún. Quitando el bucle que borra las referencias de la matriz y la llamada al recolector de basuras se ahorra mucho tiempo, mientras que el gasto de memoria es asumible y, aunque se incrementa cada vez que este método es invocado, se libera cuando se cumplen las condiciones comprobadas en otros métodos (a continuación).

"CrearFilasNuevas(...)": amplía la matriz bidimensional *"aristasSolución"*, añadiéndole filas nuevas. Para ello, se tiene que crear una matriz (llamada *"aristaAux"*) con la misma cantidad de columnas pero con más filas, y después copiar los elementos de la matriz menor a la mayor. Una vez hecho esto, se ha de crear los elementos de las filas nuevas, que serán inicializados más tarde; por último, hay que perder las referencias a los objetos de la matriz pequeña (*"aristasSolución"*), sobreescribiendo cada una con *"null"* en un bucle, para que la memoria ocupada por estos objetos sea liberada al llamar al recolector de basuras, y redireccionar la referencia *"aristasSolución"* de toda la matriz a la matriz nueva y más grande (*"aristaAux"*).

Esto se puede atajar haciendo que las filas de la nueva matriz, que a su vez son referencias a objetos de tipo *"array de DatosAristaPrim"* (ya que cada fila apunta a una matriz con sus columnas), apunten a las filas de la matriz que ya existe, en lugar de crearle objetos nuevos y copiar el contenido de los que ya existen para luego borrarlos, por lo que se ahorra el tiempo de la creación de nuevos objetos, de la copia en estos del contenido de objetos ya existentes, de la redirección de las referencias de la matriz a estos objetos y de la recolección de memoria.

"FilaCompletada(...)": lo mismo que sucede en *"CrearFilasNuevas(...)"*.

La recolección de basura se controla acumulando, en* "CrearFilasNuevas(...)", *la cantidad de filas creadas cada vez en la variable* "numFilas", *mediante la sentencia* "numFilas+=totalFilasNuevas;". *La diferencia entre el número total de filas en memoria (*"numFilas"*) y la cantidad de las mismas utilizada para albergar

soluciones (*“número Soluciones”*) es la cantidad de filas de objetos que ya no están siendo usados. Cuando esa cantidad alcanza un determinado valor fijado en la práctica mediante pruebas para un óptimo equilibrio entre consumo de memoria y consumo de tiempo (en la aplicación, 400), el método “RecolectarMemoria(...)”, que es invocado en cada iteración del bucle principal del algoritmo, llama al recolecor de basuras, y si no se supera este valor no lo llama.

Optimizaciones de consumo de memoria

Consisten en un rediseño de la interacción de ciertos métodos del algoritmo de Kruskal Automático para minimizar la cantidad de memoria consumida y reducir al mínimo la cantidad de casos en los que el algoritmo no se podía ejecutar por falta de memoria.

Esto sucedía porque, cada vez que se encontraba un conjunto de aristas con el mismo valor que no estuviesen en todas las soluciones pero que sí estuviesen en algunas, se tenía que generar una matriz bidimensional de tipo *“DatosArista”*, llamada *“soluciónParcial”*, cuyo número de columnas era igual al número de aristas con el mismo valor, y cuyo número de filas era igual al factorial de esa cifra; así, para diez aristas con el mismo valor que no estén en todas las soluciones y que sí estén en alguna, la cantidad de elementos a reservar sería 10! x 10, y cada uno de ellos es un objeto de más de diez bytes, luego la cantidad de memoria requerida para la matriz superaría los 300 MegaBytes, y el tiempo necesario para calcular 10! soluciones sería enorme, por no contar con la cantidad de elementos de cada par de filas que habría que recorrer para marcar una fila como solución repetida o dejarla como solución nueva.

De modo que decidí crear cada fila nueva sólo en el momento en el que se demostrase que esa solución no era repetida, en lugar de crear todas las filas previamente para luego calcular las soluciones y, por último, eliminar aquéllas que estuviesen repetidas. De este modo, se ahorraba mucha memoria para cantidades

elevadas de aristas con el mismo valor, y se evitaba que se produjesen errores de memoria. Aún así, si la cantidad de aristas con el mismo valor forma una cantidad de soluciones no repetidas tal que sea demasiado grande para ser albergada en memoria, la circunstancia se maneja como una excepción, que se lanza y captura en el lugar adecuado, de manera que se le muestra al usuario el mensaje de error oportuno, permitiendo que la aplicación siga ejecutándose normalmente, de forma que se puede editar el grafo para quitarle aristas o nodos, por ejemplo, y volver a intentar calcular todas sus soluciones de manera automática por Prim. Es, no obstante, necesario un grafo realmente enorme con una cantidad elevadísima de ciclos con aristas del mismo valor para generar una cantidad de soluciones tal que no quepan en la memoria, pero de suceder se maneja bien y no sucede a causa del diseño del algoritmo, sino de las limitaciones físicas de la máquina sobre la que se ejecute, por lo que de esta manera se da por válido.

Los métodos que ha habido que modificar han sido *"AristasVariasSoluciones(...)"* y *"CargarAristasMismoValor(...)"* ; *"CargarCombinación(...)"* ha cambiado y, además, ahora se llama *"CargarCombinaciónVálida(...)"*, se ha eliminado *"factorial(...)"* y *"MarcarSolucionesRepetidas(...)"* y se ha añadido *"AmpliarSoluciónParcial(...)"*.

Optimización del programa a nivel de usuario

Ahora trataré las mejoras del programa en cuanto a la capacidad del interfaz gráfico del mismo de ser utilizado de manera intuitiva y agradable por el usuario.

Por más que se haya optimizado la aplicación para que realice los cálculos de todas las soluciones de los algoritmos automáticos de la forma más eficiente posible tanto en términos de consumo de memoria como de tiempo, con grafos con un número de soluciones posibles relativamente elevado el tiempo de cálculo puede ser de bastantes segundos o incluso de varios minutos para grafos suficientemente complejos.

Tal y como estaba estructurada la aplicación anteriormente, durante el tiempo en el que se calculaban las soluciones de un grafo mediante un algoritmo automático no se podía interactuar con la aplicación de ningún modo; esto no tenía importancia para grafos con pocas soluciones, pues el tiempo en el que la aplicación no podía interactuar, o dicho de otro modo, quedaba *bloqueada*, era de unos pocos segundos, por lo que tal hecho pasaba inadvertido al usuario o, de no ser así, no resultaba particularmente molesto para la utilización de la aplicación; en cambio, si resultaba mucho más molesta de utilizar cuando ese bloqueo se prolongaba durante minutos, lo que sucedía con grafos complejos.

De esta forma, decidí modificar las clases de los algoritmos automáticos para que pueda ejecutarse *concurrentemente* (es decir: en paralelo, al mismo tiempo) con el resto de la aplicación, de manera que sea posible ejecutar, por ejemplo, el código de repintado de la ventana (necesario si la aplicación se destapa después de haber ocultado toda la ventana o una parte) o el código de gestión de la pulsación de ciertos botones, mientras se realiza el cálculo de las soluciones; uno de los botones que puede pulsarse es esencial, pues *mata* al proceso concurrente del cálculo de soluciones que corre en segundo plano, de manera que se puede volver al modo de edición para cambiar la estructura del grafo; es decir: no sólo se evita que la aplicación quede bloqueada, sino que se puede cancelar el cálculo si el usuario lo encuentra demasiado largo.

Por otra parte, el hecho de que la aplicación no quede bloqueada permite que el proceso de cálculo de soluciones que se ejecuta en segundo plano pueda pasar al proceso de la aplicación principal un mensaje de texto y, cada vez que el proceso secundario *duerma*, cediendo el control al principal, éste muestre ese mensaje por pantalla en la barra de estado; esto permite que la aplicación en segundo plano muestre, en todo momento, la cantidad de soluciones halladas, y cambios en general en el mensaje para que el usuario intuya que sucede algo en segundo plano y, por tanto, que la aplicación no está *ociosa* ni *colgada*. Esto redunda en un aumento de la usabilidad de la aplicación, pues el usuario puede estar entretenido contemplando los

avances de los cálculos mediante el mensaje, ver lo que ha avanzado el cálculo y hacerse una idea de lo que puede tardar aún.

Para ello, se ha hecho que las clases *"KruskalAutomático"* y *"PrimAutomático"* hereden de la clase *"Thread"*, lo que en Java permite gestionar la multitarea de los objetos de estas clases de manera automática. Lo único que se hace es, en el momento en el que se pulsa el botón para realizar el cálculo, crear el objeto de la clase adecuadda e invocar al método *"start(...)"*, heredado de *"Thread"*. A continuación, se pone un objeto global, *"controladorAlgoritmos"* (cuya clase también hereda de *"Thread"*) en un estado determinado, que le indica que invoque al método *"Espera(...)"* del objeto algoritmo apropiado. Como el método principal del objeto algoritmo (*"run(...)"*, heredado de *"Thread"*), que es el invocado por *"start(...)"*, y el método *"Espera(...)"* son ambos métodos de tipo *"synchronized"* del mismo objeto, son mutuamente exclusivos, es decir, no pueden ejecutarse concurrentemente sobre el mismo objeto, luego el proceso del objeto *"controladorAlgoritmos"*, que llama a un método *sincronizado* del objeto algoritmo después de que otro método también *sincronizado* del mismo objeto, *"run(...)"*, ya haya sido llamado, queda bloqueado hasta que el método *"run(...)"* termine su ejecución y, por tanto, termine todo el algortimo. Así, tenemos un lugar en el que sabemos que el objeto algoritmo ya ha terminado su ejecución y puede ser destruído: detrás de la llamada a *"Espera(...)"* en el método principal, *"run(...)"*, del objeto *"controladorAlgoritmos"*. Sólo así se puede borrar la memoria ocupada por el objeto algoritmo, que ya no se utiliza más, y restaurar el estado de la aplicación al modo de edición.

2.4- Método para resolver excesivas soluciones óptimas

El algoritmo de Kruskal trabaja con todo el grafo en su conjunto y selecciona de una vez todas las aristas con un mismo valor. Esto no sucede con el algoritmo de Prim que al no trabajar con todo el grafo a la vez (empieza desde un nodo y se desarrolla sobre una parte del grafo) no produce tantas soluciones de golpe al principio por lo que cabe esperar que sea más rápido en un primer momento, pero tiene el problema de no saber si una determinada solución sin todas sus aristas es única o no hasta que no se completan todas sus aristas solución, con lo cual, en la parte final del proceso se acumula más memoria y cabe esperar que sea más lento en esta parte final, ya que el algoritmo de Kruskal no tiene este problema.

Si la memoria es suficiente, el algoritmo de Prim es bastante más rápido porque el algoritmo de Kruskal tiene al principio un pico de cálculo excesivo.

El grafo también tiene mucha influencia en su resolución por un algoritmo u otro (situación de las aristas con el mismo valor) y también depende de cual sea el nodo de inicio en el algoritmo de Prim.

Soluciones propuestas

- Para evitar un excesivo número de soluciones, si la precisión de las aristas no es muy importante, se pueden poner decimales en los valores de las aristas para evitar la coincidencia entre dichos valores y así no tener un gran número de soluciones.
- Otra solución puede ser, aprovechando que el algoritmo de Prim no se desarrolla sobre el grafo en su totalidad, aplicarlo sobre una parte del grafo (grafo parcial) y cuando tengamos esa solución parcial, la guardamos y hacemos de dicha parte del grafo un solo nodo. Las aristas que salen de dicho nodo al resto del grafo son las mismas aristas que salen del grafo parcial

- hacia el resto del grafo. Siguiendo este proceso de resolver grafos parciales, al final, nos quedan las soluciones de dichos grafos parciales. Al unirlas, tenemos la solución a todo el grafo completo.

Bibliografía

Bibliografía de "Caminos y Circuitos Hamiltonianos y su Optimización":

- Autores: Sixto Rios, Alfonso Mateos, María Concepción Bielza, Antonio Jimenez (Año: 2004); Título: Investigación Operativa. Modelos determinísticos y estocásticos; Editorial: Centro de Estudios ramón Areces.

- Autor: Robert Faure (Año: 1975). Título: Elementos de Investigación Operativa. Editorial: ICE ediciones.

- Autor: Ángel Sarabia (Año: 2000). Título: La Investigación Operativa. Universidad Pontificia de Comillas (ICAI, ICADE).

- Autor: Juan José Salazar (Año: 2001). Título: Programación Matemática. Editorial: Díaz de Santos.

- Autor: Kenneth H. Roven (Año: 2009). Título: Matemática Discreta y sus aplicaciones. Editorial: Mc Graw Hill.

- Autores: Félix García, Gregorio Hernández y Antonio Nevot (Año: 2003). Título: Problemas resueltos de matemática discreta. Editorial: Thomson.

- Autor: Claude Berge (Año: 1970). Título: Graphs and Hypergraphs. Editorial: North – Holland Mathematical Library.

- Autores: Behzad, Chartrand y Lesniak-Foster (Año: 1988). Título: Graphs & Digraphs. Editorial: Prindle, Weber & Schmidt Internetional Series.

- Julio Suboez, Yair Caro, Roland Haggkvist, Klas Markstrom, Lai Chunhui (Año: 1979). Título: Graphs Theory and Related Topics. Editorial: J. A. Bondy U.S.R. Murty. Academic Press.

- Información en Internet:

Documentación de la cátedra de matemática discreta en la Facultad de Ciencias, Universidad de Zulia (Conectividad, Grafos Hamiltonianos y Grafos Planares):

http://mipagina.cantv.net/jhnieto1/c11.pdf

Nociones básicas sobre grafos:

http://lear.inforg.uniovi.es/ioperativa/TutorialGrafos/grafo/grafo.htm

Nociones básicas sobre grafos y algunos algoritmos sobre teoría de grafos:

http://www.fi.uba.ar/materias/6107/grafos_definiciones.pdf

Página 74 de la documentación (grafos hamiltonianos), Condiciones suficientes y resultados interesantes:

http://books.google.es/books?id=r8hvE1vgXHgC&pg=PA72&lpg=PA72&dq=circuito+hamiltoniano&source=web&ots=hWM132uPvU&sig=jOF3n9gpY2TyBfFfvnonKzZdIko&hl=es&sa=X&oi=book_result&resnum=1&ct=result#PPA74,M1

Bibliografía de "Árboles de recubrimiento mínimo":

- Autores: Sixto Rios, Alfonso Mateos, María Concepción Bielza, Antonio Jimenez (Año: 2004); Título: Investigación Operativa. Modelos determinísticos y estocásticos; Editorial: Centro de Estudios ramón Areces.

- Autores: G. Brassard, P. Bratley (Año: 1997) ; Título: Fundamentos de algoritmia; Editorial: Prentice Hall.

- Autor: Félix García (Año: 2005) ; Título: Matemática discreta. Editorial: Paraninfo
(especialmente indicado para definiciones)

- Autor: Robin J. Wilson (Año: 1983) ; Título: Introducción a la teoría de grafos; Editorial: Alianza Editorial S.A.

- Autores: M. Abellanas, D. Lodares (Año: 1991); Título: Análisis de algoritmos y teoría de grafos; Editorial: Ra-Ma.

- Autor: Juan José Salazar (Año: 2001) ; Título: Programación matemática.

Conclusiones

En mi opinión, se han cumplido bien los objetivos de este proyecto. Es muy importante que el proyecto no se pierda en el tiempo y no se han escatimado esfuerzos para ello porque tiene mucha utilidad y por ello se pueden dar soluciones a infinidad de modelos reales.

Hacer una parte gráfica completa, cómoda y de fácil manejo con una calidad de resolución alta en los cinco navegadores de Internet más importantes lleva mucho tiempo con los precarios medios de que se dispone en la actualidad pero creo que merece la pena porque una aplicación aunque sea buena si no se hace atractiva al usuario acaba por utilizarse poco.

El proyecto también tiene una parte formativa muy importante que puede ser accesible desde cualquier lugar (especialmente indicado a los alumnos con el nuevo plan de estudios Bolonia) al estar la aplicación puesta en Internet y en castellano e inglés. Si la aplicación puesta en el ordenador, desde mi punto de vista, es muy completa, la documentación aporta una continuidad en el tiempo del proyecto y una información muy valiosa y útil para los usuarios, ya sean alumnos o no. La resolución de los caminos y circuitos hamiltonianos es muy rápida y para grafos grandes con muchas soluciones, todos los procesos y situaciones excepcionales están bajo control.

Aparecen todas las soluciones desde cada nodo, todas las soluciones mínimas y todas las soluciones máximas también desde cada nodo y para cada circuito.

Lo que más tiempo me ha llevado es hacer el estudio y resolución de todos los árboles de recubrimiento mínimo utilizando como base los algoritmos de Kruskal y Prim. Para el cálculo de dichas soluciones se han optimizado los recursos del ordenador y la programación previo a un estudio exhaustivo realizando muchas pruebas. El usuario tiene el control de los tiempos de ejecución en todo momento y es informado de en que punto se encuentra el proceso. La eficacia del cálculo de todas las soluciones se puede contrastar a simple vista observando que el número de

soluciones calculado utilizando métodos tan distintos en su ejecución como los de Kruskal y Prim coincide siempre.

Los resultados de la aplicación de todos los algoritmos de este proyecto se pueden ver directamente sobre el grafo, o bien en paralelo, los arcos o aristas en una ventana de forma detallada con una total flexibilidad para seleccionar los resultados.

En definitiva, si bien es verdad que cualquier proyecto siempre es mejorable, desde mi punto de vista, se sobrepasa con creces la cantidad y la calidad que se requiere para un proyecto fin de carrera.

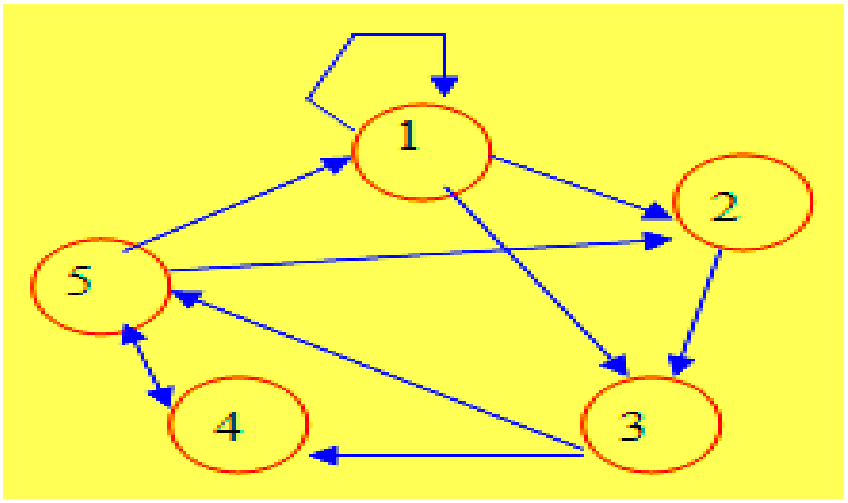
Apéndice A: Ejemplos de caminos y circuitos hamiltonianos calculando paso a paso las matrices intermedias

En este apéndice, se van a desarrollar distintos ejemplos de cómo pasar un grafo a una matriz y después calcular y mostrar las matrices intermedias hasta llegar a las matrices finales que nos muestran los caminos y circuitos hamiltonianos. Se verán métodos de simplificación (si el grafo tiene los suficientes nodos no hace falta calcular todas las matrices) y la multiplicación latina para cada elemento de las matrices. Además de ver desglosado todo el proceso de cálculo de la multiplicación latina de matrices elemento a elemento de cada matriz, estos ejemplos son muy importantes porque a través de ellos se ha comprobado después el buen funcionamiento de la aplicación informática. Es decir, los ejemplos que se ven aquí se han hecho a mano para después introducir los mismos grafos en la aplicación informática y comparar resultados, lo que representa un gran esfuerzo de trabajo manual.

En total son nueve ejemplos que nos dan una idea de los caminos y circuitos hamiltonianos que aparecen según sea el grafo y según el número de arcos que tenga. Se empieza con tres ejemplos de grafos de cinco nodos calculando todas las matrices intermedias (aunque sólo son necesarias $[M]^1$, $[M]^2$, $[M]^4$ y $[M]^5$). En el cuarto ejemplo si es obligatorio calcular todas las matrices porque el grafo sólo tiene tres nodos ($[M]^1$ para pasar el grafo a una matriz, $[M]^2$ ya contiene los caminos hamiltonianos y $[M]^3$ los circuitos hamiltonianos). A partir del quinto ejemplo ya se calculan sólo las matrices intermedias necesarias (cuanto más nodos tiene el grafo, más se simplifica el cálculo de las matrices).

Llega un momento en el que las matrices son demasiado grandes para que aparezcan en su formato normal y hay que representarlas de fila en fila a modo de texto.

Ejemplo 1: Empezaremos con un ejemplo sencillo de un grafo con cinco nodos. El bucle del nodo uno se elimina al crear la matriz $[M]^1$:



$$\text{Matriz\_grafo} = [M]^1 = \begin{pmatrix} \text{“”} & 1\#2 & 1\#3 & \text{“”} & \text{“”} \\ \text{“”} & \text{“”} & 2\#3 & \text{“”} & \text{“”} \\ \text{“”} & \text{“”} & \text{“”} & 3\#4 & 3\#5 \\ \text{“”} & \text{“”} & \text{“”} & \text{“”} & 4\#5 \\ 5\#1 & 5\#2 & \text{“”} & 5\#4 & \text{“”} \end{pmatrix}$$

$$[M]^2 = \begin{pmatrix} \text{“”} & \text{“”} & 1\#2\#3 & 1\#3\#4 & 1\#3\#5 \\ \text{“”} & \text{“”} & \text{“”} & 2\#3\#4 & 2\#3\#5 \\ 3\#5\#1 & 3\#5\#2 & \text{“”} & 3\#5\#4 & 3\#4\#5 \\ 4\#5\#1 & 4\#5\#2 & \text{“”} & \text{“”} & \text{“”} \\ \text{“”} & 5\#1\#2 & \begin{matrix} 5\#1\#3 \\ 5\#2\#3 \end{matrix} & \text{“”} & \text{“”} \end{pmatrix}$$

$$[M]^3 = \begin{pmatrix} \text{""} & 1\#3\#5\#2 & \text{""} & \begin{matrix}1\#2\#3\#4\\1\#3\#5\#4\end{matrix} & \begin{matrix}1\#2\#3\#5\\1\#3\#4\#5\end{matrix} \\ 2\#3\#5\#1 & \text{""} & \text{""} & 2\#3\#5\#4 & 2\#3\#4\#5 \\ 3\#4\#5\#1 & 3\#5\#1\#2 & \text{""} & \text{""} & \text{""} \\ \text{""} & 4\#5\#1\#2 & \begin{matrix}4\#5\#1\#3\\4\#5\#2\#3\end{matrix} & \text{""} & \text{""} \\ \text{""} & \text{""} & 5\#1\#2\#3 & \begin{matrix}5\#1\#3\#4\\5\#2\#3\#4\end{matrix} & \text{""} \end{pmatrix}$$

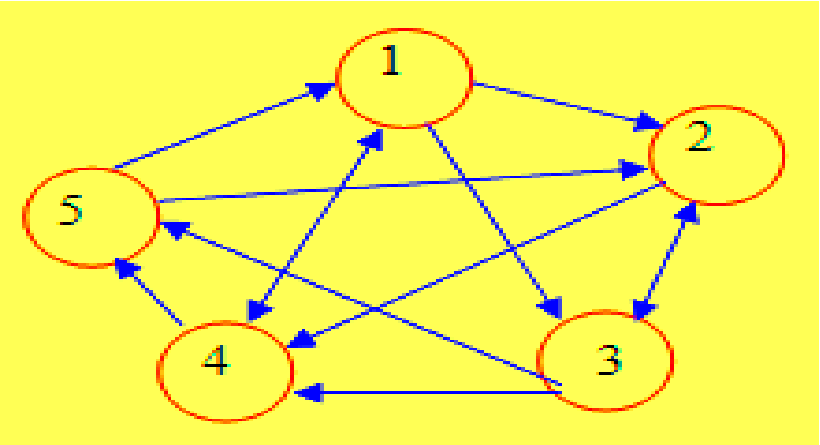
Como el grafo tiene cinco nodos, la matriz *[M]*^{4} nos muestra los caminos hamiltonianos:

$$[M]^4 = \begin{pmatrix} \text{""} & 1\#3\#4\#5\#2 & \text{""} & 1\#2\#3\#5\#4 & 1\#2\#3\#4\#5 \\ 2\#3\#4\#5\#1 & \text{""} & \text{""} & \text{""} & \text{""} \\ \text{""} & 3\#4\#5\#1\#2 & \text{""} & \text{""} & \text{""} \\ \text{""} & \text{""} & 4\#5\#1\#2\#3 & \text{""} & \text{""} \\ \text{""} & \text{""} & \text{""} & 5\#1\#2\#3\#4 & \text{""} \end{pmatrix}$$

Y en la matriz *[M]5* tenemos los circuitos hamiltonianos:

$$[M]^5 = \begin{pmatrix} 1\#2\#3\#4\#5\#1 & \text{""} & \text{""} & \text{""} & \text{""} \\ \text{""} & 2\#3\#4\#5\#1\#2 & \text{""} & \text{""} & \text{""} \\ \text{""} & \text{""} & 3\#4\#5\#1\#2\#3 & \text{""} & \text{""} \\ \text{""} & \text{""} & \text{""} & 4\#5\#1\#2\#3\#4 & \text{""} \\ \text{""} & \text{""} & \text{""} & \text{""} & 5\#1\#2\#3\#4\#5 \end{pmatrix}$$

Ejemplo 2: En este ejemplo simplemente hay más arcos y los mismos nodos con lo que el número de caminos y circuitos hamiltonianos se incrementará.



$$\text{Matriz\_grafo} = [M]^1 = \begin{pmatrix} \text{""} & 1\#2 & 1\#3 & 1\#4 & \text{""} \\ \text{""} & \text{""} & 2\#3 & 2\#4 & \text{""} \\ \text{""} & 3\#2 & \text{""} & 3\#4 & 3\#5 \\ 4\#1 & \text{""} & \text{""} & \text{""} & 4\#5 \\ 5\#1 & 5\#2 & \text{""} & \text{""} & \text{""} \end{pmatrix}$$

$$[M]^2 = \begin{pmatrix} \text{""} & 1\#3\#2 & 1\#2\#3 & \begin{matrix}1\#2\#4\\1\#3\#4\end{matrix} & \begin{matrix}1\#3\#5\\1\#4\#5\end{matrix} \\ 2\#4\#1 & \text{""} & \text{""} & 2\#3\#4 & \begin{matrix}2\#3\#5\\2\#4\#5\end{matrix} \\ \begin{matrix}3\#4\#1\\3\#5\#1\end{matrix} & 3\#5\#2 & \text{""} & 3\#2\#4 & 3\#4\#5 \\ 4\#5\#1 & \begin{matrix}4\#1\#2\\4\#5\#2\end{matrix} & 4\#1\#3 & \text{""} & \text{""} \\ \text{""} & 5\#1\#2 & \begin{matrix}5\#1\#3\\5\#2\#3\end{matrix} & \begin{matrix}5\#1\#4\\5\#2\#4\end{matrix} & \text{""} \end{pmatrix}$$

$$[M]^3 = \begin{pmatrix} \text{""} & \begin{matrix}1\#3\#5\#2\\1\#4\#5\#2\end{matrix} & \text{""} & \begin{matrix}1\#3\#2\#4\\1\#2\#3\#4\end{matrix} & \begin{matrix}1\#2\#3\#5\\1\#2\#4\#5\\1\#3\#4\#5\end{matrix} \\ \begin{matrix}2\#3\#4\#1\\2\#3\#5\#1\\2\#4\#5\#1\end{matrix} & \text{""} & 2\#4\#1\#3 & \text{""} & 2\#3\#4\#5 \\ \begin{matrix}3\#2\#4\#1\\3\#4\#5\#1\end{matrix} & \begin{matrix}3\#4\#1\#2\\3\#5\#1\#2\\3\#4\#5\#2\end{matrix} & \text{""} & \begin{matrix}3\#5\#1\#4\\3\#5\#2\#4\end{matrix} & 3\#2\#4\#5 \\ \text{""} & \begin{matrix}4\#5\#1\#2\\4\#1\#3\#2\end{matrix} & \begin{matrix}4\#5\#1\#3\\4\#1\#2\#3\\4\#5\#2\#3\end{matrix} & \text{""} & 4\#1\#3\#5 \\ 5\#2\#4\#1 & 5\#1\#3\#2 & 5\#1\#2\#3 & \begin{matrix}5\#1\#2\#4\\5\#1\#3\#4\\5\#2\#3\#4\end{matrix} & \text{""} \end{pmatrix}$$

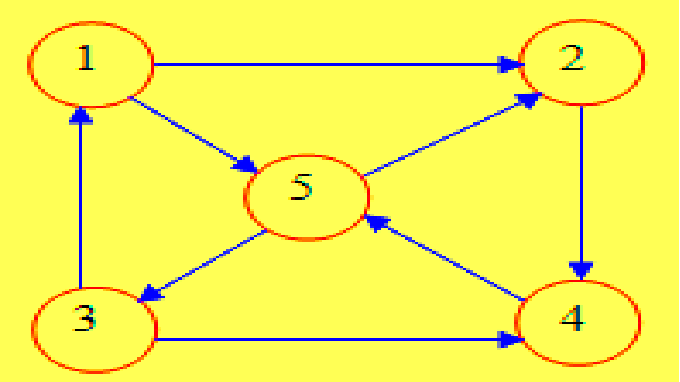
Como el número de nodos es cinco, la matriz $[M]^4$ contendrá los circuitos hamiltonianos:

$$[M]^4 = \begin{pmatrix} \text{""} & 1\#3\#4\#5\#2 & 1\#4\#5\#2\#3 & 1\#3\#5\#2\#4 & \begin{matrix}1\#3\#2\#4\#5\\1\#2\#3\#4\#5\end{matrix} \\ 2\#3\#4\#5\#1 & \text{""} & 2\#4\#5\#1\#3 & 2\#3\#5\#1\#4 & 2\#4\#1\#3\#5 \\ \begin{matrix}3\#5\#2\#4\#1\\3\#2\#4\#5\#1\end{matrix} & 3\#4\#5\#1\#2 & \text{""} & 3\#5\#1\#2\#4 & \text{""} \\ \text{""} & \begin{matrix}4\#5\#1\#3\#2\\4\#1\#3\#5\#2\end{matrix} & 4\#5\#1\#2\#3 & \text{""} & 4\#1\#2\#3\#5 \\ 5\#2\#3\#4\#1 & \text{""} & 5\#2\#4\#1\#3 & \begin{matrix}5\#1\#3\#2\#4\\5\#1\#2\#3\#4\end{matrix} & \text{""} \end{pmatrix}$$

Y en la diagonal principal de la matriz $[M]^5$ estarán los circuitos hamiltonianos:

$$[M]^5 = \begin{pmatrix} \begin{matrix}1\#2\#3\#4\#5\#1\\1\#3\#5\#2\#4\#1\\1\#3\#2\#4\#5\#1\end{matrix} & \text{""} & \text{""} & \text{""} & \text{""} \\ \text{""} & \begin{matrix}2\#3\#4\#5\#1\#2\\2\#4\#5\#1\#3\#2\\2\#4\#1\#3\#5\#2\end{matrix} & \text{""} & \text{""} & \text{""} \\ \text{""} & \text{""} & \begin{matrix}3\#2\#4\#5\#1\#3\\3\#4\#5\#1\#2\#3\\3\#5\#2\#4\#1\#3\end{matrix} & \text{""} & \text{""} \\ \text{""} & \text{""} & \text{""} & \begin{matrix}4\#1\#3\#5\#2\#4\\4\#5\#1\#2\#3\#4\\4\#5\#1\#3\#2\#4\end{matrix} & \text{""} \\ \text{""} & \text{""} & \text{""} & \text{""} & \begin{matrix}5\#1\#3\#2\#4\#5\\5\#1\#2\#3\#4\#5\\5\#2\#4\#1\#3\#5\end{matrix} \end{pmatrix}$$

Ejemplo 3: Con este grafo completamos los ejemplos de cinco nodos:



$$\text{Matriz\_grafo} = [M]^1 = \begin{pmatrix} \text{""} & 1\#2 & \text{""} & \text{""} & 1\#5 \\ \text{""} & \text{""} & \text{""} & 2\#4 & \text{""} \\ 3\#1 & \text{""} & \text{""} & 3\#4 & \text{""} \\ \text{""} & \text{""} & \text{""} & \text{""} & 4\#5 \\ \text{""} & 5\#2 & 5\#3 & \text{""} & \text{""} \end{pmatrix}$$

$$[M]^2 = \begin{pmatrix} \text{""} & 1\#5\#2 & 1\#5\#3 & 1\#2\#4 & \text{""} \\ \text{""} & \text{""} & \text{""} & \text{""} & 2\#4\#5 \\ \text{""} & 3\#1\#2 & \text{""} & \text{""} & \begin{matrix} 3\#1\#5 \\ 3\#4\#5 \end{matrix} \\ \text{""} & 4\#5\#2 & 4\#5\#3 & \text{""} & \text{""} \\ 5\#3\#1 & \text{""} & \text{""} & \begin{matrix} 5\#2\#4 \\ 5\#3\#4 \end{matrix} & \text{""} \end{pmatrix}$$

$$[M]^3 = \begin{pmatrix} \text{----} & \text{----} & \text{----} & \begin{matrix}1\#5\#2\#4\\1\#5\#3\#4\end{matrix} & 1\#2\#4\#5 \\ \text{----} & \text{----} & 2\#4\#5\#3 & \text{----} & \text{----} \\ \text{----} & \begin{matrix}3\#1\#5\#2\\3\#4\#5\#2\end{matrix} & \text{----} & 3\#1\#2\#4 & \text{----} \\ 4\#5\#3\#1 & \text{----} & \text{----} & \text{----} & \text{----} \\ \text{----} & 5\#3\#1\#2 & \text{----} & \text{----} & \text{----} \end{pmatrix}$$

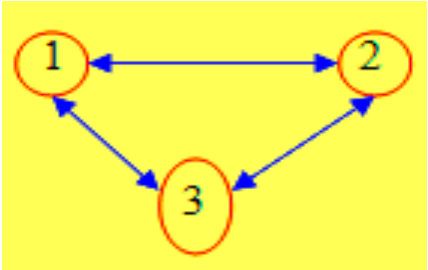
La matriz $[M]^4$ contendrá los caminos hamiltonianos:

$$[M]^4 = \begin{pmatrix} \text{----} & \text{----} & 1\#2\#4\#5\#3 & \text{----} & \text{----} \\ 2\#4\#5\#3\#1 & \text{----} & \text{----} & \text{----} & \text{----} \\ \text{----} & \text{----} & \text{----} & 3\#1\#5\#2\#4 & 3\#1\#2\#4\#5 \\ \text{----} & 4\#5\#3\#1\#2 & \text{----} & \text{----} & \text{----} \\ \text{----} & \text{----} & \text{----} & 5\#3\#1\#2\#4 & \text{----} \end{pmatrix}$$

Y la matriz $[M]^5$ los circuitos hamiltonianos:

$$[M]^5 = \begin{pmatrix} 1\#2\#4\#5\#3\#1 & \text{----} & \text{----} & \text{----} & \text{----} \\ \text{----} & 2\#4\#5\#3\#1\#2 & \text{----} & \text{----} & \text{----} \\ \text{----} & \text{----} & 3\#1\#2\#4\#5\#3 & \text{----} & \text{----} \\ \text{----} & \text{----} & \text{----} & 4\#5\#3\#1\#2\#4 & \text{----} \\ \text{----} & \text{----} & \text{----} & \text{----} & 5\#3\#1\#2\#4\#5 \end{pmatrix}$$

Ejemplo 4 : En este ejemplo bastará con calcular la matriz $[M]^2$ para saber los caminos hamiltonianos existentes y $[M]^3$ para saber los circuitos hamiltonianos ya que el número de nodos de este grafo es tres.



$$\text{Matriz\_grafo} = [M]^1 = \begin{pmatrix} \text{“”} & 1\#2 & 1\#3 \\ 2\#1 & \text{“”} & 2\#3 \\ 3\#1 & 3\#2 & \text{“”} \end{pmatrix}$$

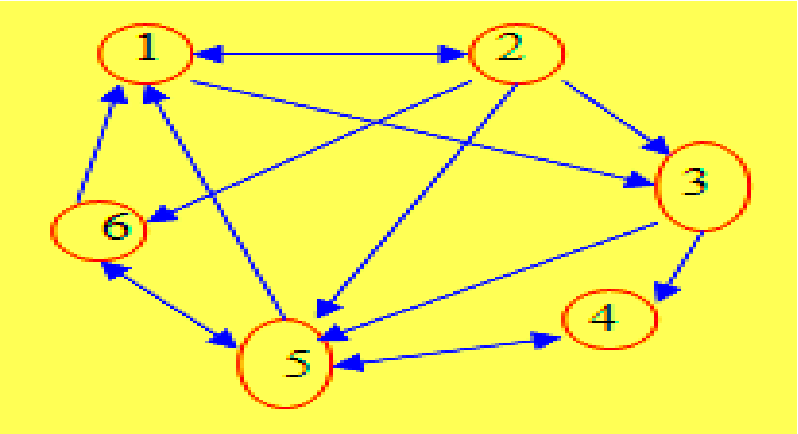
Caminos hamiltonianos:

$$[M]^2 = \begin{pmatrix} \text{“”} & 1\#3\#2 & 1\#2\#3 \\ 2\#3\#1 & \text{“”} & 2\#1\#3 \\ 3\#2\#1 & 3\#1\#2 & \text{“”} \end{pmatrix}$$

Circuitos hamiltonianos:

$$[M]3 = \begin{pmatrix} \begin{Bmatrix} 1\#2\#3\#1 \\ 1\#3\#2\#1 \end{Bmatrix} & \text{“”} & \text{“”} \\ \text{“”} & \begin{Bmatrix} 2\#1\#3\#2 \\ 2\#3\#1\#2 \end{Bmatrix} & \text{“”} \\ \text{“”} & \text{“”} & \begin{Bmatrix} 3\#1\#2\#3 \\ 3\#2\#1\#3 \end{Bmatrix} \end{pmatrix}$$

Ejemplo 5: Para un grafo de seis nodos, no hace falta calcular la matriz $[M]^3$, se hace la multiplicación latina de $[M]^2$ por $[M]^2$ y ya tenemos $[M]^4$.



Matriz_grafo = $[M]^1$ =

""	1#2	1#3	""	""	""
2#1	""	2#3	""	2#5	2#6
""	""	""	3#4	3#5	""
""	""	""	""	4#5	""
5#1	""	""	5#4	""	5#6
6#1	""	""	""	6#5	""

$$[M]^2 = \begin{pmatrix} "" & "" & 1\#2\#3 & 1\#3\#4 & \begin{matrix}1\#2\#5\\1\#3\#5\end{matrix} & 1\#2\#6 \\ \begin{matrix}2\#5\#1\\2\#6\#1\end{matrix} & "" & 2\#1\#3 & \begin{matrix}2\#3\#4\\2\#5\#4\end{matrix} & \begin{matrix}2\#3\#5\\2\#6\#5\end{matrix} & 2\#5\#6 \\ 3\#5\#1 & "" & "" & 3\#5\#4 & 3\#4\#5 & 3\#5\#6 \\ 4\#5\#1 & "" & "" & "" & "" & 4\#5\#6 \\ 5\#6\#1 & 5\#1\#2 & 5\#1\#3 & "" & "" & "" \end{pmatrix}$$

Matriz $[M]^4$:

$$\begin{pmatrix} "" & "" & "" & \begin{matrix}1\#2\#3\#5\#4\\1\#2\#6\#5\#4\end{matrix} & 1\#2\#3\#4\#5 & \begin{matrix}1\#2\#3\#5\#6\\1\#3\#4\#5\#6\end{matrix} \\ \begin{matrix}2\#3\#4\#5\#1\\2\#3\#5\#6\#1\end{matrix} & "" & \begin{matrix}2\#6\#5\#1\#3\\2\#5\#6\#1\#3\end{matrix} & \begin{matrix}2\#5\#1\#3\#4\\2\#6\#1\#3\#4\\2\#1\#3\#5\#4\end{matrix} & \begin{matrix}2\#6\#1\#3\#5\\2\#1\#3\#4\#5\end{matrix} & \begin{matrix}2\#1\#3\#5\#6\\2\#3\#4\#5\#6\end{matrix} \\ 3\#4\#5\#6\#1 & \begin{matrix}3\#4\#5\#1\#2\\3\#5\#6\#1\#2\end{matrix} & "" & "" & "" & 3\#5\#1\#2\#6 \\ "" & 4\#5\#6\#1\#2 & \begin{matrix}4\#5\#1\#2\#3\\4\#5\#6\#1\#3\end{matrix} & "" & "" & 4\#5\#1\#2\#6 \\ "" & "" & 5\#6\#1\#2\#3 & \begin{matrix}5\#6\#1\#3\#4\\5\#1\#2\#3\#4\end{matrix} & "" & "" \\ "" & "" & 6\#5\#1\#2\#3 & \begin{matrix}6\#5\#1\#3\#4\\6\#1\#2\#3\#4\\6\#1\#2\#5\#4\\6\#1\#3\#5\#4\end{matrix} & \begin{matrix}6\#1\#3\#4\#5\\6\#1\#2\#3\#5\end{matrix} & "" \end{pmatrix}$$

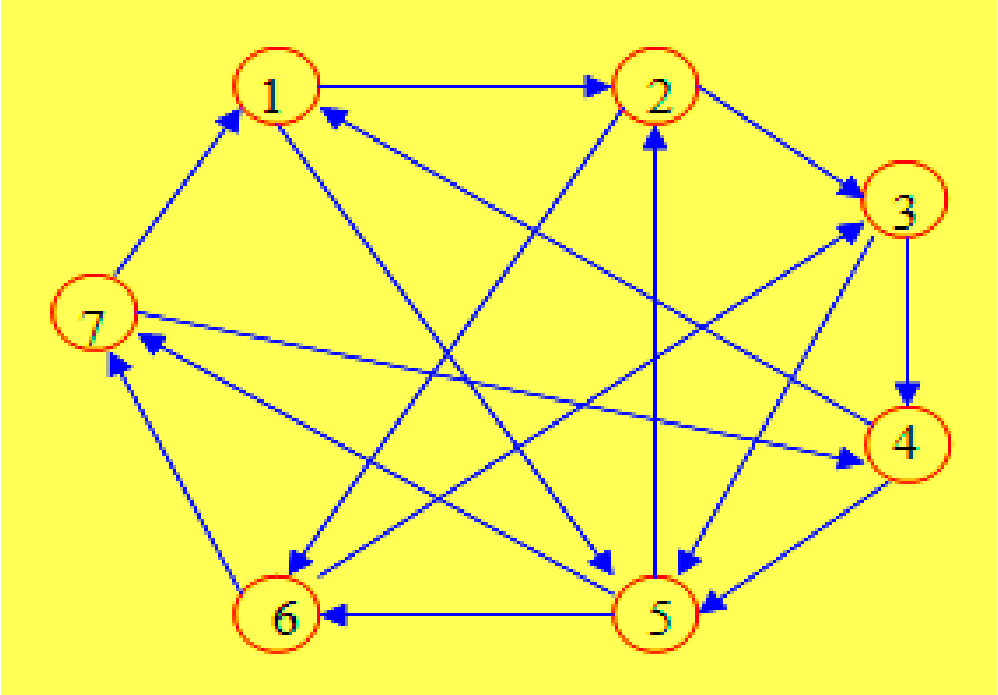
Los caminos hamiltonianos están en $[M]^5$:

“”	“”	“”	“”	“”	1#2#3#4#5#6
2#3#4#5#6#1	“”	“”	2#6#5#1#3#4 2#5#6#1#3#4 2#6#1#3#5#4	2#6#1#3#4#5	2#1#3#4#5#6
“”	3#4#5#6#1#2	“”	“”	“”	3#4#5#1#2#6
“”	“”	4#5#6#1#2#3	“”	“”	“”
“”	“”	“”	5#6#1#2#3#4	“”	“”
“”	“”	“”	6#5#1#2#3#4 6#1#2#3#5#4	6#1#2#3#4#5	“”

Y los circuitos hamiltonianos en **[M]**^{6}:

1#2#3#4#5#6#1	“”	“”	“”	“”	“”
“”	1#2#3#4#5#6#1	“”	“”	“”	“”
“”	“”	1#2#3#4#5#6#1	“”	“”	“”
“”	“”	“”	1#2#3#4#5#6#1	“”	“”
“”	“”	“”	“”	1#2#3#4#5#6#1	“”
“”	“”	“”	“”	“”	1#2#3#4#5#6#1

Ejemplo 6: El objetivo es llegar a las matrices $[M]^6$ (caminos hamiltonianos) y $[M]^7$ (circuitos hamiltonianos) lo más rápidamente posible y sólo con las matrices intermedias necesarias. Es un grafo con muchos arcos por lo que la multiplicación latina se hace más trabajosa y se esperan muchos caminos y circuitos hamiltonianos.



$$\text{Matriz\_grafo} = [M]^1 = \begin{pmatrix} \text{""} & 1\#2 & \text{""} & \text{""} & 1\#5 & \text{""} & \text{""} \\ \text{""} & \text{""} & 2\#3 & \text{""} & \text{""} & 2\#6 & \text{""} \\ \text{""} & \text{""} & \text{""} & 3\#4 & 3\#5 & \text{""} & \text{""} \\ 4\#1 & \text{""} & \text{""} & \text{""} & 4\#5 & \text{""} & \text{""} \\ \text{""} & 5\#2 & \text{""} & \text{""} & \text{""} & 5\#6 & 5\#7 \\ \text{""} & \text{""} & 6\#3 & \text{""} & \text{""} & \text{""} & 6\#7 \\ 7\#1 & \text{""} & \text{""} & 7\#4 & \text{""} & \text{""} & \text{""} \end{pmatrix}$$

$$[M]^2 = \begin{pmatrix} \text{“”} & 1\#5\#2 & 1\#2\#3 & \text{“”} & \text{“”} & \begin{matrix}1\#2\#6\\1\#5\#6\end{matrix} & 1\#5\#7 \\ \text{“”} & \text{“”} & 2\#6\#3 & 2\#3\#4 & 2\#3\#5 & \text{“”} & 2\#6\#7 \\ 3\#4\#1 & 3\#5\#2 & \text{“”} & \text{“”} & 3\#4\#5 & 3\#5\#6 & 3\#5\#7 \\ \text{“} & \begin{matrix}4\#1\#2\\4\#5\#2\end{matrix} & \text{“”} & \text{“”} & 4\#1\#5 & 4\#5\#6 & 4\#5\#7 \\ 5\#7\#1 & \text{“”} & \begin{matrix}5\#2\#3\\5\#6\#3\end{matrix} & 5\#7\#4 & \text{“”} & 5\#2\#6 & 5\#6\#7 \\ 6\#7\#1 & \text{“”} & \text{“”} & \begin{matrix}6\#3\#4\\6\#7\#4\end{matrix} & 6\#3\#5 & \text{“”} & \text{“”} \\ 7\#4\#1 & 7\#1\#2 & \text{“”} & \text{“”} & \begin{matrix}7\#1\#5\\7\#4\#5\end{matrix} & \text{“”} & \text{“”} \end{pmatrix}$$

Nos saltamos el cálculo de $[M]^3$ y $[M]^5$. La matriz $[M]^4$ se calcula por multiplicación latina entre $[M]^2$ y $[M]^2$ y la matriz $[M]^6$ por multiplicación latina entre $[M]^4$ y $[M]^2$. La matriz $[M]^4$ es muy grande para representarla como las anteriores y se representará en forma de tabla. A partir de aquí, las matrices muy grandes se representarán así.

$[M]^4$:

“”	“”	1#5#2#6#3	1#5#2#3#4 1#2#6#3#4 1#5#6#3#4 1#2#6#7#4 1#5#6#7#4	1#2#3#4#5 1#2#6#3#5	1#2#3#5#6	1#5#2#6#7 1#2#3#5#7
2#6#3#4#1 2#3#5#7#1 2#6#7#4#1	“”	“”	2#3#5#7#4	2#6#3#4#5 2#3#4#1#5 2#6#7#1#5 2#6#7#4#5	2#3#4#5#6	2#6#3#5#7 2#3#4#5#7 2#3#5#6#7
3#4#5#7#1 3#5#6#7#1 3#5#7#4#1	3#4#1#5#2 3#5#7#1#2	“”	3#5#6#7#4	“”	3#4#1#2#6 3#4#1#5#6 3#4#5#2#6	3#4#1#5#7 3#5#2#6#7 3#4#5#6#7
4#5#6#7#1	4#5#7#1#2	4#1#2#6#3 4#5#2#6#3 4#1#5#2#3 4#1#5#6#3	“”	4#1#2#3#5	4#1#5#2#6	4#1#2#6#7 4#5#2#6#7 4#1#5#6#7
5#2#3#4#1 5#6#3#4#1 5#2#6#7#1 5#6#7#4#1	5#7#4#1#2 5#6#7#1#2	5#7#1#2#3	5#2#6#3#4 5#2#6#7#4	“”	5#7#1#2#6	“”
6#3#5#7#1	6#7#1#5#2 6#3#4#1#2 6#3#4#5#2 6#7#4#1#2 6#7#4#5#2	6#7#1#2#3	6#3#5#7#4	6#3#4#1#5 6#7#4#1#5	“”	6#3#4#5#7
“”	7#4#1#5#2	7#4#1#2#3 7#1#2#6#3 7#1#5#2#3 7#1#5#6#3 7#4#5#2#3 7#4#5#6#3	7#1#2#3#4	7#1#2#3#5	7#4#1#2#6 7#4#1#5#6 7#1#5#2#6 7#4#5#2#6	“”

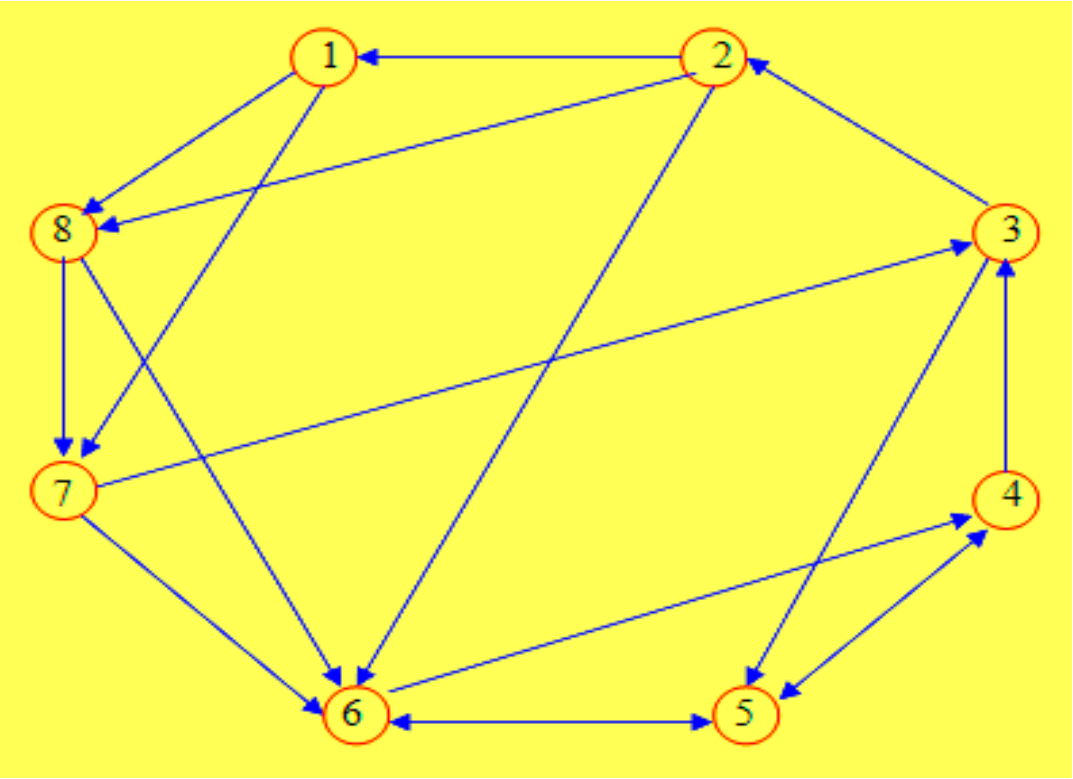
Caminos hamiltonianos (matriz $[M]^6$):

“”	“”	“”	1#2#6#3#5#7#4 1#2#3#5#6#7#4	“”	“”	1#2#6#3#4#5#7 1#2#3#4#5#6#7
2#6#3#5#7#4#1 2#3#5#6#7#4#1 2#6#3#4#5#7#1 2#3#4#5#6#7#1	“”	“”	“”	“”	“”	2#6#3#4#1#5#7 2#3#4#1#5#6#7
3#5#2#6#7#4#1 3#4#5#2#6#7#1	3#5#6#7#4#1#2 3#4#5#6#7#1#2	“”	“”	“”	3#5#7#4#1#2#6 3#4#5#7#1#2#6	3#4#1#5#2#6#7
“”	“”	4#5#6#7#1#2#3 4#5#7#1#2#6#3	“”	“”	“”	4#1#2#6#3#5#7 4#1#2#3#5#6#7
“”	“”	5#6#7#4#1#2#3 5#7#4#1#2#6#3	5#6#7#1#2#3#4 5#7#1#2#6#3#4	“”	“”	“”
“”	6#3#5#7#4#1#2 6#3#4#5#7#1#2	6#7#4#1#5#2#3	6#7#1#5#2#3#4	6#7#4#1#2#3#5 6#7#1#2#3#4#5	“”	“”
“”	“”	7#4#1#5#2#6#3	7#1#5#2#6#3#4	7#4#1#2#6#3#5 7#1#2#6#3#4#5	7#4#1#2#3#5#6 7#1#2#3#4#5#6	“”

Circuitos hamiltonianos (matriz $[M]^7$):

1#2#6#3#5#7#4#1 1#2#3#5#6#7#4#1 1#2#6#3#4#5#7#1 1#2#3#4#5#6#7#1

Ejemplo 7: Seguimos con ejemplos de grafos cada vez más grandes.



$$\text{Matriz\_grafo} = [M]^1 = \begin{pmatrix} \text{“”} & \text{“”} & \text{“”} & \text{“”} & \text{“”} & \text{“”} & 1\#7 & 1\#8 \\ 2\#1 & \text{“”} & \text{“”} & \text{“”} & \text{“”} & 2\#6 & \text{“”} & 2\#8 \\ \text{“”} & 3\#2 & \text{“”} & \text{“”} & 3\#5 & \text{“”} & \text{“”} & \text{“”} \\ \text{“”} & \text{“”} & 4\#3 & \text{“”} & 4\#5 & \text{“”} & \text{“”} & \text{“”} \\ \text{“”} & \text{“”} & \text{“”} & 5\#4 & \text{“”} & 5\#6 & \text{“”} & \text{“”} \\ \text{“”} & \text{“”} & \text{“”} & 6\#4 & 6\#5 & \text{“”} & \text{“”} & \text{“”} \\ \text{“”} & \text{“”} & 7\#3 & \text{“”} & \text{“”} & 7\#6 & \text{“”} & \text{“”} \\ \text{“”} & \text{“”} & \text{“”} & \text{“”} & \text{“”} & 8\#6 & 8\#7 & \text{“”} \end{pmatrix}$$

$$[M]^2 = \begin{pmatrix} "" & "" & 1\#7\#3 & "" & "" & \begin{matrix}1\#7\#6\\1\#8\#6\end{matrix} & 1\#8\#7 & "" \\ "" & "" & "" & 2\#6\#4 & 2\#6\#5 & 2\#8\#6 & \begin{matrix}2\#1\#7\\2\#8\#7\end{matrix} & 2\#1\#8 \\ 3\#2\#1 & "" & "" & 3\#5\#4 & "" & \begin{matrix}3\#2\#6\\3\#5\#6\end{matrix} & "" & 3\#2\#8 \\ "" & 4\#3\#2 & "" & "" & 4\#3\#5 & 4\#5\#6 & "" & "" \\ "" & "" & 5\#4\#3 & 5\#6\#4 & "" & "" & "" & "" \\ "" & "" & 6\#4\#3 & 6\#5\#4 & 6\#4\#5 & "" & "" & "" \\ "" & 7\#3\#2 & "" & 7\#6\#4 & \begin{matrix}7\#3\#5\\7\#6\#5\end{matrix} & "" & "" & "" \\ "" & "" & 8\#7\#3 & 8\#6\#4 & 8\#6\#5 & 8\#7\#6 & "" & "" \end{pmatrix}$$

$[M]^4$

“”	1#8#7#3#2	1#7#6#4#3 1#8#6#4#3	1#7#3#5#4 1#7#6#5#4 1#8#6#5#4 1#8#7#6#4	1#8#7#3#5 1#7#6#4#5 1#8#6#4#5 1#8#7#6#5	1#7#3#2#6 1#7#3#5#6	“”	1#7#3#2#8
“”	“”	2#6#5#4#3 2#8#6#4#3 2#1#8#7#3	2#8#6#5#4 2#1#7#6#4 2#8#7#6#4 2#1#8#6#4	2#6#4#3#5 2#1#7#3#5 2#8#7#3#5 2#8#6#4#5 2#1#7#6#5 2#8#7#6#5 2#1#8#6#5	2#1#8#7#6	“”	“”
“”	“”	“”	3#2#6#5#4 3#2#8#6#4	3#2#6#4#5 3#2#8#6#5	3#2#1#7#6 3#2#8#7#6 3#2#1#8#6	3#2#1#8#7	“”
“”	“”	“”	“”	4#3#2#6#5	4#3#2#8#6	4#3#2#1#7 4#3#2#8#7	4#3#2#1#8
5#4#3#2#1	5#6#4#3#2	“”	“”	“”	5#4#3#2#6	“”	5#4#3#2#8
6#4#3#2#1	6#5#4#3#2	“”	“”	“”	“”	“”	6#4#3#2#8
“”	7#6#4#3#2	7#6#5#4#3	7#3#2#6#4 7#3#5#6#4	7#6#4#3#5 7#3#2#6#5	7#3#2#8#6	“”	7#3#2#1#8
8#7#3#2#1	8#6#4#3#2	8#6#5#4#3 8#7#6#4#3	8#7#3#5#4 8#7#6#5#4	8#6#4#3#5 8#7#6#4#5	8#7#3#2#6 8#7#3#5#6	“”	“”

$[M]^5$:

“”	1#7#6#4#3#2 1#8#6#4#3#2	1#7#6#5#4#3 1#8#6#5#4#3 1#8#7#6#4#3	1#8#7#3#5#4 1#8#7#6#5#4 1#7#3#2#6#4 1#7#3#5#6#4	1#7#6#4#3#5 1#8#6#4#3#5 1#8#7#6#4#5 1#7#3#2#6#5	1#8#7#3#2#6 1#8#7#3#5#6 1#7#3#2#8#6	“”	“”
“”	“”	2#8#6#5#4#3 2#1#7#6#4#3 2#8#7#6#4#3 2#1#8#6#4#3	2#1#7#3#5#4 2#8#7#3#5#4 2#1#7#6#5#4 2#8#7#6#5#4 2#1#8#6#5#4 2#1#8#7#6#4	2#8#6#4#3#5 2#1#8#7#3#5 2#1#7#6#4#5 2#8#7#6#4#5 2#1#8#6#4#5 2#1#8#7#6#5	2#1#7#3#5#6 2#8#7#3#5#6	“”	“”
“”	“”	“”	3#2#8#6#5#4 3#2#1#7#6#4 3#2#8#7#6#4 3#2#1#8#6#4	3#2#8#6#4#5 3#2#1#7#6#5 3#2#8#7#6#5 3#2#1#8#6#5	3#2#1#8#7#6	“”	“”
“”	“”	“”	“”	4#3#2#8#6#5	4#3#2#1#7#6 4#3#2#8#7#6 4#3#2#1#8#6	4#3#2#1#8#7	“”
5#6#4#3#2#1	“”	“”	“”	“”	5#4#3#2#8#6	5#4#3#2#1#7 5#4#3#2#8#7	5#4#3#2#1#8 5#6#4#3#2#8
6#5#4#3#2#1	“”	“”	“”	“”	“”	6#4#3#2#1#7 6#4#3#2#8#7	6#4#3#2#1#8 6#5#4#3#2#8
7#6#4#3#2#1	7#6#5#4#3#2	“”	7#3#2#6#5#4 7#3#2#8#6#4	7#3#2#6#4#5 7#3#2#8#6#5	7#3#2#1#8#6	“”	7#6#4#3#2#8
8#6#4#3#2#1	8#6#5#4#3#2 8#7#6#4#3#2	8#7#6#5#4#3	8#7#3#2#6#4 8#7#3#5#6#4	8#7#6#4#3#5 8#7#3#2#6#5	“”	“”	“”

“”	1#7#6#5#4#3#2 1#8#6#5#4#3#2 1#8#7#6#4#3#2	1#8#7#6#5#4#3	1#7#3#2#6#5#4 1#8#7#3#2#6#4 1#8#7#3#5#6#4 1#7#3#2#8#6#4	1#8#7#6#4#3#5 1#7#3#2#6#4#5 1#8#7#3#2#6#5 1#7#3#2#8#6#5	“”	“”	1#7#6#4#3#2#8
“”	“”	2#1#7#6#5#4#3 2#8#7#6#5#4#3 2#1#8#6#5#4#3 2#1#8#7#6#4#3	2#1#8#7#3#5#4 2#1#8#7#6#5#4 2#1#7#3#5#6#4 2#8#7#3#5#6#4	2#1#7#6#4#3#5 2#8#7#6#4#3#5 2#1#8#6#4#3#5 2#1#8#7#6#4#5	2#1#8#7#3#5#6	“”	“”
“”	“”	“”	3#2#1#7#6#5#4 3#2#8#7#6#5#4 3#2#1#8#6#5#4 3#2#1#8#7#6#4	3#2#1#7#6#4#5 3#2#8#7#6#4#5 3#2#1#8#6#4#5 3#2#1#8#7#6#5	“”	“”	“”
“”	“”	“”	“”	4#3#2#1#7#6#5 4#3#2#8#7#6#5 4#3#2#1#8#6#5	4#3#2#1#8#7#6	“”	“”
“”	“”	“”	“”	“”	5#4#3#2#1#7#6 5#4#3#2#8#7#6 5#4#3#2#1#8#6	5#6#4#3#2#1#7 5#4#3#2#1#8#7 5#6#4#3#2#8#7	5#6#4#3#2#1#8
“”	“”	“”	“”	“”	“”	6#5#4#3#2#1#7 6#4#3#2#1#8#7 6#5#4#3#2#8#7	6#5#4#3#2#1#8
7#6#5#4#3#2#1	“”	“”	7#3#2#8#6#5#4 7#3#2#1#8#6#4	7#3#2#8#6#4#5 7#3#2#1#8#6#5	“”	“”	7#6#4#3#2#1#8 7#6#5#4#3#2#8
8#6#5#4#3#2#1 8#7#6#4#3#2#1	8#7#6#5#4#3#2	“”	8#7#3#2#6#5#4	8#7#3#2#6#4#5	“”	8#6#4#3#2#1#7	“”

Esta es la matriz $[M]^6$

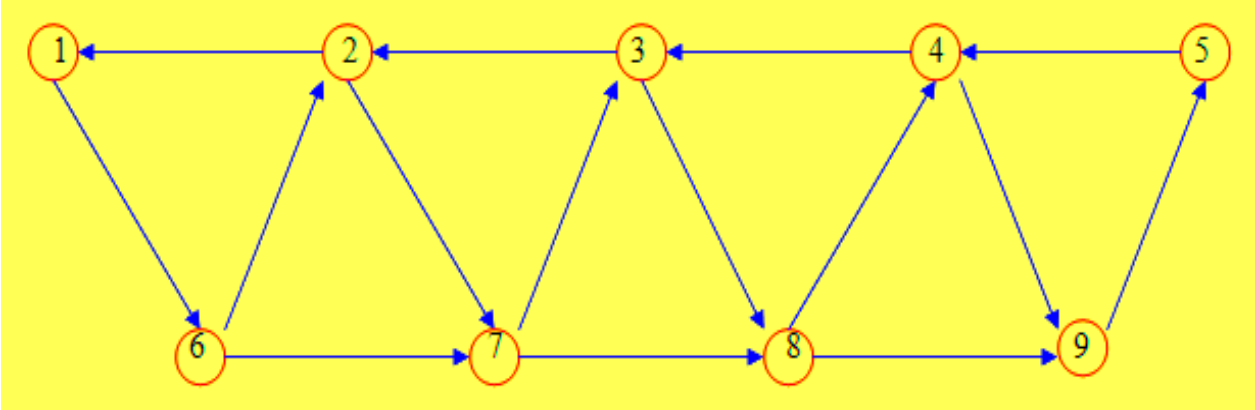
“”	1#8#7#6#5#4#3#2	“”	1#8#7#3#2#6#5#4 1#7#3#2#8#6#5#4	1#8#7#3#2#6#4#5 1#7#3#2#8#6#4#5	“”	“”	1#7#6#5#4#3#2#8
“”	“”	2#1#8#7#6#5#4#3	2#1#8#7#3#5#6#4	2#1#8#7#6#4#3#5	“”	“”	“”
“”	“”	“”	3#2#1#8#7#6#5#4	3#2#1#8#7#6#4#5	“”	“”	“”
“”	“”	“”	“”	4#3#2#1#8#7#6#5	“”	“”	“”
“”	“”	“”	“”	“”	5#4#3#2#1#8#7#6	5#6#4#3#2#1#8#7	“”
“”	“”	“”	“”	“”	“”	6#5#4#3#2#1#8#7	“”
“”	“”	“”	7#3#2#1#8#6#5#4	7#3#2#1#8#6#4#5	“”		7#6#5#4#3#2#1#8
8#7#6#5#4#3#2#1	“”	“”	“”	“”	“”	8#6#5#4#3#2#1#7	“”

Esta matriz contiene los caminos hamiltonianos (matriz $[M]^7$)

Circuitos hamiltonianos (matriz $[M]^8$):

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

Ejemplo 8: En este ejemplo de un grafo de nueve nodos, sólo hace falta calcular las matrices $[M]^1$, $[M]^2$, $[M]^4$, $[M]^8$ y $[M]^9$. Las matrices que realmente nos interesan son $[M]^8$ y $[M]^9$, es decir, las que contienen los caminos y circuitos hamiltonianos. Con esta simplificación se ahorra mucho tiempo de proceso en el ordenador y también se reduce el espacio de memoria ocupada considerablemente.



Matriz_grafo = $[M]^1$ =

					1#6			
2#1						2#7		
	3#2						3#8	
		4#3						4#9
			5#4					
	6#2					6#7		
		7#3					7#8	
			8#4					8#9
				9#5				

$$[M]^2 = \begin{pmatrix} \text{“”} & 1\#6\#2 & \text{“”} & \text{“”} & \text{“”} & \text{“”} & 1\#6\#7 & \text{“”} & \text{“”} \\ \text{“”} & \text{“”} & 2\#7\#3 & \text{“”} & \text{“”} & 2\#1\#6 & \text{“”} & 2\#7\#8 & \text{“”} \\ 3\#2\#1 & \text{“”} & \text{“”} & 3\#8\#4 & \text{“”} & \text{“”} & 3\#2\#7 & \text{“”} & 3\#8\#9 \\ \text{“”} & 4\#3\#2 & \text{“”} & \text{“”} & 4\#9\#5 & \text{“”} & \text{“”} & 4\#3\#8 & \text{“”} \\ \text{“”} & \text{“”} & 5\#4\#3 & \text{“”} & \text{“”} & \text{“”} & \text{“”} & \text{“”} & 5\#4\#9 \\ 6\#2\#1 & \text{“”} & 6\#7\#3 & \text{“”} & \text{“”} & \text{“”} & 6\#2\#7 & 6\#7\#8 & \text{“”} \\ \text{“”} & 7\#3\#2 & \text{“”} & 7\#8\#4 & \text{“”} & \text{“”} & \text{“”} & 7\#3\#8 & 7\#8\#9 \\ \text{“”} & \text{“”} & 8\#4\#3 & \text{“”} & 8\#9\#5 & \text{“”} & \text{“”} & \text{“”} & 8\#4\#9 \\ \text{“”} & \text{“”} & \text{“”} & 9\#5\#4 & \text{“”} & \text{“”} & \text{“”} & \text{“”} & \text{“”} \end{pmatrix}$$

$[M]^4$

“”	1#6#7#3#2	1#6#2#7#3	1#6#7#8#4	“”	“”	“”	1#6#7#3#8	1#6#7#8#9
“”	“”	2#7#8#4#3 2#1#6#7#3	2#7#3#8#4	2#7#8#9#5	“”	“”	2#1#6#7#8	2#7#8#4#9 2#7#3#8#9
“”	“”	“”	3#8#9#5#4 3#2#7#8#4	3#8#4#9#5	“”	3#2#1#6#7	“”	3#2#7#8#9
“”	“”	“”	“”	4#3#8#9#5	4#3#2#1#6	“”	4#3#2#7#8	“”
5#4#3#2#1	“”	“”	“”	“”	“”	5#4#3#2#7	“”	5#4#3#8#9
6#7#3#2#1	“”	6#7#8#4#3	6#7#3#8#4 6#2#7#8#4	6#7#8#9#5	“”	“”	6#2#7#3#8	6#7#8#4#9 6#7#3#8#9 6#2#7#8#9
“”	7#8#4#3#2	“”	7#8#9#5#4	7#8#4#9#5 7#3#8#9#5	7#3#2#1#6	“”	“”	7#3#8#4#9
8#4#3#2#1	“”	8#9#5#4#3	“”	“”	“”	8#4#3#2#7	“”	“”
“”	9#5#4#3#2	“”	“”	“”	“”	“”	9#5#4#3#8	“”

Recorrido óptimo de los nodos en una red

Caminos hamiltonianos de la matriz $[M]^8$ (cinco primeras columnas):

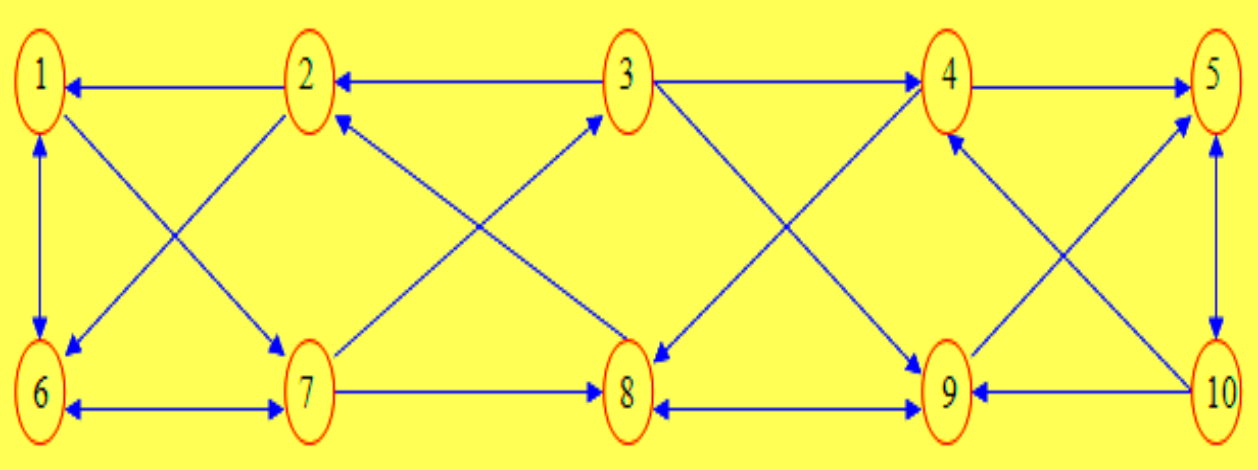
“”	1#6#7#8#9#5#4#3#2	1#6#2#7#8#9#5#4#3	1#6#2#7#3#8#9#5#4	1#6#2#7#3#8#4#9#5
“”	“”	2#1#6#7#8#9#5#4#3	2#1#6#7#3#8#9#5#4	2#1#6#7#3#8#4#9#5
“”	“”	“”	3#2#1#6#7#8#9#5#4	3#2#1#6#7#8#4#9#5
“”	“”	“”	“”	4#3#2#1#6#7#8#9#5
“”	“”	“”	“”	“”
6#7#8#9#5#4#3#2#1	“”	“”	“”	“”
“”	“”	“”	“”	“”
“”	“”	“”	“”	“”
“”	“”	“”	“”	“”

Cuatro últimas columnas:

“”	“”	“”	“”
“”	“”	“”	“”
“”	“”	“”	“”
“”	“”	“”	“”
“”	“”	“”	5#4#3#2#1#6#7#8#9
“”	“”	“”	“”
7#8#9#5#4#3#2#1#6	“”	“”	“”
“”	8#9#5#4#3#2#1#6#7	“”	“”
“”	“”	9#5#4#3#2#1#6#7#8	“”

Circuitos hamiltonianos (Matriz $[M]^9$): 1#6#7#8#9#5#4#3#2#1

Ejemplo 9: Seguimos la misma filosofía del ejemplo anterior para no calcular tantas matrices.



$$\text{Matriz\_grafo} = [M]^1 = \begin{pmatrix} & & & & & 1\#6 & 1\#7 & & & \\ 2\#1 & & & & & 2\#6 & & & & \\ & 3\#2 & & 3\#4 & & & & & 3\#9 & \\ & & & & 4\#5 & & & 4\#8 & & \\ & & & & & & & & & 5\#10 \\ 6\#1 & & & & & & 6\#7 & & & \\ & & 7\#3 & & & 7\#6 & & 7\#8 & & \\ & 8\#2 & & & & & & & 8\#9 & \\ & & & & 9\#5 & & & 9\#8 & & \\ & & & 10\#4 & 10\#5 & & & & 10\#9 & \end{pmatrix}$$

$[M]^2$:

$$
\begin{pmatrix}
"" & "" & 1\#7\#3 & "" & "" & 1\#7\#6 & 1\#6\#7 & 1\#7\#8 & "" & "" \\
2\#6\#1 & "" & "" & "" & "" & 2\#1\#6 & \begin{cases} 2\#1\#7 \\ 2\#6\#7 \end{cases} & "" & "" & "" \\
3\#2\#1 & "" & "" & "" & \begin{cases} 3\#4\#5 \\ 3\#9\#5 \end{cases} & 3\#2\#6 & "" & \begin{cases} 3\#4\#8 \\ 3\#9\#8 \end{cases} & "" & "" \\
"" & 4\#8\#2 & "" & "" & "" & "" & "" & "" & 4\#8\#9 & 4\#5\#10 \\
"" & "" & "" & 5\#10\#4 & "" & "" & "" & "" & 5\#10\#9 & "" \\
"" & "" & 6\#7\#3 & "" & "" & "" & 6\#1\#7 & 6\#7\#8 & "" & "" \\
7\#6\#1 & \begin{cases} 7\#3\#2 \\ 7\#8\#2 \end{cases} & "" & 7\#3\#4 & "" & "" & "" & "" & \begin{cases} 7\#3\#9 \\ 7\#8\#9 \end{cases} & "" \\
8\#2\#1 & "" & "" & "" & 8\#9\#5 & 8\#2\#6 & "" & "" & "" & "" \\
"" & 9\#8\#2 & "" & "" & "" & "" & "" & "" & "" & 9\#5\#10 \\
"" & "" & "" & "" & \begin{cases} 10\#4\#5 \\ 10\#9\#5 \end{cases} & "" & "" & \begin{cases} 10\#4\#8 \\ 10\#9\#8 \end{cases} & "" & ""
\end{pmatrix}
$$

$[M]^4$:

“”	1#6#7#3#2 1#6#7#8#2	“”	1#6#7#3#4	1#7#3#4#5 1#7#3#9#5 1#7#8#9#5	1#7#3#2#6 1#7#8#2#6	“”	1#7#3#4#8 1#7#3#9#8	1#6#7#3#9 1#6#7#8#9	“”
“”	“”	2#6#1#7#3 2#1#6#7#3	2#1#7#3#4 2#6#7#3#4	“”	“”	“”	2#6#1#7#8 2#1#6#7#8	2#1#7#3#9 2#6#7#3#9 2#1#7#8#9 2#6#7#8#9	“”
3#4#8#2#1 3#9#8#2#1	“”	“”	3#9#5#10#4	3#4#8#9#5	3#4#8#2#6 3#9#8#2#6 3#2#1#7#6	3#2#6#1#7 3#2#1#6#7	3#2#1#7#8 3#2#6#7#8	3#4#5#10#9	“”
4#8#2#6#1	“”	“”	“”	“”	4#8#2#1#6	4#8#2#1#7 4#8#2#6#7	4#5#10#9#8	“”	4#8#9#5#10
“”	5#10#4#8#2 5#10#9#8#2	“”	“”	“”	“”	“”	“”	5#10#4#8#9	“”
6#7#3#2#1 6#7#8#2#1	6#1#7#3#2 6#1#7#8#2	“”	6#1#7#3#4	6#7#3#4#5 6#7#3#9#5 6#7#8#9#5	“”	“”	6#7#3#4#8 6#7#3#9#8	6#1#7#3#9 6#1#7#8#9	“”
7#3#2#6#1 7#8#2#6#1	7#3#4#8#2 7#3#9#8#2	“”	“”	“”	7#3#2#1#6 7#8#2#1#6	“”	“”	7#3#4#8#9	7#3#4#5#10 7#3#9#5#10 7#8#9#5#10
“”	“”	8#2#1#7#3 8#2#6#7#3	8#9#5#10#4	“”	8#2#1#7#6	8#2#6#1#7 8#2#1#6#7	“”	“”	“”
9#8#2#6#1	“”	“”	“”	“”	9#8#2#1#6	9#8#2#1#7 9#8#2#6#7	9#5#10#4#8	“”	“”
10#4#8#2#1 10#9#8#2#1	“”	“”	“”	10#4#8#9#5	10#4#8#2#6 10#9#8#2#6	“”	“”	“”	“”

Cinco primeras columnas de $[M]^8$:

“”	1#7#3#9#5#10#4#8#2 1#7#3#4#5#10#9#8#2	“”	“”	“”
“”	“”	“”	2#6#1#7#3#9#5#10#4 2#1#6#7#3#9#5#10#4 2#6#1#7#8#9#5#10#4 2#1#6#7#8#9#5#10#4	2#6#1#7#3#4#8#9#5 2#1#6#7#3#4#8#9#5
3#9#5#10#4#8#2#6#1 3#4#5#10#9#8#2#6#1	“”	“”	3#2#1#7#8#9#5#10#4 3#2#6#7#8#9#5#10#4	“”
“”	“”	4#5#10#9#8#2#1#7#3 4#5#10#9#8#2#6#7#3	“”	4#8#2#6#1#7#3#9#5 4#8#2#1#6#7#3#9#5
“”	“”	5#10#4#8#2#6#1#7#3 5#10#9#8#2#6#1#7#3 5#10#4#8#2#1#6#7#3 5#10#9#8#2#1#6#7#3	5#10#9#8#2#1#7#3#4 5#10#9#8#2#6#7#3#4	“”
“”	6#7#3#9#5#10#4#8#2 6#7#3#4#5#10#9#8#2	“”	“”	“”
7#3#9#5#10#4#8#2#1 7#3#4#5#10#9#8#2#1	“”	“”	“”	“”
	“”	“”	8#2#1#7#3#9#5#10#4 8#2#6#7#3#9#5#10#4	“”
“”	“”	9#5#10#4#8#2#1#7#3 9#5#10#4#8#2#6#7#3	“”	9#8#2#6#1#7#3#4#5 9#8#2#1#6#7#3#4#5
“”	“”	“”	10#9#8#2#6#1#7#3#4 10#9#8#2#1#6#7#3#4	10#9#8#2#1#7#3#4#5 10#9#8#2#6#7#3#4#5 10#4#8#2#1#7#3#9#5 10#4#8#2#6#7#3#9#5

Cinco últimas columnas de $[M]^8$:

“”	“”	1#6#7#3#9#5#10#4#8 1#6#7#3#4#5#10#9#8	“”	1#6#7#3#4#8#9#5#10
“”	“”	2#1#7#3#9#5#10#4#8 2#6#7#3#9#5#10#4#8 2#1#7#3#4#5#10#9#8 2#6#7#3#4#5#10#9#8	2#6#1#7#3#4#5#10#9 2#1#6#7#3#4#5#10#9	2#1#7#3#4#8#9#5#10 2#6#7#3#4#8#9#5#10
3#9#5#10#4#8#2#1#6 3#4#5#10#9#8#2#1#6	3#9#5#10#4#8#2#1#7 3#4#5#10#9#8#2#1#7 3#9#5#10#4#8#2#6#7 3#4#5#10#9#8#2#6#7	“”	“”	3#2#6#1#7#8#9#5#10 3#2#1#6#7#8#9#5#10
4#5#10#9#8#2#1#7#6	4#5#10#9#8#2#6#1#7 4#5#10#9#8#2#1#6#7	“”	“”	4#8#2#1#7#3#9#5#10 4#8#2#6#7#3#9#5#10
“”	“”	“”	5#10#4#8#2#1#7#3#9 5#10#4#8#2#6#7#3#9	“”
“”	“”	6#1#7#3#9#5#10#4#8 6#1#7#3#4#5#10#9#8	“”	6#1#7#3#4#8#9#5#10
7#3#9#5#10#4#8#2#6 7#3#4#5#10#9#8#2#6	“”	“”	“”	“”
“”	“”	“”	8#2#1#7#3#4#5#10#9 8#2#6#7#3#4#5#10#9	8#2#6#1#7#3#4#5#10 8#2#1#6#7#3#4#5#10 8#2#6#1#7#3#9#5#10 8#2#1#6#7#3#9#5#10
9#5#10#4#8#2#1#7#6	9#5#10#4#8#2#6#1#7 9#5#10#4#8#2#1#6#7	“”	“”	9#8#2#1#7#3#4#5#10 9#8#2#6#7#3#4#5#10
“”	“”	“”	10#4#8#2#6#1#7#3#9 10#4#8#2#1#6#7#3#9	“”

Cinco primeras columnas de la matriz **[M]^{9}** con los caminos hamiltonianos:

“”	1#6#7#3#9#5#10#4#8#2 1#6#7#3#4#5#10#9#8#2	“”	“”	“”
“”	“”	“”	“”	“”
“”	“”	“”	3#2#6#1#7#8#9#5#10#4 3#2#1#6#7#8#9#5#10#4	“”
“”	“”	4#5#10#9#8#2#6#1#7#3 4#5#10#9#8#2#1#6#7#3	“”	“”
“”	“”	“”	5#10#9#8#2#6#1#7#3#4 5#10#9#8#2#1#6#7#3#4	“”
6#7#3#9#5#10#4#8#2#1 6#7#3#4#5#10#9#8#2#1	6#1#7#3#9#5#10#4#8#2 6#1#7#3#4#5#10#9#8#2	“”	“”	“”
7#3#9#5#10#4#8#2#6#1 7#3#4#5#10#9#8#2#6#1	“”	“”	“”	“”
“”	“”	“”	8#2#6#1#7#3#9#5#10#4 8#2#1#6#7#3#9#5#10#4	“”
“”	“”	9#5#10#4#8#2#6#1#7#3 9#5#10#4#8#2#1#6#7#3	“”	“”
“”	“”	“”	“”	10#9#8#2#6#1#7#3#4#5 10#9#8#2#1#6#7#3#4#5 10#4#8#2#6#1#7#3#9#5 10#4#8#2#1#6#7#3#9#5

Cinco últimas columnas de la matriz **[M]**[9] con los caminos hamiltonianos:

1#7#3#9#5#10#4#8#2#6 1#7#3#4#5#10#9#8#2#6	“”	“”	“”	“”
“”	“”	2#6#1#7#3#9#5#10#4#8 2#1#6#7#3#9#5#10#4#8 2#6#1#7#3#4#5#10#9#8 2#1#6#7#3#4#5#10#9#8	“”	2#6#1#7#3#4#8#9#5#10 2#1#6#7#3#4#8#9#5#10
3#9#5#10#4#8#2#1#7#6 3#4#5#10#9#8#2#1#7#6	3#9#5#10#4#8#2#6#1#7 3#4#5#10#9#8#2#6#1#7 3#9#5#10#4#8#2#1#6#7 3#4#5#10#9#8#2#1#6#7	“”	“”	“”
“”	“”	“”	“”	4#8#2#6#1#7#3#9#5#10 4#8#2#1#6#7#3#9#5#10
“”	“”	“”	5#10#4#8#2#6#1#7#3#9 5#10#4#8#2#1#6#7#3#9	“”
“”	“”	“”	“”	“”
7#3#9#5#10#4#8#2#1#6 7#3#4#5#10#9#8#2#1#6	“”	“”	“”	“”
“”	“”	“”	8#2#6#1#7#3#4#5#10#9 8#2#1#6#7#3#4#5#10#9	“”
“”	“”	“”	“”	9#8#2#6#1#7#3#4#5#10 9#8#2#1#6#7#3#4#5#10
“”	“”	“”	“”	“”

Matriz **[M]**[10] con los circuitos hamiltonianos:

1#6#7#3#9#5#10#4#8#2#1 1#6#7#3#4#5#10#9#8#2#1 1#7#3#9#5#10#4#8#2#6#1 1#7#3#4#5#10#9#8#2#6#1

Apéndice B: Notación, Símbolos y Abreviaciones

EXPRESIÓN	SIGNIFICADO
$\in$	Pertenece.
$\notin$	No pertenece.
$\forall$	Para todo.
$\sum$	Sumatorio.
n!	Factorial del valor numérico *"n"*, es decir: *n * (n - 1) * (n - 2) * … * 2 * 1*
<	Menor que.
>	Mayor que.
≤	Menor o igual.
≥	Mayor o igual.
≠	Distinto.
G <N, A>	Grafo; *N* es el conjunto de nodos y *A* el conjunto de aristas.
G (A)	Grado de un nodo *"A"*. Número de aristas que inciden en el nodo *"A"*.
G_n	Grafo completo de *"n"* nodos: tiene una arista entre cada par de vértices.
| N |	Número de nodos del conjunto de nodos *"N"*.

{“A”, 6}, {“B”, 8}, …	Vector compuesto por elementos que contienen el nombre de un nodo del grafo y su valor asociado.
$\cap$	Intersección.
U	Unión.
$\subset$	Incluido.
$\subseteq$	Incluido o igual.
φ	Conjunto vacío.
S = (i1, i2, … , ip) T = (j1, j2, … , jp)	Cadenas “*S*” y “*T*” de caracteres.
#	Separador de caracteres.
“”	Cadena de caracteres vacía.
1 # 2 # … # n	Cadena de nodos de un grafo separados por el carácter #.
$[M]^i$	Matriz de orden “*i*” de $(n * n)$ elementos (siendo “*n*” el número de nodos del grafo). Representa los caminos elementales de orden “*i*” de un grafo.
Matriz_grafo = $[M]^1$	Matriz que representa un grafo.
$[M]^{n-1}$	Matriz que contiene los caminos hamiltonianos de un grafo de “*n*” nodos.
$[M]^n$	Matriz que contiene los circuitos hamiltonianos de un grafo de “*n*” nodos.
m^i_{kp}	Elemento de la matriz $[M]^i$ situado en la fila “*k*” columna “*p*”.

*	Multiplicación aritmética.
*L	Multiplicación latina.
n(x, y)	Coordenadas de un punto en el área de dibujo de la aplicación informática.
{(x1,y1), (x2, y2)}	Representación de una línea recta en el área de dibujo.

Apéndice C: Traducción de la aplicación al inglés

Se ha querido presentar la aplicación también en inglés para que desde Internet sea más internacional. Es una traducción muy profesional y minuciosa traduciendo términos informáticos que se salen del lenguaje inglés más habitual. Para ello las personas encargadas de la traducción han sido:

- Ulrike Lelickens: Traductora profesional con el título de "Studienassesorin".

- Ludger Wagner: Traductor e informático con el título de "Diplom-Informatiker".

Entre los traductores se da una mezcla de conocimientos de traducción al inglés e informática imprescindible para traducir trabajos informáticos.

Apéndice D: Ayuda dentro de la aplicación

Documentación teórica dentro de la aplicación puesta en Internet

Antes de utilizar el área de dibujo y resolver sobre el grafo creado los algoritmos de Kaufmann, Kruskal o Prim, la aplicación dispone de una documentación teórica en los enlaces "Documentación teórica Kruskal y Prim" y "Documentación teórica Hamilton" en donde se explican conceptos de teoría de grafos, los algoritmos antes citados y ejemplos completos resueltos. Los conceptos se van explicando poco a poco y de forma muy sencilla y clara, nunca aparece un término o idea nuevo que no haya sido explicado con antelación. Con esta forma de ordenar las explicaciones, la lectura y comprensión del texto se hace más fácil.

La información sobre conceptos básicos y algoritmos es completa (incluyendo demostraciones y ejemplos) de tal forma que no hay que acudir a ningún texto adicional para completar dichos algoritmos.

El texto se completa con ilustraciones de grafos para seguirlo mejor

Botón de "Ayuda"

Antes de editar el grafo, se puede ver el contenido del botón de "Ayuda" en dos idiomas, castellano e inglés según la selección previa que hayamos hecho del idioma. Se va a mostrar el contenido de la Ayuda sólo en castellano para no excedernos en la documentación.

Según estemos en la ventana que calcula los algoritmos de Kruskal y Prim o en la ventana que calcula el algoritmo de Kaufmann, el contenido del botón de ayuda es diferente.

Apéndice D: Ayuda dentro de la aplicación

La ayuda se divide en dos partes fundamentales: Explicación de cada botón y pasos a seguir para la utilización de los elementos de la vantana. A continuación se muestra el contenido completo de los dos botones de Ayuda:

Ayuda Caminos y Circuitos Hamiltonianos

Índice

Botones

[Índice]

Dejando el puntero del ratón encima de cada botón de la aplicación aparecerá automáticamente un mensaje explicativo de las funciones que puede hacer el botón.

[Botones][Índice]

Botón Nodos

Este botón tiene dos funciones:

Haciendo un click de ratón sobre la ventana de diseño del grafo aparece un nuevo nodo, y arrastrando con el ratón un nodo ya creado, este nodo se desplaza de su posición siguiendo la dirección del puntero del ratón. Esto último es útil cuando queremos situar los elementos del grafo de tal forma que podamos verlos con claridad.

Al iniciarse la aplicación, como lo primero que tenemos que hacer es crear los nodos del grafo, no es necesario pulsar el botón Nodos, ya que esta función está activada por defecto.

Botón Arcos

Este botón se utiliza para añadir arcos al grafo. El valor del arco viene dado por el dato situado en Próximo dentro del cuadro Valores Arcos.

La forma de operar es la siguiente:

Primero hacemos un click de ratón sobre el nodo origen del arco y después hacemos otro click de ratón sobre el nodo destino.

El color del arco es gris, pero si dos nodos están unidos por arcos de distinto sentido, el valor de un arco estará en color azul y el valor del otro arco en color verde. Las puntas de las flechas de cada arco estarán en color azul (cuyo valor corresponde al valor en color azul) y en color verde (cuyo valor corresponde al valor en color verde).

Botón Valor

Se utiliza para cambiar el valor de un arco ya creado.

Pasos:

1. Hacemos click sobre el arco cuyo valor queremos cambiar (imaginemos que queremos cambiar el valor del arco (b,c)).
2. Si había otro arco entre dichos nodos (arco (c,b) en nuestro ejemplo) se mostrará un cuadro de diálogo informando de la situación, ofreciendo las

posibilidades de escoger el arco b->c o el c->b para la operación. Pulsaremos sobre aquél cuyo valor nos interese cambiar (también podremos cancelar la operación en este momento).

3. Aparece una ventana en la que se nos pide que introduzcamos el valor deseado.
4. Pulsamos el botón Aceptar.

[Botones][Índice]

Botón Borrar Nodo

Tras pulsar este botón, cada vez que hacemos click sobre un nodo, éste desaparece al igual que todos los arcos que tienen en este nodo su origen o su destino.

Botón Borrar Arco

Tras pulsar este botón, cada vez que hacemos click sobre un arco, éste desaparece.

Botón Borrar Último

Hace desaparecer un elemento del grafo (nodo o arco) en orden inverso a como se creó. Es decir, que cada vez que pulsamos el botón Último desaparece el nodo o arco que se creó en último lugar.

Botón Borrar Grafo

Elimina el grafo completo. Antes, lanza un mensaje de advertencia para preguntarnos si deseamos borrarlo.

Si pulsamos Borrar, el grafo desaparecerá. Si pulsamos Cancelar, el grafo permanecerá intacto.

[Botones][Índice]

Valores Arcos

En este marco debemos dar valores a los arcos, bien poniendo el valor directamente o bien a través de la barra de desplazamiento horizontal que suma o resta media unidad cada vez que pulsamos sus flechas de desplazamiento.

La casilla de verificación Ver nos permite visualizar u ocultar los valores de los arcos en el grafo.

Antes de crear un nuevo arco, debemos poner su valor en el cuadro de texto de este marco para que cuando se cree adquiera ese valor introducido.

Botón de Estadísticas

Este botón muestra una ventana con información relativa a los contadores de arcos del grafo.
Concretamente, agrupa los arcos por sus valores, informando de la cantidad de arcos de cada grupo.

[Botones][Índice]

Botón Caminos hamiltonianos

Teniendo el grafo completado, cuando pulsamos este botón, se calculan todos los caminos hamiltonianos del grafo, si existen.

Botón Circuitos hamiltonianos

Teniendo el grafo completado, cuando pulsamos este botón, se calculan todos los circuitos hamiltonianos del grafo, si existen.

Botón de Ayuda

Este botón, como su propio nombre indica, nos da información acerca de cómo un usuario puede manejar la aplicación.

Básicamente, se divide en dos partes:

- Una parte nos informa de las funciones que realiza cada botón que aparece en la parte inferior de la pantalla.
- La otra parte hace un seguimiento de los pasos ordenados que tenemos que hacer para manejar correctamente la aplicación.

Lista para seleccionar el idioma

Por defecto, la aplicación se abre en castellano pero existe la opción de trabajar con el idioma inglés seleccionándolo en esta lista de idiomas.
Dado que la aplicación está disponible en Internet, el idioma inglés es siempre interesante.

[Botones][Índice]

Botón Desplazar área dibujo

Al crear un grafo encontraremos con frecuencia que éste, a medida que crece, no está posicionado dentro del área de dibujo en el emplazamiento ideal para seguir añadiendo nodos.

Una vez pulsado este botón, se puede desplazar el grafo por el área de dibujo, de manera que quede colocado en una posición más cómoda para su edición.

Para ello, tras pulsar este botón basta con pulsar el botón izquierdo del ratón sobre el área de dibujo y arrastrarla en la dirección adecuada.

Botón Fondo

Basta con pulsar este botón sucesivamente para que aparezcan distintos colores de fondo sin afectar al grafo. El color que más nos guste quedará fijado.

Pasos a seguir

[Índice]

I. Edición del grafo.
II. Realización de los algoritmos.

[Pasos a seguir][Índice]

Edición del grafo

Crear nodos

Al iniciar la aplicación, lo primero que hay que hacer son los nodos y por tanto, está activado automáticamente el botón Nodos (si pasamos a otras fases de la aplicación y queremos crear nodos tenemos que activar el botón Nodos con un click de ratón).

Cada vez que hacemos un click de ratón sobre la pantalla de diseño aparece un nuevo nodo. También se puede arrastrar un nodo con el ratón para cambiarlo de sitio.

Crear arcos

Para trazar los arcos entre los nodos que hemos creado, lo primero que hay que hacer es un click de ratón sobre el botón Arcos. Después, tenemos que fijar el valor del arco en el marco Valores arcos directamente en la caja de texto Próximo o a través de la pequeña barra horizontal.

Para trazar un arco entre dos nodos, tenemos que hacer un click de ratón sobre el nodo inicial y después otro click de ratón sobre el otro nodo (nodo final).

Si queremos cambiar el valor de algún arco, hacemos click en el botón Valor; a continuación seleccionamos el arco con un click de ratón sobre él y nos sale una ventana para que demos un nuevo valor al arco seleccionado.

Borrar arcos

Para eliminar arcos debemos activar el botón Arco con un click de ratón y a continuación, cada vez que hacemos click con el ratón sobre un arco, éste desaparece del grafo.

Borrar nodos

Para borrar nodos tenemos que hacer un click de ratón sobre el botón Nodo y a continuación, cada vez que hacemos un click de ratón sobre un nodo, éste desaparece junto con todos sus arcos (tanto entrantes como salientes).

Deshacer última acción

Cada vez que hacemos un click de ratón sobre el botón Último, desaparece la última acción que se ha hecho al editar el grafo, es decir, si lo último que hemos hecho es crear un arco, desaparecerá dicho arco y si lo último que hemos hecho es crear un nodo, desaparecerá dicho nodo.

Borrar el grafo

Si queremos hacer desaparecer el grafo completo, tenemos que activar el botón Grafo con un click de ratón, a continuación, aparecerá una ventana informativa para asegurarnos de que lo que queremos hacer es eliminar el grafo y así proceder con el criterio adecuado.

Cambiar el color del fondo de la ventana de diseño del grafo

Para terminar la edición, sólo nos queda, si queremos, cambiar el color del fondo sin modificar el color del grafo. Para ello, cada vez que hacemos click sobre el botón Fondo aparece un color de fondo distinto.

El número de colores que puede tomar el fondo de la ventana es finito y se repite cíclicamente.

Cálculo de todos los caminos hamiltonianos

Cuando hacemos un click de ratón sobre este botón es el propio programa el que se encarga de calcular todas las soluciones siguiendo el algoritmo de Kaufmann.

A medida que se calculan las soluciones, el índice de la matriz de cálculo actual, así como los índices de la fila, columna y solución actuales que están siendo procesados se muestran en la barra de estado. Mientras se espera a que el cálculo finalice, los nodos pueden ser desplazados arrastrándolos con el ratón, o todo el grafo en sí puede ser movido usando el botón Desplazar grafo**. Si el proceso resulta muy lento, existe un botón para interrumpirlo si el usuario lo considera oportuno.**

Cuando ya se han calculado todos los caminos, si existen, aparece una ventana con varios botones partiendo de cada nodo del grafo:

- *Siguiente*: **nos permite mostrar la siguiente solución.**
- *Anterior*: **nos permite mostrar la solución anterior.**
- *Terminar*: **se utiliza para salir de la ventana de soluciones y terminar con el proceso de cálculo de todos los caminos hamiltonianos.**
- *Mostrar/Ocultar detalles*: **pulsando este botón, se muestran las coordenadas de los arcos en el orden que constituye la solución para la solución óptima que está reflejada en el grafo, o se ocultan estas coordenadas si ya estaban siendo mostradas.**
- *Soluciones menores/Todas las soluciones*: **conmuta entre el modo de muestra de todos los caminos hamiltonianos para el grafo actual y el de muestra de tan solo los caminos de menor valor acumulado entre todos los caminos hamiltonianos que parten del mismo nodo, por cada nodo del grafo.**
- *Soluciones mayores/Todas las soluciones*: **conmuta entre el modo de muestra de todos los caminos hamiltonianos para el grafo actual y el de muestra de tan solo los caminos de mayor valor acumulado entre todos los caminos hamiltonianos que parten del mismo nodo, por cada nodo del grafo.**
- *Mostrar matrices/Ocultar matrices*: **abre o cierra la ventana que contiene las tablas representando las matrices de cálculo que se han utilizado para calcular los caminos hamiltonianos para el grafo actual.**

Los arcos solución aparecen en color rojo, así como el reborde del nodo origen del camino que corresponde a la solución actualmente mostrada. Si sólo hay una solución, se mostrará una ventana con las coordenadas de los arcos en el orden que constituye la solución y un botón para finalizar.

Cálculo de todos los circuitos hamiltonianos

Cuando hacemos un click de ratón sobre este botón es el propio programa el que se encarga de calcular todas las soluciones siguiendo el algoritmo de Kaufmann.

A medida que se calculan las soluciones, el índice de la matriz de cálculo actual, así como los índices de la fila, columna y solución actuales que están siendo procesados se muestran en la barra de estado. Mientras se espera a que el cálculo finalice, los nodos pueden ser desplazados arrastrándolos con el ratón, o todo el grafo en sí puede ser movido usando el botón Desplazar grafo. Si el proceso resulta muy lento, existe un botón para interrumpirlo si el usuario lo considera oportuno.

Cuando ya se han calculado todos los circuitos, si existen, aparece una ventana con varios botones partiendo de cada nodo del grafo:

- *Siguiente*: nos permite mostrar la siguiente solución.
- *Anterior*: nos permite mostrar la solución anterior.
- *Terminar*: se utiliza para salir de la ventana de soluciones y terminar con el proceso de cálculo de todos los circuitos hamiltonianos.
- *Mostrar/Ocultar detalles*: pulsando este botón, se muestran las coordenadas de los arcos en el orden que constituye la solución para la solución óptima que está reflejada en el grafo, o se ocultan estas coordenadas si ya estaban siendo mostradas.
- *Soluciones menores/Todas las soluciones*: conmuta entre el modo de muestra de todos los circuitos hamiltonianos para el grafo actual y el de muestra de tan solo los circuitos de menor valor acumulado entre todos los circuitos hamiltonianos que parten del mismo nodo, por cada nodo del grafo.
- *Soluciones mayores/Todas las soluciones*: conmuta entre el modo de muestra de todos los circuitos hamiltonianos para el grafo actual y el de muestra de tan solo los circuitos de mayor valor acumulado entre todos los circuitos hamiltonianos que parten del mismo nodo, por cada nodo del grafo.
- *Mostrar matrices/Ocultar matrices*: abre o cierra la ventana que contiene las tablas representando las matrices de cálculo que se han utilizado para calcular los circuitos hamiltonianos para el grafo actual.

Los arcos solución aparecen en color rojo, así como el reborde del nodo origen del circuito que corresponde a la solución actualmente mostrada. Si sólo hay una solución, se mostrará una ventana con las coordenadas de los arcos en el orden que constituye la solución y un botón para finalizar.

[Pasos a seguir][Índice]

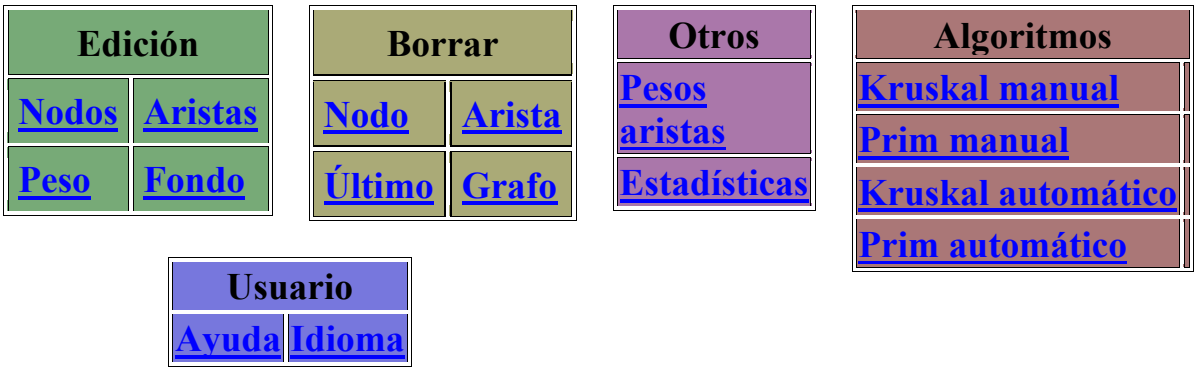
Ayuda Árboles de Recubrimiento Mínimo

Índice

Botones

[Índice]

Dejando el puntero del ratón encima de cada botón aparecerá automáticamente un mensaje explicativo de las funciones que puede hacer el botón.



[Botones][Índice]

Botón Nodos

Este botón tiene dos funciones:

Haciendo un click de ratón sobre la ventana de diseño del grafo aparece un nuevo nodo, y arrastrando con el ratón un nodo ya creado, este nodo se desplaza de su posición siguiendo la dirección del puntero del ratón. Esto último es útil cuando queremos situar los elementos del grafo de tal forma que podamos verlos con claridad.

Al iniciarse la aplicación, como lo primero que tenemos que hacer es crear los nodos del grafo, no es necesario pulsar el botón Nodos, ya que esta función está activada por defecto.

Botón Aristas

Este botón se utiliza para añadir aristas al grafo. El valor de la arista viene dado por el dato situado en Próximo dentro del cuadro Pesos Aristas.

La forma de operar es la siguiente:

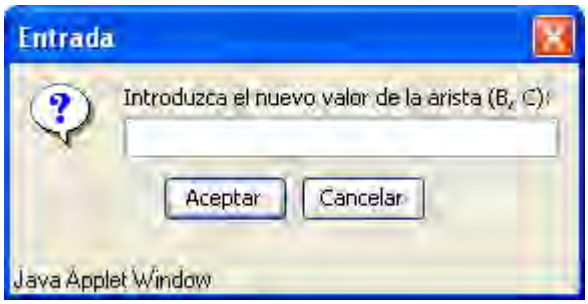
Primero hacemos un click de ratón sobre uno de los nodos en donde va a incidir la arista y después hacemos otro click de ratón sobre el otro nodo de la arista.

Botón Peso

Se utiliza para cambiar el valor de una arista ya creada.

Pasos:

1. Hacemos click sobre la arista cuyo valor queremos cambiar (imaginemos que queremos cambiar el valor de la arista (b , c)).
2. Aparece una ventana en la que se nos pide que introduzcamos el valor deseado.



3. **Pulsamos el botón** Aceptar.

Botón Fondo

Basta con pulsar este botón sucesivamente para que aparezcan distintos colores de fondo sin afectar al grafo. El color que más nos guste quedará fijado.

[Botones][Índice]

Botón Borrar Nodo

Tras pulsar este botón, cada vez que hacemos click sobre un nodo, éste desaparece al igual que todas las aristas que inciden sobre él.

Botón Borrar Arista

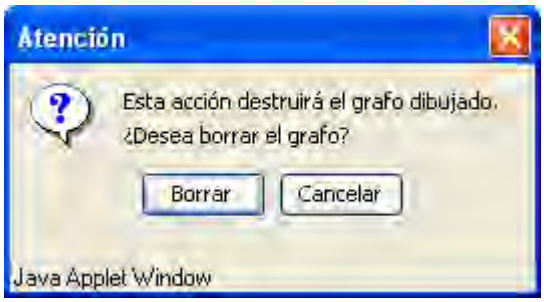
Tras pulsar este botón, cada vez que hacemos click sobre una arista, ésta desaparece.

Botón Borrar Último

Hace desaparecer un elemento del grafo (nodo o arista) en orden inverso a como se creó. Es decir, que cada vez que pulsamos el botón Último desaparece el nodo o arista que se creó en último lugar.

Botón Borrar Grafo

Elimina el grafo completo. Antes, lanza un mensaje de advertencia para preguntarnos si deseamos borrarlo.



Si pulsamos Borrar, el grafo desaparecerá. Si pulsamos Cancelar, el grafo permanecerá intacto.

[Botones][Índice]

Pesos Aristas

En este marco debemos dar valores a las aristas, bien poniendo el valor directamente o bien a través de la barra de desplazamiento horizontal que suma o resta media unidad cada vez que pulsamos sus flechas de desplazamiento.

La casilla de verificación Ver nos permite visualizar o no los valores de las aristas en el grafo.

Antes de crear una nueva arista, debemos poner su valor en el cuadro de texto de este marco para que cuando se cree adquiera el valor mostrado en ese cuadro.

Botón de Estadísticas

Este botón muestra una ventana con información relativa a los contadores de aristas del grafo.

Concretamente, agrupa las aristas por sus pesos, informando de la cantidad de aristas de cada grupo.

[Botones][Índice]

Botón Kruskal manual

Este botón se utiliza, una vez editado el grafo, para que el usuario realice el algoritmo de Kruskal sobre el grafo que él mismo ha creado.

El usuario debe seleccionar aristas con un click de ratón sobre ellas siguiendo el algoritmo de Kruskal y debe fijarse en las indicaciones de la parte inferior para saber si las aristas que está seleccionando son las correctas.

Cuando se hace un click de ratón sobre el botón Kruskal manual, este botón se convierte en el botón Modo edición para poder volver a editar un grafo cuando terminemos el proceso.

Botón Prim manual

Este botón se utiliza, una vez editado el grafo, para que el usuario realice el algoritmo de Prim sobre el grafo que él mismo a creado.

En primer lugar, el usuario debe seleccionar con un click de ratón un nodo de partida.

Después, se debe seleccionar haciendo click de ratón las aristas hasta completar el algoritmo y obtener una solución.

El usuario debe fijarse en las indicaciones de la parte inferior para saber si está haciendo bien el algoritmo de Prim.

Cuando se pulsa el botón Prim manual, este botón se convierte en el botón Modo edición para poder volver a editar un grafo cuando terminemos el proceso.

Botón Kruskal automático

Teniendo el grafo completado, cuando pulsamos este botón, se calcula todas las soluciones óptimas posibles por medio del algoritmo de Kruskal automáticamente, sin intervención del usuario, y cuando se completa el proceso nos aparece una ventana indicándonos el número de soluciones y dándonos las opciones de pasar a la siguiente solución, a la anterior solución o terminar, a no ser que sólo exista una solución y en ese caso sólo aparece la opción de terminar el proceso pulsando el botón de Aceptar.

Botón Prim automático

Teniendo el grafo completado, cuando pulsamos este botón, se calcula todas las soluciones óptimas posibles por medio del algoritmo de Prim automáticamente, sin intervención del usuario, y cuando se completa el proceso nos aparece una ventana indicándonos el número de soluciones y dándonos las opciones de pasar a la siguiente solución, a la anterior solución o terminar, a no ser que sólo exista una solución y en ese caso sólo aparece la opción de terminar el proceso pulsando el botón de Aceptar.

[Botones][Índice]

Botón de Ayuda

Este botón, como su propio nombre indica, nos da información acerca de cómo un usuario puede manejar la aplicación.

Básicamente, se divide en dos partes:

- Una parte nos informa de las funciones que realiza cada botón que aparece en la parte inferior de la pantalla.
- La otra parte hace un seguimiento de los pasos ordenados que tenemos que hacer para manejar correctamente la aplicación.

Lista de Idiomas

Este control es, en realidad, una lista desplegable, que muestra los idiomas que están disponibles en la aplicación.

Al escoger un idioma de la lista, todos los elementos del interfaz de la aplicación quedarán traducidos a dicho idioma (texto de botones, de ventanas, mensajes de la barra de estado y la ayuda).

Los idiomas actualmente disponibles son español e inglés.

Pasos a seguir

[Índice]

I. Edición del grafo.
II. Realización de los algoritmos.

[Pasos a seguir][Índice]

Edición del grafo

Crear nodos

Al iniciar la aplicación, lo primero que hay que hacer son los nodos y por tanto, está activado automáticamente el botón Nodos (si pasamos a otras fases de la aplicación y queremos crear nodos tenemos que activar el botón Nodos con un click de ratón).

Cada vez que hacemos un click de ratón sobre la pantalla de diseño aparece un nuevo nodo. También se puede arrastrar un nodo con el ratón para cambiarlo de sitio.

Crear aristas

Para trazar las aristas entre los nodos que hemos creado, lo primero que hay que hacer es un click de ratón sobre el botón Aristas. Después, tenemos que fijar el valor de la arista en el marco Pesos aristas directamente en la caja de texto Próximo o a través de la pequeña barra horizontal.

Para trazar una arista entre dos nodos, tenemos que hacer un click de ratón sobre uno de ellos y después otro click de ratón sobre el otro nodo.

Si queremos cambiar el valor de alguna arista, hacemos click en el botón Peso; a continuación seleccionamos la arista con un click de ratón sobre ella y nos sale una ventana para que demos un nuevo valor a la arista seleccionada.

Borrar aristas

Para eliminar aristas debemos activar el botón Arista con un click de ratón y a continuación, cada vez que hacemos click con el ratón sobre una arista, ésta desaparece del grafo.

Borrar nodos

Para borrar nodos tenemos que hacer un click de ratón sobre el botón Nodo y a continuación, cada vez que hacemos un click de ratón sobre un nodo, éste desaparece junto con todas las aristas que inciden sobre él.

Deshacer última acción

Cada vez que hacemos un click de ratón sobre el botón Último, desaparece la última acción que se ha hecho al editar el grafo, es decir, si lo último que hemos hecho es crear una arista, desaparecerá dicha arista y si lo último que hemos hecho es crear un nodo, desaparecerá dicho nodo.

Borrar el grafo

Si queremos hacer desaparecer el grafo completo, tenemos que activar el botón Grafo con un click de ratón, a continuación, aparecerá una ventana informativa para asegurarnos de que lo que queremos hacer es eliminar el grafo y así proceder con el criterio adecuado.

Cambiar el color del fondo de la ventana de diseño del grafo

Para terminar la edición, sólo nos queda, si queremos, cambiar el color del fondo sin modificar el color del grafo. Para ello, cada vez que hacemos click sobre el botón Fondo aparece un color de fondo distinto.

El número de colores que puede tomar el fondo de la ventana es finito y se repite cíclicamente.

[Pasos a seguir][Índice]

Realización de los algoritmos

Algoritmo de Kruskal manual

A partir de hacer un click de ratón sobre el botón Kruskal manual, el usuario debe ir seleccionando aristas según el algoritmo de Kruskal para encontrar una solución óptima de todas las posibles soluciones. Estas aristas se seleccionan haciendo un click de ratón sobre ellas.

En la parte inferior de la ventana aparecen mensajes que nos indican cómo se está realizando el algoritmo de Kruskal: si el usuario selecciona la arista correcta, ésta toma el color rojo y aparece un mensaje indicando la correcta elección de la arista. Por el contrario, si no se ha elegido la arista adecuada, aparece un mensaje avisando del tipo de error que se ha cometido y de qué es lo que se tiene que hacer para seleccionar una arista correcta.

Cuando las aristas seleccionadas se completan y se termina el proceso, aparece un mensaje que nos lo indica.

Algoritmo de Prim manual

A partir de hacer un click de ratón sobre el botón Prim manual, lo primero que hay que hacer es seleccionar un nodo cualquiera de partida haciendo un click de ratón sobre cualquier nodo. A continuación, hay que seleccionar aristas según el algoritmo de Prim haciendo un click de ratón sobre cada arista que conforma una solución óptima de entre todas las posibles soluciones óptimas posibles.

En la parte inferior de la ventana aparecen mensajes indicándonos si las acciones que se hacen son las correctas o no, y también se informa, en caso de no seguir correctamente el algoritmo de Prim, de qué es lo que se debe hacer.

Las aristas correctas y los nodos en los que inciden pasan a tener un color rojo y cuando el algoritmo termina, se muestra un mensaje avisándonos.

Algoritmo de Kruskal que calcula todas las soluciones

Cuando hacemos un click de ratón sobre el botón Kruskal automático, el usuario no tiene que seleccionar aristas y es el propio programa el que se encarga de

calcular todas las soluciones siguiendo el algoritmo de Kruskal.

Mientras se calculan las soluciones van apareciendo el número de ellas que ya se han calculado y se pueden mover los nodos del grafo arrastrando con el ratón sobre ellos. También aparece un botón para interrumpir el proceso si tarda demasiado.
Cuando ya se han calculado todas las soluciones, aparece una ventana con varios botones:

- *Siguiente*: nos permite mostrar la siguiente solución.
- *Anterior*: nos permite mostrar la solución anterior.
- *Terminar*: se utiliza para salir de la ventana de soluciones y terminar con el proceso de cálculo de todas las soluciones óptimas mediante el algoritmo de Kruskal.
- *Mostrar/Ocultar detalles*: pulsando este botón, se muestra las coordenadas de las aristas solución para la solución óptima que está reflejada en el grafo, o se oculta estas coordenadas si ya estaban siendo mostradas.

Algoritmo de Prim que calcula todas las soluciones

Cuando hacemos un click de ratón sobre el botón Prim automático, el usuario no tiene que seleccionar ni el nodo inicial ni ninguna arista y es el propio programa el que se encarga de calcular todas las soluciones siguiendo el algoritmo de Prim.

Mientras se calculan las soluciones van apareciendo el número de ellas que ya se han calculado y se pueden mover los nodos del grafo arrastrando con el ratón sobre ellos. También aparece un botón para interrumpir el proceso si tarda demasiado.

Cuando ya se han calculado todas las soluciones, aparece una ventana con varios botones:

- *Siguiente*: nos permite mostrar la siguiente solución.
- *Anterior*: nos permite mostrar la solución anterior.
- *Terminar*: Se utiliza para salir de la ventana de soluciones y terminar con el proceso de cálculo de todas las soluciones óptimas mediante el algoritmo de Prim.
- *Mostrar/Ocultar detalles*: pulsando este botón, se muestra las coordenadas de las aristas solución para la solución óptima que está reflejada en el grafo, o se oculta estas coordenadas si ya estaban siendo mostradas.