En la ciudad de Hermosillo, Sonora, México siendo las 12:00 horas del día 1º de Octubre de 1993, se reunieron en el aula 9K-205 del Departamento de Matemáticas de la Universidad de Sonora los señores:

Pedro Flores Pérez
Pablo Barrera Sánchez
Héctor A. Hernández Hdez

bajo la Presidencia del primero y fungiendo como Secretario el último para efectuar el examen profesional de la carrera de:

Acta No. 81
Hoja No. 80
Libro No. 01
Exp. No. 8910476-3

Licenciado en Matemáticas

a la señorita: Edelmira Rodríguez Alcantar

Después de haber presentado su tesis intitulada: "Los Problemas de Arbol de Mínima Expansión en Gráficas y Ruta Mínima en Digráficas" la que previamente le fué aprobada por el jurado, los señores sinodales replicaron al sustentante y después de debatir entre sí, secreta y libremente la declararon:

APROBADO POR UNANIMIDAD

Firma de la Sustentante

Acto continuo el presidente del Jurado le hizo saber el resultado de su examen y para constancia se levanta la presente y firman los que han intervenido:

MATS.

Universidad de Sonora

Departamento de Matemáticas

Tesis

Los problemas de: Arbol de mínima expansión en gráficas y ruta mínima en digráficas.

Que para obtener el titulo de

Licenciado en Matemáticas

Presenta

Edelmira Rodríguez Alcántar

Hermosillo, Sonora, 1 de Octubre de 1993

A mis padres.

A mis hermanos.

A mis amigos.

Agradecimientos

Quiero agradecer infinitamente a mis padres por la oportunidad que me brindaron, así como a mi maestro y director: Pedro Flores Pérez, quién no sólo dirigió con mano firme este trabajo sino que, además, trabajó a la par conmigo hasta verlo terminado.

Contenido

Introducción 4

1 Arbol de mínima expansión en gráficas **5**

1.1 Problema 5

1.2 Descripción del problema con elementos de gráficas 8

1.2.1 Propiedades de los árboles 9

1.3 Algoritmo de Kruskal 24

1.3.1 Justificación teórica del algoritmo 25

1.3.2 Descripción de la implementación con estructuras de datos 26

1.3.3 Ejemplo 27

1.4 Algoritmo de Prim 41

1.4.1 Justificación teórica del algoritmo 41

1.4.2 Descripción de la implementación con estructura de datos 42

1.4.3 Ejemplo 43

2 Ruta más corta en digráficas **49**

2.1 Descripción del problema con elementos de digráficas 52

2.2 Algoritmo de Dijkstra para costos no negativos 55

2.2.1 Justificación teórica del algoritmo 57

2.2.2 Descripción de la implementación con estructuras de datos 58

2.2.3 Ejemplo: 60

2.3 Caso General 65

2.3.1 Justificación teórica del algoritmo 67
2.3.2 Implementación con estructuras de datos 68
2.3.3 Ejemplo: 70
2.4 Algoritmo de Floyd 74
2.4.1 Justificación teórica del algoritmo. 76
2.4.2 Implementación con estructuras de datos 76
2.4.3 Ejemplo 78

Conclusiones **83**

A Algoritmos para gráficas **85**
A.1 Formato en los archivos 85
A.2 Algoritmo de Prim 87
A.3 Algoritmo de Kruskal 95

B Algoritmos para digráficas **109**
B.1 Formato en los archivos 109
B.2 Algoritmo de Dijkstra General 112
B.3 Algoritmo de Floyd 149

Bibliografía **165**

Introducción

El constante y desmesurado aumento de la población ha provocado el surgimiento de nuevos problemas y acentuado otros ya viejos. Algunos ejemplos de estos problemas son los derivados de: Planeación de tráfico urbano, lineas de comunicaciones, tuberías, oleoductos y transporte colectivo entre otros.

La necesidad de resolver estos problemas, o cuando menos disminuir su impacto, ha provocado que se intente resolverlos aplicando diversas técnicas, como por ejemplo: Programación Lineal, Programación Dinámica, Métodos Eurísticos y Teoría de Redes.

Debido al éxito de esta última nos hemos interesado en los resultados obtenidos al modelar el problema como una red y resolverlo con los algoritmos específicos del área. Algunos de estos problemas se han convertido en clásicos dentro de esta teoría y en este trabajo se resuelven dos de ellos.

Los problemas que resolveremos aquí son: encontrar un árbol de peso mínimo en gráficas y encontrar una ruta mínima en digráficas; para esto revisaremos los aspectos teóricos que justifican los algoritmos que resuelven estos problemas así como daremos una propuesta de implementacional para los mismos. Con esto, pretendemos que este trabajo sirva como material de apoyo a los interesados en la implementación computacional de modelos de redes pues existe poca difusión sobre este aspecto.

Aunado a esto, como los problemas reales que se modelan mediante redes son de tamaño considerable por lo que se tiene que recurrir a implementaciones computacionales, que no son triviales, de los algoritmos con lo cual se logra resolverlos con precisión y rápidez.

Para la implementación computacional utilizamos estructuras de datos y apuntadores pues facilitan algunas operaciones y además de que reducen el tiempo de cómputo, factor importante cuando se trata de problemas grandes.

En el capítulo 1 veremos cómo obtener un árbol de mínima expansión en gráficas mediante dos algoritmos: el de Kruskal y el de Prim. Antes de la presentación de los algoritmos se describe el problema y se enuncian algunas propiedades de los árboles.

En el capítulo 2 se resuelve el problema de encontrar una ruta más corta entre dos nodos específicos y entre un nodo dado y los restantes de una digráfica. Para resolver estos problemas se utilizan dos algoritmos de Dijkstra: uno para redes con costos en los arcos no negativos y otro

aplicable a cualquier red. También se presenta el problema de encontrar una ruta más corta entre todo par de nodos en una digráfica para lo cual se presenta el algoritmo de Floyd que puede aplicarse a redes con costos reales.

Todos estos algoritmos tienen una característica común: son del tipo glotón, esto es, toman los arcos o aristas de menor costo en cada iteración.

Se anexan los códigos de la implementación, en lenguaje C, de los algoritmos mencionados. Estan divididos en dos partes al igual que este trabajo: los que resuelven el problema planteado en gráficas y los que resuelven los problemas planteados en digráficas. Cada uno de los anexos tiene un pequeño instructivo sobre el manejo de archivos necesario para el problema y la solución, así como un ejemplo resuelto para ilustrar.

Los conceptos básicos de Teoría de Gráficas aquí manejados pueden consultarse en [2].

Capítulo 1

Arbol de mínima expansión en gráficas

En este capítulo se aborda el problema de encontrar el árbol de mínima expansión en una gráfica.

Para esto, en la primera sección se presentan algunos ejemplos del tipo de problemas que podremos resolver encontrando el árbol de mínima expansión en una gráfica. Mostrando, además, la que modela el problema planteado.

La descripción del problema con elementos de teoría de gráficas queda cubierta en la sección dos, donde se concluye que la solución a nuestro problema se obtiene al encontrar el árbol de mínima expansión de la gráfica. Con el fin de encontrar el árbol de mínima expansión se presentan en las secciones tres y cuatro, respectivamente, los algoritmos de Kruskal y Prim, así como su justificación teórica, la descripción de su implementación con estructuras de datos y un ejemplo resuelto a detalle.

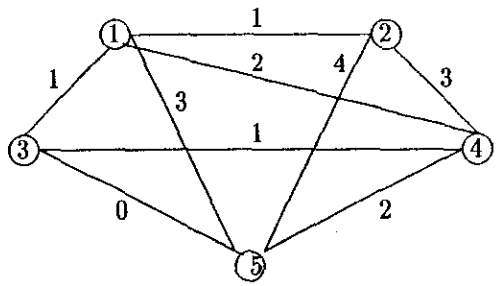
1 Problema

1. El señor Zazueta desea establecer un negocio de entrega de paquetería a cinco ciudades del sur del estado: Cd. Obregón, Hermosillo, Navojoa, Guaymas y Alamos. Existen diferentes maneras de transportar los paquetes entre estas ciudades, cada una con un costo de franquicia. El problema que tiene el Sr. Zazueta consiste en escoger de entre los caminos posibles aquellos que le permitan los envíos entre todas de tal manera que pague lo menos posible en franquicia.

   La siguiente tabla muestra los distintos caminos que existen y su costo. Las ciudades entre las cuales no hay camino se representan en la tabla con un guión.

	Hermosillo	Obregón	Navojoa	Alamos	Guaymas
Hermosillo	—	1	—	4	3
Obregón	1	—	1	3	2
Navojoa	—	1	—	0	1
Alamos	4	3	0	—	2
Guaymas	3	2	1	2	—

Esta gráfica modela el problema:

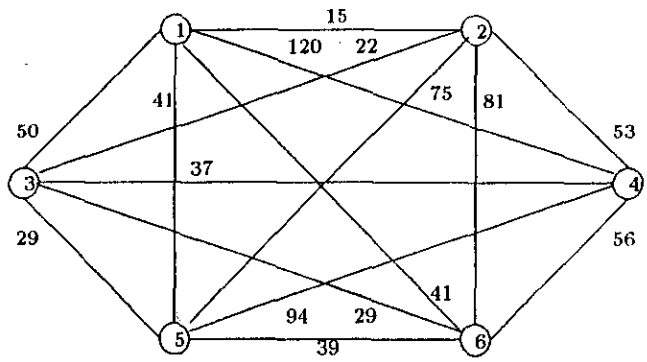


El nodo 1 representa a Hermosillo, el 2 a Cd. Obregón, el 3 a Navojoa, el 4 a Alamos y el 5 a Guaymas, el número asociado a las aristas representa el costo del camino.

2. Una compañía telefónica quiere introducir sus lineas en seis ciudades de tal manera que se minimize el cable usado. Debe considerarse que, una vez puestas las lineas en una ciudad, pueden tenderse de esta a cualquier otra. El cable necesario entre las ciudades se muestra en la siguiente tabla:

	1	2	3	4	5	6
1	—	15	50	120	41	12
2	15	—	22	53	75	81
3	50	22	—	37	46	29
4	120	53	37	—	94	56
5	41	75	46	94	—	39
6	12	81	29	56	39	—

Esta gráfica modela el problema:



A continuación modelaremos estos problemas para resolverlos con teoría de gráficas.

1.2 Descripción del problema con elementos de gráficas

Sea $G = [X, A, C]$ una red, donde $C : A \rightarrow R$. Con esto vamos a modelar el siguiente problema:

Tenemos un conjunto de n ciudades denotadas $x_1, x_2, ..., x_n$ y un conjunto de caminos que las comunican. El transporte por el camino que comunica a la ciudad x_i con la x_j tiene un costo de C_{ij} pesos. El problema consiste en encontrar la manera en que exista transportación entre todas las ciudades de la forma más barata posible.

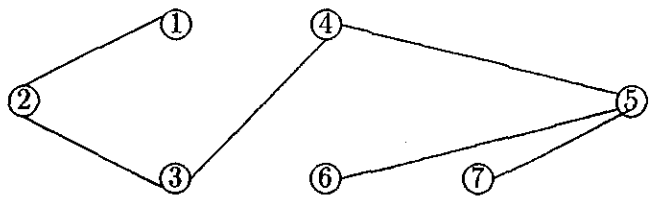
En nuestro modelo el conjunto de nodos X representará el conjunto de ciudades y cada elemento (x_i, x_j) del conjunto de aristas A, representa un camino en la red de una ciudad x_i a una ciudad x_j. La función C será llamada función de costo y asociará a cada arista (i, j) un costo, C_{ij}, de transportación. El problema planteado se resuelve con una gráfica parcial $T = [X, A']$ de G, que además, deberá cumplir los siguientes tres puntos.

- Deberá existir una cadena que una a todo par de nodos, por lo tanto, T deberá ser conexa.
- El costo de transportación, total, deberá ser mínimo.
- T no deberá tener ciclos puesto que, de ser así, se incurrirá en un costo innecesario.

Afirmamos que la solución al problema es un árbol expandido de costo minímo de la gráfica. Para demostrarlo vamos a revisar la definición de árbol expandido así como la de costo de un árbol.

Definición 1.1 *Un árbol es una gráfica $T = [X, A]$ conexa y acíclica.*

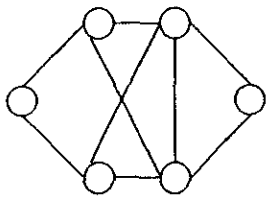
Ejemplo:



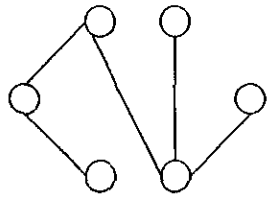
Arbol

Definición 1.2 *Sea $G = [X, A]$ una gráfica. Un árbol expandido de G es una gráfica parcial $T = [X, A']$, de G, que es un árbol.*

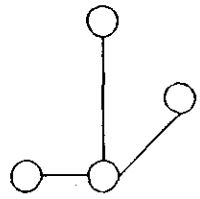
Ejemplo:



Gráfica G



Arbol expandido de G



Arbol no expandido de G

Así, la gráfica T que será solución para el problema de las ciudades debe ser un árbol expandido de G. Nótese que existen varios árboles expandidos de una misma gráfica, es decir, hay varias posibles soluciones al problema de las ciudades. Nos interesa, sin embargo, aquella donde el costo de transportación sea lo más barato posible. Esto nos obliga a definir el costo de un árbol.

Definición 1.3 *El costo de un árbol es la suma de los costos de las aristas que lo forman.*

Basándonos en estas definiciones vemos que, en efecto, la solución óptima al problema de las ciudades está dada por el árbol expandido de costo mínimo asociado a la gráfica G.

El tema siguiente, Propiedades de los Arboles, nos será de suma utilidad para justificar los algoritmos de obtención del árbol de mínima expansión de una gráfica que se presentarán después.

1.2.1 Propiedades de los árboles

Existen varias definiciones de árboles, todas ellas equivalentes entre si. Antes de presentarlas, y para demostrar su equivalencia, debemos analizar ciertas propiedades elementales de una gráfica. La primera que veremos nos muestra lo que puede pasar al añadir una arista a una gráfica con varias componentes conexas.

Proposición 1.1 *Sea $G = [X, A]$ una gráfica. Si agregamos la arista (i, j), tal que $(i, j) \notin A$, entonces*

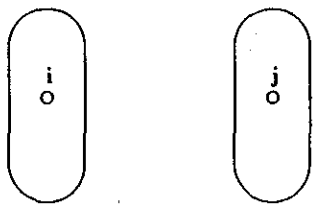
1. *Si i y j están en dos componentes conexas de G distintas se tiene que el número de componentes conexas disminuye en uno; además, no hay ningún ciclo que contenga a (i, j) en $G' = [X, A \cup \{(i, j)\}]$.*
2. *Si se agrega la arista a una misma componente conexa el número de éstas no se altera y además aparece un ciclo que contiene a (i, j) en $G' = [X, A \cup \{(i, j)\}]$.*

Demostración:

1. Supongamos que G consiste de dos componentes conexas, G_1 y G_2, es decir $G = G_1 \cup G_2$. Sean

$$G_1 = [X_1, A_1] \quad y \quad G_2 = [X_2, A_2]$$

Supongamos, además que $i \in G_1$ y $j \in G_2$

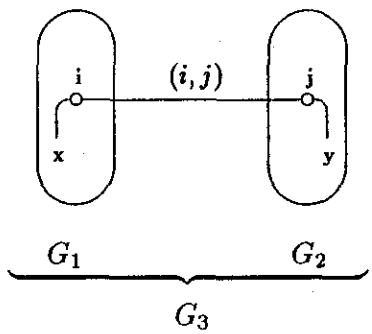


tomemos

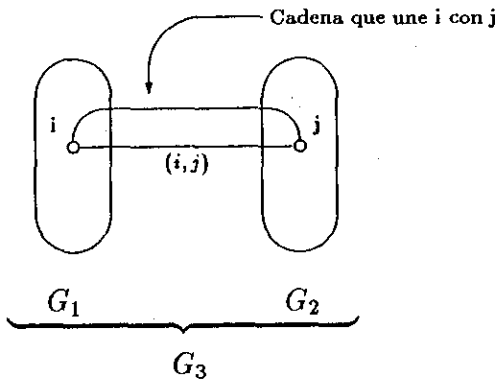
$$G_3 = [X_1 \cup X_2, A_1 \cup A_2 \cup \{(i,j)\}]$$

Afirmamos que G_3 es una componente conexa de G'

Efectivamente, para cualquiera nodos x y y en este conjunto se tiene que, si $x, y \in X_1$ ó $x,y \in X_2$ existe una cadena, $x = i_1, ..., i_k = y$, que los conecta. Cuando $x \in X_1$ (X_2) y $y \in X_2$ (X_1) existe una cadena, $x = i_1, ..., i_k = i$ $(y = i_1, ..., i_k = i)$, que conecta a x (y) con i y una cadena, $j = l_1, ..., l_k = y$ $(j = l_1, ..., l_k = x)$ que conecta a j con $y(x)$; si a estas cadenas agregamos la arista (i,j) se obtiene una cadena, $x = i_1, ..., i_k = i, j = l_1, ..., l_k = y$ $(y = i_1, ..., i_k = i, j = l_1, ..., l_k = x)$, en G_3 que conecta a x con y (y con x) en G_3 y por lo tanto en G'.

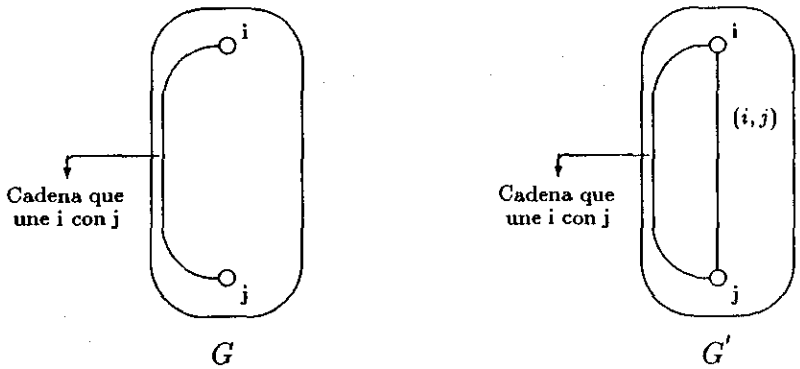


Si la arista (i,j) estuviera contenido en un ciclo en G', al quitarla, quedaría una cadena que conecta a i y j, lo cual no es posible pues i y j están en componentes conexas distintas.



Por lo tanto (i,j) no puede estar contenido en un ciclo.

2. Si i y j están en la misma componente este conjunto sigue siendo conexo y además componente conexa. Además, existe una cadena que une i con j por lo que, al agregar a esta cadena la arista (i,j), aparece un ciclo.



■

El número de aristas, en un árbol es igual, al número de nodos menos uno, para demostrarlo presentamos la siguiente proposición.

Proposición 1.2 *Sea $G = [X, A]$ una gráfica con n nodos y m aristas.*

1. *Si G es conexa $m \geq n - 1$.*
2. *Si G es acíclica $m \leq n - 1$.*

Demostración:

Tomemos la gráfica $G' = [X, \phi]$ que consiste de n componentes conexas (pues no hay aristas que conecten los nodos). Vamos a construir la gráfica $G = [X, A]$ agregando una a una las aristas a la gráfica G'.

1. Nótese que cada vez que se agrega una arista, disminuye en uno el número de componentes conexas (en el mejor de los casos) por lo que concluimos que si G es conexa el número de aristas que hay que agregar es mayor o igual a $n-1$.

2. Si G es acíclica toda gráfica parcial de G es acíclica. Entonces cada vez que se agrega una arista, el número de componentes conexas disminuye en uno (para que siga siendo acíclica). Dado que G tiene al menos una componente conexa, se concluye que el número de aristas que hay no debe superar las $n-1$.

■

Al ser un árbol una gráfica conexa y acíclica se concluye el resultado anticipado de que el número de arcos es uno menos que el número de nodos.

Contamos ya con la herramienta necesaria para enunciar las distintas definiciones de árbol y demostrar su equivalencia.

Proposición 1.3 *Sea $G = [X, A]$ una gráfica con n nodos ($n \geq 2$) y m aristas. Entonces los siguientes postulados son equivalentes:*

1. *G es conexa y acíclica.*

2. *G es acíclica y tiene $n-1$ aristas.*

3. *G es acíclica y si se agrega una arista, se forma un único ciclo.*

4. *G es conexa y tiene $n-1$ aristas.*

5. *G es conexa y deja de serlo si se quita alguna arista.*

6. *Existe en G una única cadena entre cada par de nodos.*

Demostración:

La demostración será en el siguiente orden: $1 \Rightarrow 4 \Rightarrow 2 \Rightarrow 3 \Rightarrow 5 \Rightarrow 6 \Rightarrow 1$.

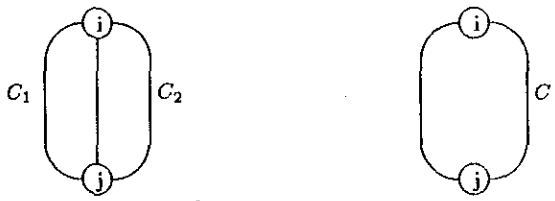
($1 \Rightarrow 4$) Trivial. (Proposición 1.2.)

($4 \Rightarrow 2$) Trivial. (Proposición 1.2.)

($2 \Rightarrow 3$) Lo haremos por contradicción. Supongamos que se agrega la arista (x, y) a G. Sea $G' = [X, A']$ donde $A' = A \cup \{(x, y)\}$. Entonces G' tiene al menos un ciclo, por la proposición 1.2 y el hecho de tener n aristas. Supongamos que éste ciclo no es único, es decir, que se formaron dos ciclos al menos. Sean éstos $C_1 : (x = i_1, ..., i_k = y, x)$ y $C_2 : (x = j_1, ..., j_r = y, x)$; entonces el ciclo

$$C : (x = i_1, ..., i_k = y, j_r, j_{r-1}, ..., j_1 = x)$$

es un ciclo que no contiene a la arista (x, y) y que, además, está contenido en G. Esto es una contradicción pues G es acíclica por hipótesis. Por lo tanto G tiene exactamente un ciclo.



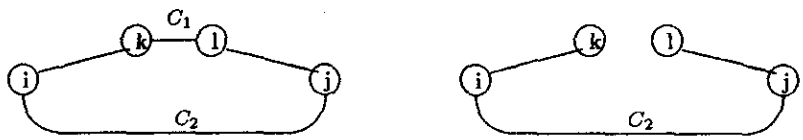
(3 $\Rightarrow$ 5) Vamos a demostrar que G es conexa. Supongamos que G no es conexa. Así por lo menos tiene dos componentes conexas y al agregar una arista que las una no aparece un ciclo (proposición 1.1). Esto contradice la hipótesis. Por lo tanto G es conexa. Como G es acíclica, por hipótesis, y ahora conexa, tiene $n-1$ aristas por lo que, al quitar una deja de ser conexa (por la proposición 1.2).

(5 $\Rightarrow$ 6) Vamos a demostrar que la cadena es única.

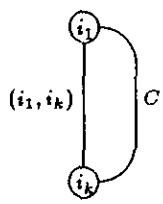
Como G es conexa existe, al menos, una cadena entre todo par de nodos. Supóngase que entre los nodos x y y existen, al menos, dos cadenas. Sean C_1 y C_2 dos de tales cadenas. Sea (k, l) una arista de C_1 que no está en C_2. Entonces la gráfica

$$G' = [X, A - \{(k,l)\}]$$

es conexa. Esto es una contradicción con la hipótesis. Por lo tanto, la cadena que une a un par de nodos es única.



(6 $\Rightarrow$ 1) El que G sea conexa se sigue de la hipótesis. Ahora vamos a demostrar que G es acíclica. Supongamos que G tiene un ciclo. Sea éste $(i_1, ..., i_k, i_1)$. Entonces existen dos cadenas entre i_1 y i_k, a saber: $C : (i_1, ..., i_k)$ y el arco (i_k, i_1). Esto contradice la hipótesis por lo tanto, G es acíclica.



■

Existen dos operaciones en los árboles que nos resultarán especialmente importantes, estas son:

- Agregar un nodo y conectarlo por medio de una arista,
- eliminar un nodo con una sola arista adyacente y esta arista.

En los dos algoritmos presentados en este capítulo se hace uso de estas propiedades. Es el momento de presentar el concepto de nodo pendiente de los cuales, se demuestra que un árbol tiene al menos dos.

Definición 1.4 *Sea $G = [X, A]$ una gráfica. Sea x un nodo de G. Se dice que x es un nodo pendiente si su grado es uno.*

Notemos que en las dos operaciones mencionadas, el concepto de nodo pendiente es fundamental: Al agregar un nodo y conectarlo por una arista en realidad estamos agregando un nodo pendiente a la gráfica; del mismo modo al quitar un nodo con una única arista y dicha arista se está quitando de la gráfica un nodo pendiente. De aquí que en adelante este concepto será, como ya dijimos, fundamental.

Proposición 1.4 *Si $G = [X, A]$ es una gráfica acíclica entonces*

1. *O todos sus nodos tiene grado cero, o*
2. *existe, por lo menos, un nodo pendiente.*

Demostración:

Para la demostración partiremos el conjunto de nodos en dos partes: uno que contenga a los nodos de grado cero y en otro el resto. Llamemos A al primero y B al segundo, es decir,

$$A = \{x \in X | g(x) = 0\} \quad y \quad B = \{x \in X | g(x) \neq 0\}$$

1. Si el conjunto $B = \phi$, el resultado se obtiene.
2. Supongamos que $B \neq \phi$. Sea n el número de nodos en B.

   Obsérvese que la suma de los grados en una gráfica, es dos veces el número de aristas puesto que cada arista contribuye dos veces a la suma. Así en una gráfica acíclica, donde el número de aristas es a lo más $n - 1$, la suma de los grados de los nodos es menor o igual a $2(n - 1)$, es decir, $\sum_{i \in X} g(i) \leq 2(n - 1) < 2n$. Si en B no existen nodos pendientes, de grado uno, entonces el grado de todos de los nodos de B es mayor o igual a 2, y se tendría que

$$\sum_{i \in X} g(i) \geq 2n. \quad (\Rightarrow\Leftarrow)$$

   De donde concluimos que B debe tener por lo menos un nodo pendiente.

■

Proposición 1.5 *Sea $G = [X, A]$ un árbol con n nodos, donde $n \geq 2$. Entonces G tiene, al menos, dos nodos pendientes.*

Demostración:

Por ser G un árbol, es conexa y acíclica, es decir

$$\sum_{i \in X} g(i) = 2(n-1).$$

Pasemos a demostrar que G tiene dos nodos pendientes, para esto, suponer ahora que todos los nodos de G, excepto uno (por la proposición 1.4), tienen grado mayor que uno, es decir

$$g(j) = 1 \quad \textit{para algún } j \in X$$

$$g(i) \geq 2 \quad \forall i \in X, i \neq j$$

Así

$$\sum_{i \in X} g(i) \geq 2n - 1 > 2n - 2. \quad (\Rightarrow\Leftarrow)$$

Por lo tanto, G tiene, al menos, dos nodos pendientes.

■

En la siguiente proposición se demuestra que un árbol conserva sus propiedades bajo las operaciones mencionadas anteriormente.

Proposición 1.6 *Sea $G = [X, A]$ un árbol. Entonces:*

1. *Si x es un nodo pendiente de G y a es la única arista adyacente a él, la gráfica $G' = [X - \{x\}, A - \{a\}]$ es un árbol.*
2. *Si se agrega un nodo x a G y se conecta a G por medio de una arista, a, la gráfica resultante, $G' = [X \cup \{x\}, A \cup \{a\}]$ es un árbol.*

Demostración:

1. Por construcción de G' el número de aristas que contiene es igual al número de aristas de G menos uno y el número de nodos es el número de nodos de G menos uno. Además G' es conexa, pues G lo es. Así, por la proposición 1.3, G' es un árbol.
2. El razonamiento es análogo a la demostración de (1).

■

Obsérvese que al quitar un nodo pendiente solo baja el grado del único nodo conectado con él.

Proposición 1.7 *Si $G = [X, A]$ es una gráfica acíclica con n nodos y se eliminan sucesivamente los nodos pendientes, entonces, se llega a una gráfica donde todos sus nodos tienen grado cero.*

Demostración:

(Vamos a demostrarlo por inducción en el número de nodos)

Sea k el número de nodos.

- k=1. Trivial.
- Supongámoslo válido para k=n−1
- Supongamos que se tiene una gráfica con n nodos; si tiene un nodo pendiente, se elimina y nos queda una gráfica acíclica con $n-1$ nodos. Por hipótesis de inducción puedo llevar a un conjunto de nodos de grado cero.

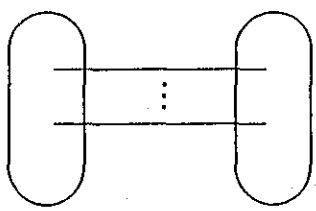
■

La siguiente proposición caracteriza un ciclo.

Proposición 1.8 *Si $G = [X, A]$ es una gráfica con un solo ciclo al eliminar sucesivamente los nodos pendientes se llega a una gráfica cuyos nodos tienen grado cero o grado dos y todos los nodos de grado dos forman el ciclo.*

Demostración:

Dividamos el conjunto de nodos en dos partes: los que forman parte del ciclo y lo que no forman parte de él.



G_1 G_2

Sea G_1 la gráfica que se obtiene al tomar el conjunto de nodos que no forman parte del ciclo y las aristas que los conectan, y G_2 el conjunto de nodos que forman parte del ciclo junto con las aristas que lo forman. Observe que los nodos en G_1 pueden conectar a un solo nodo en G_2, porque existe un solo ciclo (el de G_2). Además la gráfica de G_1 es acíclica.

Al eliminar sucesivamente los nodos pendientes, encontramos que, en G_1 solo hay nodos de grado cero y en G_2 los nodos tienen grado dos. (Los nodos de grado cero se obtienen cuando la gráfica no es conexa.)

■

El resultado anterior es base para construir el algoritmo que usaremos para detectar ciclos.

Algoritmo para encontrar ciclos

El procedimiento a seguir será cortar sucesivamente los nodos pendientes (de los cuales siempre habrá al menos uno, a menos que la gráfica se reduzca a un ciclo). La proposición 1.8 nos garantiza que con este procedimiento detectaremos el ciclo (en caso de existir).

Primeramente, mostraremos la estructura de datos utilizada en este algoritmo. Debemos aclarar que se partirá de una gráfica dada y a partir de ella se buscará un ciclo.

La gráfica la vamos a guardar con listas enlazadas: en una lista, ordenada por el número del nodo, guardaremos la información referente al mismo. Cada elemento de la lista tendrá los siguientes datos:

- **Número de nodo.**
- **Grado del nodo,** es el número de aristas conectadas con él,
- **Copia del grado,** este dato no es esencial en el calculo del ciclo pero se utiliza posteriormente en el algoritmo de Kruskal.

La estructura usada para los nodos se muestra a continuación:

Nodo Inicial	Grado	Copia del grado

La copia del grado de los nodos nos será útil en el Kruskal pues, como el grado bajará al buscar el ciclo, debemos guardar el grado original para recuperarlo con facilidad al final del proceso.

Las aristas conectadas con un nodo, también estarán guardadas en listas ordenadas por el número del nodo extremo de la arista. Cada miembro de estas listas tendrá lo siguiente:

- **número del nodo extremo de la arista,**
- **costo de la arista,**
- **etiqueta,** puede tomar uno de estos valores: 0, 1 ó 2.

El 0 indicará que la arista está en la solución (esta la usaremos en el Kruskal),

el 1 indicará que la arista no está en la solución y

el 2 es una marca temporal útil en la búsqueda de ciclos.

La estructura usada para las aristas se ilustra aquí.

Nodo final	Costo	Etiqueta

Los nodos tienen además dos apuntadores: uno al siguiente nodo en la lista y otro a la lista de aristas conectadas con él. Las aristas tienen un apuntador al siguiente en la lista. Estos apuntadores, aunque no los representaremos explícitamente, deben tenerse presentes.

En este procedimiento se cortan los nodos pendientes, junto con la arista que lo conecta a la gráfica. Si quedan aristas en la gráfica solución es que se formó un ciclo.

---

1. Sea C_k el conjunto de nodos que inicialmente son pendientes.
2. Sacar un elemento de C_k.
   (a) Si x tiene grado uno guardar, en s, el número del nodo extremo y eliminar la arista que lo conecta a la gráfica. Con s, ir a 2(a).
   (b) Si el grado no es uno, ir a 2.
3. Se revisan los nodos, si alguno tiene grado distinto de cero es que se formó un ciclo.

Revisemos este algoritmo paso por paso, mencionando las estructuras de datos que usaremos en su implementación.

1. Sea C_k el conjunto de nodos que inicialmente son pendientes.

   Revisar el grado de los nodos. Los que tengan grado uno entran en C_k. Este conjunto, C_k, lo manejaremos, por facilidad, como una pila.

2. Sacar un elemento de C_k.

   Sacar un elemento de la pila, llamarlo x. Es importante, al momento de sacarlo, revisar si aún tiene grado uno pues, en una iteración anterior pudimos haberlo disminuido.

(a) Si tiene grado uno guardar, en s, el número del nodo extremo y eliminar la arista que conecta a x con la solución. Con s, ir a 2 (a).

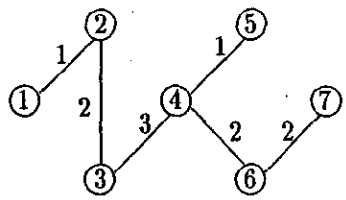
Si x tiene grado uno, debemos quitar la arista que lo conecta con el conjunto solución. Para esto, bajamos el grado de x en uno y a esta arista la marcamos con un 1. Buscar el nodo extremo de s, y, e ir a 2(a).

(b) Si el grado no es uno, ir a (2).

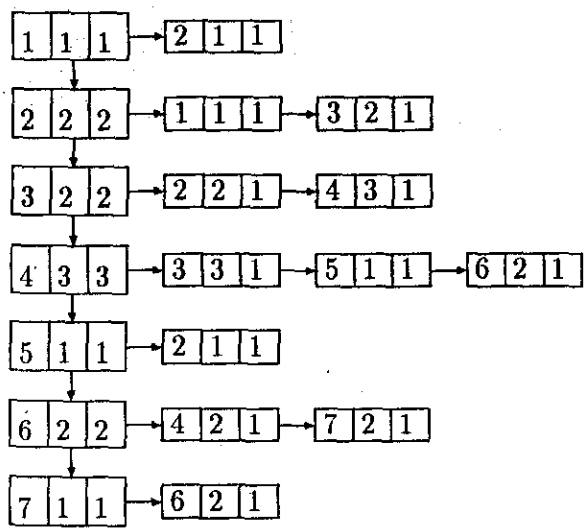
3. Se revisan los nodos, si alguno tiene grado distinto de cero es que se formó un ciclo.

Presentaremos el siguiente ejemplo para ilustrar el procedimiento.

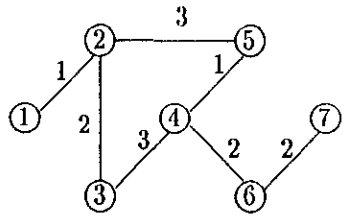
Considere la siguiente gráfica acíclica



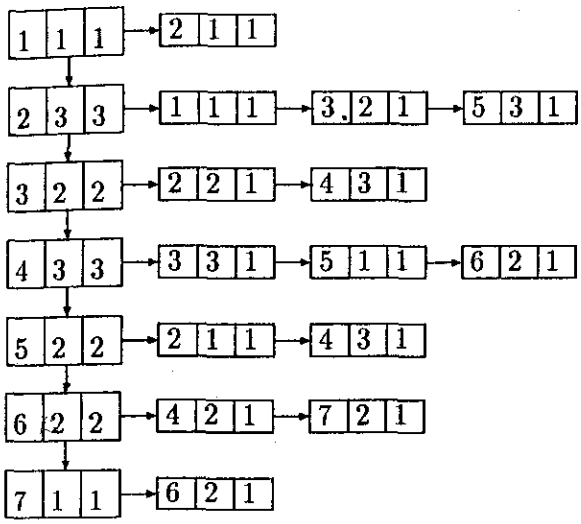
Esta representada por:



Agregando la arista $(2,5)$ a G, obtenemos.

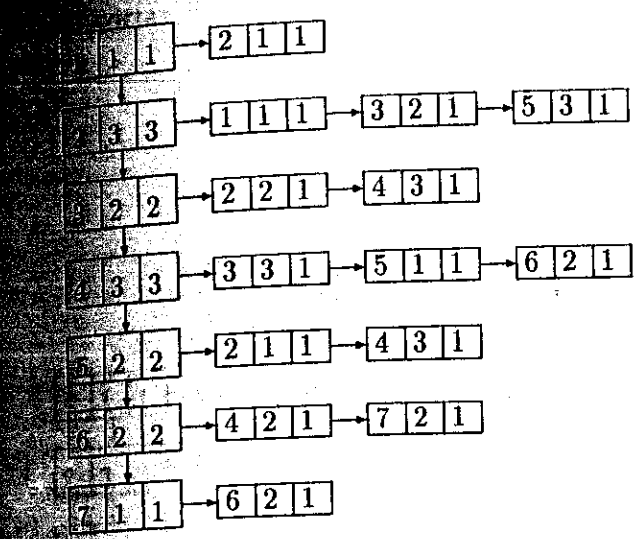


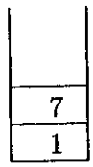
$G' = [X, A \cup \{(2,5)\}]$



Como podemos ver en la ilustración se ha formado un ciclo. Vamos a detectarlo cortando nodos pendientes.

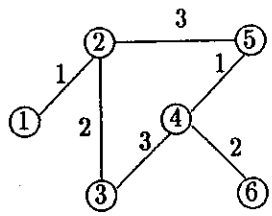
Primero hacemos una pila, donde guardaremos los nodos de grado uno: 1 y 7.

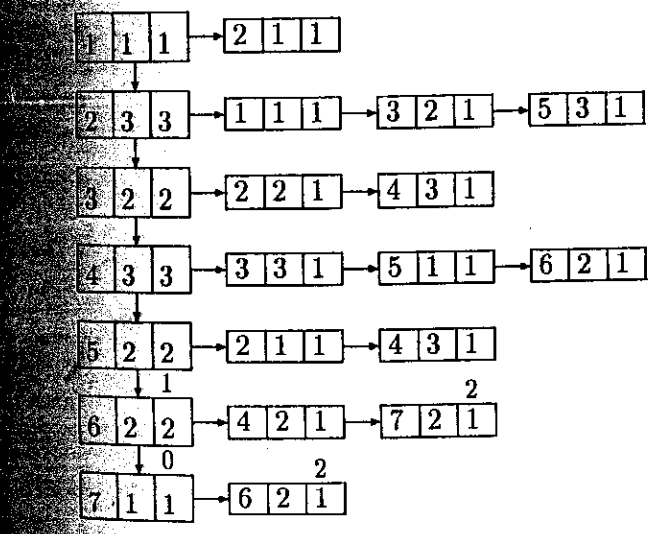


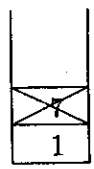


Sacamos un elemento de la pila: el 7. Quitamos este nodo de la gráfica junto con la arista que lo conecta.

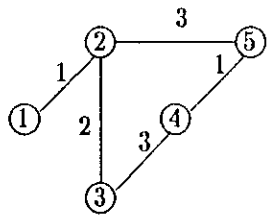
Los cambios que se hagan en cada iteración se indicarán poniéndolos sobre la casilla correspondiente.

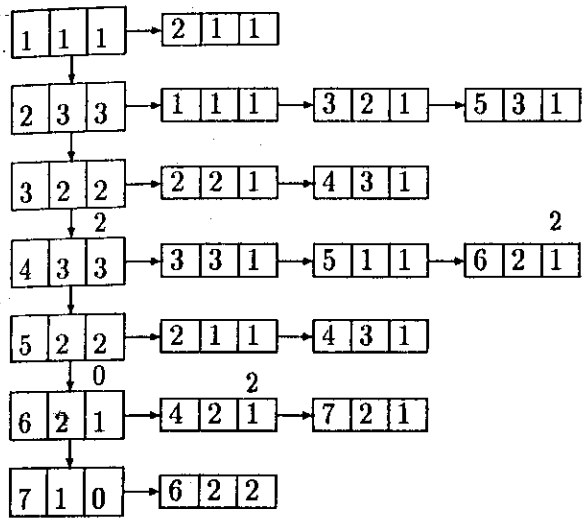




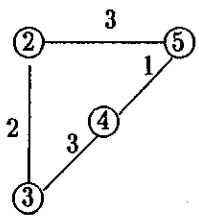


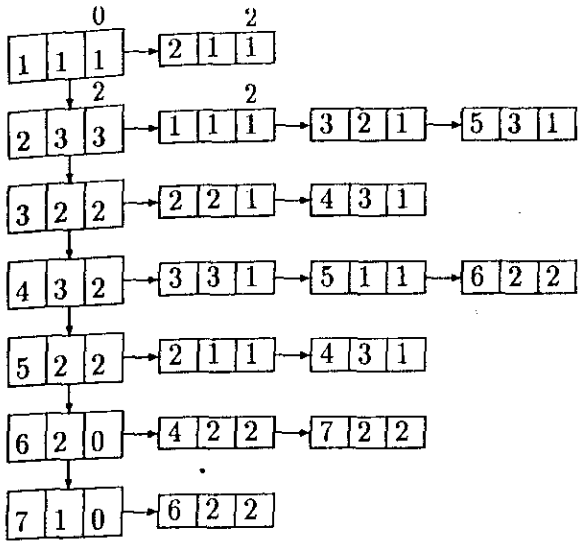
Al cortar el nodo 7 el 6 se convierte en nodo pendiente y se corta da la gráfica junto con su arista.

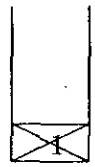




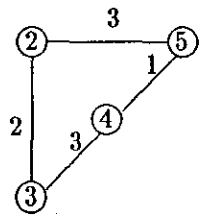
Como el 4 no es pendiente, sacamos de la pila otro nodo: el 1. Cortamos el 1 de la gráfica junto con la arista con la que está conectado.

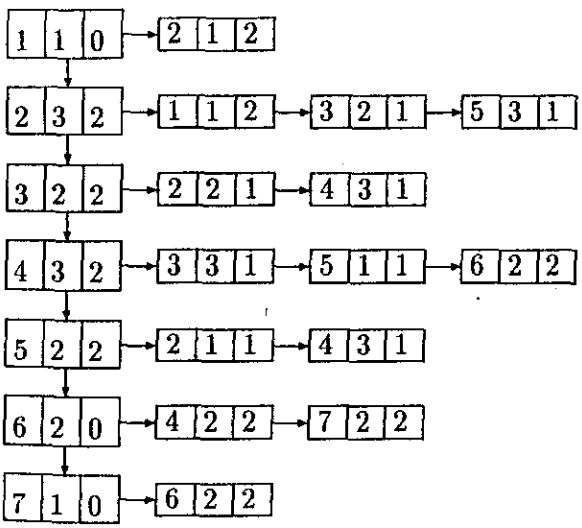


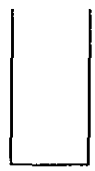




El nodo 2 no se convierte en pendiente al cortar el 1, por lo tanto, sacamos otro de la pila: está vacía.







Los nodos que quedaron en la gráfica tienen grado temporal igual a cero o a dos; lo cual indica que se ha detectado un ciclo.

Considerando el problema original, el de encontrar el árbol expandido de costo mínimo asociado a una gráfica cuyas aristas tengan un costo asociado, debemos hacer notar que, para que el problema tenga solución, la gráfica deberá ser conexa. Cuando una gráfica cumple con esta característica podemos concluir la presencia de, al menos, un árbol expandido, aún más de un número finito de ellos, por lo cual podemos encontrar uno de costo mínimo. Esto se resume en la siguiente proposición.

Proposición 1.9 *Si $G = [X, A]$ es una gráfica conexa entonces existe un árbol de espansión para G y, por lo tanto, solución al problema del árbol de mínima expansión.*

Demostración: Si no existe una arista $a \in A$, tal que la gráfica $G' = [X, A - \{a\}]$ es conexa, entonces G es un árbol expandido. En caso de existir esta arista, considérese ahora la gráfica conexa G' en vez de G y repítase esta operación.

■

Presentaremos dos métodos de solución para el problema de encontrar el árbol expandido de costo mínimo en una red conexa $R = [X, A, C]$ con n nodos: el de Kruskal y el de Prim.

1.3 Algoritmo de Kruskal

En éste algoritmo primero se ordenan las aristas, por el costo, de menor a mayor. En este orden se revisan las aristas, si la arista considerada no forma ciclo con las que ya estaban en la solución entra en ella. Así se crea una gráfica acíclica aunque no necesariamente conexa. Cuando en la solución están consideradas $n - 1$ aristas el algoritmo termina. Se garantiza así la creación de un árbol expandido de G. En la última iteración la gráfica generada es conexa.

Este es el algoritmo de Kruskal para determinar el árbol de costo mínimo de una gráfica.

Sea $G = [X, A]$ una gráfica conexa con n nodos y m aristas. Sea j el número de iteraciones, k el número de aristas en la solución y A' el conjunto de aristas que están en la solución.

1. Ordénese el conjunto de aristas de manera creciente con respecto a la función de costo. Sean $a_1, ..., a_m$ las aristas ordenadas.
   Hacer $k = j = 0$ y $A' = \phi$.
2. Hacer $j = j + 1$. Elegir a_j, la arista de menor costo. $A' = A' \cup a_j$. Si a_j forma ciclo con las aristas marcadas en la solución se quita de A'. Si no lo forma, hacer $k = k + 1$.
3. Si $k = n - 1$ terminar. La gráfica $T = [X, A']$ es el árbol expandido de costo mínimo de G. Si $k < n - 1$ regresar a (2).

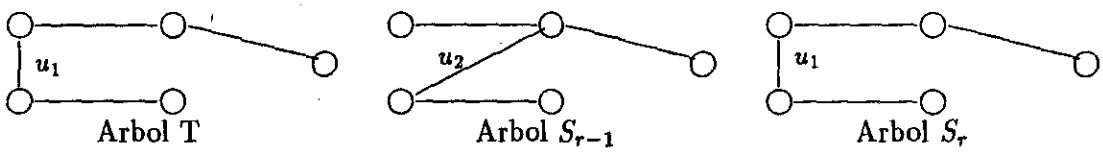
1.3.1 Justificación teórica del algoritmo

Sea $T = [X, A']$ la gráfica generada por el algoritmo. Por construcción T es acíclica y tiene $n-1$ aristas; por lo tanto, T es un árbol expandido de G. Ahora se demostrará que T es de costo mínimo.

Sea $S = [X, A'']$ un árbol de costo mínimo de G y supóngase que T es distinto de S. Basta probar que T y S tienen el mismo costo para que el algoritmo quede justificado. Para ello se procederá de la siguiente manera:

A partir de los árboles T y S se construirá otro árbol, $S_1 = [X, A''']$, de costo mínimo de G con la característica de que es más 'parecido' a T que a S; es decir, si T y S tienen z aristas en común, T y S_1 tendrán $z+1$ aristas en común. Después, si $T \neq S_1$, a partir de estos dos árboles se construirá otro árbol de costo mínimo, S_2, que tendrá $z+2$ aristas en común con T. Como el procedimiento propuesto es finito, llegará un momento en que se construirá un árbol S_k de costo mínimo de G que tendrá $n-1$ aristas en común con T; es decir, se tendrá $S_k = T$ y se podrá concluir que T es un árbol de costo mínimo de G.

Para la construcción del árbol S_r, (r=1, 2, ... , k), considérese lo siguiente: Si $T \neq S_{r-1}$ ($S_0 = S$) difieren, al menos, en una arista. Sea u_1 la arista de menor costo que está en T pero no en S_{r-1}; es decir, $C(u_1) = min\{C(a)\}$, para toda arista $a \in T$ tal que $a \notin S_{r-1}$. Obsérvese que u_1 es la primera arista examinada, durante el algoritmo de Kruskal, que pertenece a T pero no a S_{r-1}. Por otro lado, como S_{r-1} es árbol, existe en él una cadena que los une los extremos de u_1. Esta cadena contiene una arista, u_2, que no pertenece a T, puesto que si estuviera toda la cadena se formaría un ciclo con u_1 en T. Agréguese u_1 a S_{r-1} y elíminese u_2 de S_{r-1}. Sea S_r la gráfica resultante y obsérvese que ésta es un árbol de G.



Ahora debe probarse que S_r es de costo mínimo. Esto se hará por inducción. Para $k = 0$, se tiene que $S_0 = S$ es un árbol de costo mínimo de G. Supóngase válida la afirmación para $k = r-1$; esto es, supóngase que S_{r-1} es un árbol de costo mínimo de G. Esto implica que

$$C(S_{r-1}) \leq C(S_r)$$

y como S_{r-1} y S_r difieren sólo en una arista se tiene:

$$C(u_2) \leq C(u_1) \quad \ldots \quad (1)$$

hasta probar que $C(u_1) \leq C(u_2)$ para poder concluir que $C(S_{r-1}) = C(S_r)$. Para ésto, supóngase que $C(u_1) > C(u_2)$; nótese que esta suposición implica que la arista u_2 fué examinada antes que la arista u_1 durante el algoritmo de Kruskal que generó T. Por otro lado, como u_2 no fué incluida en T, esta arista forma ciclo con las aristas examinadas antes que ella y que fueron incluidas en T. Sean éstas $b_1, b_2, ..., b_j$. Debe observarse que $C(b_i) < C(u_1)$, para i=0,1,...j; luego, dada la definición de u_1, se concluye que $b_1, b_2, ..., b_j$ pertenecen a S_{r-1}. Esto es una contradicción, puesto

que $b_1, b_2, ..., b_j$ y u_2 formarían ciclo en S_{r-1}. Por lo tanto se concluye

$$C(u_1) \leq C(u_2)$$

Esta expresión junto con la expresión (1) implica

$$C(u_1) = C(u_2)$$

y por lo tanto

$$C(S_{r-1}) = C(S_r)$$

es decir, S_r es un árbol de costo mínimo de G. Nótese que T y S_r contienen ambos a la arista u_1 (que no estaba contenida en S_{r-1}); luego, si T y S_{r-1} tienen z aristas en común, T y S_r tienen $z+1$ aristas en común. Si $T = S_r$ entonces T es de costo mínimo; en caso contrario, constrúyase S_{r+1}, del mismo modo que como se construyó S_r a partir de T y S_r. Este procedimiento es finito puesto que el número de aristas de T es finito.

■

1.3.2 Descripción de la implementación con estructuras de datos

Revisemos ahora el algoritmo de Kruskal paso a paso. En cada uno de ellos diremos como será implementado con estructuras de datos.

1. • Ordénese el conjunto de aristas de manera creciente con respecto al costo.
   Para esto usaremos un montículo con la siguiente información en sus elementos:
   - número del nodo inicial,
   - número del nodo final
   - y costo de la arista.

   La estructura aquí usada se muestra enseguida:

Nodo inicial	Nodo final	costo

   • $A' = \phi$.
   El grado y la copia de éste, en todos los nodos, será 0. La etiqueta de las aristas es 1.

2. • Elegir a_j la arista de menor costo.
   Sacar un elemento del montículo y guardarlo en a_j.
   • $A' = A' \cup a_j$.
   La arista A_j entra en la solución: A los nodos extremos se les aumenta el grado en uno y en sus respectivas listas de aristas se marcar a_j con un 0.

- Revisar si a_j forma ciclo con los elementos de A'.

  Copiar el grado de los nodos. Se revisa la gráfica buscando nodos cuya copia del grado sea 1. Los que se encuentren se meten en una pila cuya única información será el número del nodo. Esto es equivalente a determinar todos los nodos pendiente.

  Sacar un elemento de la pila.

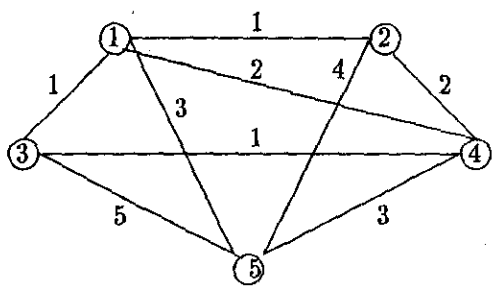
  Revisar si aún tiene grado 1 en la copia del grado, de ser así bajar a 0 en la copia del grado y la arista marcarla con 2. Ir al nodo extremo de la arista, bajar su grado temporal (la copia) en uno y marcar la arista con un 2. Revisar si este nodo tiene grado temporal 1, de serlo repetir el procedimiento, si no, sacar otro de la pila y repetir.

  Cuando la pila esté vacía se revisan las copias de los nodos: si todos tienen grado 0 entonces a_j no formó ciclo y se queda en la solución. Pero si alguno de los nodos tiene grado distinto de 0 se saca de la solución bajando el grado de las aristas extremos en uno marcando las aristas con un 1. Las aristas marcadas con 2 cambian a 0 y se copia de nuevo el grado de los nodos.

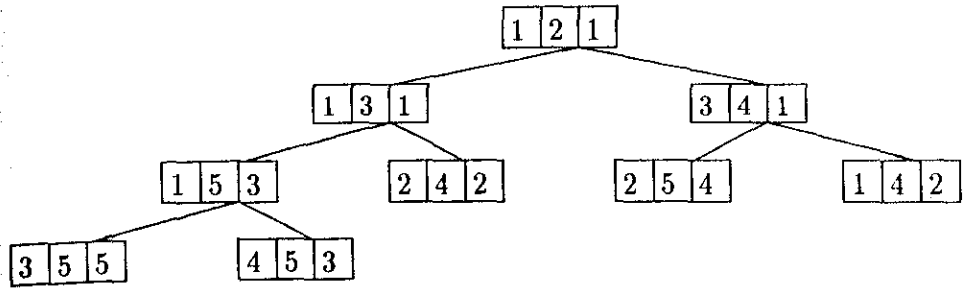
A continuación veremos un ejemplo resuelto:

1.3.3 Ejemplo

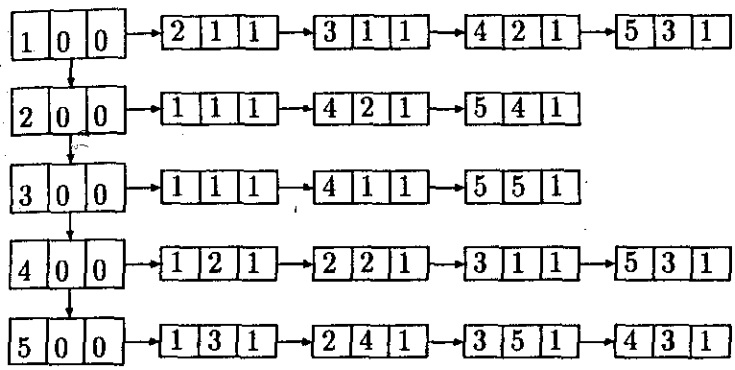
Supongamos que tenemos la gráfica de la figura. Vamos a determinar el árbol de costo mínimo en la gráfica por medio del algoritmo de Kruskal



Primero que nada debemos ordenar las aristas de forma creciente, utilizando un montículo quedan ordenados como sigue:



Las listas enlazadas serán las estructuras de datos utilizadas para manejar la información. A continuación se muestra la gráfica inicial. La etiqueta en las aristas puede ser 0, 1 ó 2; cero significa que está en la solución, uno no está y dos es una marca temporal para revisar si se forma ciclo.



Comencemos a utilizar el algoritmo:

Inicialmente $A' = \phi, \quad k = j = 0.$

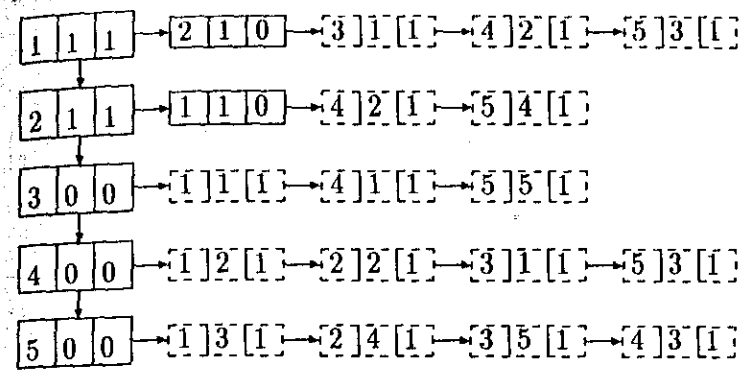
Primera iteración

j=1 Sacamos un elemento del montículo: $a_1 = (1, 2, 1)$.

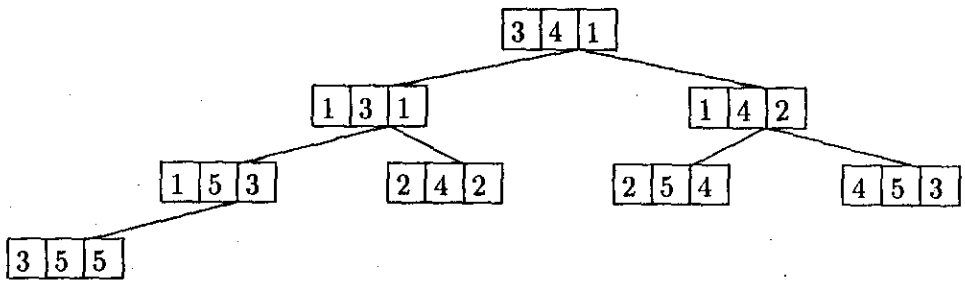
$A' = \{(1, 2, 1)\}$

Evidentemente, a_1 no forma ciclo pues es la única arista en A'.

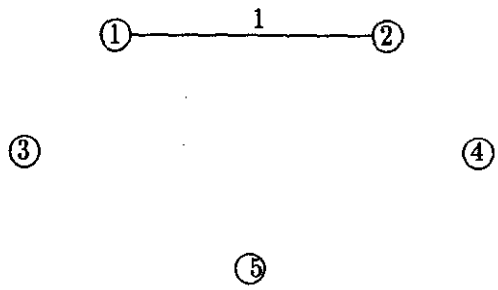
En la gráfica debemos marcar la arista en la solución, lo cual haremos etiquetándolo con un cero. Las aristas que no están en la solución se muestran con lineas punteadas.



El montículo, ya reordenado, se muestra a continuación:



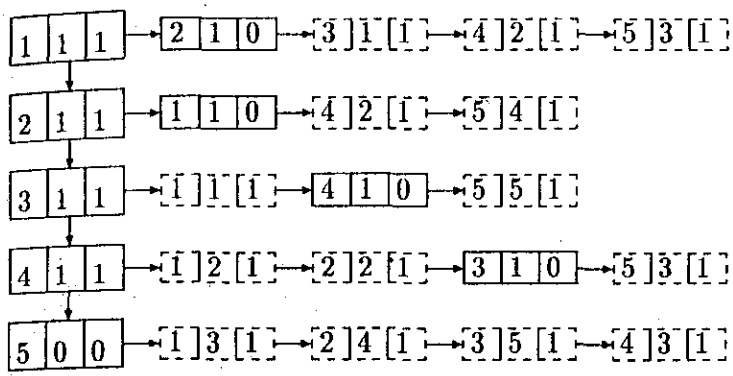
Hasta ahora ésta es la solución:



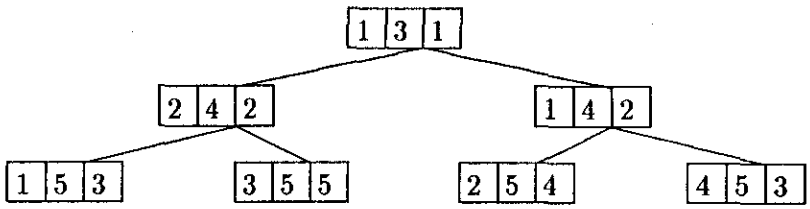
k=1

Segunda iteración

j=2 $a_2 = (3,4,1)$ $A' = \{(1,2,1),(3,4,1)\}$

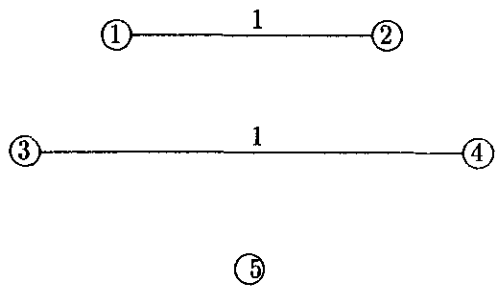


El nuevo montículo se muestra abajo.



Aún no puede formarse ciclo, pues sólo hay dos aristas.

Hasta aquí la solución es:

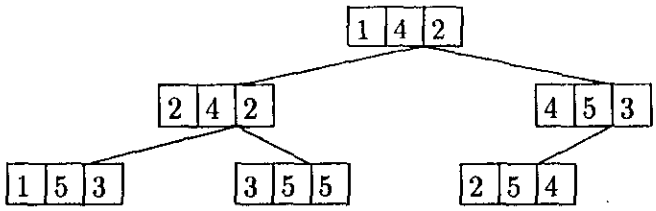


k=2

Tercera iteración

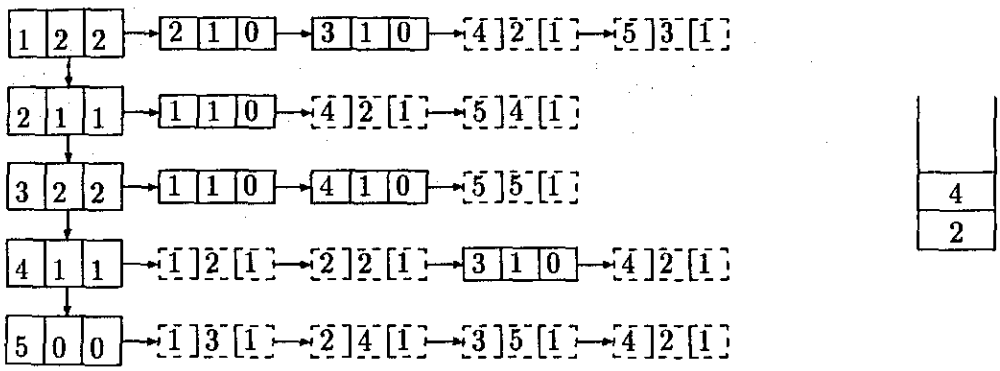
j=3 $a_3 = (1,3,1)$ $A' = \{(3,4,1),(1,3,1),(1,2,1)\}$

El montículo queda:

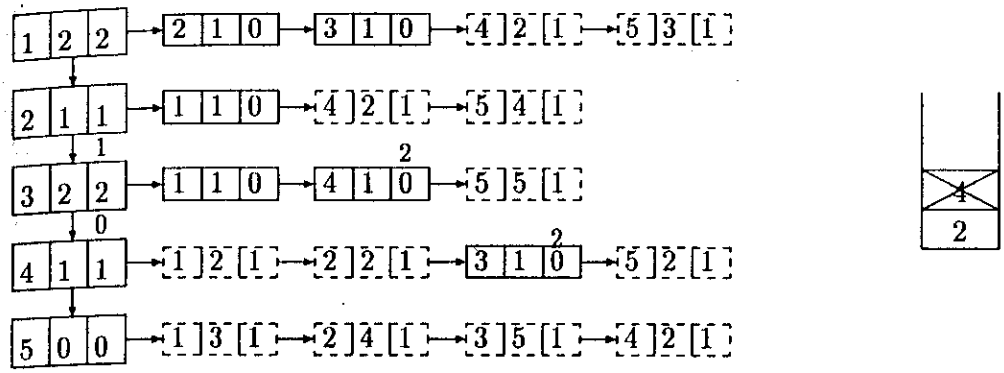


Con tres aristas es posible que se forme un ciclo, revisemos que eso no suceda. Los cambios que se harán en la gráfica en cada iteración, para mayor claridad, se pondrán encima de las casillas correspondientes.

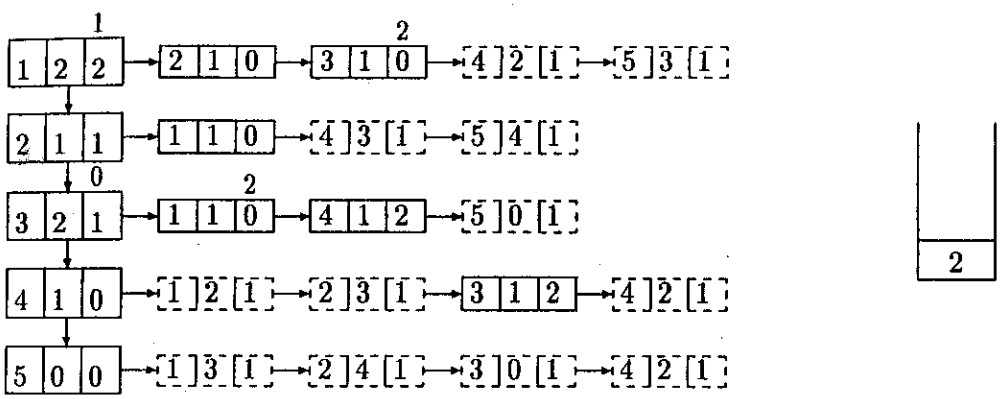
Hacer una pila con los nodos que tienen grado uno: 2 y 4.



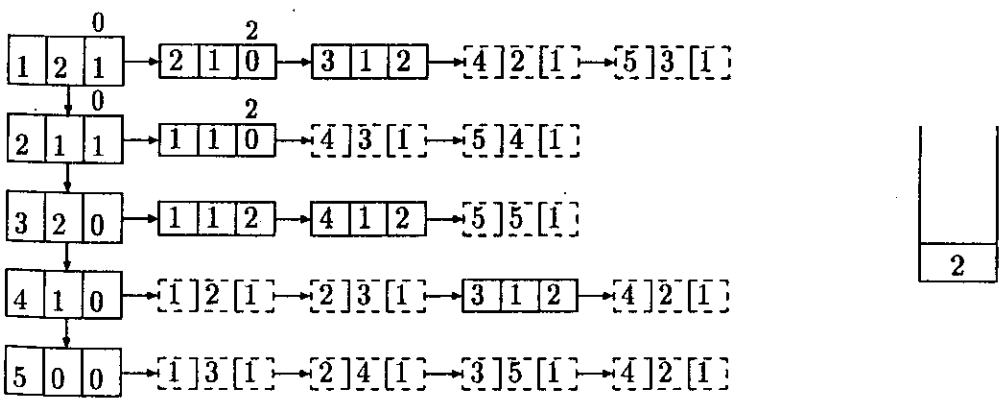
Sale el 4 de la pila. Bajar su grado temporal (la copia) a 0 y marcar la arista $(4,3)$ con una etiqueta temporal (2); ir al nodo 3, bajar su grado a uno y marcar la arista (3,4) con un 2.



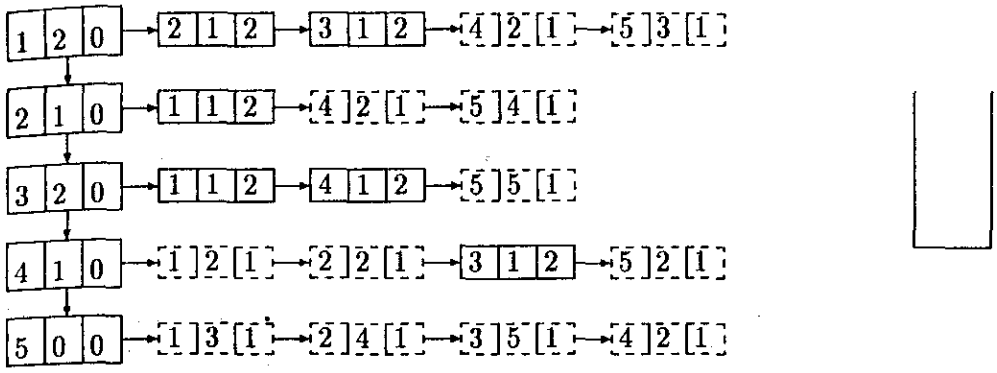
Como el tres es ahora pendiente, bajar su grado a 0 y marcar la arista $(3,1)$ con un 2; ir al nodo 1, bajar su grado a 1 y marcar la arista $(1,3)$ como temporal.



El nodo 1 es ahora pendiente, bajar su grado a 0 y marcar la arista $(1,2)$ con un 2; ir al nodo 2, bajar su grado a 0 y marcar la arista $(2,1)$ con un 2.

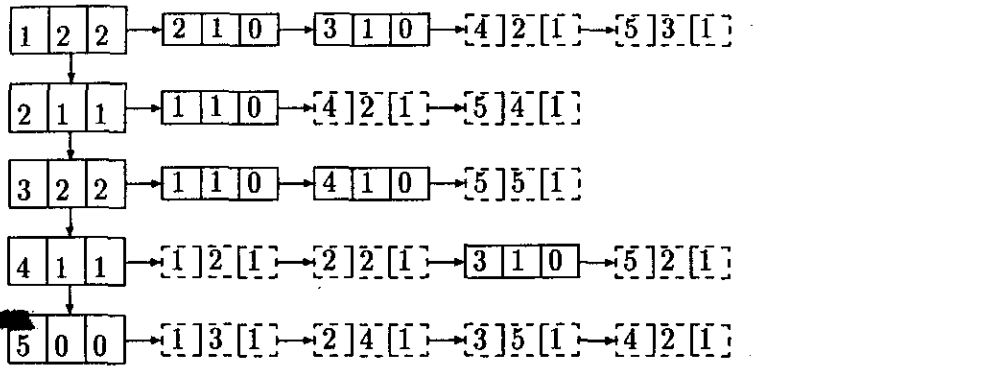


Como el nodo 2 tiene grado temporal 0, sacar otro de la pila: el 2. Sacar otro de la pila: está vacía.

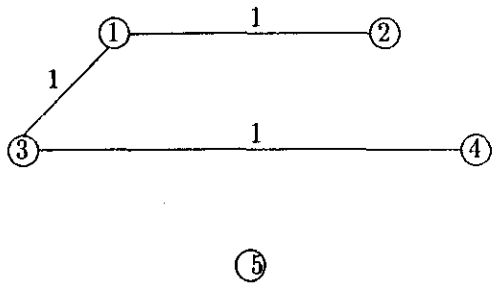


Al revisar los grados temporales de todos los nodos nos damos cuenta de que no se formó ciclo al meter a_3 a la solución.

Quitar las marcas temporales del ciclo: cambiar la etiqueta a las aristas que tengan un 2 por un 0 y copiar de nuevo los grados de los nodos.



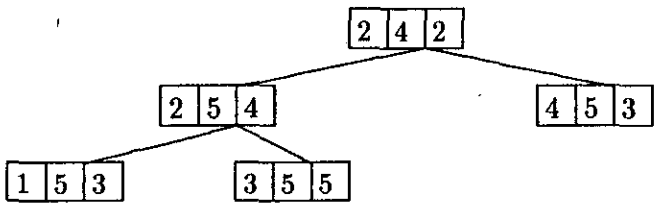
Gráficamente, la solución hasta ahora es:



k=3

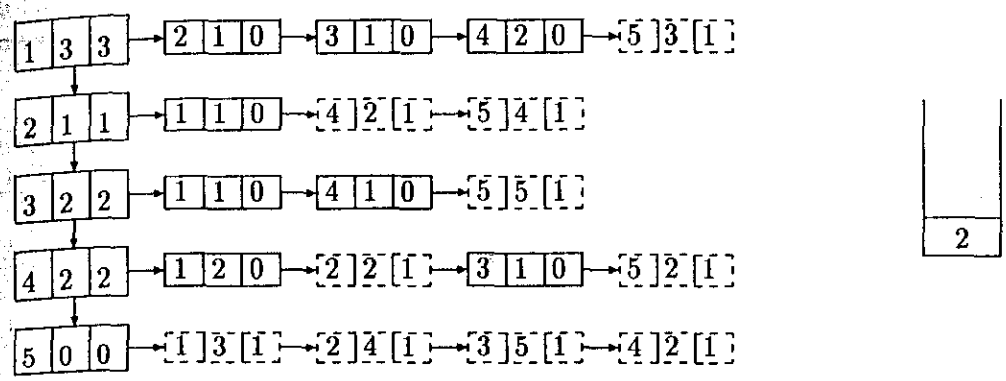
Cuarta iteración

j=4 $a_4 = (1,4,2)$ $A' = \{(1,2,1),(3,4,1),(1,3,1),(1,4,2)\}$

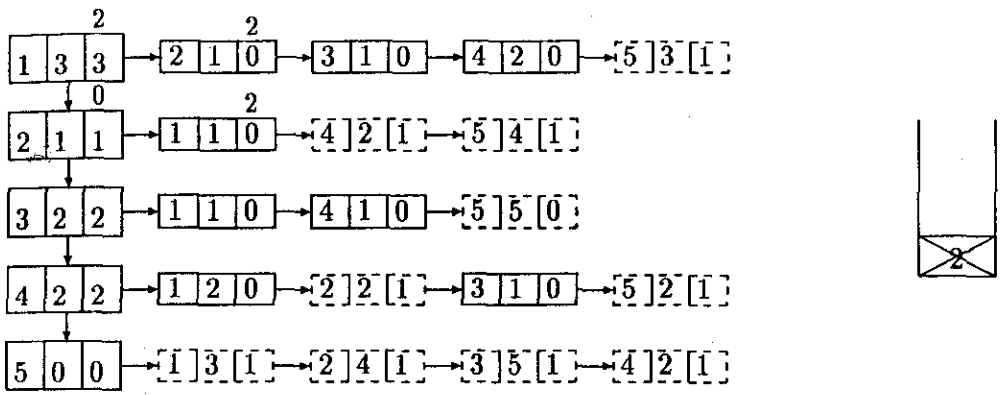


Revisemos si se formó ciclo con la nueva arista:

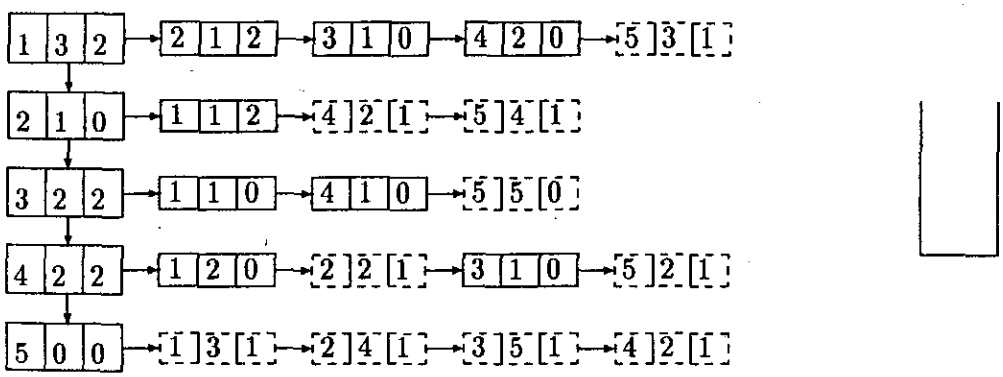
Hacer una pila con los nodos que tiene grado uno: 2.



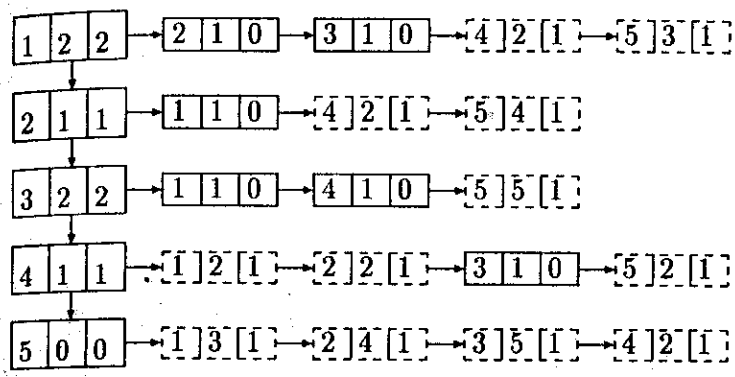
Sale el 2 de la pila. Bajar su grado temporal (la copia) a 0 y marcar la arista $(2,1)$ con una etiqueta temporal (2); ir al nodo 1, bajar su grado a dos y marcar la arista (1,2) con un 2.



El nodo 2 no tiene grado 1. Sacar otro elemento de la pila: está vacía.

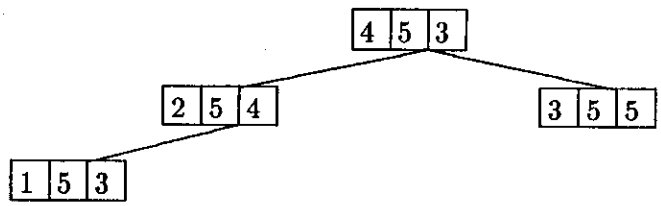


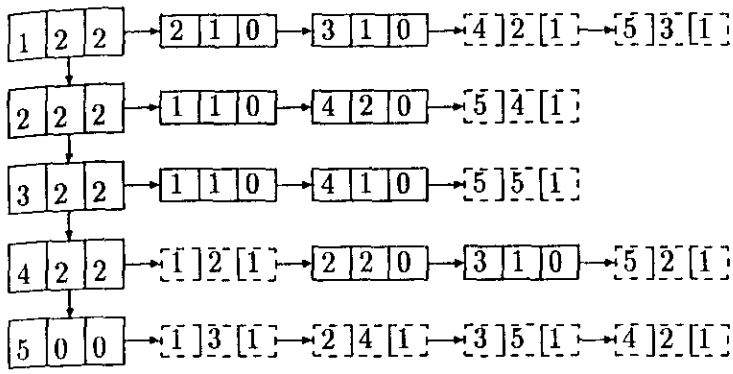
Revisando el grado temporal de los nodos, vemos que se ha detectado un ciclo: 1,2,4. Por lo tanto, la arista $(1,4)$ con peso 2, sale de la solución. La gráfica toma de nuevo la forma que tenía al comenzar esta iteración.



Quinta iteración

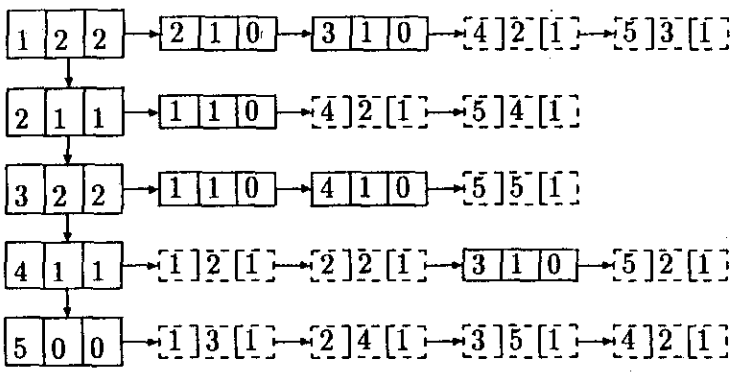
j=5 $\quad a_5 = (2,4,2) \quad A' = \{(1,2,1),(3,4,1),(1,3,1),(2,4,2)\}$





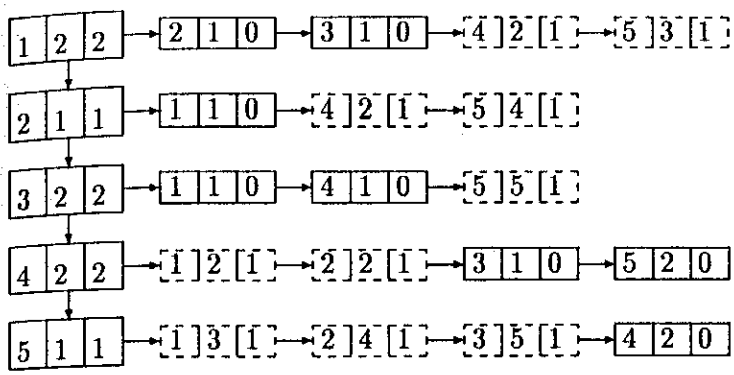
Para revisar si se ha formado ciclo, meter en una pila los nodos de grado 1. No hay. Nos damos cuenta que hay nodos con grado distinto de cero, por lo tanto se formó ciclo y la arista $(2,4)$ debe salir de la solución.

Nuevamente, la gráfica regresa a la forma que tenía al inicio de la iteración.

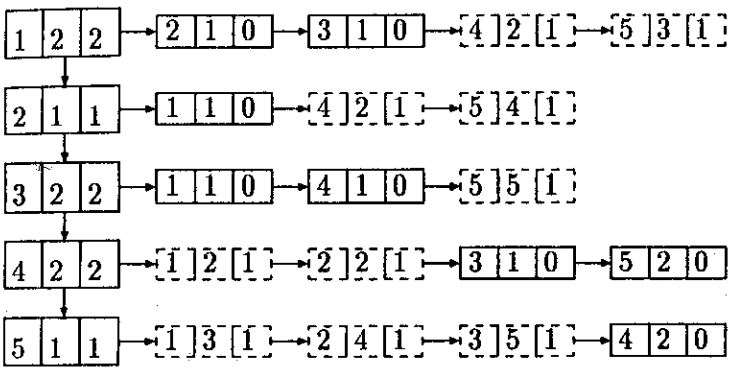


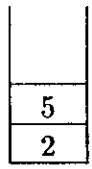
Sexta iteración

j=6 $a_6 = (4,5,3)$ $A' = \{(1,2,1),(3,4,1),(1,3,1),(4,5,3)\}$

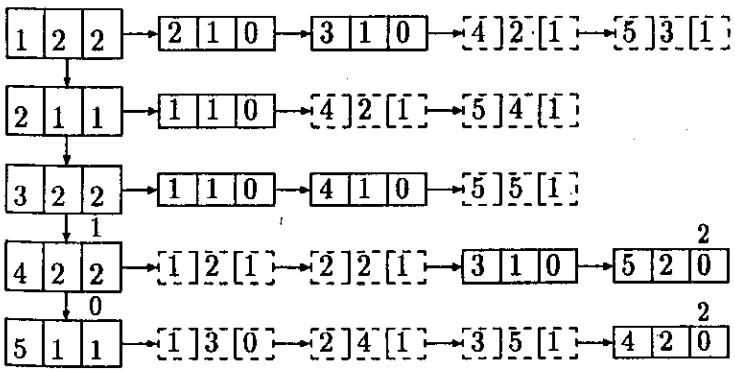


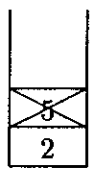
Los nodos de grado 1 entran en una pila: el 2 y el 5.



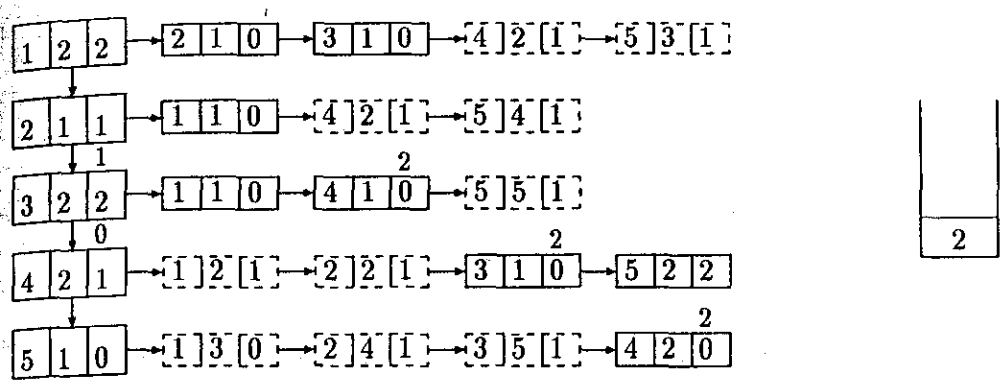


Sale el 5 de la pila. Cortarlo.

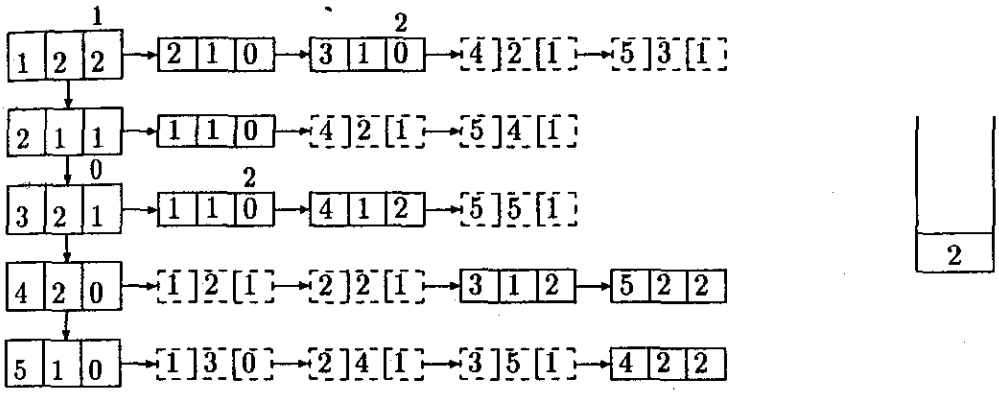




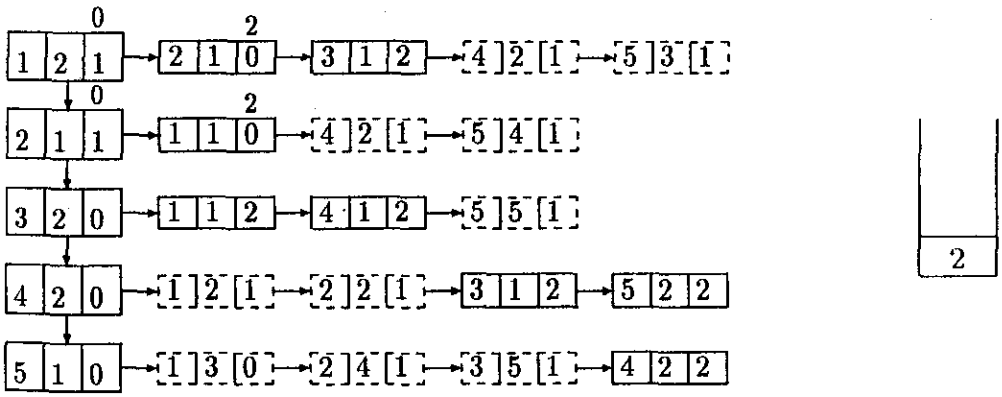
El 4 es pendiente. cortarlo



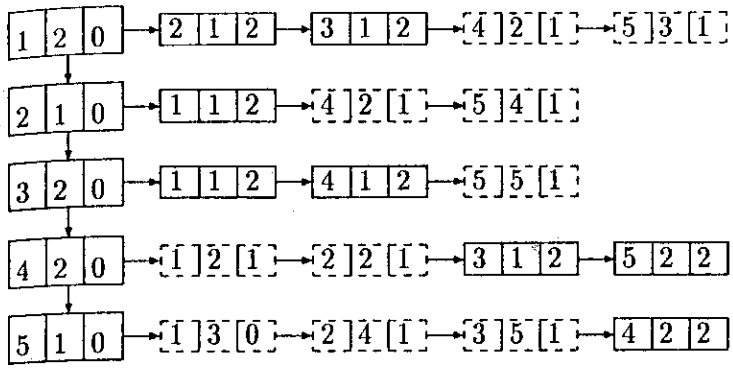
El 3 es pendiente. Cortarlo.

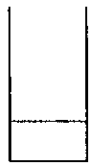


El 1 es pendiente. Cortarlo.

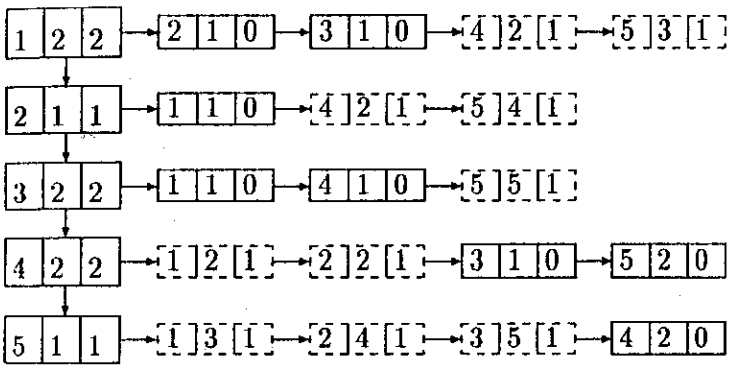


El 2 no es pendiente. Sacar de la pila: el 2. Sacar otro: está vacía .



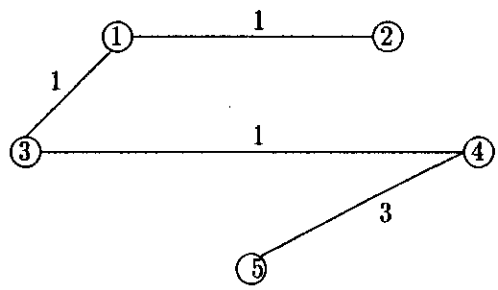


No se forma ciclo con la arista a_6, por lo tanto, quitar las marcas temporales y copiar nuevamente los grados de los nodos.



k=4 Por lo tanto el algoritmo terminó.

La solución del Kruskal es:



1.4 Algoritmo de Prim

Existe otro método para determinar el árbol de costo mínimo de una red conexa con n nodos: El algoritmo de Prim. Este algoritmo construye la gráfica solución partiendo de un nodo cualquiera de la gráfica; a continuación agrega la arista de menor costo de las que están conectadas con él y el extremo opuesto. Después se aumenta la arista más pequeña que tiene un extremo en la gráfica solución, junto con su otro extremo siempre y cuando este no esté en la solución. El procedimiento termina cuando se tienen n nodos en la gráfica solución. El procedimiento es el siguiente:

Sea k el número de nodos en la solución, X_i el conjunto de nodos en la solución en la $i-$ésima iteración y A_i el conjunto de aristas en la solución en la $i-$ésima iteración.

1. Sea x_0 cualquier nodo de la gráfica G; sean $X_0 = \{x_0\}$ y $A_0 = \phi$. Hacer $k = 0$.

2. $k = k + 1$, si $k = n$ terminar.

3. Sea C_k el conjunto de aristas que tienen exactamente un extremo en X_{k-1}. Sea a_k la arista de costo mínimo de C_k y sea x_k el extremo de a_k que no pertenece a X_{k-1}. Hacer

   $X_k = X_{k-1} \cup \{x_k\}$

   $A_k = A_{k-1} \cup \{a_k\}$ Ir a (2).

La gráfica $T_{n-1} = [X_{n-1}, A_{n-1}]$ es un árbol expandido de costo mínimo de G.

1.4.1 Justificación teórica del algoritmo

Basta demostrar que cada gráfica $T_k = [X_k, A_k]$, $k = 1, ..., n-1$ es una subgráfica conexa, con $k+1$ nodos, de un árbol de costo mínimo de G. Esto se hará por inducción sobre k.

Para $k = 0$ es trivial. (T_0 consta de un solo nodo.)

Supóngase que la afirmación es válida para $k = r - 1$.

Falta demostrarla para $k = r$.

Como T_{r-1} tiene r nodos y es conexa entonces, por construcción, T_r tiene $r + 1$ nodos. Sea T^* un árbol de peso mínimo de G que contiene todas las aristas de T_r. Si a_r es una arista de T^*, entonces T_r es una subgráfica conexa de T^*. Si al contrario, no contiene a a_r, entonces al agregar a_r a T^* se formará un ciclo con a_r y la cadena que une sus extremos en T^*. Dada la definición del conjunto C_r, dicha cadena contiene una arista de él; sea ésta u.

Sea el árbol $T' = [X, A']$, donde $A' = (A^* \cup \{a_r\}) - \{u\}$. Nótese que T_r es una subgrálica conexa del árbol T'. Ahora, puesto que

$$C(a_r) = \min_{a \in C_k} \{C(a)\}, \quad \textit{para todo} \quad k = 1, 2, ..., n-1$$

y

$$C(a_r) \leq C(u)$$

se tiene que

$$C(T') \leq C(T^*)$$

y como T^* es un árbol expandido de costo mínimo de G, entonces T' también lo es.

■

1.4.2 Descripción de la implementación con estructura de datos

La gráfica la mantendremos en listas enlazadas: En una lista, ordenada por el número del nodo, guardaremos la información referente al mismo. Cada elemento de la lista tendrá los siguientes datos:

- Número del nodo.
- Etiqueta, puede ser 0 ó 1.

  Es 0 si el nodo está en la solución y

  es 1 si el nodo no está en la solución.

Inicialmente todos los nodos tienen etiqueta igual a 1.

La estructura usada en los nodos se muestra enseguida:

Nodo Inicial	Etiqueta

Las aristas conectadas con un nodo también estarán guardadas con listas, ordenadas por el número del nodo extremo de la arista. Cada miembro de estas listas contendrá lo siguiente:

- Número del nodo extremo de la arista.
- Costo de la misma.
- Etiqueta, puede ser 0 ó 1.

  Es 0 si la arista está en la solución y

  es 1 si la arista no está en la solución.

Inicialmente todas tienen la etiqueta igual a 1.

La estructura usada para las aristas se muestra aquí:

Nodo final	costo	Etiqueta

Cada elemento de C_k tendrá los siguientes datos:

- Número de nodo inicial.
- Número de nodo final.
- Costo del arco.

La estructura se ilustra a continuación.

Nodo inicial	Nodo final	Etiqueta

Revisemos ahora el algoritmo de Prim paso a paso. En cada uno de ellos diremos la manera en que será implementado con estructuras de datos.

1. $X_0 = \{x_0\}, A_0 = \phi$.

   El nodo x_0 se etiqueta con un 0. Las aristas todavía no presentan cambios.

2. $k = k + 1, \quad A_0 = \phi$.

3. Sea C_k el conjunto de arista que tienen exactamente un extremo en X_{k-1}. Sea a_k la arista de costo mínimo de C_k y sea x_k el extremo de a_k que no pertenece a X_{k-1}. Hacer $X_k = X_k \cup \{x_k\}$ y $A_k = A_{k-1} \cup \{a_k\}$. Ir a (2).

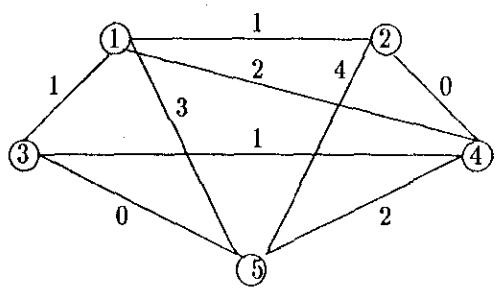
   Revisemos todos los nodos en la solución. Cada uno de ellos tiene un conjunto de aristas conectadas. En el conjunto C_k, guardado con una lista ordenada por el costo [1] van aquellas que aún no entran en la solución (las que tienen etiqueta 1) y se escoge de entre estas a la menor costo, a_k, y el nodo extremo que no estaba en la solución x_k se marca con un 0 y la arista se marcan con un 0.

   Repetir hasta que el número de nodos en la solución, sea igual a n.

1.4.3 Ejemplo

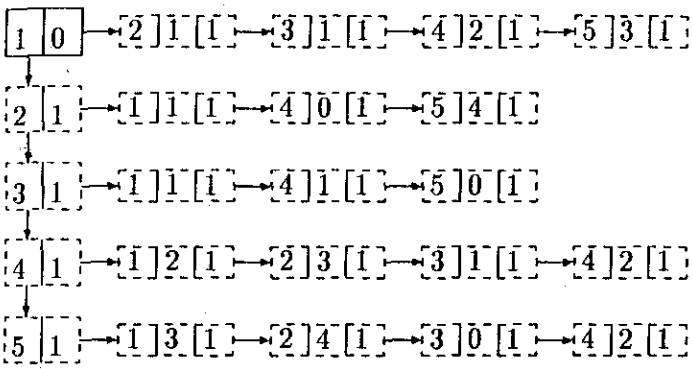
Consideremos el ejemplo siguiente:

[1] Pudimos haber usado un montículo, se optó por una lista ordenada para ejemplificar otro método de ordenamiento.



vamos a determinar el árbol de costo mínimo en la gráfica por medio del algoritmo de Prim partiendo del nodo 1.

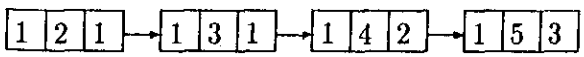
$X_0 = \{1\}, \quad A_0 = \phi, \quad k = 0$



Primera iteración

$k = 1$

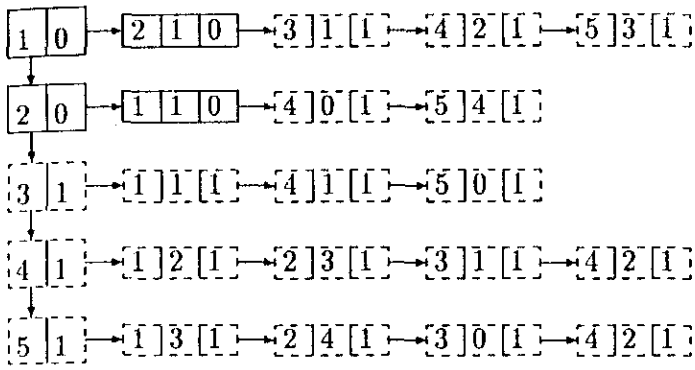
Los arcos candicatos a entrar en la solución son aquellos conectados con el nodo 1, esto es:

C_1: 

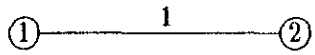
Sale de C_1 el arco $(1,2)$ y entra en la gráfica.

$a_1 = (1,2,1), \quad X_3 = \{1,2\}$

$A_1 = \{(1,2,1)\}$



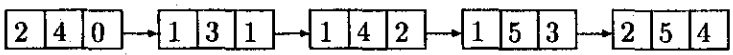
Esta es la gráfica solución hasta ahora.



Segunda iteración

$k = 2$

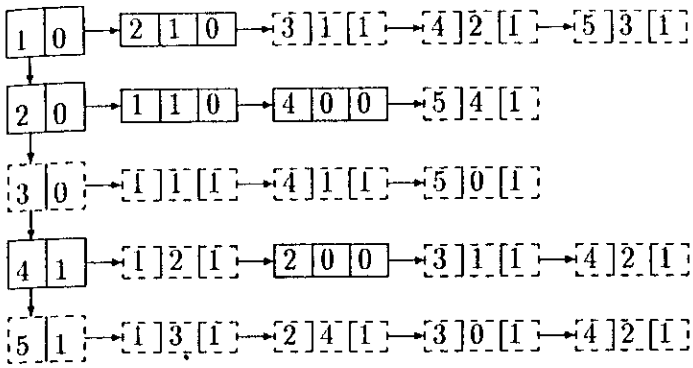
En C_2 irán ahora los arcos conectados con los nodos 1 y 2, así,

C_2 : 

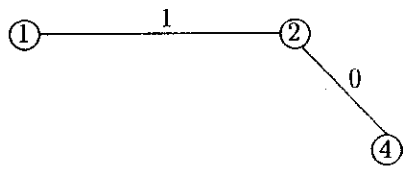
Entra el $(2,4)$ a la gráfica y sale de C_2.

$a_2 = (2,4,0), \quad X_3 = \{1,2,4\}$

$A_2 = \{(1,2,1),(2,4,0)\}$



Esta es la solución hasta ahora:



Continuando con el procedimiento se tienen las siguientes iteraciones.

Tercera iteración

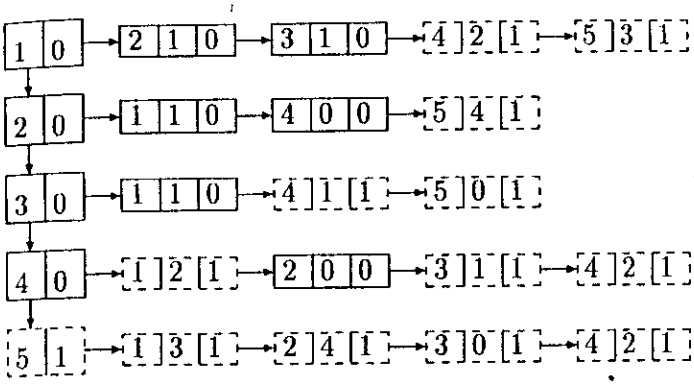
$k = 3$

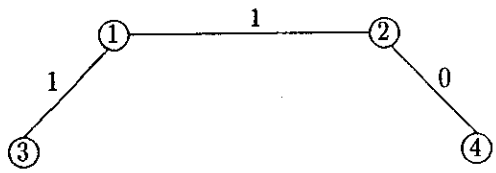
El conjunto de arcos candidatos son ahora los conectados con los nodos 1, 2 y 3.

C_3 : 1 3 1 → 3 4 1 → 1 4 2 → 4 5 2 → 1 5 3 2 5 4

$a_3 = (1,3,1), \quad X_3 = \{1,2,3,4\}$

$A_3 = \{(1,2,1),(2,4,0),(1,3,1)\}$





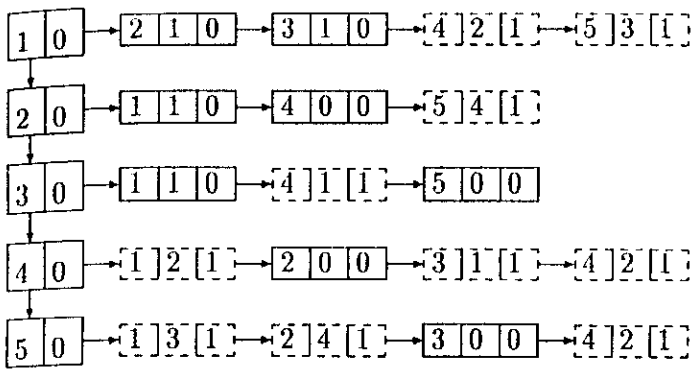
Cuarta iteración

$k = 4$

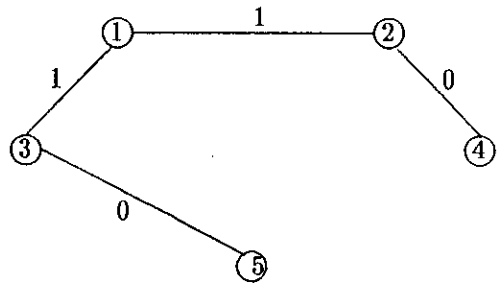
C_4 : 3 5 0 → 3 4 1 → 1 4 2 → 4 5 2 → 2 4 3 → 2 4 3

$a_4 = (3,5,0)$ $\quad X_3 = \{1,2,3,4,5\}$

$A_4 = \{(1,2,1),(2,4,0),(1,3,1),(3,5,0)\}$



Esta es la gráfica al meter a_4 en la solución.



$k = 5$, terminar.

En [5] se menciona que en general (confirmado esto por la experimentación) el método de Prim es mejor si se utiliza para una gráfica densa mientras que el Kruskal lo es tratándose de gráficas ralas.

En el siguiente capítulo, abordaremos el problema de encontrar la ruta mínima en una digráfica.

Capítulo 2

Ruta más corta en digráficas

En este capítulo se presentan métodos de solución para los siguientes problemas de rutas más cortas en una red $R = [X, A, C]$:

1. Dados dos nodos, $i, j \in X$, encontrar el camino de costo mínimo que va de i a j.
2. Dado $i \in X$ encontrar los caminos más cortos para todos los $x \in X$ tal que exista algún camino de i a x.
3. Para todo $i \in X$ encontrar los caminos más cortos para todos los $x \in X$ tal que existe algún camino de i a x.

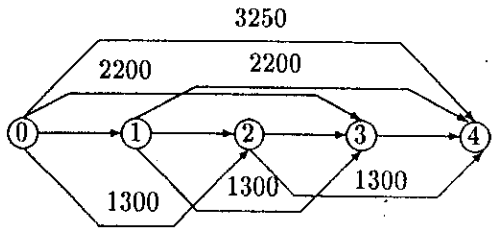
Estos problemas aparecen en numerosas situaciones reales y teóricas de ahi la importancia de conocer métodos eficientes para resolverlos. A continuación, veremos algunos problemas que utilizan para su solución la resolución de problemas de rutas más cortas.

1. Una oficina desea establecer una política racional de uso para sus aires acondicionados para lo cual contrata una compañía dedicada al mantenimiento de estos aparatos. Al comenzar el trabajo cuentan con aparatos nuevos que costaron $ 1500.00. Esta compañía sabe que los aparatos pueden tenerse en funcionamiento todo el año al final del cual los dueños deben decidir si los cambian o los reparan. Tanto el costos de mantenimiento del aparato como el de venta dependen del tiempo que tenga funcionando. En la siguiente tabla se muestran estos costos

Años de uso	Costo de venta	Costo de mantenimiento
1	750	300
2	500	600
3	200	875
4	25	1000

El problema es minimizar el costo de mantenimiento durante un período de tres años.

Para resolverlo se utiliza la siguiente red:



Los nodos 0, 1, 2, 3 y 4 representan el inicio o el final de un año. Los costos en los arcos es lo que se ha gastado en el lapso que se indica [1], el nodo inicial del arco indica el año en que fué comprado el aparato y el final cuándo lo vendieron.

2. La Constructora Zarate, una de las firmas en la ciudad constrtuirá canchas deportivas en la escuela preparatoria CBTIS #188 de Cd. Obregón, Sonora. Las multas por no terminar a tiempo un contrato son muy altas, por lo que el presidente de la compañía instruyó al departamento de planeación para que determine que partes del proyecto son importantes para terminar el proyecto a tiempo.

   Como un paso preliminar el departamento de planeación dividió el proyecto en subproyectos ajenos y designó un tiempo de terminación para cada uno tratando así de disminuir el tiempo requerido por el proyecto total. La manera en que se dividió el proyecto así como los subproyectos precedentes se muestran en la siguiente tabla.

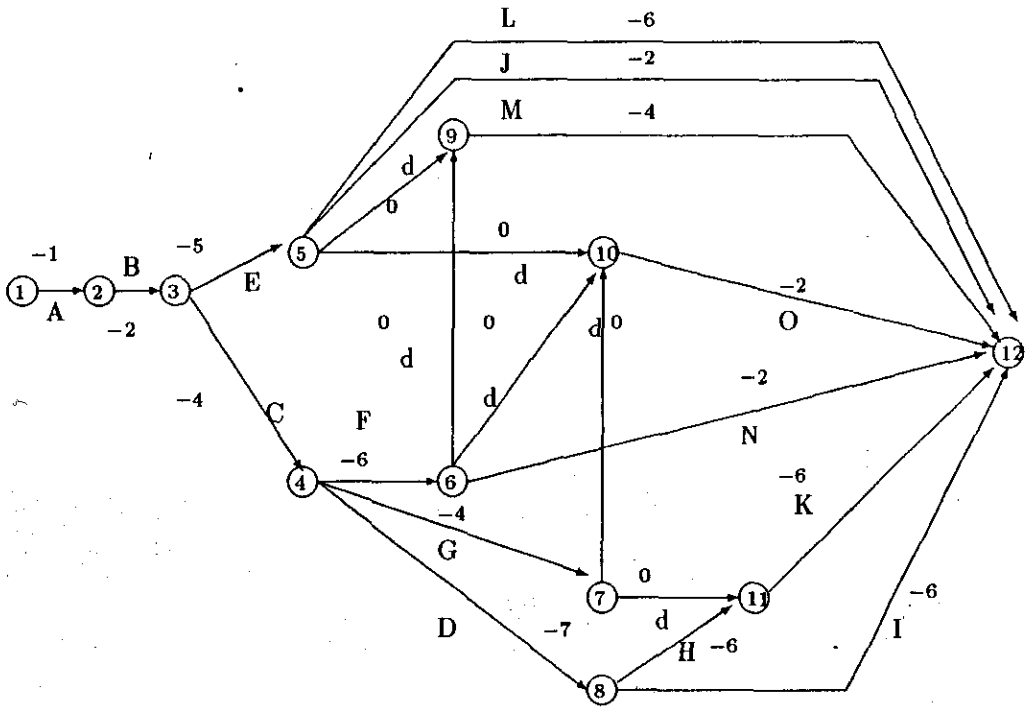
Subproyecto	Realización de	Dias requeridos	Subproyecto anterior
A	Inspección del terreno	1	–
B	Cimientos	2	A
C	Paredes	4	B
D	Techos	7	C
E	Pisos de Madera	5	B
F	Acabado de interiores	6	C
G	Tuberias	4	C
H	Instalación eléctrica	4	D
I	Canchas de Basquet boll	1	D
J	Pintar canchas	2	E
K	Cocina/Refrigeración	6	G,H
L	Camino	6	E
M	Asientos del estadio	4	E,F
N	Tablero electrónico	2	F
O	Construir la refresqueriå	2	E,F,G

La figura de abajo representa la red con la información donde los arcos representan los varios subproyectos que hacen el proyecto total. Este problema se soluciona encontrando la ruta más larga en la digráfica lo cual es equivalente, cambiando el signo de los costos, a encontrar

[1] Recuérdese que en el año cero se parte del costo de compra por la adquisición del equipo.

una ruta mínima. Esta es la razón del negativo de los costos de los arcos. Los arcos etiquetados con d y que tienen costo cero indican que hay actividades que no pueden realizarse simultáneamente. Por ejemplo, la actividad O requiere de la terminación de E, F y G para realizarse.

Los nodos de la figura representan el principio o final del subproyecto o ambos. Los nodos 1 y 14 representan, respectivamente, el principio y el final del proyecto total.

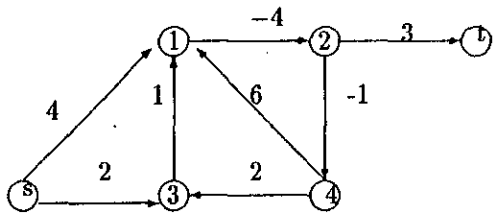


2.1 Descripción del problema con elementos de digráficas

En una red $R = [X, A, C]$, la función C asociada a cada arco, $C(a)$, se llama costo o longitud de a.

El costo de un arco no es necesariamente positivo, pues la función de costo, C, puede representar no solo distancia o tiempo, sino también costos o cualquier otra cantidad. Además el costo de una ruta o camino es la suma de los costos de los arcos que la forman, aquella ruta tal que su costo sea mínimo se le llama ruta más corta o camino más corto.

Si la red contiene arcos con costo negativo pueden presentarse circuitos negativos en cuyo caso el problema puede ser no acotado puesto que, dada cualquier ruta entre s y t que contenga el circuito negativo existe otra mejor (aquella que contiene una vez más al circuito). Para ejemplificar considerese la siguiente red:



Una ruta más corta entre s y t es $s, 3, 1, 2, t$ cuyo costo es 2. Puede encontrarse una ruta mejor entre s y t: $s, 1, 2, 4, 3, 1, 2, t$ de costo 0. Si se considera nuevamente el ciclo, el costo bajará aún más.

Por lo tanto, para que el problema de encontrar la ruta más corta entre los nodos i y j tenga solución se requiere que

1. Exista algún camino de i a j.
2. No exista un circuito negativo con un nodo que pase por un camino de i a j.

Resolveremos el problema de encontrar la ruta mínima entre i y x, para todo $x \in X$. Cuando existen estos elementos, i se llama *raíz* y a la digráfica resultante se le llama *arborescencia.*

Obsérvese que una arborescencia resuelve parte de nuestro problema: encontrar un camino entre un nodo específico y los demás en una digráfica. Esto motiva su estudio.

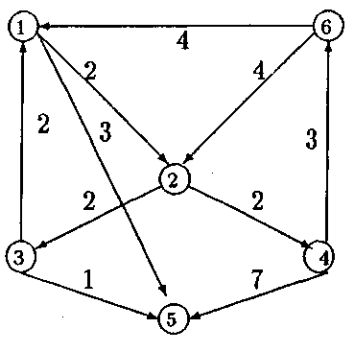
Definición 2.1 *Sea $G = [X, A]$ una digráfica. Una arborescencia de G es un árbol expandido de G que tiene un nodo que es raíz.*

En una arborescencia de raíz s el camino de s a x, para todo $x \in X$, es único, la demostración de esto se ve en la proposición 2.1.

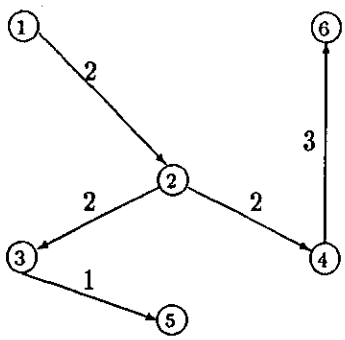
Supóngase que ahora se quiere conocer las rutas más cortas entre todo par de nodos en la red. Asociemos a este problema la red $R = [X, A, C]$ definida anteriormente. Este problema es una generalización inmediata de los anteriores. Por ésto puede concluirse que, para que exista solución en cualquier red $R = [X, A, C]$ se deberá cumplir lo siguiente:

1. Existir al menos un camino entre todo par de nodos.
2. No existir circuitos negativos en la red R.

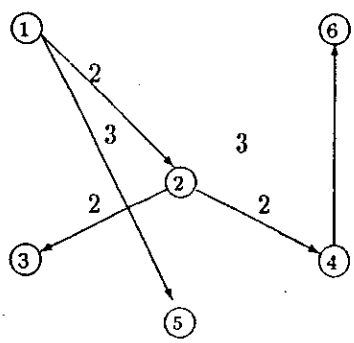
Vamos a ver los métodos de solución para los tres problemas expuestos, pero antes veremos algunas propiedades de las arborescencias. Algunas de estas serán utilizadas en la búsqueda de la solución del problema de la arborescencia de rutas más cortas de raíz s. En el siguiente teorema se demuestran estas propiedades, postulando las distintas caracterizaciones de este tipo de digráfica. En general, la arborescencia de ruta más corta es distinta del árbol de peso mínimo de una red puesto que el segundo concepto es no orientado y el primero sólo es aplicable a redes dirigidas. Por ello, para determinar la arborescencia de rutas más cortas no es posible aplicar ninguno de los algoritmos presentados en el capítulo anterior. Podría suceder incluso que un árbol de peso mínimo resulte ser una arborescencia; sin embargo, de ningún modo puede garantizarse que ésta sea de rutas más cortas. Para ilustrar esto, en el siguiente ejemplo presentamos una red $G = [X, A, C]$, una arborescencia de peso mínimo y una arborescencia de rutas más cortas para G, ambas con raíz 1.



G



Arborescencia de peso mínimo de G



Arborescencia de rutas más cortas de G

Las propiedades mencionadas son las siguientes:

Proposición 2.1 *Sea $G = [X, A]$ una digráfica con n nodos, $n \geq 2$. Los postulados siguientes son equivalentes:*

1. *G es un árbol y tiene un nodo s que es raíz.*
2. *Para todo nodo x existe un camino único de s a x.*
3. *G tiene al nodo s como raíz y si se elimina un arco entonces s ya no es raíz.*

4. *G es concxa,* $g^-(s) = 0$ *y* $g^-(x) = 1 \; \forall \; x \neq s$

5. *G es acíclica,* $g^-(s) = 0$ *y* $g^-(x) = 1 \; \forall \; x \neq s$

6. *G tiene como raíz a s y es acíclica.*

7. *G tiene como raíz a s y posee* $n - 1$ *arcos.*

Demostración:

Se hará en el siguiente orden: $1 \Rightarrow 7 \Rightarrow 6 \Rightarrow 5 \Rightarrow 4 \Rightarrow 3 \Rightarrow 2 \Rightarrow 1$.

- $(1 \Rightarrow 7)$ Por hipótesis G tiene una raíz y por ser árbol posee $n - 1$ arcos.
- $(7 \Rightarrow 6)$ G tiene como raíz a s por hipótesis, por lo tanto es conexa y, al tener $n - 1$ arcos, también acíclica.
- $(6 \Rightarrow 5)$ G es conexa y acíclica por hipótesis. Entonces

$$\sum_{x \in X} g^-(x) = n - 1$$

. Supóngamos $g^-(s) \neq 0$ entonces existiría un arco, (y, s), conectado a s pero al ser s raíz un camino desde el a y con el cual se formaría un ciclo contradiciendo la hipótesis.

Al ver s raíz $g^-(x) \geq 1 \; \forall x \in X, x \neq s$.

Supongamos $g^-(x) > 1 \; \forall x \in X, x \neq s$. Existirían entonces dos caminos desde la raíz hasta x con lo cual

$$\sum_{x \in X} g^-(x) > n - 1$$

con lo cual se contradice la hipótesis.

- $(5 \Rightarrow 4)$ G es acíclica por hipótesis y tiene n-1 arcos (pues $g^-(s) = 0$ y $g^-(x) = 1 \; \forall \; x \neq s$), por lo tanto es conexa.
- $(4 \Rightarrow 3)$ G es conexa y tiene $n - 1$ arcos (puesto que $g^-(s) = 0$ y $g^-(x) = 1 \; \forall \; x \neq s$) por lo tanto es también acíclica.

Supongamos que s no es raíz, es decir, existe j al que no hay camino desde s. Por otro lado, sabemos que $g^-(j) = 1$. Sea x el antecesor de j $x \neq s$ pues si lo fuera existiría un camino desde s hasta x. El antecesor de x no puede ser s por la misma razón. Así, debe llegar el momento en que el antecesor de k para algún k en la cadena sea j, con lo cual se forma un ciclo en G. Lo cual contradice la hipótesis. Por lo tanto, s es raíz de G.

Quitemos un arco (i, j) de la digráfica G para formar una nueva, $G' = [X, A - \{(i, j)\}]$ de tal manera que s es aún raíz de G'. Esto implica que $g^-(j) > 1$ lo cual contradice la hipótesis.

- $(3 \Rightarrow 2)$ Al ser s raíz existe un camino de s a x, $\forall\, x \in X, x \neq s$. Por demostrar que el camino es único.

  Supongamos que existen dos caminos, C_1 y C_2, entre s y y para algún $y \in X$. Sea (k,l) un arco que está en el camino C_1 pero no en C_2. Si quitamos este arco existe camino entre s y y. Contradicción con la hipótesis. Por lo tanto, el camino es único.

- $(2 \Rightarrow 1)$ El que G tenga una raíz es inmediato de la hipótesis. Por demostrar que G es un árbol.

  Al existir un camino entre s y x $\forall\, x \in X$ se implica que G es conexa.

  El que este camino sea único $\forall x \in X$ implica que G es acíclica.

  Por lo tanto, G es un árbol.

■

En las dos secciones siguientes veremos los algoritmos presentados por Dijkstra para resolver al problema de encontrar la arborescencia de rutas más cortas de raíz s en una red, $R = [X, A, C]$. El primero se aplica sólo cuando la función de costo toma valores no negativos y el segundo que llamaremos General pues es una generalización del primero, admite que esta función tome cualquier valor. Ambos métodos sirven para encontrar la solución óptima al problema de la ruta más corta entre todo par de nodos.

El método para costos no negativos es importante por dos cosas:

- La gran cantidad de problemas que pueden modelarse con éste tipo de redes justifica el algoritmo y
- Es el primer paso del algoritmo General presentado después.

Estos algoritmo son capaces de detectar no sólo la solución óptima sino también de decir cuando no existe, lo cual puede ocurrir por una de dos razones:

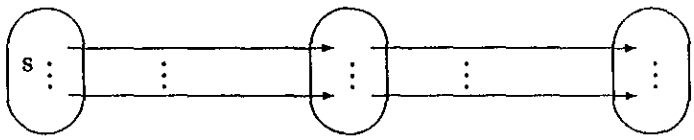
- El nodo, s, dado como raíz no es tal, o
- existe, al menos, un circuito negativo.

Vamos a presentar ahora el primero de los algoritmos mencionados.

2.2 Algoritmo de Dijkstra para costos no negativos

Este método se basa en la asignación de etiquetas "permanentes" a los nodos para los cuales ya se conocen los costos de las rutas más cortas de la raíz s a ellos. Sea R este conjunto de nodos. Los nodos sucesores de estos se etiquetan "temporalmente" y se ordenan por la distancia mínima, de

menor a mayor, de la raíz hasta ellos. En la primera iteración, el conjunto R contendrá únicamente el nodo raíz; es decir, sólo la raíz estará etiquetada de manera permanente. La distancia de la raíz a los nodos etiquetados de manera temporal se mejora continuamente y en cada iteración se agrega exactamente un nodo x a R. Este nodo x es tal que su distancia desde la raíz es la más corta posible. Cuando todos los nodos están en S se ha encontrado la solución deseada.



Nodos etiquetados como permanentes | Nodos etiquetados como temporales | Nodos con etiqueta inicial

Presentemos a continuación del seudocódigo del algoritmo:

Algoritmo

En este algoritmo se utilizan tres etiquetas para los nodos: inicial, temporal y definitiva.

---

1. Inicialización de etiquetas.

   $Costo\_Min(x) = 0 \quad \forall \ x \in X$

   $Ant\_Ruta\_Min(x) = x \quad \forall\, x \in X$

   Todos los nodos tienen una etiqueta de revisión del algoritmo.

2. Sea C el conjunto de nodos que están marcados de manera temporal junto con la distancia mínima que hay desde la raíz hasta él. (Los elementos de C están ordenados por la distancia mínima desde la raíz. Inicialmente en C sólo está la raíz.

3. Se saca un elemento de C y guardarlo en p.

4. Se marca el nodo p como definitivo, después se considera cada nodo $i \in \Gamma^+(p)$ y se revisa

   - Si no lo ha tocado el algoritmo:
     - Marcar i temporalmente.
     - Hacer $Costo\_Min(i) = Costo\_Min(p) + Costo(p, i)$.
     - Ant_Ruta_Min(i)=p.
     - Meter i en C.
   - Si está marcado como temporal y si $Costo\_Min(i) > Costo\_Min(p) + Costo(p, i)$:
     - Hacer $Costo\_Min(i) = Costo\_Min(p) + Costo(p, i)$.

- Hacer Ant_Ruta_Min(i)=p.
- Actualizar en C el costo mínimo de i.

5. Ir a (3), mientras $C \neq \phi$

Obsérvese que si al terminar el algoritmo quedan nodos con etiqueta inicial no existe una arborescencia de peso mínimo.

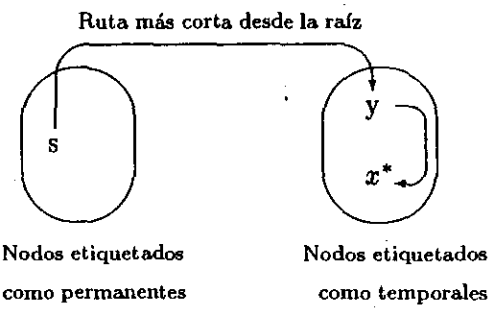
Presentaremos ahora la justificación de este algoritmo.

2.2.1 Justificación teórica del algoritmo

Nótese que si, al terminar el algoritmo, todos los nodos tienen etiqueta definitiva la digráfica generada tendrá $n-1$ nodos y una raíz, por esta razón dicha digráfica es una arborescencia. Falta demostrar que la arborescencia generada es de rutas más cortas. Demostraremos que, para los nodos etiquetados de forma permanente la distancia de la raíz al nodo, dada por el algoritmo, es la longitud más corta desde la raíz a ellos, para lo cual usaremos el método de inducción sobre el número de iteraciones. Es claro para $k = 1$. Supongámoslo válido para la k-ésima iteración. Sea S el conjunto de nodos etiquetados de forma definitiva y S' el conjunto de nodos con etiquetas temporales en la k-ésima iteración. Al final de la iteración $k+1$ la distancia de x para $x \in S'$ es la costo más corta de la raíz a x que contiene nodos de S. Esto porque sólo se marca definitivo un nodo en cada iteración. En particular, sucede para x^* (el primer nodo en C, es decir, el que tiene menor distancia desde la raíz).

Supóngamos que la ruta más corta desde la raíz hasta x^* no contiene solo nodos de S. Sea y el primer nodo, en el camino más corto de s a x^*, que no está en S. La longitud del camino que une y con x^* es no negativa, sea D esta costo. El tramo del camino de s a x^* que une s con y, contiene solo nodos de S. Así, $C(y)$ es el costo de la ruta má s corta que contiene todos sus nodos en S, luego:

$$C(y) + D = C(x^*)$$



lo cual implica

$$C(y) = C(x^*) - D < C(x^*)$$

lo cual es un contradicción pues x^* es el mínimo de los etiquetados temporales. Podemos entonces concluir que la ruta más corta de s a x^* contiene solo nodos de S y por lo tanto $C(x^*)$ es su costo.

Obsérvese finalmente, que si al finalizar el algoritmo algún nodo conservó la etiqueta inicial, (-1), no hay camino hasta el por lo que el nodo dado como raíz no es tal.

■

Las estructuras de datos que usaremos en la implementación computacional se presentan aquí.

2.2.2 Descripción de la implementación con estructuras de datos

La digráfica la vamos a guardar con listas enlazadas: en una lista, ordenada por el número del nodo, guardaremos la información referente al mismo. Cada elemento de esta lista tendrá los siguientes datos:

- Número de nodo,
- Nodo antecesor en la ruta.
- Distancia mínima desde la Raíz.
- Etiqueta, puede tomar uno de estos valores: $-1, 0$ ó 1.

  El -1 indicará que el nodo no ha sido revisado por el algoritmo.

  El 0 marca temporal.

  El 1 marca definitiva.

El nodo antecesor en la ruta es útil al recuperar el camino desde la raíz al nodo.

La estructura usada para los nodos se muestra a continuación:

Num_Nodo	Ant_Ruta_Min	Costo_Min	Etiqueta

Los arcos conectados con un nodo, también estarán guardadas en listas ordenadas por el número del nodo extremo del arco. Cada miembro de estas listas tendrá lo siguiente:

- Número del nodo extremo del arco.
- Costo del arco.

La estructura usada para los arcos se ilustra aquí.

Nodo final	Costo

Los nodos tienen además dos apuntadores: uno al siguiente nodo en la lista y otro a la lista de arcos conectadas con él. Los arcos tienen un apuntador al siguiente en la lista. Estos apuntadores no los representaremos explícitamente.

Revisemos ahora el algoritmo de Dijkstra paso a paso. En cada uno de ellos diremos como será implementado con estructuras de datos.

1. Inicialización de etiquetas.

   $Costo\_Min(x) = 0 \quad - x - X$

   $Ant\_Ruta\_Min(x) = x \quad - x - X$

   Todos los nodos tienen una etiqueta de revisión del algoritmo.

   La etiqueta que usaremos para indicar que el nodo no ha sido usado por el algoritmo será -1.

2. Sea C el conjunto de nodos que están marcados de manera temporal junto con la distancia mínima que hay desde la raíz hasta él. (Los elementos de C están ordenados por la distancia mínima desde la raíz. Inicialmente en C sólo está la raíz.

   Este conjunto, C, lo vamos a implementar como un monticulo que inicialmente tendrá solo al nodo raíz y su distancia mínima.

3. Se saca un elemento de C y guardarlo en p.

   Quitar la raíz del montículo y asignar su valor a p.

4. Se marca el nodo p como definitivo, después se considera cada nodo $i - \Gamma^+(p)$ y se revisa

   Etiquetar p como definitivo y para cada nodo en su lista de sucesores, revisar la etiqueta; en caso de que esta sea

   - Inicial:
     - Marcar i temporalmente,con un 0.
     - Hacer $Costo\_Min(i) = Costo\_Min(p) + Costo(p, i)$.
     - Ant_Ruta_Min(i)=p.
     - Meter i en C.
   - Temporal

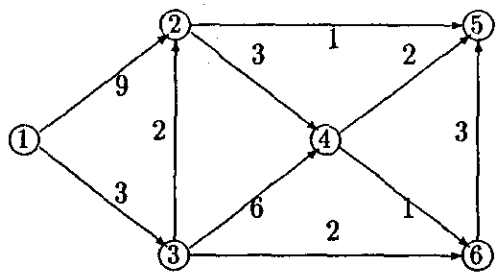
     Revisar la distancia mínima desde la raíz, si es mayor que la de un camino que pase por p, cambiarlo de la manera siguiente:
     - Hacer $Costo\_Min(i) = Costo\_Min(p) + Costo(p, i)$.
     - Hacer Ant_Ruta_Min(i)= p.
     - Actualizar en C la distancia mínima de i.

5. Ir a (3), mientras $C - \neq \phi$

Veamos ahora un ejemplo que ilustre el procedimiento:

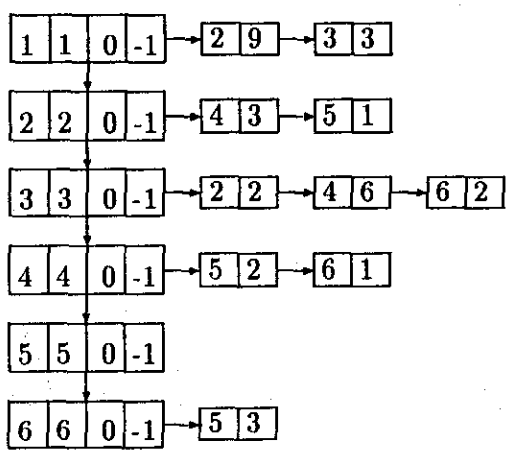
2.3 Ejemplo:

Considérese la siguiente digráfica:



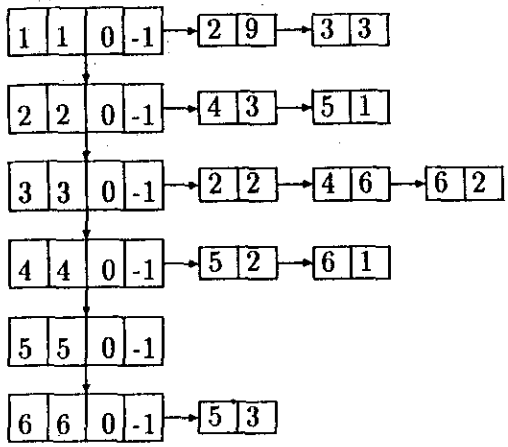
Encontraremos la arborescencia de rutas más cortas de raíz el nodo 1.

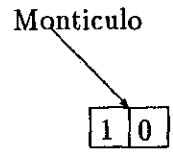
Inicializando etiquetas y al montículo.



Monticulo= ϕ

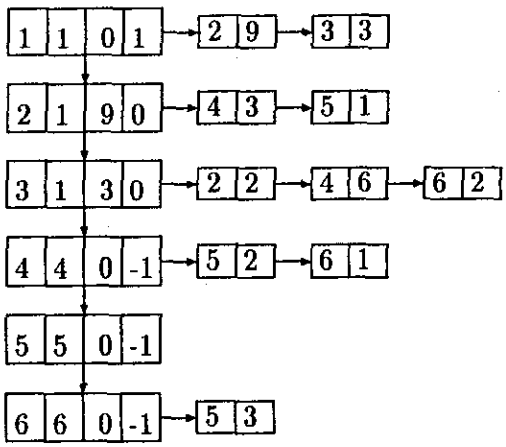
Entra la raíz al montículo.

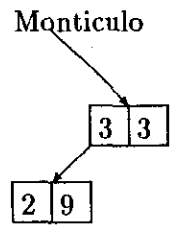




Primera iteración

Sacar un elemento del montículo: el 1. Marcarlo definitivo en la solución y marcar como temporales a los nodos 2 y 3 y ponerles como costo mínimo el costo del arco que los conecta con el 1. Entran al montículo junto con su costo mínimo.





Segunda iteración

Sacar un elemento del montículo: el 3. Marcarlo definitivo y etiquetar a el 4 como temporal y ponerle costo como sigue:

$$Costo\_Minimo(4) = Costo\_Minimo(3) + Costo(3,4) = 3 + 6 = 9$$

lo mismo con el nodo 6, marcarlo temporal y ponerle como costo mínimo

$$Costo\_Minimo(6) = Costo\_Minimo(3) + Costo(3,6) = 3 + 2 = 5$$

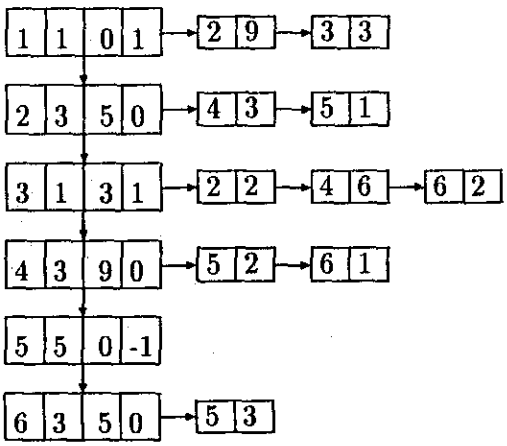
Comparar el costo mínimo del 2, que ya está marcado temporal, con el costo de la ruta que pasa

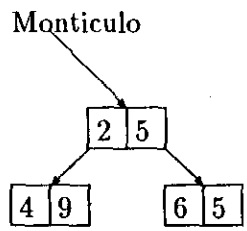
por el 3, si la segunda ruta mejora el costo, cambiarla, así,

$$Costo\_Minimo(2) = 9 > 3 + 2 = Costo\_Minimo(3) + Costo(3, 2)$$

por lo tanto, cambiar la ruta al 2: El antecesor será ahora el 3 y su costo mínimo desde la raíz será 5.

Tanto el 4 como el 6 entran en el montículo con sus costos mínimos, al 2 solo se le actualiza el costo con el que había entrado.





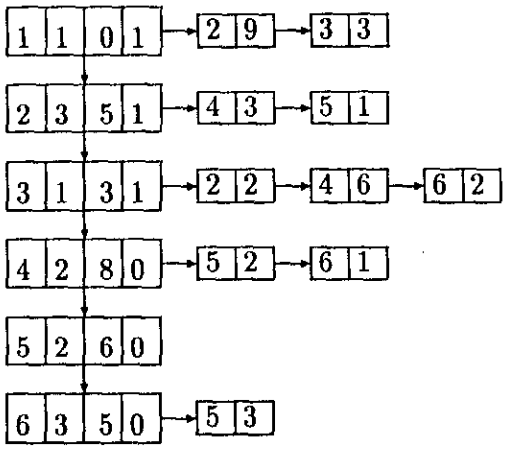
Tercera iteración

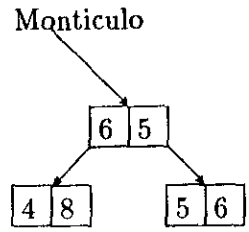
Sacar el 2 del montículo. Marcarlo definitivo y revisar sus sucesores. Revisar si la ruta al nodo 4, que ya tiene etiqueta temporal, es más corta que la ruta que obtenemos al pasar por el 2.

$$Costo\_Minimo(4) = 9 > 5 + 3 = Costo\_Minimo(2) + Costo(2, 4)$$

Como si mejora la ruta al nodo 4 se cambia. Ahora el antecesor en la ruta mínima del 4 será el 2 y su costo mínimo es 8. Actualizar este costo en el montículo. El nodo 5 se marca temporal y se cambia su costo mínimo así:

$$Costo\_Minimo(5) = Costo\_Minimo(2) + Costo(2, 5) = 5 + 1 = 6$$

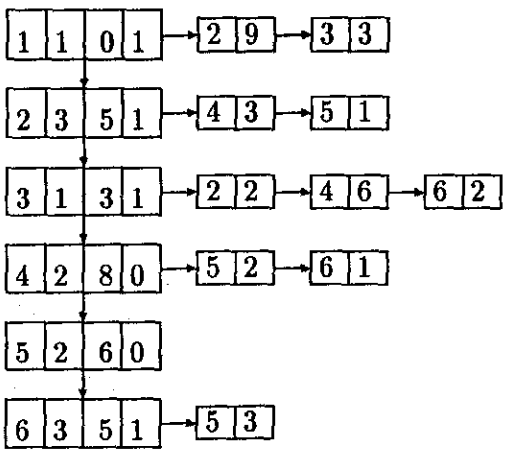


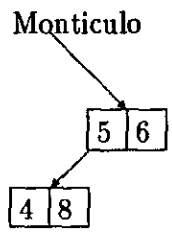


Cuarta iteración

Sacar el nodo 6 del montículo. Marcarlo definitivo y revisar el sucesor: el 5. El nodo 5 tiene etiqueta temporal, por lo que debemos revisar si se mejora la ruta. Esta no se mejora pues

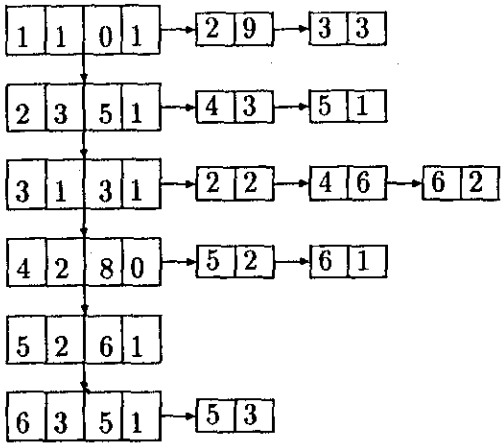
$$Costo\_Minimo(5) = 6 < 5 + 3 = Costo\_Minimo(6) + Costo(6,5)$$

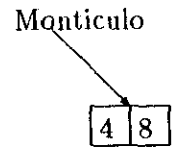




Quinta iteración

Sacar el 5 del montículo, marcarlo definitivo y como no tiene sucesores, sacar otro del montículo.

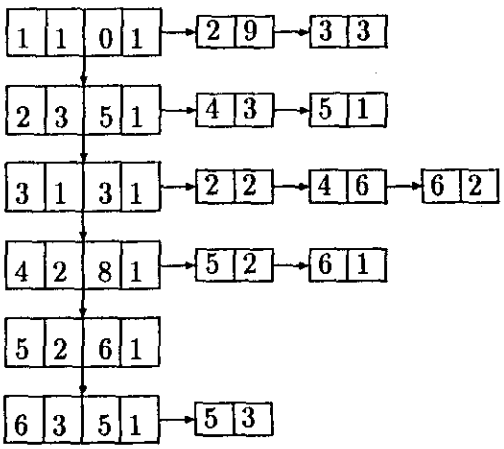




Sexta iteración

Sacar el 4 del montículo y marcarlo como definitivo, los sucesores están marcados como tal por lo que hay que sacar otro del montículo. Es vacío. El algoritmo terminó.

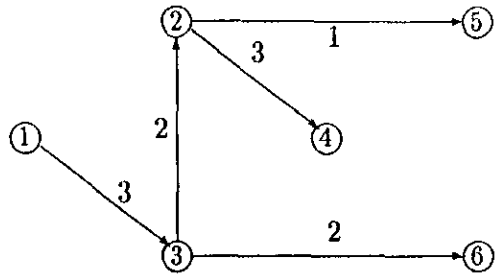
Esta es la digráfica resultante.



Monticulo= ϕ

Todos los nodos están marcados como definitivos por lo que si hay solución al problema de la arborescencia más corta. La solución se dá a continuación.

Solución



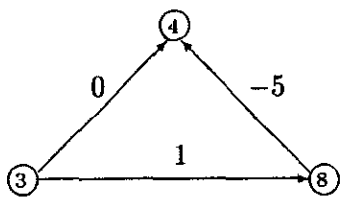
Ruta más corta del nodo 1 al

Nodo	Ruta	Costo
2	$1 \rightarrow 3 \rightarrow 2$	Costo= 5
3	$1 \rightarrow 3$	Costo=3
4	$1 \rightarrow 3 \rightarrow 2 \rightarrow 4$	Costo=8
5	$1 \rightarrow 3 \rightarrow 2 \rightarrow 5$	Costo=6
6	$1 \rightarrow 3 \rightarrow 6$	Costo=5

El caso general, para costos reales, de este algoritmo lo vamos a presentar en la siguiente sección.

2.3 Caso General

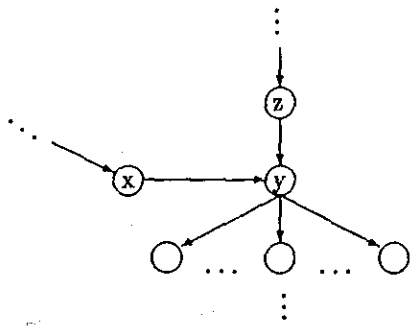
Veamos un ejemplo, ejemplificando porqué es necesario un algoritmo distinto al anterior de Dijkstra para el caso donde la función de costo puede tomar cualquier valor, en particular negativos.



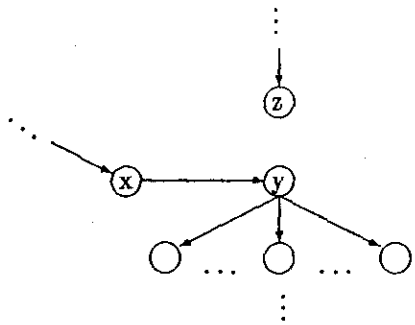
En esta red al aplicar el algoritmo de Dijkstra, el camino más corto entre 3 y 4 tiene costo 0, cuando en realidad el camino más corto pasa por el nodo 8 y tiene costo -4. Así, el algoritmo de Dijkstra no puede usarse pues puede conducir al error.

Es por eso que se hace necesario presentar otro algoritmo que solucione estos casos. El método que presentaremos es una generalización del algoritmo de Dijkstra para costos no negativos. (Lo llamaremos Dijkstra General.)

El Dijkstra General comienza con una arborescencia cualquiera y la optimiza. Esta arborescencia inicial la obtiene al aplicar el algoritmo de Dijkstra para costos no negativos. Una arborescencia no es de rutas más cortas si existe algún nodo para el cual el único camino de la raíz hasta el no es el más corto; de ahi que, para verificar que una arborescencia es de rutas más cortas, debe agregarse un arco $a = (x, y)$ que estando en la digráfica no está en la solución. Si esto pasa se intercambian a y el arco que, estando en la arborescencia, tiene como nodo final a y para obtener una arborescencia mejor. Es importante, al hacer el intercambio conocer la parte de la arborescencia que cuelga del nodo y pues la ruta más corta a todos los nodos en esa parte cambiará. Si al revisar todos los arcos que no están en la arborescencia no hubo necesidad de cambios, la solución dada por Dijkstra es la óptima.



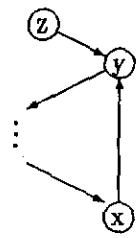
Comparando las rutas.



El arco (x, y) mejoró la ruta por lo tanto se cambió.

Otro problema que puede presentarse es que, al poder tomar costos negativos, existan ciclos negativos en la red. Debemos prevenir este caso contando con una herramienta que los detecte. Para esto, cada vez que encontremos un arco que mejore las rutas deberemos observar que, o el nodo final de este arco sea la raíz o si al agregar un arco y quitar el correspondiente resulta que este no formaba parte del ciclo que se formó. En ambos casos la red resultante no es arborescencia pues contiene un circuito.

Si y =raíz se forma un circuito



El arco (z, y) no está en el ciclo.

Afirmamos que este circuito es negativo.

En efecto, la operación de agregar un arco adecuado, a, y eliminar el correspondiente, b, mejora la ruta al nodo y pero también las que tengan como nodo intermedio a y. De esta manera al formarse un circuito se mejoran las rutas de tal circuito a través de él mismo.

Este es el algoritmo Dijkstra General.

---

1. Encontrar una arborescencia cualquiera de raíz s en la red.
2. Calcular, para un arco (i,j) que no esté en la arborescencia la suma del costo de la ruta mínima hasta el nodo i más el costo del arco (i,j) y compararla con el costo de la ruta mínima a el nodo j. Si sucede que esta suma es mayor o igual que el costo de la ruta hasta j para todo arco (i,j) que no está en la red terminar. De otro modo, si para algún nodo j se mejora la ruta al meter el arco (i,j) a la arborescencia, Ir a 3.
3. Si $j = s$ terminar (se ha detectado un circuito negativo). En otro caso sea $(k,j) \in A$ el único arco, incidente en j, en la arborescencia. Reemplazar este arco por (i,j). Si la digráfica resultante es una arborescencia actualizar las etiquetas de todos los nodos x, para los cuales la ruta de s a x contenga a j como nodo intermedio de acuerdo con

$$Costo\_Minimo(x) = Costo\_Minimo(x) - Costo$$

donde $Costo\_Minimo(x)$ es el costo de la ruta mínima desde la raíz hasta x, $Costo = Costo\_Minimo(j) - Costo\_Minimo(i) - Costo(i,j)$. Ir al paso (2). En caso contrario, si la digráfica resultante no es arborescencia, terminar; existe un circuito negativo y por lo tanto no hay solución.

La justificación de este algoritmo se presenta a continuación.

2.3.1 Justificación teórica del algoritmo

Si

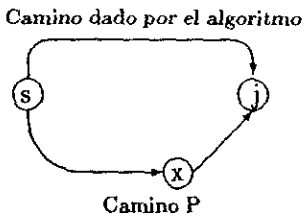
$$Costo\_Minimo(i) + Costo(i,j) \geq Costo\_Minimo(j), \quad \forall (i,j) \in A$$

el algoritmo termina.

Suponer que el Costo_Minimo(j), al final del algoritmo, no es la longitud de una ruta más corta de la raíz s a j para algún j (si existe mas de un nodo con esta característica tomar aquel tal que la ruta en la arborescencia de s a j tenga el menor número de arcos). Por construcción, Costo_Minimo(j) es el costo de un camino de s a j. Sea P el camino más corto de s a j entonces el costo de P, C(P), es menor que Costo_Minimo(j). Sea C(x) el costo en la ruta más corta de s a x, por lo tanto,

$$C(P) = C(x) + Costo(x,j) < Costo\_Minimo(j)$$

lo cual es una contradicción. Así que la arborescencia de raíz s generada es de rutas más cortas.



Debemos conocer la profundidad de cada nodo así y los nodos antecesor en la ruta y sucesor en preorden para facilitar la operación de intercambio de arcos. Veamos como serán las estructuras de datos que usaremos.

2.3.2 Implementación con estructuras de datos

La digráfica la vamos a guardar con listas enlazadas: en una lista, ordenada por el número del nodo, guardaremos la información referente al mismo. Cada elemento de la lista tendrá los siguientes datos:

- **Número de nodo.**
- **Nodo antecesor en la ruta mínima.**
- **Costo mínimo desde la Raíz, costo mínimo desde la raíz al nodo.**
- **Profundidad**, profundidad del nodo en la arborescencia. Se cal cula de la siguiente manera:

  Al nodo raíz se le asigna profundidad 0 al resto se le asigna la profundidad del padre más uno.
- **Sucesor en preorden**, es el número del nodo que es sucesor en el sentido del recorrido preorden.

La estructura usada para los nodos se muestra a continuación:

Num_Nodo	Ant_Ruta_Min	Costo_Min	Profundidad	Suc_Preorden

Los arcos conectados con un nodo, también estarán guardadas en listas ordenadas por el número del nodo extremo del arco. Cada miembro de estas listas tendrá lo siguiente:

- Número del nodo extremo del arco.
- Costo del arco.

La estructura usada para los arcos se ilustra aquí.

Nodo final	Costo

Los nodos tienen además dos apuntadores: uno al siguiente nodo en la lista y otro a la lista de arcos conectadas con él. Las arcos tienen un apuntador al siguiente en la lista. Estos apuntadores, no los representaremos explícitamente.

La manera de implementar el algoritmo General de Dijkstra se presenta ahora:

1. Encontrar una arborescencia cualquiera de raíz s en la red.

   Aplicaremos un Dijkstra para costos no negativos.

2. Calcular, para un arco (i,j) que no esté en la arborescencia la suma del costo de la ruta mínima hasta el nodo i más el costo del arco (i,j) y compararla con el costo de la ruta mínima a el nodo j. Si sucede que esta suma es mayor o igual que el costo de la ruta hasta j para todo arco (i,j) que no está en la red terminar. De otro modo, si para algún nodo j se mejora la ruta al meter el arco (i,j) a la arborescencia.

   Para todo arco (i,j) que no esté en la arborescencia comparar

   $$Costo\_Minimo(j) \;\; y \;\; Costo\_Minimo(i) + Costo(i,j)$$

   y si sucede que

   $$Costo\_Minimo(j) > Costo\_Minimo(i) + Costo(i,j)$$

   meter el arco (i,j) en una pila.

3. Si $j = s$ terminar (se ha detectado un circuito negativo). En otro caso sea $(k,j) \in A$ el único arco, incidente en j, que está en la arborescencia. Reemplazar este arco por (i,j). Si la digráfica resultante es una arborescencia actualizar las etiquetas de todos los nodos x, para los cuales la ruta de s a x contenga a j como nodo intermedio o final de acuerdo con

   $$Costo\_Minimo(x) = Costo\_Minimo(x) - Costo$$

   donde $Costo = Costo\_Minimo(j) - Costo\_Minimo(i) - Costo(i,j)$. Ir al paso (2). En caso contrario, terminar; existe un circuito negativo y por lo tanto no hay solución.

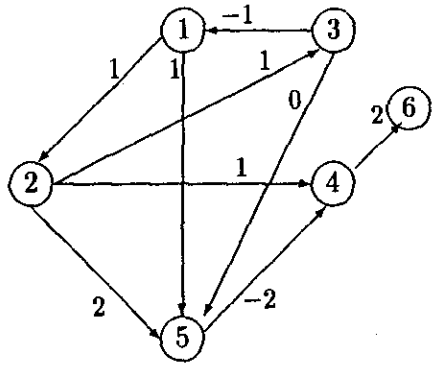
   Sacar un arco (i,j) de la pila y si j es la raíz, terminar pues existe un ciclo negativo.

   En caso de que $j \neq s$ buscar si se formó un circuito negativo al meter a (i,j). Para buscar este circuito, debemos buscar si hay camino, en la solución, de j a i. Con este fin, calcular la diferencia de profundidades de los nodos j e i y asignarlo a la variable *Resta*, con esto ir

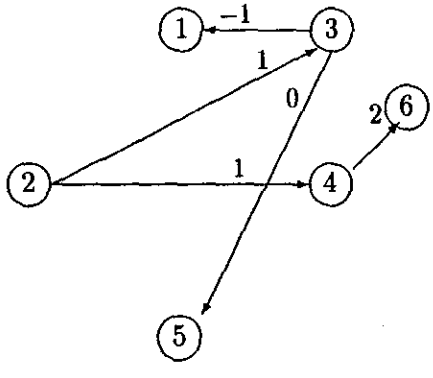
al nodo i y tantas veces como indique *Resta* buscar el antecesor en la ruta mínima. Si se llega al nodo j, hemos encontrado un circuito negativo. En otro caso quitar el arco (k, j) de la solución y actualizar la información de los nodos que tienen como intermedio al nodo j del modo indicado.

2.3.3 Ejemplo:

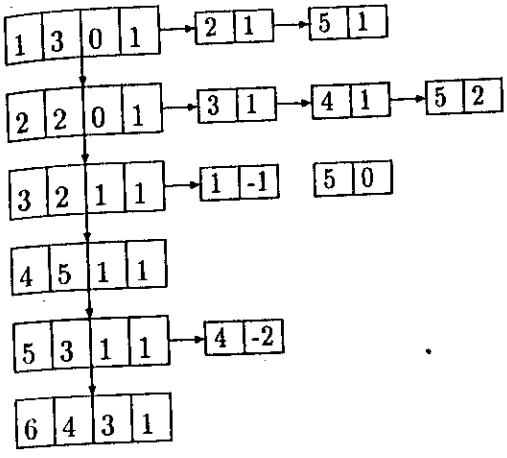
Considerese la siguiente digráfica



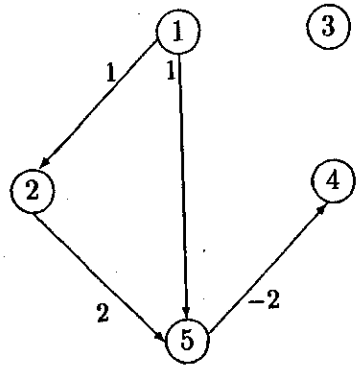
Solución inicial enraizada en el nodo 2:



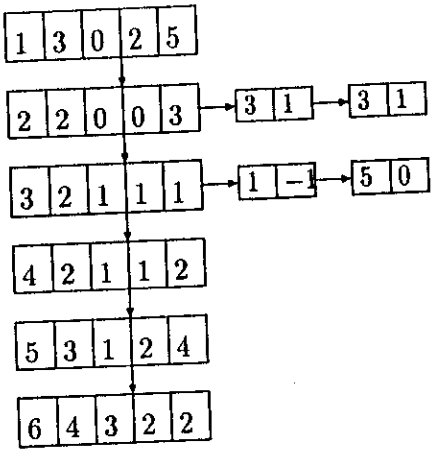
3. CASO GENERAL



Estos arcos no están en la solución:



Grafica solución:



Vamos a escoger los arcos candidatos en la solución, es decir, aquellos que, de entrar, disminuyen el costo de la ruta a su nodo final. Los vamos a revisar en el orden en que aparecen recorriendo la digráfica inicial. Los arcos que mejoren la ruta entran en una pila.

- $(1,2)$
  $\text{Costo\_Min}(2) = 0$
  $\text{Costo\_Min}(1)+\text{Costo}(1,2) = 0+1 = 1$
- $(1,5)$
  $\text{Costo\_Min}(5) = 1$
  $\text{Costo\_Min}(1)+\text{Costo}(1,5) = 0+1 = 1$
- $(2,5)$
  $\text{Costo\_Min}(5) = 1$
  $\text{Costo\_Min}(2)+\text{Costo}(2,5) = 0+2 = 2$
- $(5,4)$
  $\text{Costo\_Min}(4) = 1$
  $\text{Costo\_Min}(5)+\text{Costo}(5,4) = 1-2 = -2$

Sacamos el arco que está en la pila, debemos checar si este arco forma un ciclo negativo al entrar en la solución en cuyo caso no existe solución al problema de encontrar la arborescencia de ruta más cortas enraizada en el nodo 2.

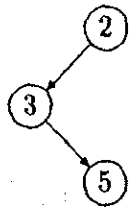
Para saber si el arco $(5,4)$ forma un ciclo con los que ya están en la solución hay que buscar si existe camino del 4 al 5, de ser así , un ciclo negativo ha sido detectado.

Sea *Resta* la diferencia de profundidades de los nodos 5 y 4.

$$Resta = Profundidad(4) - Profundidad(5) = 3 - 2 = 1$$

Tantas veces como indique el número *Resta* debemos retroceder por la ruta mínima al nodo 5. Si llegamos al nodo 4, existe camino del 4 al 5.

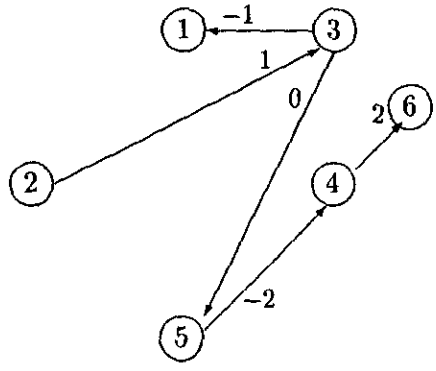
Esta es la ruta mínima desde la raíz al 5.

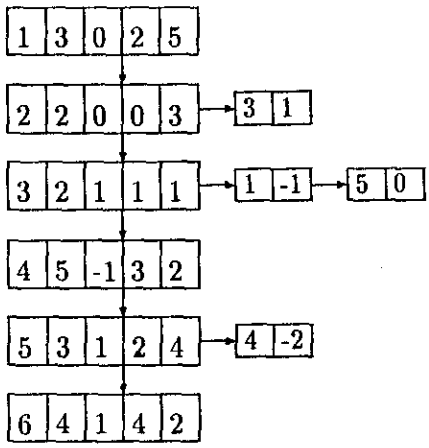


Como vemos no se ha formado ningún ciclo por lo que cambiaremos el arco $(2,4)$ por el $(5,4)$. El nodo 6, único sucesor del 4, sufre cambios: Ahora su costo mínimo será

$$Costo\_Min(6) = Costo\_Min(6) - [Costo\_Min(4) - Costo\_Min(5) - Costo(5,4)] = 3 - [1 - 1 + 2] = 1$$

La profundidad del 4 es igual a la del 5 mas uno, es decir 3 y la del 6, 4. La digráfica resultante es la siguiente:





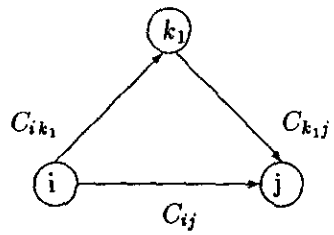
Evidentemente, un caso particular del problema de encontrar la arborescencia de rutas más cortas de raíz x es encontrar la ruta más corta entre todo par de nodos en una red $R = [X, A, C]$ (aplicando un Dijkstra General $\forall x \in X$, el problema queda resuelto).

Existen, sin embargo, procedimientos más efectivos como el que presentamos en esta sección: el algoritmo de Floyd, para el cual la función de costo puede tomar cualquier valor real.

2.4 Algoritmo de Floyd

Cada arco (i, j) tiene asociado un costo C_{ij}; se considera que el costo del camino de un nodo i a si mismo es cero.

Lo que hace el algoritmo en la primera iteración es comparar este camino con longitud de paso igual a 1 (tomando como longitud de paso el número de arcos necesario para comunicar i con j) con los caminos existentes con longitud de paso igual a 2, es decir, con el costo del camino $i \rightarrow k_1 \rightarrow j$ donde k_1 es el único nodo intermedio en este camino. Si el costo se mejora para alguna k_1 se cambia la ruta de i a j haciéndola pasar por k_1, indicando el nodo antecesor a j en la ruta.



Comparando el arco (i, j) con el camino ikj

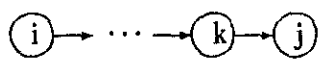
Puede suceder que el arco (i, j) no exista, pero que en la primera iteración se obtuvo un camino que pasa por k_1 y que comunica a los nodos i y j ; en este caso el arco (i, j) debe agregarse a la solución indicando que tiene a k_1 como nodo intermedio y con un costo igual a

$$C_{ij} = C_{ik_1} + C_{k_1j}.$$

Al finalizar esta iteración, tenemos que los caminos tienen como nodo intermedio al nodo k_1 o a ninguno.

En la segunda iteración comparamos los caminos de i a j que tenemos con el costo de caminos de longitud 3 que pasen por k_2. Así, al finalizar esta iteración los caminos obtenidos tendrán a k_1 o k_2 como nodos intermedios o bien a ninguno.

Continuando del mismo modo, en la k -ésima iteración del algoritmo de Floyd se calcula la ruta más corta entre i y j que puede incluir a algunos o a todos los primeros $k-1$ nodos intermedios. De esta manera, al final de la n -ésima iteración se tendrá la ruta más corta entre toda pareja de nodos conteniendo esta algunos o todos los nodos como nodos intermedios.



k es nodo intermedio en el camino de i a j

Debemos, por lo tanto, observar que si al finalizar la aplicación del algoritmo no existe el arco (i, j) es porque no existe ruta entre ellos.

Por otro lado, si el costo de (i, i), para algún i, cambia tomando un valor negativo se ha detectado un circuito negativo.

Este es el algoritmo de Floyd que resuelve el problema.

Sea C_{ij} el costo del camino del nodo i al j. Sea k el número de iteraciones.

1. $C_{ii} = 0, \forall\, i \in X$. Hacer $k = 0$.
2. Hacer $k = k + 1$. Para todo $i \in \Gamma^-(k)$, y $j \in \Gamma^+(k)$, hacer

$$C_{ij} = min\{C_{ij}, C_{ik} + C_{kj}\}$$

3. (a) Si $C_{ii} < 0$ para alguna $i \in X$, terminar. En este caso existe un circuito negativo que contiene al nodo i y, por lo tanto, no hay solución.
   (b) Si $C_{ij} \geq 0 \; \forall \; i \in X$, y $k = n$, terminar; C_{ij} es la longitud del camino más corto de i a j.
   (c) Si $C_{ii} \geq 0 \; \forall \; i \in X$ y $k < n$, ir al paso 2.

Para recuperar la ruta mínima entre los nodos i y j seguir los siguientes pasos:

1. Si $j \neq i$ buscar a j como sucesor de i, y hacer Ant_Ruta_Min = antecesor en la ruta mínima del nodo j.
2. Hacer $j = Ant\_Ruta\_Min$. Ir a 1.

2.4.1 Justificación teórica del algoritmo.

El número de iteraciones que necesita este algoritmo es igual al número de nodos en la red, n, a menos que termine por haber encontrado un ciclo negativo. Para demostrar que el algoritmo encuentra un óptimo usaremos inducción sobre el número de iteraciones.

Al comenzar la iteración uno el arco (i, j) tiene asociado un número que representa el costo de la ruta más corta entre i y j que no contiene nodos intermedios. Durante esta iteración se compara este precio con el de la ruta que pasa por el nodo 1, es decir, compara C_{ij} y $C_{i1} + C_{1j}$, obteniéndose la ruta más corta entre i y j que, o no tiene nodos intermedios o tiene al 1 como tal.

Supongamos que al final de la iteración $k-1$, C_{ij} representa la longitud de la ruta más corta que, o no tiene nodos intermedios o tiene algunos o a todos los $k-1$ primeros nodos revisados como intermedios.

Durante la k-ésima iteración se realiza la comparación entre esta ruta y la que pasa por el nodo k, es decir, compara C_{ij} y $C_{ik} + C_{kj}$, así, al final de esta iteración C_{ij} representa la longitud de la ruta más corta entre i yj que puede contener hasta k nodos intermedios.

De aquí, podemos concluir que, al final de la iteración n se obtendrá la ruta más corta entre los nodos i y j, para todo i y j en X.

■

2.4.2 Implementación con estructuras de datos

La digráfica la vamos a guardar con listas enlazadas: en una lista, ordenada por el número del nodo, guardaremos la información referente al mismo. Cada elemento de la lista tendrá los siguientes datos:

- Número de nodo.
- Costo del nodo a si mismo.

La estructura usada para los nodos se muestra a continuación:

Num_Nodo	Costo

Los arcos conectados con un nodo, también estarán guardadas en listas ordenadas por el número del nodo extremo del arco. Cada miembro de estas listas tendrá lo siguiente:

- Número del nodo extremo del arco.
- Costo del arco.
- Nodo intermedio predecesor inmediato al nodo final del arco.

La estructura usada para los arcos se ilustra aquí.

Nodo final	Costo	Predecesor

Los nodos tienen además dos apuntadores: uno al siguiente nodo en la lista y otro a la lista de arcos conectados con él. Las arcos tienen un apuntador al siguiente en la lista. Estos apuntadores, no los representaremos explícitamente.

El algoritmo de Floyd, seguido detalladamente aclarando las estructuras de datos usadas, se muestra ahora.

1. $C_{ii} = 0, \forall\, i \in X$. Hacer $k = 0$.

   Este costo se guarda en la estructura del nodo.

2. Hacer $k = k + 1$. Para todo $i \in \Gamma^{-}(k)$, y $j \in \Gamma^{+}(k)$, hacer

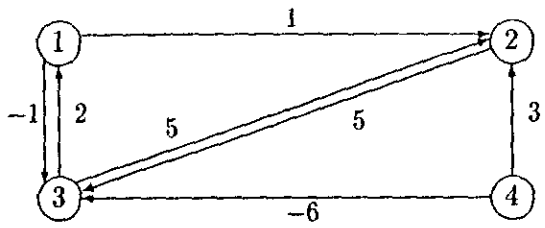
$$C_{ij} = min\{C_{ij}, C_{ik} + C_{kj}\}$$

   Recorrer los antecesores y sucesores de k buscando que nodos se pueden conectar pasando por k. Se compara el costo de este camino con el de uno que ya estaba, si resulta mejor se cambia.

3. (a) Si $C_{ii} < 0$ para alguna $i \in X$, terminar. En este caso existe un circuito negativo que contiene al nodo i y, por lo tanto, no hay solución.
   (b) Si $C_{ij} \geq 0\ \forall\ i \in X$, y $k = n$, terminar; C_{ij} es la longitud del camino más corto de i a j.
   (c) Si $C_{ii} \geq 0\ \forall\ i \in X$ y $k < n$, ir al paso 2.

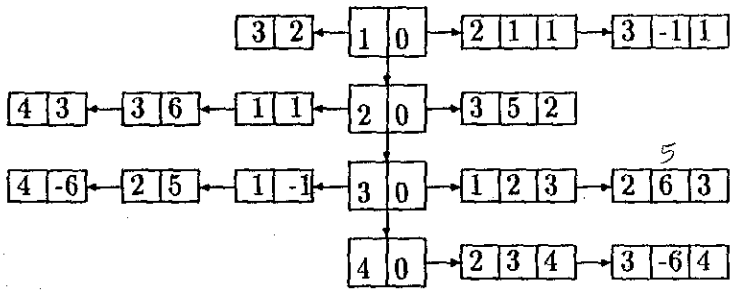
2.4.3 Ejemplo

Supongamos que tenemos la siguiente red:



deseamos encontrar la ruta más corta entre todo par de nodos utilizando el algoritmo de Floyd.

Esta es la representación con estructuras de datos:



Recorrer las listas de arcos antecesores y sucesores.

Primera iteración k=1

i=3

j=2

Comparar el costo del camino del 3 al 2 que tiene al 1 como nodo intermedio con el del arco $(3, 2)$.

$$C_{31} + C_{12} = 2 + 1 = 3 < 6 = C_{32}$$

El camino cambia.

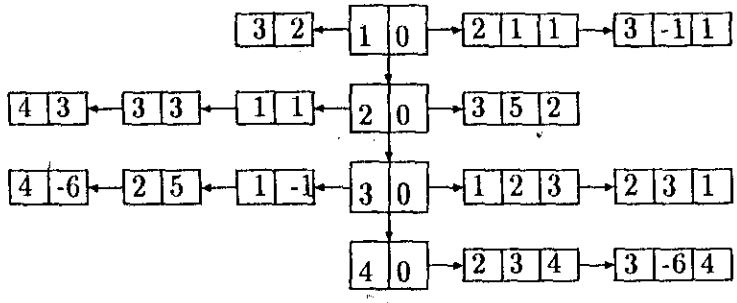
i=3

j=3

Comparar el costo del camino del 3 al 3 pasando por el 1 con el inicial, 0.

$$C_{31} + C_{13} = 2 - 1 = 1 > 0 = C_{11}$$

No cambia.



Segunda iteración k=2

i=1

j=3

Comparando costo del camino de 1 a 3 pasando por el 2 con el del arco $(1,3)$.

$$C_{12} + C_{23} = 1 + 5 = 6 > 5 = C_{1,3}$$

No Cambia.

i=3

j=3

Comparando costo del camino de 3 a 3 pasando por el 2 con el del arco $(3,3)$.

$$C_{32} + C_{23} = 3 + 5 = 8 > 0 = C_{33}$$

No cambia.

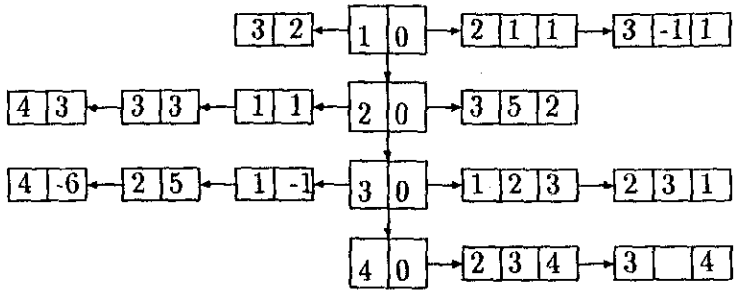
i=4

j=3

Comparando costo del camino de 4 a 3 pasando por el 2 con el del arco $(4,3)$.

$$C_{42} + C_{23} = 3 + 5 = 8 > -6 = C_{43}$$

No cambia.



Tercera iteración k=3

- C_{11}

  $C_{13} + C_{31} = -1 + 2 = 1$. No Cambia.

- C_{21}

  $C_{23} + C_{31} = 5 + 2 = 7$. El arco $(2, 1)$ no existe, se agrega.

- C_{41}

  $C_{43} + C_{31} = -6 + 2 = -3$. El arco $(4, 1)$ no existe, se agrega.

- C_{12}

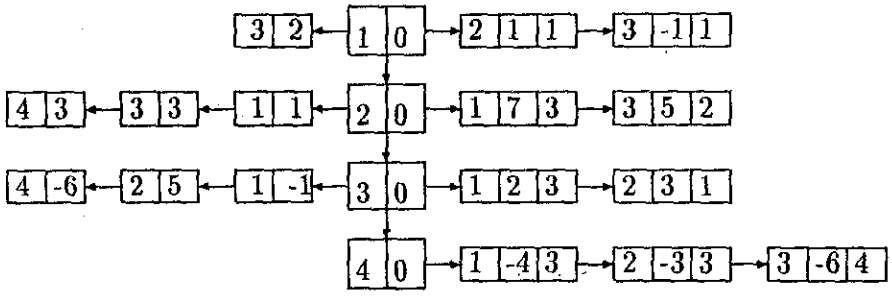
  $C_{13} + C_{32} = -1 + 3 = 2$. No cambia.

- C_{22}

  $C_{23} + C_{32} = 5 + 3 = 8$. No cambia.

- C_{42}

  $C_{43} + C_{32} = -6 + 3 = 3$. Cambia.



El nodo 4 no tiene antecesores y como es el último en la lista de nodos, el algoritmo terminó.

En la siguiente tabla se resumen los resultados.

	1		2		3		4	
	Ruta	Costo	Ruta	Costo	Ruta	Costo	Ruta	Costo
1	1→1	0	1→2	1	1→3	−1	—	—
2	2→3→1	7	2→2	0	2→3	5	—	—
3	3→1	2	3→1→2	3	3→3	0	—	—
4	4→3→1	−4	4→3→2	−6	4→3	−6	4→4	0

Conclusiones

En un principio la idea original de este trabajo era presentar las implementaciones más eficientes para el problema de ruta más corta, sin embargo al darnos cuenta de que aún las implementaciones de los algoritmos clásicos para este tipo de problema no estaban a la mano, se decidió tratar de llenar este vacío elaborando este trabajo con los aspectos teóricos de dichos algoritmos así como con propuestas probadas de implementación de los mismos. Por lo anterior el resolver el problema de ruta más corta con las estructuras de datos más eficientes (Prunning, Running y Buckets) queda como una linea de trabajo a continuar.

De igual manera cabe señalar que existen formas de implementar el algoritmo de Dijkstra General considerándolo como un caso específico del Método Simplex Especializado en Redes pero que para ser comprendido cabalmente se requiere conocer dicho método lo que queda fuera de los alcances de este trabajo.

El presente trabajo forma parte de un proyecto de desarrollo de software para flujo en redes llamado PAREIMM en el cual es seguro que a futuro se ampliará la producción de algoritmos y de material escrito, como el presente.

Apéndice A

Algoritmos para gráficas

A.1 Formato en los archivos

Deberemos tener el siguiente formato de entrada:

1. Una primera linea que contenga un 0. (Indica el editor para estos algoritmos.)
2. Una lista de aristas con su costo (un espacio deberá separar el nodo inicial, nodo final y el costo). Una arista por renglón.
3. Para terminar de teclear las aristas poner 0 0 0 en el siguiente renglón.
4. Terminar poniendo el número de nodos, el número de arcos y el peso total de la gráfica.

El nombre del archivo debe tener la extensión PRB.

El problema no deberá exceder de 20 nodos.

El archivo de salida tomará el nombre del problema y le pondrá terminación SOL. Tendrá la siguiente información:

1. Una primera linea que contiene un 0 si el problema se solucionó con el algoritmo de Prim y un 1 si se solucionó con un Kruskal.
2. La lista de aristas que están en la solución y su costo, uno por renglón. Al igual que en el archivo de entrada un espacio separa la información de cada arco.
3. Un renglón que indique que terminó de listarse la solución (0 0 0).
4. El número de nodos, el número de arcos y el costo del árbol.

Por ejemplo para el problema 1 del capítulo 1, el archivo de entrada es:

0		
1	3	1
1	2	1
1	5	3
1	4	2
3	4	1
3	5	0
5	4	2
5	2	4
2	4	3
0	0	0
5	9	17

Solucionado con el algoritmo de Prim obtenemos:

0		
1	3	1
1	2	1
3	4	1
3	5	0
0	0	0
5	4	3

Estos trabajos son parte de un software para flujo en redes, llamando PAREIMM, que incluye un manejador central. Si se quieren usar por separado, es decir, fuera del PAREIMM requieren un manejo especial. El algoritmo de KRUSKAL necesita el número de aristas en la gráfica como argumento al llamarlo. Supongamos que el archivo de nuestro ejemplo se llama PROB.PRB la manera de correr un PRIM para solucionarlo seria:

PRIM PROB.PRB

y con un Kruskal

KRUSKAL PROB.PRB 9

En ambos casos la solución quedará grabada en PROB.SOL.

A.2 Algoritmo de Prim

```
/* ========================================================================
   VERANO 93                                                    PAREIMM

   NOMBRE DEL ARCHIVO: PRIM.C
    Este archivo aplica el algoritmo PRIM a una grafica.

   FUNCIONES QUE INCLUYE:
    main: Organiza la informacion de entrada y salida.
  ======================================================================== */

#include <io.h>
#include <math.h>
#include <ctype.h>
#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <process.h>
#define         IO_ERR          0
#define         NO_PROBLEM      1
#define         DISCONECTED     2
#define         CICLO           3
#define         NO_MEM          4
#define         TOO_BIG         5
#define         NO_PRIM         6
#define         EXITO           100
#define         PRIM            0
#define  SI   'S'
#define  NO   'N'
#define  TOL    .0001
typedef unsigned short unsh;
struct Arco {
 char Revisado;                     //  s  fue revisado,  n  si no
 char En_Sol;                 //  S  esta en la solucion,  N  si no
 double Peso;
 struct Arco *Siguiente;            //  Puntero al siguiente arco
 struct Nodo *Extremo;              //  Puntero al nodo extremo
};
struct Nodo {
 unsh Numero;                       //  Numero de nodo
 char Revisado;                     //  s  si se ha revisado, n  si no
 struct Arco *Arcos;                //  Puntero a su arco de menor peso
 struct Nodo *Siguiente;            //  Puntero al siguiente nodo de la
};                                    //  grafica (en la lista de nodos)
struct Graficas {
 unsh Num_Nodos;
 unsh Num_Arcos;
 double Peso;                       //  Peso total de la grafica
```

```
truct Nodo *Nodo_ini;              //  Puntero al primer nodo
Grafica;
h PAREIMM;
nclude "primgraf.h"
nclude "prim.h"
/*-------------------------------------------------
id main(int argc, char *argv[])
    Esta funcion organiza la informacion de entrada y de salida.
- Se leen los datos de un archivo reordenando la informacion mediante la
  funcion Agrega_Arco.
- Aplica el algoritmo mediante la funcion Prim.
- Da formato de salida a la solucion.
as variables principales son:
  Peso: Contiene la informacion del peso del arco que se va a agregar
 la grafica.
  Ni,Nf: Son los numeros de los nodos extremos del arco que va a agre-
garse a la grafica.
  Peso_Sol: Es el peso del arbol de minima expansion.
as funciones que se utilizan:
  Mem_Agot: Indica que no hubo memoria disponible.
  Agrega_Arco: Agrega un arco a la grafica.
  Prim: Aplica el algoritmo.                                        */

char  C;
unsh Temp_Estuvo, Tipo_Prob;
unsh   Ni, Nf;                     //  Nodos leidos del teclado
double   Peso;                     //  Peso leido del teclado
double   Peso_Sol;
struct Nodo *Dir_Nodo;             //  Se usa en la impresion
struct Arco *Dir_Arco;
FILE *Arch;
char Prim(double *);
unsh Agrega_Arco(unsh, unsh, double);
void Chk_Nomb_PRB(char *);
void Chk_Nomb_SOL(char *);
void Quitar_Copia(struct Nodo *, unsh);

Grafica.Num_Nodos  =  0;
Grafica.Num_Arcos  =  0;
Grafica.Peso       =  0;
Grafica.Nodo_ini   =  NULL;
if ( access("pareimm.tmp", 0) ) {     //  Si no fue llamado por el manager
 PAREIMM = 0;
 if ( argc == 1 ) {
  cputs("\nUso: prim problema.prb\r\n\n");
  cputs("La extension .PRB es obligada para el nombre del archivo\r\n");
  cputs("que contiene al problema. Tal archivo debe tener una\r\n");
  cputs("primera linea que solo contenga el numero 0, y una ultima\r\n");
  cputs("linea que contenga 3 ceros (0 0 0).\r\n");
  exit( 1 );
```

```
 }
 cputs("\nPAREIMM  Version 1.0,  1992-1993\r\nDEPARTAMENTO DE MATEMATICAS");
 cputs("\r\nUNIVERSIDAD DE SONORA\r\n");
} else
 PAREIMM = 1;
Chk_Nomb_PRB( argv[1] );
if ( access(argv[1], 0) ) {
 if ( !PAREIMM )  cprintf("ERROR: Archivo %s no encontrado\r\n", argv[1]);
 exit( IO_ERR );
}
if ( !( Arch = fopen(argv[1], "rt") ) ) {
 if ( !PAREIMM )  cprintf("ERROR: Archivo %s no abierto\r\n", argv[1]);
 exit( IO_ERR );
}
fscanf(Arch, "%d", &Tipo_Prob);
if ( Tipo_Prob != PRIM ) {
 fclose( Arch );
 if ( !PAREIMM )  cprintf("ERROR: Archivo %s no adecuado\r\n", argv[1]);
 exit( NO_PRIM );
}
do {
 fscanf(Arch, "%d %d %lf", &Ni, &Nf, &Peso);
 if ( Ni == 0  &&  Nf == 0 ) continue;
 Temp_Estuvo = Agrega_Arco( Ni, Nf, Peso );
 if ( Temp_Estuvo == 0 ) {
   fclose( Arch );
   if ( !PAREIMM ) cprintf("ERROR: Memoria insuficiente para
      continuar\r\n", argv[1]);
   exit( NO_MEM );
 }
 if ( Grafica.Num_Nodos > 20 ) {
   fclose( Arch );
   if ( !PAREIMM ) cprintf("ERROR: Archivo %s demasiado grande\r\n",
      argv[1]);
   exit( TOO_BIG );
 }
} while( Ni != 0  ||  Nf != 0 );
fclose( Arch );
if( !Grafica.Nodo_ini ) {
 if ( !PAREIMM )  cprintf("ERROR: Archivo %s vacio\r\n", argv[1]);
 exit( NO_PROBLEM );
}

//      A P L I C A N D O   E L   A L G O R I T M O

 C  =  Prim ( &Peso_Sol ) ;

if ( C == NO ) {
 if ( !PAREIMM )  cprintf("ERROR: Grafica disconexa\r\n", argv[1]);
 exit( DISCONECTED );
}
```

```
SOL( argv[1] );
rch = fopen(argv[1], "wt") ) ) {
REIMM )  cprintf("ERROR: Archivo %s no abierto\r\n", argv[1]);
_ERR );

ntf(Arch, "0\n") == EOF ) {
Arch );
argv[1] );
REIMM )  cputs("ERROR: Problemas al escribir la solucion\r\n");
_ERR );

= Grafica.Nodo_ini;
r_Nodo ) {
= Dir_Nodo->Arcos;
ir_Arco ) {
r_Arco->En_Sol == SI ) {
printf(Arch, "%u %u %lf\n", Dir_Nodo->Numero,
Arco->Extremo->Numero,
Arco->Peso ) == EOF ) {
ae( Arch );
nk( argv[1] );
(REIMM) cputs("ERROR: Problemas al escribir la solucion\r\n");
( IO_ERR );

Copia( Dir_Arco->Extremo, Dir_Nodo->Numero);

= Dir_Arco->Siguiente;

= Dir_Nodo->Siguiente;

tf(Arch, "0 0 0\n%u %u %f", Grafica.Num_Nodos,
um_Nodos - 1, Peso_Sol) == EOF ) {
rch );
rgv[1] );
M)cputs("ERROR:Problemas al escribir la solucion\r\n");
_ERR );

ch );
M)cprintf("\nSolucion grabada en: %s\r\n", argv[1]);
);

----------
omb_PRB(char *Nomb)


har(char *, char);

Char(Nomb, '.') ? Pos_Char(Nomb, '.')-1 : strlen(Nomb));

= '.';
1] = 'P';
```

```
 Nomb[Pos+2] = 'R';
 Nomb[Pos+3] = 'B';
 Nomb[Pos+4] = NULL;
}
void Chk_Nomb_SOL(char *Nomb)
{
 unsh Pos;
 unsh Pos_Char(char *, char);

 Pos=(Pos_Char(Nomb, '.') ? Pos_Char(Nomb, '.')- 1 : strlen(Nomb));
 Nomb[Pos] = '.';
 Nomb[Pos+1] = 'S';
 Nomb[Pos+2] = 'O';
 Nomb[Pos+3] = 'L';
 Nomb[Pos+4] = NULL;
}
//   ------------------------
unsh Pos_Char(char *s, char c)
{
 unsh i=1;

 while ( *s ) {
  if ( *s == c )
    return ( i );
  s++;
  i++;
 }
 return ( 0 );
}
//   ----------------------------------------
void Quitar_Copia(struct Nodo *Nodo_Ini,unsh Nf)
{
 struct Arco *Arco;

 Arco = Nodo_Ini->Arcos;
 while (Arco) {
  if ( Arco->Extremo->Numero == Nf ) {
   Arco->En_Sol = NO;
   return;
  }
  Arco = Arco->Siguiente;
 }
 return;
}
```

```
/* ================================================================
     VERANO 93                                          PAREIMM

     NOMBRE DEL ARCHIVO: PRIM.H
Este archivo aplica el algoritmo PRIM.

     FUNCIONES QUE INCLUYE:
    Prim: Aplica el algoritmo.
    Enc_Arc_Sol: Encuentra el arco que va a entrar a la solucion.
================================================================ */

// -------------------------
char Prim( double *Peso_Sol )

/*    Esta funcion aplica el Prim.
   Trabaja de la siguiente manera:
 1.- Se marca el primer nodo.
 2.- Se entra a un ciclo que se mantiene mientras que el numero
    de nodos seleccionados (NNS) sea diferente del numero total
    de nodos de la grafica.
 3.- Se recurre a una funcion que encuentra el arco de menor
    peso entre los que no han sido marcados (Enc_Arc_Sol), salen
    de algun nodo que ya se ecuentre en la solucion y cuyo nodo
    extremo no este en la solucion.
 4.- Si no hay un arco con esas caracteristicas, entonces la
    grafica es disconexa y se regresa a la funcion main().
 5.- Se marcan el arco encontrado, su nodo extremo y el arco
    par (el que se encuentra en el nodo extremo).
 6.- Se actualizan el peso de la solucion y el numero de nodos
    marcados.
 Las variables principales son:
  Num_Nod_Sol: Cuenta el numero de nodos en la solucion.
  Dir_Nodo_Rev: Direccion de nodos en la solucion.
  Dir_Ext: Direccion del nodo extremo del arco en la solucion.
  Dir_Arc_Sol: Direccion del arco que va a entrar a la solucion.
  Dir_Arc_Par: Direccion de la copia del arco que entra a la solucion.
 La funcion que se utiliza es:
Enc_Arc_Sol: Encuentra el arco que entrara en la solucion.    */
{
 char   C;
 unsigned short Num_Nod_Sol = 1;
 struct Nodo *Dir_Nodo_Rev, *Dir_Ext;
 struct Arco *Dir_Arc_Sol, *Dir_Arc_Par;
 char Enc_Arc_Sol(struct Nodo **,struct Arco **);

 Dir_Nodo_Rev = Grafica.Nodo_ini;
 *Peso_Sol = 0;
 Dir_Nodo_Rev->Revisado = SI;
 while( Num_Nod_Sol != Grafica.Num_Nodos ) {
  if(C == NO)           // Grafica disconexa
```

```
    return(NO);
  Dir_Arc_Sol->En_Sol = SI ;    // Se marca el arco
  Dir_Arc_Sol->Revisado = SI ;
  Dir_Ext = Dir_Arc_Sol->Extremo;  // Se marca el
  Dir_Ext->Revisado = SI;          //nodo extremo
  Dir_Arc_Par = Dir_Ext->Arcos;
  while(Dir_Arc_Par->Revisado==SI)//Salta arcos marcados
    Dir_Arc_Par = Dir_Arc_Par->Siguiente;
  while(Dir_Arc_Par->Extremo!=Dir_Nodo_Rev) //Busca arco par
    Dir_Arc_Par = Dir_Arc_Par->Siguiente;
  Dir_Arc_Par->En_Sol = SI;     //  Marca arco par
  Dir_Arc_Par->Revisado = SI;
  *Peso_Sol += Dir_Arc_Sol->Peso;
  Num_Nod_Sol++;
 }
 return( SI );
}
// ------------------------------------------------------------
char Enc_Arc_Sol( struct Nodo **Dir_Nodo_a_Marcar,
struct Arco **Dir_Arco_a_Marcar )
/*    Esta funcion encuentra el arco que va a entrar a la solucion.
 a.- Encuentra el arco de minimo peso del conjunto de arcos no marcados
  (arco->Revisado = NO) que salen de los nodos que estan en la solucion.
 b.- Es una busqueda tipica en la que se va guardando el menor hasta
  el momento. Como las listas de arcos estan ordenadas por el peso,
  en cada nodo hay que saltarse aquellos arcos que ya esten marcados
  o cuyo nodo extremo este en la solucion.
 c.- De encontrarlo, devuelve  s , y en caso contrario  n .
    Las variables principales son:
    Dir_Nodo_Rev: Direccion del nodo que se esta revisando.
    Dir_Arc_SMarca: Direccion para busqueda de arcos que no estan en la
      solucion.                                                  */
{
 struct Nodo *Dir_Nodo_Rev;
 struct Arco *Dir_Arc_SMarca;

 *Dir_Nodo_a_Marcar  =  NULL;
 *Dir_Arco_a_Marcar  =  NULL;
 Dir_Nodo_Rev = Grafica.Nodo_ini;
 while( Dir_Nodo_Rev ) {
  if( Dir_Nodo_Rev->Revisado == NO ) {  //  Salta nodos no
     Dir_Nodo_Rev = Dir_Nodo_Rev->Siguiente;    // marcados
     continue;
  }
  Dir_Arc_SMarca = Dir_Nodo_Rev->Arcos; //  Busca arco sin marca
  while((Dir_Arc_SMarca->Revisado==SI ||
   Dir_Arc_SMarca->Extremo->Revisado==SI) && Dir_Arc_SMarca)
     Dir_Arc_SMarca = Dir_Arc_SMarca->Siguiente;
  if( !Dir_Arc_SMarca ) {                       //  Ese nodo no tiene
    Dir_Nodo_Rev = Dir_Nodo_Rev->Siguiente;     // arcos disponibles
    continue;
```

```
 }
 if(!*Dir_Arco_a_Marcar || (Dir_Arc_SMarca->Peso <
      (*Dir_Arco_a_Marcar)->Peso)){
   *Dir_Nodo_a_Marcar = Dir_Nodo_Rev;  //  Se actualiza el
   *Dir_Arco_a_Marcar = Dir_Arc_SMarca;        // candidato
 }
 Dir_Nodo_Rev = Dir_Nodo_Rev->Siguiente;       //  Avanza un nodo
}
return( SI );
}
```

A.3 Algoritmo de Kruskal

```
/*============================================================
      VERANO 93                                        PAREIMM

      NOMBRE DEL ARCHIVO: KRUSKAL.C
  Este archivo aplica el algoritmo KRUSKAL.

      FUNCIONES QUE INCLUYE:
  Main: Organiza la informacion de entrada y salida.
 ============================================================*/

#include <io.h>
#include <stdio.h>
#include <process.h>
#include <stdlib.h>
#include <string.h>
#include <alloc.h>
#define       IO_ERR          0
#define       NO_PROBLEM      1
#define       DISCONECTED     2
#define       CICLO           3
#define       NO_MEM          4
#define       TOO_BIG         5
#define       NO_TIPO         6
#define         EXITO           100
#define         GRAFICA         0
#define         KRUSKAL         1
#define  SI         'S'
#define  NO         'N'
#define  ARBOL      'a'
#define  CCICLO     'c'
typedef unsigned short unsh;
struct Arco {
 char Revisado;       // SI si fue revisado, NO si no.
 char En_Sol;         // SI si esta en la solucion, NO si no.
 float Peso;          // Peso del arco.
 struct Nodo *Dir_Nodo_Ini, *Dir_Nodo_Final;//  Direccion
    //a nodos extremos.
};
struct Nodo {
 unsh Numero;                  //  Numero de nodo.
 char Revisado;           // SI si se ha revisado, NO  si no.
 unsh Grado, Grado_Temp;  // Grado y grado temporal del nodo.
 struct Nodo *Dir_Siguiente;// Dir. al siguiente nodo de la
};                        // grafica (en la lista de nodos).
struct Graficas {
 unsh Num_Nodos;          // Numero de nodos de la grafica.
 unsh Num_Arcos;          // Numero de arcos de la grafica.
```

```
float Peso;              // Peso total de la grafica.
struct Nodo *Dir_Grafica;// Dir. del primer nodo de la grafica.
struct Arco *Dir_Raiz;   // Puntero a la raiz del monton.
} Grafica;
struct Arco *Dir_Arco_Rev;
unsh Num_Arcos_Sol=0;  //  Numero de arcos que van en solucion.
unsh PAREIMM;
#include "krusmont.h"  // Archivo donde se crea monton de arcos.
#include "kruskal.h"   // Archivo donde esta el algoritmo KRUSKAL.
//   ----------------------------
void main(int argc, char *argv[])
/*    Esta funcion organiza la informacion de entrada y salida.
 a.- Captura los arcos y nodos de la grafica.
 b.- Crea un monton con los arcos de captura ordenandolos por
  peso de menor a mayor, quedando disponible el de menor peso.
 c.- Aplica el algoritmo mediante la funcion Kruskal.
 d.- Despliega la solucion obtenida.
  Las variables principales que utiliza son:
 Nodo_Ini: Numero del nodo inicial en la captura de la grafica.
 Nodo_Final:  Numero del nodo final en la captura de la grafica.
 Peso: Contiene el peso del arco en la captura de la grafica.
 Peso_Sol: Peso del arbol de minima expansion.
   Las funciones que se utilizan son:
 Mem_Agot: Indica que no hubo memoria disponible.
 Crea_Monton: Crea un monton ascendente de arcos por peso.
 Kruskal: Aplica el algoritmo Kruskal.                          */
{
 char  C, Nomb_Prob[16];
 unsh   Tipo_Prob;
 unsh   Ni, Nf;
 char   Temp_Estuvo;
 double   Peso;
 double   Peso_Sol;
 struct Arco *Dir_Arco;
 FILE *Arch;
 char Crea_Monton( unsh, unsh, double );
 char Kruskal( double * );
 char Kruskal(double *);
 void Chk_Nomb_PRB(char *);
 void Chk_Nomb_SOL(char *);
 Grafica.Num_Nodos    =  0;   //  Valores iniciales de grafica.
 Grafica.Peso         =  0;
 Grafica.Dir_Grafica  =  NULL;
 Grafica.Dir_Raiz     =  NULL;

 if(access("pareimm.tmp",0)){ // Si no fue llamado por el manager
  PAREIMM = 0;
  if ( argc < 3 ) {
   puts("\nUso: kruskal problema.prb num_arcos\n");
   puts("La extension .PRB es obligada para el nombre del archivo");
   puts("que contiene al problema. Tal archivo debe tener una");
```

```
  puts("primera linea que solo contenga el numero 0, y una ultima");
  puts("linea que contenga 3 ceros (0 0 0).");
  exit( 1 );
 }
 puts("\nPAREIMM Version 1.0, 1992-1993\r\nDEPARTAMENTO DE
 MATEMATICAS");
 puts("\r\nUNIVERSIDAD DE SONORA");
} else
 PAREIMM = 1;
strcpy(Nomb_Prob, argv[1]);
Grafica.Num_Arcos = atoi(argv[2]);
Chk_Nomb_PRB( Nomb_Prob );
if ( access(Nomb_Prob, 0) ) {
 if ( PAREIMM ) exit( IO_ERR );
 else {
   printf("ERROR: Archivo %s no encontrado\n", Nomb_Prob);
   exit( 1 );
 }
}
if ( !( Arch = fopen(Nomb_Prob, "rt") ) ) {
 if ( PAREIMM ) exit( IO_ERR );
 else {
   printf("ERROR: Archivo %s no abierto\n", Nomb_Prob);
   exit( 1 );
 }
}
fscanf(Arch, "%d", &Tipo_Prob);
if ( Tipo_Prob != GRAFICA ) {
 fclose( Arch );
 if ( PAREIMM ) exit( NO_TIPO );
 else {
   printf("ERROR: Archivo %s no adecuado\n", Nomb_Prob);
   exit( 1 );
 }
}
do {
 fscanf(Arch, "%d %d %lf", &Ni, &Nf, &Peso);
 if ( Ni == 0  &&  Nf == 0 ) continue;
 Temp_Estuvo = Crea_Monton( Ni, Nf, Peso );
 if ( Temp_Estuvo == NO ) {
  fclose( Arch );
  if ( PAREIMM ) exit( NO_MEM );
  else {
   printf("ERROR: Memoria insuficiente para continuar\n",Nomb_Prob);
   exit( 1 );
  }
 }
 if ( Grafica.Num_Nodos > 20 ) {
   fclose( Arch );
   if ( PAREIMM ) exit( TOO_BIG );
   else {
```

```
      printf("ERROR: Archivo %s demasiado grande\n", Nomb_Prob);
      exit( 1 );
    }
  }
 } while( Ni != 0  ||  Nf != 0 );
 fclose( Arch );
 if( !Grafica.Dir_Grafica ) {
  if ( PAREIMM ) exit( NO_PROBLEM );
  else {
    printf("ERROR: Archivo %s vacio\n", Nomb_Prob);
    exit( 1 );
  }
 }
//   A P L I C A N D O   E L   A L G O R I T M O

C = Kruskal( &Peso_Sol );

 if ( C == NO ) {
  if ( PAREIMM ) exit( DISCONECTED );
  else {
   printf("ERROR: Grafica disconexa\n", Nomb_Prob);
   exit( 1 );
  }
 }
 Chk_Nomb_SOL( Nomb_Prob );
 if ( !( Arch = fopen(Nomb_Prob, "wt") ) ) {
  if ( PAREIMM ) exit( IO_ERR );
  else {
   printf("ERROR: Archivo %s no abierto\n", Nomb_Prob);
   exit( 1 );
  }
 }
 if ( fprintf(Arch, "1\n") == EOF ) {
  fclose( Arch );
  unlink( Nomb_Prob );
  if ( PAREIMM )  exit( IO_ERR );
  else {
   puts("ERROR: Problemas al escribir la solucion");
   exit( 1 );
  }
 }
 Dir_Arco = Grafica.Dir_Raiz + Grafica.Num_Arcos - 1 ;
 while ( Dir_Arco != Grafica.Dir_Raiz ) {
  if(Dir_Arco->En_Sol == SI ) {
   if(fprintf(Arch,"%u %u %lf\n",Dir_Arco->Dir_Nodo_Ini->Numero,
      Dir_Arco->Dir_Nodo_Final->Numero,Dir_Arco->Peso)==EOF){
     fclose(Arch);
     unlink(Nomb_Prob);
     if ( PAREIMM ) exit( IO_ERR );
     else {
      puts("ERROR: Problemas al escribir la solucion\n");
```

```
      exit( 1 );
     }
    }
   }
   Dir_Arco--;
  }
  if ( Dir_Arco->En_Sol == SI ) {
   if(fprintf(Arch, "%u %u %lf\n", Dir_Arco->Dir_Nodo_Ini->Numero,
Dir_Arco->Dir_Nodo_Final->Numero,Dir_Arco->Peso)==EOF){
    fclose( Arch );
    unlink( Nomb_Prob );
    if ( PAREIMM ) exit( IO_ERR );
    else {
     puts("ERROR: Problemas al escribir la solucion\n");
     exit( 1 );
    }
   }
  }
 if ( fprintf(Arch, "0 0 0\n%u %u %f", Grafica.Num_Nodos,
Grafica.Num_Nodos - 1, Peso_Sol) == EOF ) {
  fclose( Arch );
  unlink( Nomb_Prob );
  if ( PAREIMM ) exit( IO_ERR );
  else {
    puts("ERROR: Problemas al escribir la solucion\n");
    exit( 1 );
  }
 }
 fclose( Arch );
 if ( !PAREIMM ) {
  puts("\nSolucion grabada en: ");
  puts(Nomb_Prob);
  putchar('\n');
 }
 exit( EXITO );
}
//   ----------------------
void Chk_Nomb_PRB(char *Nomb)
{
  unsh Pos;
  unsh Pos_Char(char *, char);

  Pos=(Pos_Char(Nomb, '.') ? Pos_Char(Nomb, '.')-1 : strlen(Nomb));

  Nomb[Pos] = '.';
  Nomb[Pos+1] = 'P';
  Nomb[Pos+2] = 'R';
  Nomb[Pos+3] = 'B';
  Nomb[Pos+4] = NULL;
}
//   ----------------------
```

```
void Chk_Nomb_SOL(char *Nomb)
{
 unsh Pos;
 unsh Pos_Char(char *, char);

 Pos=(Pos_Char(Nomb, '.') ? Pos_Char(Nomb, '.')-1 : strlen(Nomb));

 Nomb[Pos] = '.';
 Nomb[Pos+1] = 'S';
 Nomb[Pos+2] = 'O';
 Nomb[Pos+3] = 'L';
 Nomb[Pos+4] = NULL;
}
//   ------------------------
unsh Pos_Char(char *s, char c)
{
 unsh i=1;

 while ( *s ) {
   if ( *s == c )
     return ( i );
   s++;
   i++;
 }
 return ( 0 );
}
```

```
/* ========================================================
    VERANO 93                                        PAREIMM

    NOMBRE DEL ARCHIVO: KRUSKAL.H

    Este archivo aplica el algoritmo KRUSKAL
      FUNCIONES QUE INCLUYE:

    Kruskal: Aplica el algoritmo Kruskal.
    Escoge_Arco: Escoge el arco de minimo peso del monton.
    Busca_Ciclo: Revisa si el arco escogido forma un ciclo.
 ======================================================== */

char Kruskal( double *Peso_Sol )
 /*    Esta funcion aplica  el algoritmo KRUSKAL a una grafica.
    Opera de la siguiente manera:
 a.- En un ciclo de 1 hasta el numero de arcos menos 1, va marcando los
  arcos de menor a mayor, en orden del peso, que son proporcionados de
  esta forma por la funcion Escoge_Arco. Tal funcion escoge solo arcos
  que no formen un ciclo al entrar a la solucion.
 b.- La direccion de cada arco que se escoge se guarda en Dir_Dispon,
  y se utiliza para marcar al mismo arco como dentro de la solucion
  (Arco->En_Sol = SI), asi como para aumentar el grado de los nodos
  extremos (Arco->Dir_Nodo_Ini o Dir_Nodo_Final->Grado++) y marcarlos
  como ya dentro de la solucion (Arco-Dir_Nodo_Ini o
  Dir_Nodo_Final->Revisado = SI).
 c.- Si la direccion del arco que va a entrar a la solucion es el valor
  NULL, no hay arcos disponibles, y el problema no tiene solucion.
  En este caso la funcion regresa  NO. Si el ciclo se termina, entonces
  si se pudo encontrar solucion, y se regresa SI.
    Las variables principales que utiliza son:
  Dir_Dispon: Direccion del arco disponible obtenido en Escoge_Arco
  para revisar si entra en la solucion.
    La funcion que utiliza es:
     Escoge_Arco: Escoge el arco de minimo peso del monton.  */
{
 unsh i;
 struct Arco *Dir_Dispon;  //  Direccion del arco disponible.
 struct Arco *Escoge_Arco(void); // Funcion que escoge arco del monton.

 for ( i = 1 ; i < Grafica.Num_Nodos ; i++ ) {
 Dir_Dispon = Escoge_Arco();       //  Escogiendo el arco.
 if ( !Dir_Dispon ) return( NO );
 Dir_Dispon->En_Sol = SI;          //  Etiquetando
 Dir_Dispon->Dir_Nodo_Ini->Revisado = SI;
 Dir_Dispon->Dir_Nodo_Final->Revisado = SI;
 Dir_Dispon->Dir_Nodo_Ini->Grado++;
 Dir_Dispon->Dir_Nodo_Final->Grado++;
 *Peso_Sol = *Peso_Sol + Dir_Dispon->Peso;
 Num_Arcos_Sol++;
```

```
 }
 return( SI );
}
//-----------------------------
struct Arco *Escoge_Arco( void )
/* Esta funcion es la encargada de escoger el arco de menor peso que
   puede entrar a la solucion.
   Opera de la siguiente manera:
a.- Los arcos se encuentran guardados en un monton en el que el
 elemento menor se encuentra arriba. Se intercambian el menor y
 el mayor, se marca el menor  y se reordena el monton sin tomar
 en cuenta los arcos marcados.
b.- Si el arco que se escogio tiene ambos nodos extremos marcados
 se revisa si no forma ciclo. Si no lo forma, se regresa ese arco,
 en caso contrario se toma el nuevo arco de menor peso.
   Las variables principales que utiliza:
  Cont: Cuenta el numero de arcos que faltan por marcar.
  Lugar: Asigna el lugar en el monton de un arco.
  Dir_Padre: Direccion del nodo padre.
  Dir_Hijo: Direccion del nodo hijo.
 La funcion que utiliza es:
  Busca_Ciclo: Revisa si se forma ciclo con el arco escogido.        */
{
 char C;
 static unsh Cont = Grafica.Num_Arcos; // Numero de arcos no marcados.
 unsh Lugar;                           // Lugar en el monton.
 struct Arco *Dir_Arco, *Dir_Padre, *Dir_Hijo; // Dirs. del padre e hijo.
 static struct Arco *Dir_Temp_Arco = new Arco; // Puntero temporal para
       //  asignacion de memoria.
 char Busca_Ciclo( void );

 while ( Cont > 0 ) {
  Dir_Arco = Grafica.Dir_Raiz + Cont-- - 1 ;
  *Dir_Temp_Arco  =  *Dir_Arco;
  *Dir_Arco    = *( Grafica.Dir_Raiz );
  *( Grafica.Dir_Raiz )  =  *Dir_Temp_Arco;
  Dir_Arco->Revisado  =  SI;
  if ( Cont >= 3 )                     // Reordenando monton
   if((Grafica.Dir_Raiz+1)->Peso>(Grafica.Dir_Raiz+2)->Peso) Lugar=3;
   else Lugar = 2;
  else if(Cont==2 && Grafica.Dir_Raiz->Peso>(Grafica.Dir_Raiz+1)->Peso)
Lugar=2;
       else Lugar=1;
  Dir_Padre=Grafica.Dir_Raiz;
  Dir_Hijo=Grafica.Dir_Raiz + Lugar-1;
  while(Dir_Padre->Peso>Dir_Hijo->Peso){ // Inician intercambios
   *Dir_Temp_Arco=*Dir_Hijo;       // hasta encontrar la
   *Dir_Hijo   =  *Dir_Padre;      // posicion del arco
   *Dir_Padre  =  *Dir_Temp_Arco; // en el monton
   Lugar *= 2;
   if ( Lugar + 1 <= Cont ) {
```

```
    Dir_Padre = Dir_Hijo;
    if((Grafica.Dir_Raiz+Lugar-1)->Peso>(Grafica.Dir_Raiz+Lugar)->Peso)
       Lugar++;
    Dir_Hijo = Grafica.Dir_Raiz + Lugar - 1 ;
   }else if(Lugar==Cont && (Grafica.Dir_Raiz+Lugar-1)->Peso<Dir_Hijo->Peso){
    Dir_Padre = Dir_Hijo;
    Dir_Hijo = Grafica.Dir_Raiz + Lugar - 1 ;
   }
  }
  if(Dir_Arco->Dir_Nodo_Ini->Revisado == SI &&
Dir_Arco->Dir_Nodo_Final->Revisado==SI){
   (Dir_Arco->Dir_Nodo_Ini->Grado)++;
   (Dir_Arco->Dir_Nodo_Final->Grado)++;
   Dir_Arco_Rev = Dir_Arco;     //  Arco en proceso de revision.
   C = Busca_Ciclo( );
   if ( C == ARBOL ) return( Dir_Arco );
  } else return( Dir_Arco );
 }
 return( NULL );
}
// ----------------
char Busca_Ciclo( )
/* Esta funcion revisa si forma ciclo el arco disponible del monton.
   Opera de la siguiente manera:
a.- Se hace una copia de los Grados en los nodos en los
   Grados_Temporales.
b.- Se busca un nodo pendiente (Grado_Temp = 1).
c.- Se baja el Grado_Temp a cero.
d.- Se busca el arco del nodo pendiente.
e.- Se borra el arco.
f.- Se le baja el Grado_Temp en una unidad al otro nodo extremo.
g.- Se continua hasta ya no encontrar nodos pendientes.
h.- Si el Grado_Temp = 0 de todos los nodos en solucion, no hay
   ciclo. Contrariamente si algun nodo tiene Grado_Temp >0, hay
   ciclo.
i.- Regresa 'a' si hay solucion (hay arbol).
j.- Regresa 'c' si hay un ciclo.
  Las variables principales que utiliza son:
     N: Cuenta los nodos borrados.                          */
{
 unsh i=0;
 struct Nodo *Nodo;     // al ir revisando ciclos
 struct Arco *Dir_Arco;

//  Haciendo copia del Grado en Grado_Temp.
 for(Nodo=Grafica.Dir_Grafica;Nodo;Nodo=Nodo->Dir_Siguiente)
    Nodo->Grado_Temp = Nodo->Grado;
 Nodo = Grafica.Dir_Grafica;
 while ( i < (Grafica.Num_Nodos)*(Num_Arcos_Sol+1) ){
  while(Nodo&&Nodo->Revisado==SI&&Nodo->Grado_Temp==1){
   Nodo->Grado_Temp = 0;
```

```
   i++;
   Dir_Arco  =  Grafica.Dir_Raiz + Grafica.Num_Arcos - 1 ;
   while ( Dir_Arco ) {
    if(Dir_Arco->Dir_Nodo_Ini==Nodo&&
Dir_Arco->Dir_Nodo_Final->Grado_Temp>0){
     Dir_Arco->Dir_Nodo_Final->Grado_Temp--;//Otro extremo del arco.
     Nodo = Dir_Arco->Dir_Nodo_Final;
     break;
    } else if ( Dir_Arco->Dir_Nodo_Final == Nodo
&& Dir_Arco->Dir_Nodo_Ini->Grado_Temp>0){
     Dir_Arco->Dir_Nodo_Ini->Grado_Temp--;// Otro extremo del arco.
     Nodo = Dir_Arco->Dir_Nodo_Ini;
     break;
    }
    Dir_Arco--;
   }
  }
  if(Nodo&&Nodo->Revisado==SI&&Nodo->Grado_Temp!=1){
    i++;
    Nodo = Nodo->Dir_Siguiente;
  }else if(Nodo&&!Nodo->Revisado==SI&&Nodo->Grado_Temp==1){
    i++;
    Nodo = Nodo->Dir_Siguiente;
  }else if(Nodo&&Nodo->Revisado==NO&&Nodo->Grado_Temp!=1){
    i++;
    Nodo = Nodo->Dir_Siguiente;
  }
  if (!Nodo ) Nodo = Grafica.Dir_Grafica;
  }
 Nodo=Grafica.Dir_Grafica;// Verificando si hay ciclo
 while ( Nodo ) {
  if ( Nodo->Grado_Temp > 0 ) {
    Dir_Arco_Rev->Dir_Nodo_Ini->Grado--;
    Dir_Arco_Rev->Dir_Nodo_Final->Grado--;
    return ( CCICLO );
   }
  Nodo = Nodo->Dir_Siguiente;
 }
 return (ARBOL);
}
```

```
/* ================================================
   VERANO 93                              PAREIMM

     NOMBRE DEL ARCHIVO: CREAMONT.H

   Este archivo crea un monton para el algoritmo KRUSKAL.

FUNCIONES QUE INCLUYE:

   Crea_Monton: Crea un monton de arcos por el peso.
  Agrega_Nodo: Agrega un nodo en una lista.
  Mem_Agot: Indica si hubo memoria requerida.

================================================================ */

char Crea_Monton( unsh Ni, unsh Nf, double Peso )
/* Esta funcion crea un monton con los arcos de la grafica por el
  peso.
   Opera de la siguiente manera:
a.- Al momento en que se captura la grafica el monton se va creando.
b.- Se hace una busqueda de  nodo inicial y final para ver si ya se
   encuentran en el monton.
c.- Si no fueron encontrados se agregan mediente la funcion
   Agrega_Nodo
d.- Se busca ahora el arco en el monton, si este ya existe
   regresa 'y'.
e.- De no haber estado, el arco se agrega al monton.
f.- Se busca el padre que le correspondio al arco.
g.- Se Realiza el intercambio de padre-hijo si es necesario para
  poner en el lugar indicado al arco.
h.- Se calcula el nuevo padre del arco ya en posicion correcta.
 Las variables principales que utiliza son:
  Cont: Indica el numero de arcos que hay en el monton.
  Lugar: Posicion de un arco en el monton.
  Dir_Nodo_Ini: Direccion del nodo inicial de un arco.
  Dir_Nodo_Final: Direccion del nodo final de un arco.
  Dir_Arco: Direccion de un arco.
  Dir_Arco_Padre: Direccion del arco padre de un arco en el monton.
La funcion que utiliza es:
  Agrega_Nodo: Esta funcion agrega un nodo ala lista.          */
{
 char   Se_Agrego_Nodo;  //  Indica si se pudo agregar un nodo.
 static unsh  Cont = 0 ;
 unsh Lugar;
 struct Nodo *Dir_Nodo_Ini, *Dir_Nodo_Final;
 struct Arco *Dir_Arco, *Dir_Arco_Padre;
 static struct Arco *Dir_Temp_Arco = new Arco;// Puntero temporal de
      //arco.
 char Agrega_Nodo( int, struct Nodo ** );

 if ( !Dir_Temp_Arco ) return (NO);//exit (0);
```

```
Dir_Nodo_Ini = Grafica.Dir_Grafica;//Se busca el nodo inicial.
while ( Dir_Nodo_Ini ){
  if(Dir_Nodo_Ini->Numero != Ni && Dir_Nodo_Ini){
    Dir_Nodo_Ini=Dir_Nodo_Ini->Dir_Siguiente;
    continue;
  }
  break;
}
Dir_Nodo_Final = Grafica.Dir_Grafica;//Se busca el nodo final.
while ( Dir_Nodo_Final ) {
  if ( Dir_Nodo_Final->Numero != Nf ) {
     Dir_Nodo_Final = Dir_Nodo_Final->Dir_Siguiente;
     continue;
  }
  break;
}
if ( !Dir_Nodo_Ini ) {          //  No existe el nodo
   Se_Agrego_Nodo = Agrega_Nodo(Ni, &Dir_Nodo_Ini);// inicial.
   if (  Se_Agrego_Nodo == NO ) return(NO);
   Grafica.Num_Nodos++;
}
if ( !Dir_Nodo_Final ) {        //  No existe el nodo
   Se_Agrego_Nodo = Agrega_Nodo(Nf, &Dir_Nodo_Final);// final.
   if (  Se_Agrego_Nodo == NO ) return( NO );
   Grafica.Num_Nodos++;
}
Cont++;                         //  Un arco mas.
if ( !Grafica.Dir_Raiz ) {      //  Primer arco.
 Grafica.Dir_Raiz=(struct Arco *)malloc(Grafica.Num_Arcos*sizeof(struct Arco));
 if ( !Grafica.Dir_Raiz ) return( NO );
 Grafica.Dir_Raiz->Revisado        =  NO;
 Grafica.Dir_Raiz->En_Sol          =  NO;
 Grafica.Dir_Raiz->Peso            =  Peso;
 Grafica.Dir_Raiz->Dir_Nodo_Ini    =  Dir_Nodo_Ini;
 Grafica.Dir_Raiz->Dir_Nodo_Final  =  Dir_Nodo_Final;
 Grafica.Peso                   =  Peso;
 return( SI );
}
Dir_Arco=Grafica.Dir_Raiz+Cont-1;   // Lugar libre.
Dir_Arco->Revisado        =  NO;
Dir_Arco->En_Sol          =  NO;
Dir_Arco->Peso            =  Peso;
Dir_Arco->Dir_Nodo_Ini    =  Dir_Nodo_Ini;
Dir_Arco->Dir_Nodo_Final  =  Dir_Nodo_Final;
Grafica.Peso += Peso;
      //Buscando el padre.
if ( Cont%2 ) Lugar  =  int( .5*(Cont - 1) );
else Lugar  =  int( .5*Cont );
Dir_Arco_Padre  =  Grafica.Dir_Raiz + Lugar - 1 ;
while(Dir_Arco_Padre->Peso>Dir_Arco->Peso){//Inician intercambios
 *Dir_Temp_Arco       =  *Dir_Arco; // hasta encontrar la
```

```
  *Dir_Arco            =  *Dir_Arco_Padre;// posicion del arco.
  *Dir_Arco_Padre      =  *Dir_Temp_Arco;
   if(Dir_Arco_Padre != Grafica.Dir_Raiz){
   Dir_Arco = Dir_Arco_Padre;
//  Calculando nuevo padre.
   if(!Lugar%2)  Lugar  =  int( .5*(Lugar - 1) );
   else  Lugar  =  int( .5*(Lugar) );
   Dir_Arco_Padre  =  Grafica.Dir_Raiz + Lugar - 1 ;
  }
 }
 return( SI );
}
//-----------------------------------------
char Agrega_Nodo(int N,struct Nodo **Dir_Arco)

/*    Esta funcion agrega un Nodo a una lista.
   Opera de la siguiente manera:
a.- Se reserva memoria para el nodo.
b.- Se llenan los parametros.
c.- Se busca el final de la lista y se enlaza el nodo.
d.- Se regresa un SI si tuvo exito la funcion.
e.- Regresa un NO si no hubo memoria disponible.
   Las variables principales son:
   Dir_Nodo_R: Direccion del nodo por agregar.          */
{
 struct Nodo  *Dir_Nodo_Q, *Dir_Nodo_R, *Dir_Nodo_Anterior;
 Dir_Nodo_R = new Nodo;   //  Se dispone memoria para el nuevo nodo.
 Dir_Nodo_R->Numero         =  N; //  Se agrega y crea el
 Dir_Nodo_R->Revisado       =  NO; // enlace con la lista
 Dir_Nodo_R->Grado            =  0;
 Dir_Nodo_R->Dir_Siguiente  =  NULL;

 *Dir_Arco = Dir_Nodo_R;
 if ( !Grafica.Dir_Grafica )    //  Primer nodo
  Grafica.Dir_Grafica = Dir_Nodo_R;
 else {
  Dir_Nodo_Q = Grafica.Dir_Grafica;
  while ( Dir_Nodo_Q ) {        // Buscando el final
   Dir_Nodo_Anterior = Dir_Nodo_Q;
   Dir_Nodo_Q = Dir_Nodo_Q->Dir_Siguiente;
  }
  Dir_Nodo_Anterior->Dir_Siguiente = Dir_Nodo_R;
 }
 return( SI );
}
//-----------------
void Mem_Agot( void )
/* Esta funcion manda un mensaje de que no se encontro memoria
   disponible.*/
{
   printf( "\nERROR: Memoria agotada.\n" );
```

```
    exit( 1 );
}
```

Apéndice B

Algoritmos para digráficas

B.1 Formato en los archivos

Deberemos tener el siguiente formato de entrada:

1. Una primera linea que contenga un 1. (Indica el tipo de editor para estos algoritmos.)
2. Una lista de arcos con su costo (un espacio deberá separar el nodo inicial, nodo final y el costo), uno por renglón.
3. Para terminar de meter los arcos teclear 0 0 0 en el siguiente renglón.
4. Poner el número de nodos, el número de arcos y el peso total de la digráfica en el siguiente renglón.

El algoritmo de Dijkstra para costos no negativos así como el General están implementados como un solo programa pues el primero está contenido en el segundo.

El nombre del archivo debe tener la extensión PRB.

El problema no deberá exceder de 20 nodos.

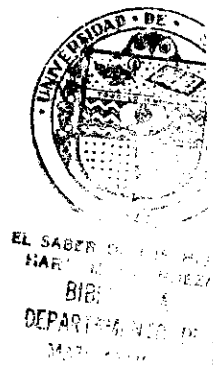
El archivo de salida contiene

1. Una primera linea que contiene un 2 si el problema se solucionó con el algoritmo de Dijkstra General y un 3 si se solucionó con un Floyd.
2. La lista de arcos que están en la solución junto con el costo al nodo final desde la raíz. Esta solución será la ruta entre dos nodos dados si se corre desde fuera del PAREIMM y un archivo con la información de la arborescencia completa en caso contrario.
3. Un renglón que indique que terminó de listarse la solución (0 0 0).

4. El número de nodos, el número de arcos y la raíz de la arborescencia.

Por ejemplo para el problema 2 del capítulo 2, el archivo de entrada es:

1		
1	2	-1
2	3	-2
3	5	-5
3	4	-4
4	6	-6
4	7	-4
4	8	-7
5	9	0
5	12	-2
5	13	-6
5	10	0
6	9	0
6	12	-2
7	10	0
6	10	0
7	11	0
8	11	-6
8	12	-6
9	12	-4
10	12	-2
11	12	-6
0	0	0
13	21	-55



La solución a este problema es la ruta más corta entre los nodos 1 y 12. Esta ruta es :

2		
1	2	-1
2	3	-3
3	4	-7
4	8	-14
8	11	-20
11	12	-26
0	0	0
7	6	1

Para llamarlo fuera del PAREIMM se necesita como argumentos los nodos entre los cuales se quiere encontrar la ruta mínima. Supongamos que el archivo del ejemplo anterior se llama

PROB.PRB y que deseamos conocer la ruta mínima entre 1 y 12. Para resolverlo con un DIJKSTRA es necesario poner lo siguiente

DIJKSTRA PROB.PRB 1 12

la solución quedará grabada en PROB.SOL.

B.2 Algoritmo de Dijkstra General

```
/* ========================================================================

      INVIERNO 92-93                                          PAREIMM

      NOMBRE DEL ARCHIVO: DIJKSTRA.C

  Este archivo aplica el algoritmo DIJKSTRA_GENERAL.

      FUNCIONES QUE INCLUYE:

  Main: Organiza la informacion de entrada y salida.
   ======================================================================== */

#include <io.h>
#include <math.h>
#include <ctype.h>
#include <conio.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <process.h>

#define         IO_ERR          0
#define         NO_PROBLEM      1
#define         NO_SOL          2
#define         CICLO           3
#define         PROB_MONTON     4
#define         NO_MEM          5
#define         TOO_BIG         6
#define         NO_DIJKSTRA     7

#define         EXITO           100


typedef unsigned short unsh;


struct Nodo{
   unsigned short int Num_Nodo;     //  Numero del nodo.
   unsigned short int Ant_Ruta_Min;//  Numero del nodo anterior en la ruta
   // minima.
   float Dist_Min;                  //  Distancia minima desde la raiz hasta
   // el nodo.
   short int Etiqueta;              // -1  si el algoritmo no lo ha tocado.
   //  0  etiqueta temporal.
   //  1  etiqueta definitiva.
   struct Arco  *Arco;              //  Direccion al primer elemento de la
   // lista de arcos del nodo.
```

```
   struct Nodo *Siguiente;          //  Direccion al siguiente elemento de la
   // lista de Nodos.
};

struct Nodo_Solucion{
   unsh Num_Nodo;                   //  Numero del nodo en la solucion.
   unsh Ant_Ruta_Min;               //  Numero del nodo anterior en la ruta
   // minima.
   float Dist_Min;                  //  Distancia minima desde la raiz hasta
   // el nodo solucion.
   short int Profundidad;           //  Profundidad del nodo en la grafica
   // solucion.
   unsh S_Preorden;                 //  Sucesor en preorden del nodo solucion.
   struct Arco  *Arco;              //  Direccion al primer elemento de la
   // lista de arcos del nodo solucion.
   struct Nodo_Solucion *Siguiente;//  Direccion al siguiente elemento de la
   // lista de Nodos en la Solucion.
};

struct Graficas {
//   unsh Num_Nodos;
   unsh Raiz;                          //Primer nodo de la ruta.
   unsh Destino;                       //Ultimo nodo de la ruta.
   struct Nodo *Dir_Grafica;           //Puntero al primer nodo.
   struct Nodo_Solucion *Dir_Graf_Sol;//Puntero al primer nodo de la solucion.
} Grafica;

struct Arco{
   unsh Nodo_Final;         //  Numero del nodo final del arco.
   float Peso;              //  Peso del arco.
   struct Arco  *Siguiente;//  Direccion al siguiente elemento de la
   // lista de arcos.
};

struct Plato_Preorden{
   unsh Num_Nodo;                     // Numero del nodo.
   struct Plato_Preorden *Siguiente; // Direccion al siguiente elemento.
};

struct PLATO_IMP{
   unsh Num_Nodo;
   unsh Ant_Ruta_Min;
   float Dist_Min;
   struct PLATO_IMP *Siguiente;
};

struct Plato_Imp{
   unsh Num_Nodo;
   struct Plato_Imp *Siguiente;
};
```

```
struct Plato{
   unsh Nodo_Inicial;       // Numero del nodo inicial del arco.
   unsh Nodo_Final;         // Numero del nodo final del arco.
   float Peso;              // Peso del arco.
   struct Plato  *Siguiente;// Direccion al siguiente elemento.
};

struct Monton{
   unsh Num_Nodo;     // Numero del nodo.
   float Dist_Min;    //  Distancia minima desde la raiz hasta
      // el nodo.
};

struct Hojas{
   struct Nodo_Solucion *Dir_Ultimo; //  Numero del ultimo nodo en preorden
     // de un arbol.
   struct Nodo_Solucion *Dir_A_Preorden;//  Antecesor en preorden del primer
// elemento de un arbol.
};

struct Ciclo{
   struct Nodo_Solucion *Dir_Nodo;     // Numero de nodo que esta en un ciclo.
   struct Nodo_Solucion *Dir_Antecesor;// Antecesor en el ciclo de Nodo.
};

unsh PAREIMM;

#include "dijkstra.h"


//   ------------------------------
void main ( int argc, char *argv[] )

/*    Esta funcion organiza la informacion de entrada y salida.

   a.- Captura los arcos de la digrafica y agrega:

      . los nodos extremos en caso de que no esten, y
      . los arcos sucesor y antecesor.

   b.- Aplica el algoritmo mediante la funcion Ruta_Minima_Dijkstra para
      una primera solucion.

   c.- Crea la grafica solucion.

   d.- Realiza un recorrido preorden sobre la grafica solucion mediante
      la funcion Recorrido_Preorden.

   e.- Optimiza la solucion mediante la funcion Cambia_Ruta.

   f.- Despliega la solucion obtenida o un ciclo si se forma.
```

```
Las variables principales que utiliza son:

  Grafica.Dir_Grafica: Direccion del primer elemento de la lista de nodos.
  Grafica.Dir_Graf_Sol: Direccion del primer elemento de la lista de nodos
       de la digrafica solucion.
  Dir_Monton: Direccion del arreglo para el monton.
  Nodo_Inicial: Numero del nodo inicial en la captura de un arco en
       la digrafica.
  Nodo_Final: Numero del nodo final en la captura de un arco en
     la digrafica.
  Peso: Peso del arco capturado.
  Raiz: Numero del nodo a partir del cual se desea la ruta minima.
  Pila: Direccion al primer elemento de la pila de arcos candidatos
       a entrar en la solucion.
  Ciclo: Estructura con la informacion de dos nodos consecutivos
de un ciclo.

Las funciones que se utilizan son:


   Nodo_Agrega: Agrega un elemento a la lista de nodos.
   Agregar_Arco: Agrega un elemento a la lista de arcos de un nodo.
   Cambia_Ruta: Optimiza la solucion mediante la funcion Cambia_Ruta.
   Ruta_Minima_Dijkstra: Aplica el algoritmo mediante para
una primera solucion.
   Recorrido_Preorden: Realiza el recorrido preorden sobre la
      grafica solucion.                                  */

{
 char Nomb_Prob[16];
 int Num_Nodos,Num_Arcos,Tipo_Prob,Nodo_Inicial,Nodo_Final,Flag,Indica=0;
 int Hay_Campo,Ind,Resultado,Contador=0;
 float Peso;
 struct Nodo *TEMP,*Dir_NI,*Dir_NF,*d,*Ant,*Dir_Destino,*Antecesor;
 struct Nodo_Solucion *Dir_Ant,*temp;
//   struct Plato  *Pila;
 struct PLATO_IMP *pa_tras,*PILA_IMP;
 struct Plato_Imp *para_tras, *Pila_Imp;
 struct Ciclo Ciclo;

 FILE *Arch;

 struct Nodo *Nodo_Agrega( int );
 struct Nodo *Nodo_Busca ( int,int * );
 int Agregar_Arco ( struct Arco  **,int,float );
//   int Recorrido_Preorden ( struct Plato **,int );
 int Ruta_Minima_Dijkstra ( int,int );
 int Grafica_Solucion ( int );
 int Mete_Plato_Imp ( struct PLATO_IMP  **,unsh,unsh,float);
 int METE_PLATO_IMP ( struct Plato_Imp  **,unsh );
```

```
struct Ciclo Cambia_Ruta ( );
void Chk_Nomb_PRB(char *);
void Chk_Nomb_SOL(char *);


//   Grafica.Num_Nodos    =  0;
 Grafica.Dir_Grafica  =  NULL;
 Grafica.Dir_Graf_Sol =  NULL;

//   Pila=NULL;
 Pila_Imp=NULL;
 PILA_IMP=NULL;

 if ( access("pareimm.tmp", 0) ) {    //  Si no fue llamado por el manager

  PAREIMM = 0;
  if ( argc == 1 ) {
     cputs("\nUso: dijkstra problema.prb\r\n\n");
     cputs("La extension .PRB es obligada para el nombre del archivo\r\n");
     cputs("que contiene al problema. Tal archivo debe tener una\r\n");
     cputs("primera linea que solo contenga el numero 1, y una ultima\r\n");
     cputs("linea que contenga 3 ceros (0 0 0).\r\n");
     exit( 1 );
  }
  cputs("\nPAREIMM  Version 1.0,  1992-1993\r\nDEPARTAMENTO DE MATEMATICAS");
  cputs("\r\nUNIVERSIDAD DE SONORA\r\n");
  strcpy(Nomb_Prob, argv[1]);

 } else PAREIMM = 1;

 strcpy(Nomb_Prob, argv[1]);
 Grafica.Raiz=(unsigned) atoi(argv[2]);
 Grafica.Destino=  (unsigned) atoi(argv[3]);

 Chk_Nomb_PRB( Nomb_Prob );
 if ( access(Nomb_Prob, 0) ) {
  if ( PAREIMM )
     exit( IO_ERR );
  else {
     cprintf("\nERROR: Archivo %s no encontrado\r\n", Nomb_Prob);
     exit( 1 );
  }
 }
 if ( !( Arch = fopen(Nomb_Prob, "rt") ) ) {
  if ( PAREIMM )
     exit( IO_ERR );
  else {
     cprintf("\nERROR: Archivo %s no abierto\r\n", Nomb_Prob);
     exit( 1 );
  }
 }
```

```
fscanf(Arch, "%d", &Tipo_Prob);
if ( Tipo_Prob != 2 ) {
 fclose( Arch );
 if ( PAREIMM )
    exit( NO_DIJKSTRA );
 else {
    cprintf("\nERROR: Archivo %s no adecuado\r\n", Nomb_Prob);
    exit( 1 );
 }
}
do {
 fscanf(Arch, "%u %u %f", &Nodo_Inicial, &Nodo_Final, &Peso);
 if ( Nodo_Inicial == 0  &&  Nodo_Final == 0 ) continue;

 if ( Peso < 0 ) Indica=1;

 Dir_NI=Nodo_Agrega ( Nodo_Inicial );
 if( !Dir_NI ) {
  fclose( Arch );
  if ( PAREIMM )
   exit( NO_MEM );
  else {
   cprintf("\nERROR: Memoria insuficiente para continuar\r\n", Nomb_Prob);
   exit( 1 );
  }
 }
 Dir_NF=Nodo_Agrega ( Nodo_Final );
 if( !Dir_NF ) {
  fclose( Arch );
  if ( PAREIMM )
   exit( NO_MEM );
  else {
   cprintf("\nERROR: Memoria insuficiente para continuar\r\n", Nomb_Prob);
   exit( 1 );
  }
 }
 Ind=Agregar_Arco ( & ( Dir_NI->Arco ),Nodo_Final,Peso );
 if( !Ind ) {
  fclose( Arch );
  if ( PAREIMM )
   exit( NO_MEM );
  else {
   cprintf("\nERROR: Memoria insuficiente para continuar\r\n", Nomb_Prob);
   exit( 1 );
  }
 }
} while( Nodo_Inicial != 0  ||  Nodo_Final != 0 );

fscanf(Arch, "%d %d %lf\n",&Num_Nodos, &Num_Arcos, &Peso);
fclose( Arch );
```

```
if ( Num_Nodos > 20 ) {
 fclose( Arch );
 if ( PAREIMM )
    exit( TOO_BIG );
 else {
    cprintf("\nERROR: Archivo %s demasiado grande\r\n", Nomb_Prob);
    exit( 1 );
 }
}


fclose( Arch );

d = Nodo_Busca(Grafica.Destino,&Resultado);
if( d == NULL) Dir_Destino=Grafica.Dir_Grafica;
else Dir_Destino=d->Siguiente;

if( !Grafica.Dir_Grafica ) {
 if ( PAREIMM )
    exit( NO_PROBLEM );
 else {
    cprintf("\nERROR: Archivo %s vacio\r\n", Nomb_Prob);
    exit( 1 );
 }
}

Flag=Ruta_Minima_Dijkstra ( Num_Nodos, Grafica.Raiz );

if ( Flag==2 ) {
 fclose( Arch );
 if ( PAREIMM )
    exit( NO_MEM );
 else {
    cprintf("\nERROR: Memoria insuficiente para continuar\r\n", Nomb_Prob);
    exit( 1 );
 }
}

if ( Flag==3 ) {
 fclose( Arch );
 if ( PAREIMM )
  exit( PROB_MONTON );
 else {
  cprintf("\nERROR: Problemas en la ejecucion del algoritmo\r\n", Nomb_Prob);
  exit( 1 );
 }
}

if ( Dir_Destino->Etiqueta==-1 ) Flag=0;

if ( Flag==0 ){
```

```
 fclose( Arch );
 if ( PAREIMM )
    exit( NO_SOL );
 else {
    cprintf("\nERROR: No existe solucion al problema\r\n", Nomb_Prob);
    exit( 1 );
 }
}

Chk_Nomb_SOL( Nomb_Prob );
if ( !( Arch = fopen(Nomb_Prob, "wt") ) ) {
 if ( PAREIMM ) exit( IO_ERR );
 else {
    cprintf("\nERROR: Archivo %s no abierto\r\n", Nomb_Prob);
    exit( 1 );
 }
}
if ( fprintf(Arch, "2\n") == EOF ) {
 fclose( Arch );
 unlink( Nomb_Prob );
 if ( PAREIMM )
  exit( IO_ERR );
 else {
  cputs("\nERROR: Problemas al escribir la solucion\r\n");
  exit( 1 );
 }
}

if ( Indica==0 ) {

 TEMP=Dir_Destino;

 while ( TEMP->Num_Nodo!=TEMP->Ant_Ruta_Min ) {
  if ( ( TEMP->Num_Nodo != Grafica.Raiz )  &&  (  TEMP ->Etiqueta==1 ) ) {

     Ind=Mete_Plato_Imp(&PILA_IMP,TEMP->Num_Nodo,TEMP->Ant_Ruta_Min,TEMP->Dist_Min);
     if(Ind==0){
fclose( Arch );
unlink( Nomb_Prob );
if ( PAREIMM )
   exit( IO_ERR );
else {
   cputs("\nERROR: Problemas al escribir la solucion\r\n");
   exit( 1 );
}
     }

  }
  Ant=Nodo_Busca(TEMP->Ant_Ruta_Min,&Resultado);
  if(Ant==NULL) Antecesor=Grafica.Dir_Grafica;
  else Antecesor=Ant->Siguiente;
```

```
  TEMP=Antecesor;
 }

 pa_tras=PILA_IMP;

 while(pa_tras){
  if(fprintf(Arch,"%u %u %f\n",pa_tras->Ant_Ruta_Min,pa_tras->Num_Nodo,
  pa_tras->Dist_Min)==EOF){
   fclose( Arch );
   unlink( Nomb_Prob );
   if ( PAREIMM ) exit( IO_ERR );
   else {
    cputs("ERROR: Problemas al escribir la solucion\r\n");
    exit( 1 );
   }
  }
  Contador++;
  pa_tras=pa_tras->Siguiente;
 }

 if(fprintf(Arch, "0 0 0\n%u %u %u", Contador+1,Contador,Grafica.Raiz)==EOF){
  fclose( Arch );
  unlink( Nomb_Prob );
  if ( PAREIMM )
     exit( IO_ERR );
  else {
     cputs("\nERROR: Problemas al escribir la solucion\r\n");
     exit( 1 );
  }
 }
 fclose( Arch );
 if ( !PAREIMM ) {
    cputs("\nSolucion grabada en: ");
    cputs(Nomb_Prob);
    putch('\n');
 }
 exit ( EXITO );

}

Hay_Campo=Grafica_Solucion ( Grafica.Raiz );
if ( Hay_Campo==0 ) {
 fclose( Arch );
 if ( PAREIMM )
    exit( NO_MEM );
 else {
    cprintf("\nERROR: Memoria insuficiente para continuar\r\n", Nomb_Prob);
    exit( 1 );
 }
}
```

```
Ciclo=Cambia_Ruta ();
if ( Ciclo.Dir_Nodo ) {
 if ( Ciclo.Dir_Antecesor==NULL ) {
  fclose( Arch );
  if ( PAREIMM )
     exit( NO_MEM );
  else {
     cprintf("\nERROR: Memoria insuficiente para continuar\r\n", Nomb_Prob);
     exit( 1 );
  }
 }

 Dir_Ant=Ciclo.Dir_Antecesor;

 while ( Dir_Ant->Num_Nodo!= ( Ciclo.Dir_Nodo )->Num_Nodo ) {

  Ind=METE_PLATO_IMP(&Pila_Imp,Dir_Ant->Num_Nodo );
  if(Ind==0){
   fclose( Arch );
   if ( PAREIMM )
    exit( IO_ERR );
   else {
    cputs("\nERROR: Problemas al escribir el ciclo (memoria)\r\n");
    exit( 1 );
   }
  }

  temp=Dir_Ant;

  Dir_Ant=Grafica.Dir_Graf_Sol;
  while ( Dir_Ant && Dir_Ant->Num_Nodo!=temp->Ant_Ruta_Min )
     Dir_Ant=Dir_Ant->Siguiente;

 }

 Ind=METE_PLATO_IMP(&Pila_Imp,Dir_Ant->Num_Nodo );
 if(Ind==0){
  fclose( Arch );
  if ( PAREIMM )
   exit( IO_ERR );
  else {
   cputs("\nERROR: Problemas al escribir el ciclo (memoria)\r\n");
   exit( 1 );
  }
 }

 para_tras=Pila_Imp;
 while(para_tras){
  if ( fprintf ( Arch,"%d\n",para_tras->Num_Nodo) ==EOF ) {
   fclose( Arch );
```

```
    unlink( Nomb_Prob );
    if ( PAREIMM ) exit( IO_ERR );
    else {
     cputs("\nERROR: Problemas al escribir la solucion\r\n");
     exit( 1 );
    }
   }
   Contador++;
   para_tras=para_tras->Siguiente;
  }

  if(fprintf(Arch, "0\n%u %u %u",Contador,Grafica.Raiz,Grafica.Destino)==EOF){
   fclose( Arch );
   unlink( Nomb_Prob );
   if ( PAREIMM ) exit( IO_ERR );
   else {
    cputs("\nERROR: Problemas al escribir la solucion\r\n");
    exit( 1 );
   }
  }
  fclose( Arch );
  if ( !PAREIMM ) {
   cputs("\nSe encontro un ciclo, grabado en: ");
   cputs(Nomb_Prob);
   putch('\n');
  }

  exit ( CICLO );

}

TEMP=Dir_Destino;

while ( TEMP->Num_Nodo!=TEMP->Ant_Ruta_Min ) {
 if ( ( TEMP->Num_Nodo != Grafica.Raiz )  &&  (  TEMP ->Etiqueta==1 ) ){

  Ind=Mete_Plato_Imp(&PILA_IMP,TEMP->Num_Nodo,TEMP->Ant_Ruta_Min,TEMP->Dist_Min);
  if(Ind==0){
   cputs("\nERROR: Problemas al escribir la solucion\r\n");
   exit( 1 );
  }

  Ant=Nodo_Busca(TEMP->Ant_Ruta_Min,&Resultado);
  if(Ant==NULL) Antecesor=Grafica.Dir_Grafica;
  else Antecesor=Ant->Siguiente;

  TEMP=Antecesor;
 }
}

pa_tras=PILA_IMP;
```

```
 while(pa_tras){

  if(fprintf(Arch,"%u %u %f\n",pa_tras->Ant_Ruta_Min,pa_tras->Num_Nodo,
    pa_tras->Dist_Min)==EOF){
   fclose( Arch );
   unlink( Nomb_Prob );
   if ( PAREIMM ) exit( IO_ERR );
   else {
    cputs("\nERROR: Problemas al escribir la solucion\r\n");
    exit( 1 );
   }
  }
  Contador++;
  pa_tras=pa_tras->Siguiente;

 }

 if(fprintf(Arch, "0 0 0\n%u %u %u",Contador+1,Contador,Grafica.Raiz)==EOF){
  fclose( Arch );
  unlink( Nomb_Prob );
  if ( PAREIMM ) exit( IO_ERR );
  else {
   cputs("\nERROR: Problemas al escribir la solucion\r\n");
   exit( 1 );
  }
 }
 fclose( Arch );
 if ( !PAREIMM ) {
  cputs("\nSolucion grabada en: ");
  cputs(Nomb_Prob);
  putch('\n');
 }
 exit( EXITO );

}


//   ----------------------
void Chk_Nomb_PRB(char *Nomb)
{
  unsh Pos;
  unsh Pos_Char(char *, char);

  Pos = ( Pos_Char(Nomb, '.') ? Pos_Char(Nomb, '.') - 1 : strlen(Nomb) );

  Nomb[Pos] = '.';
  Nomb[Pos+1] = 'P';
  Nomb[Pos+2] = 'R';
  Nomb[Pos+3] = 'B';
  Nomb[Pos+4] = NULL;
```

```
}


//   ------------------------
void Chk_Nomb_SOL(char *Nomb)
{
  unsh Pos;
  unsh Pos_Char(char *, char);

  Pos = ( Pos_Char(Nomb, '.') ? Pos_Char(Nomb, '.') - 1 : strlen(Nomb) );

  Nomb[Pos] = '.';
  Nomb[Pos+1] = 'S';
  Nomb[Pos+2] = 'O';
  Nomb[Pos+3] = 'L';
  Nomb[Pos+4] = NULL;
}


//   ------------------------
unsh Pos_Char(char *s, char c)
{
  unsh i=1;

  while ( *s ) {
    if ( *s == c )
      return ( i );
    s++;
    i++;
  }
  return ( 0 );
}
```

```
/* ========================================================================

INVIERNO 92-93                                          PAREIMM

NOMBRE DEL ARCHIVO: GENERAL.H

   Este archivo agrega y borra elementos de listas
       utilizadas por las funciones del algoritmo incluidas
tambien en este archivo.

FUNCIONES QUE INCLUYE:

 Monton: Crea un arreglo para un monton.
 Agregar_Arco: Agrega un elemento a la lista de arcos
      de un nodo.
 Grafica_Solucion: Crea una lista de Nodos y asigna la
   profundidad correspondiente a cada nodo.
 Arbol: Encuentra el arbol que cuelga de un nodo dado.
 Ciclo: Dice si existe un camino de j a i.
 Mete_Plato: Mete un elemento a la pila  ( para solucion ).
 Agr_Plato_Preorden: Mete un elemento a la pila
     ( para Preorden ).
 Saca_Plato: Saca un Plato de la pila  ( para solucion ).
 Quita_Plato_Preorden: Saca un Plato de la pila
       ( para Preorden ).
 Quita_Monton: Quita el menor elemento del monton.
 Sucesores: Marca de manera temporal los sucesores de un Nodo
    y los mete en un monton.
 Actualiza_Monton: Actualiza un elemento del monton y lo
   ordena.
 Cambia_Ruta: Optimiza la solucion.
 Ruta_Minima_Dijkstra: Aplica el algoritmo para una primera
      solucion.
 Recorrido_Preorden: Realiza el recorrido preorden sobre la
    grafica solucion.
 Nodo_Agrega: Agrega un elemento a la lista de nodos.
 Nodo_Busca: Busca un elemento en la lista de nodos.


   ======================================================================== */


//  ---------------------------
int Grafica_Solucion ( int Raiz )

/*    Esta funcion crea un arreglo de Nodos con la solucion del
     Dijkstra y asigna la profundidad correspondiente a cada nodo.

 Opera de la siguiente manera:
```

```
a.- Se crea y llena el arreglo de nodos en la solucion.
b.- Al nodo raiz se le asigna profundidad 0.
c.- A lo sucesores ( en la solucion del Dijkstra ) se les
 aumenta la profundidad en 1.

 Las variables principales que utiliza son:

    Grafica.Dir_Graf_Sol: Direccion del arreglo de nodos de la solucion.
    Raiz: Numero de nodo a partir del cual se desea calcular la
 ruta minima.
    Cont_Arcos: Cuenta el numero de arcos desde la raiz al nodo
       donde se busca la profundidad.                          */

{
 unsigned short Cont_Arcos=0,Antecesor;
 struct Nodo *TEMP;
 struct Nodo_Solucion *r,*temp,*Temp;


 TEMP=Grafica.Dir_Grafica;

 while ( TEMP ) {

  r=new Nodo_Solucion();
  if ( r==NULL ) return ( 0 );

  r->Num_Nodo=TEMP->Num_Nodo;
  r->Ant_Ruta_Min=TEMP->Ant_Ruta_Min;
  r->Dist_Min=TEMP->Dist_Min;
  r->S_Preorden=TEMP->Num_Nodo;
  r->Arco=NULL;
  r->Siguiente=NULL;

  if ( !Grafica.Dir_Graf_Sol ) Grafica.Dir_Graf_Sol=r;
  else {
   temp=Grafica.Dir_Graf_Sol;
   while ( temp->Siguiente ) temp=temp->Siguiente;
   temp->Siguiente=r;
  }

  TEMP=TEMP->Siguiente;

 }

 // Encontrando profundidad

 temp=Grafica.Dir_Graf_Sol;

 while ( temp ) {
  if ( temp->Num_Nodo==Raiz )  temp->Profundidad=0;
  else
```

```
   if ( temp->Ant_Ruta_Min==temp->Num_Nodo )  temp->Profundidad=-1;
   else{
    Temp=temp;
    while ( Temp->Ant_Ruta_Min != Raiz ) {
     Cont_Arcos=Cont_Arcos + 1;
     Antecesor=Temp->Ant_Ruta_Min;
     Temp=Grafica.Dir_Graf_Sol;
     while ( Temp  &&  Temp->Num_Nodo != Antecesor ) Temp= Temp->Siguiente;
    }
    temp->Profundidad=Cont_Arcos+1;
    Cont_Arcos=0;
   }
  temp=temp->Siguiente;
 }
 return ( 1 );
}

//      ------------------------------------------------
int Recorrido_Preorden ( struct Plato  **Pila,int Raiz )

/*      Esta funcion encuentra el sucesor en el sentido de
preorden de cada elemento de una arborescencia

   Opera de la siguiente manera:

a.- Se revisan los nodos marcados definitivos por Dijkstra, a los
   cuales se les revisan los sucesores:
      - si el nodo sucesor tiene como antecesor en la ruta minima al nodo
       inicial del arco revisado se agrega este a la solucion,
      - si no, revisar si la distancia minima del sucesor es mayor que la
       distancia minima del nodo inicial dada por Dijkstra mas el peso del
       arco revisado en cuyo caso entra a la pila de candidatos.
PREORDEN
b.- Entra en una pila preorden la raiz, se actualizan sus sucesores en la
   solucion y entran en la pila preorden.
c.- Se va al sucesor en preorden y se actualizan, a su vez, los sucesores,
   si no tiene sale uno de la pila preorden y se marca.
d.- Ir a (c) mientras el sucesor en preorden sea distinto de la raiz.

   Las variables principales que se utilizan son:

      Grafica.Dir_Grafica: Direccion del arreglo de nodos en la grafica.
      Grafica.Dir_Graf_Sol: Direccion del arreglo de nodos en la grafica
   Solucion.
      Pila_Preorden: Direccion al primer elemento de la pila de
    nodos  para preorden.

   Las funciones que se utilizan son:

      Agregar_Arco: Agrega un elemento a la lista de arcos de un nodo.
      Mete_Plato: Agrega un elemento a la pila de arcos candidatos
```

```
a entrar en la solucion.
     Agr_Plato_Preorden: Agrega un elemento a la pila para preorden.
     Quita_Plato_Preorden: Quita un elemento de la pila para preorden. */

{
unsigned short Nodo_Final,Ind,Sucesor;
int Resultado;
struct Arco  *temp;
struct Nodo  *ant,*apt,*TEMP;
struct Nodo_Solucion  *APT,*Apt;
struct Plato_Preorden  *Pila_Preorden;

int Agregar_Arco ( struct Arco  **,int,float );
int Mete_Plato ( struct Plato  **,int,int,float );
int Agr_Plato_Preorden ( struct Plato_Preorden  **,int );
int Quita_Plato_Preorden ( struct Plato_Preorden  ** );
struct Nodo *Nodo_Busca ( int ,int * );


Pila_Preorden=NULL;

TEMP=Grafica.Dir_Grafica;

while ( TEMP ) {
 if (  TEMP->Etiqueta==1 ) {
  temp= TEMP->Arco;
  while ( temp ) {
   ant = Nodo_Busca ( temp->Nodo_Final,&Resultado );
   if ( !ant ) apt=Grafica.Dir_Grafica;
   else apt=ant->Siguiente;

   if ( apt->Ant_Ruta_Min== TEMP->Num_Nodo ) {

    Apt=Grafica.Dir_Graf_Sol;

    while ( Apt && ( Apt->Num_Nodo != TEMP->Num_Nodo ) ) Apt=Apt->Siguiente;

    Ind=Agregar_Arco ( & ( Apt->Arco ) ,temp->Nodo_Final,temp->Peso );
    if( Ind==0 ) return ( 0 );

     if ( apt->Dist_Min >  TEMP->Dist_Min+temp->Peso ) {
     Ind=Mete_Plato ( Pila,TEMP->Num_Nodo,temp->Nodo_Final,temp->Peso );
     if( Ind==0 ) return ( 0 );
    }
   }

  temp=temp->Siguiente;
 }
}

TEMP=TEMP->Siguiente;
```

```
 }

 Ind=Agr_Plato_Preorden ( &Pila_Preorden,Raiz );
 if( Ind==0 ) return ( 0 );

 APT=Grafica.Dir_Graf_Sol;
 while ( APT && APT->Num_Nodo!=Raiz ) APT=APT->Siguiente;

 temp=APT->Arco;
 Sucesor=temp->Nodo_Final;
 APT->S_Preorden=Sucesor;

 temp=temp->Siguiente;
 while ( temp ) {
  Ind=Agr_Plato_Preorden ( &Pila_Preorden,temp->Nodo_Final );
  if( Ind==0 ) return ( 0 );
  temp=temp->Siguiente;
 }

 APT=Grafica.Dir_Graf_Sol;
 while ( APT && APT->Num_Nodo!=Sucesor ) APT=APT->Siguiente;

 while ( APT->Num_Nodo!=Raiz ) {
  temp=APT->Arco;
  if ( temp ) {
   Sucesor=temp->Nodo_Final;
   APT->S_Preorden=temp->Nodo_Final;
  }
  else{
   Nodo_Final=Quita_Plato_Preorden ( &Pila_Preorden );
   Sucesor=Nodo_Final;
   APT->S_Preorden=Sucesor;
   APT=Grafica.Dir_Graf_Sol;
   while ( APT && APT->Num_Nodo!=Nodo_Final ) APT=APT->Siguiente;
   continue;
  }
  temp=temp->Siguiente;
  while ( temp ) {
   Ind=Agr_Plato_Preorden ( &Pila_Preorden,temp->Nodo_Final );
   if( Ind==0 ) return ( 0 );
   temp=temp->Siguiente;
  }

  APT=Grafica.Dir_Graf_Sol;
  while ( APT && APT->Num_Nodo!=Sucesor ) APT=APT->Siguiente;
 }
 return ( 1 );
}

//     ------------------------------------------------------------------
struct Hojas Arbol(int i,int j,float Difer,struct Plato_Preorden **Elem_Arbol)
```

```
/*    Esta funcion encuentra el arbol que cuelga de un nodo dado

   Opera de la siguiente manera:

a.- Se saca un elemento de la pila de Arcos candidatos a entrar en
   la solucion.
b.- Se calcula la diferencia entre la distancia minima del arco
   actual y el arco candidato.
c.- Si es mayor la distancia dada por Dijkstra se revisa si el candidato
   forma ciclo al entrar, si es asi el problema no tiene solucion.
d.- Encuentra el arbol que cuelga del nodo final del candidato y
   actualiza la informacion de sus elementos.

   Las variables principales que utiliza son:

      Grafica.Dir_Graf_Sol: Direccion del arreglo de nodos de la digrafica
   solucion.
//       *Pila: Direccion del primer elemento de la pila de arcos
//       candidatos a entrar en la solucion.
      Hojas: Estructura que contiene el numero del ultimo nodo en
    Preorden de un arbol y el antecesor en preorden del primer
    elemento del arbol.
      Elem_Arbol: Direccion del primer elemento de la lista de nodos
 que constituyen el arbol calculado.
      Hay_Ciclo: Estructura con la informacion de dos nodos consecutivos
    de un ciclo.

   Las funciones que se utilizan son:

      Ciclo: Revisa si se forma un ciclo entre dos nodos.
      Arbol: Calcula el arbol que cuelga de un nodo dado.
      Quita_Plato_Preorden: Saca un elemento de la pila para
   preorden.
      Saca_Plato: Saca un elemento de la pila de arcos candidatos
 a entrar en la solucion.                                    */

{
 unsigned short Sucesor,Ind,Indicador;
 struct Nodo *TEMP;
 struct Nodo_Solucion  *temp,*Dir_j,*Dir_Ant,*Dir_Ultimo;
 struct Hojas Hojas;

 int Agr_Plato_Preorden ( struct Plato_Preorden  **,int );


 temp=Grafica.Dir_Graf_Sol;
 while ( temp && temp->Num_Nodo!=j ) temp=temp->Siguiente;
 Dir_j=temp;

 Indicador=temp->Profundidad;
```

```
 Dir_Ant=Grafica.Dir_Graf_Sol;
 while ( Dir_Ant && Dir_Ant->Num_Nodo!=i ) Dir_Ant=Dir_Ant->Siguiente;

 do{
  Ind=Agr_Plato_Preorden ( Elem_Arbol,temp->Num_Nodo );
   Hojas.Dir_Ultimo=NULL;
   Hojas.Dir_A_Preorden=NULL;
   return ( Hojas );
  }

  temp->Profundidad= Dir_Ant->Profundidad + 1;
  temp->Dist_Min=temp->Dist_Min - Difer;

  TEMP=Grafica.Dir_Grafica;
  while ( TEMP && TEMP->Num_Nodo!=temp->Num_Nodo ) TEMP=TEMP->Siguiente;
  TEMP->Dist_Min=TEMP->Dist_Min - Difer;

  Dir_Ultimo=temp;

  Dir_Ant=temp;

  Sucesor=temp->S_Preorden;

  temp=Grafica.Dir_Graf_Sol;
  while ( temp && temp->Num_Nodo!=Sucesor ) temp=temp->Siguiente;

 }while ( temp->Profundidad>Indicador );

 Hojas.Dir_Ultimo=Dir_Ultimo;

 temp=Grafica.Dir_Graf_Sol;
 while ( temp && temp->Num_Nodo!=Dir_j->Ant_Ruta_Min ) temp=temp->Siguiente;

 while ( temp->S_Preorden!=j ) {
  Sucesor=temp->S_Preorden;
  temp=Grafica.Dir_Graf_Sol;
  while ( temp && temp->Num_Nodo!=Sucesor ) temp=temp->Siguiente;
 }

 Hojas.Dir_A_Preorden=temp;

 return ( Hojas );
}

//  --------------------
int Ciclo ( int i,int j )

/*    Esta funcion dice si existe un camino de j a i

  Opera de la siguiente manera:
```

```
a.- Se revisa si j es la raiz en cuyo caso hay camino.
b.- Si la profundidad de j es mayor o igual que la de i no hay camino.
c.- Se calcula la diferencia de profundidades.
d.- Se busca la profundidad del antecesor de i con profundidad igual a
  la profundidad de i menos la diferencia de profundidades.
e.- Si este antecesor es j hay camino.

  La variable principal que utiliza es:

     Grafica.Dir_Graf_Sol: Direccion del arreglo de nodos de la digrafica
  solucion.                                                    */

{
 unsigned short resta,k,Antecesor;
 struct Nodo_Solucion *Dir_i,*Dir_j,*temp;

 Dir_j=Grafica.Dir_Graf_Sol;
 while ( Dir_j && Dir_j->Num_Nodo!=j ) Dir_j=Dir_j->Siguiente;

 Dir_i=Grafica.Dir_Graf_Sol;
 while ( Dir_i && Dir_i->Num_Nodo!=i ) Dir_i=Dir_i->Siguiente;


 if ( Dir_j->Profundidad==0 )  return ( 1 );

 if ( Dir_j->Profundidad >= Dir_i->Profundidad )  return ( 0 );
 else{
  resta= Dir_i->Profundidad - Dir_j->Profundidad;

  temp=Dir_i;

  for ( k=0;k<resta;k++ ) {
   Antecesor=temp->Ant_Ruta_Min;
   temp=Grafica.Dir_Graf_Sol;                  // desde i a la posicion de j
   while ( temp && temp->Num_Nodo!=Antecesor ) temp=temp->Siguiente;
  }
  if ( temp->Num_Nodo==j )  return ( 1 );
  else return ( 0 );
 }
}

//  ------------------------------------------------------
int Agregar_Arco ( struct Arco  **p,int Nodo_Final,float Peso )

/*    Esta funcion es para insertar un arco a la lista

  Opera de la siguiente manera:

a.- Se pide memoria para el nuevo arco.
b.- En caso de que haya se llena con la informacion pertinente.
```

```
c.- Se agrega al final de la lista.

  La variable principal que utiliza es:

     *p: Direccion del primer elemento de la lista de arcos de
un nodo.                                                    */

{
 struct Arco  *t,*r,*Anterior;

 r=new Arco;
 if ( r==NULL ) return (0);

 r->Nodo_Final=Nodo_Final;
 r->Peso=Peso;
 r->Siguiente=NULL;

 t= ( *p );
 if ( t==NULL ) {
    ( *p )=r;
 }else{
    while ( t ) {
       Anterior=t;
       t=t->Siguiente;
    }
    Anterior->Siguiente=r;
 }
 return (1);
}

//  ------------------------------------------------------------------------
int Mete_Plato(struct Plato **prin,int Nodo_Inicial,int Nodo_Final,float Peso)

/*    Esta funcion mete un elemento a la pila  ( para solucion )

  Opera de la siguiente manera:

a.- Se pide memoria para el nuevo elemento de la pila.
b.- En caso de que haya se llena con la informacion pertinente.

  La variable principal que utiliza es:

   *prin: Direccion del primer elemento de la pila.              */

{
 struct Plato  *temp,*r;

 temp=*prin;
 r=new Plato;
 if ( r==NULL )  return (0);
```

```
 r->Nodo_Inicial=Nodo_Inicial;
 r->Nodo_Final=Nodo_Final;
 r->Peso=Peso;
 r->Siguiente=temp;

 *prin=r;
 return (1);
}

// ------------------------------------------------------------------------
int Mete_Plato_Imp(struct PLATO_IMP **prin,unsh Num_Nodo,unsh Ant_Ruta_Min,
    float Dist_Min)

/*    Esta funcion mete un elemento a la pila  ( para solucion )

  Opera de la siguiente manera:

a.- Se pide memoria para el nuevo elemento de la pila.
b.- En caso de que haya se llena con la informacion pertinente.

  La variable principal que utiliza es:

     *prin: Direccion del primer elemento de la pila.                */

{
 struct PLATO_IMP  *temp,*r;

 temp=*prin;

 r=new PLATO_IMP;
 if(r==NULL)  return( 0 );

 r->Num_Nodo=Num_Nodo;
 r->Ant_Ruta_Min=Ant_Ruta_Min;
 r->Dist_Min=Dist_Min;
 r->Siguiente=temp;

 *prin=r;
 return (1);
}

// ------------------------------------------------------
int METE_PLATO_IMP ( struct Plato_Imp  **prin,unsh Num_Nodo )

/*    Esta funcion mete un elemento a la pila  ( para solucion )

  Opera de la siguiente manera:

a.- Se pide memoria para el nuevo elemento de la pila.
b.- En caso de que haya se llena con la informacion pertinente.
```

```
  La variable principal que utiliza es:

     *prin: Direccion del primer elemento de la pila.                */

{
 struct Plato_Imp  *temp,*r;

 temp=*prin;

 r=new Plato_Imp;
 if(r==NULL)  return( 0 );

 r->Num_Nodo=Num_Nodo;
 r->Siguiente=temp;

 *prin=r;
 return (1);
}

//  ------------------------------------------------------------
int Agr_Plato_Preorden ( struct Plato_Preorden  **prin,int Num_Nodo )

/*    Esta funcion mete un elemento a la pila  ( para Preorden )

  Opera de la siguiente manera:

a.- Se pide memoria para el nuevo elemento de la pila.
b.- En caso de que haya se llena con la informacion pertinente.

  La variable principal que utiliza es:

     *prin: Direccion del primer elemento de la pila                 */

{
 struct Plato_Preorden  *temp,*r;


 temp=*prin;
 r=new Plato_Preorden;
 if ( r==NULL ) return (0);

 r->Num_Nodo=Num_Nodo;
 r->Siguiente=temp;

 *prin=r;
 return (1);
}

//           ----------------------------------
struct Plato  *Saca_Plato ( struct Plato  **prin )
```

```
/*    Esta funcion saca un Plato de la pila  ( para solucion )

 La variable principal que utiliza es:

    *prin: Direccion del primer elemento de la pila.                */

{
 struct Plato  *temp;


 temp=*prin;
 *prin=temp->Siguiente;
 return ( temp );
}

//  ------------------------------------------------
int Quita_Plato_Preorden ( struct Plato_Preorden  **prin )

/*    Esta funcion saca un Plato de la pila  ( para Preorden )

 La variable principal que utiliza es:

    *prin: Direccion del primer elemento de la pila.                */

{
 unsigned short Num_Nodo;
 struct Plato_Preorden  *temp;


 temp=*prin;
 Num_Nodo=temp->Num_Nodo;
 *prin=temp->Siguiente;
 delete temp;
 return ( Num_Nodo );
}

//  ------------------------------------------------------------------------
int Monton(int n,struct Monton **Dir_Monton,int Num_Nodo,float Dist_Min,
int *Contador)

/*    Esta funcion crea un monton con nodos de la digrafica ordenandolos
 por la distancia minima desde la raiz hasta ellos.

  Opera de la siguiente manera:

a.- Se busca el nodo dado en el monton, si este ya existe regresa 1.
b.- De no haber estado, el nodo se agrega al monton.
c.- Se busca el padre que le correspondio al nodo.
d.- Se realiza el intercambio de padre-hijo si es necesario para poner
  en el lugar indicado al nodo.
e.- Se calcula el nuevo padre del nodo ya en posicion correcta.
```

```
  Las variables principales que utiliza son:

     *Contador: Indica el numero de nodos que hay en el monton.
     Lugar: Posicion de un nodo en el monton.
     Dir_Nodo_Mon: Direccion de un nodo en el monton.
     Dir_Nodo_Padre: Direccion del nodo padre de un nodo en
    el monton.                                          */

{
  unsigned short Lugar, i;
  struct Monton *apt,temp,*Dir_Nodo_Padre,*Dir_Nodo_Mon;


  if ( *Dir_Monton ){              //  Busqueda del nodo en el monton.
   for ( i=1, apt = *Dir_Monton ; i <= *Contador ; i++, apt++ )

      if ( apt->Num_Nodo == Num_Nodo ) return( 1 );

  } else {                                          //  Primer nodo.
   *Dir_Monton = ( struct Monton  * ) malloc ( n*sizeof ( struct Monton ) );
   if ( !( *Dir_Monton ) ) return( 0 );

   ( *Dir_Monton )->Num_Nodo=Num_Nodo;
   ( *Dir_Monton )->Dist_Min=Dist_Min;

   ( *Contador )++;
   return( 2 );
  }

  ( *Contador )++;                                   //  Un nodo mas.

  Dir_Nodo_Mon->Num_Nodo=Num_Nodo;
  Dir_Nodo_Mon->Dist_Min=Dist_Min;

 if(Dir_Nodo_Mon==*Dir_Monton)  return(2);

  if ( ( *Contador )%2 )                        //  Buscando el padre.
     Lugar=int( .5*( ( *Contador ) - 1) );
  else
     Lugar=int( .5* ( *Contador ) );

  Dir_Nodo_Padre= ( *Dir_Monton )  + Lugar -1;

  //  Inician intercambios hasta encontrar la posicion del Nodo.
  while ( Dir_Nodo_Padre->Dist_Min > Dir_Nodo_Mon->Dist_Min ) {
   temp=*Dir_Nodo_Mon;
   *Dir_Nodo_Mon=*Dir_Nodo_Padre;
   *Dir_Nodo_Padre=temp;

   if ( Dir_Nodo_Padre !=  ( *Dir_Monton )  ) {
```

```
    Dir_Nodo_Mon=Dir_Nodo_Padre;
    if ( Lugar%2 )                    //  Calculando nuevo padre.
     Lugar=int( .5*(Lugar - 1) );
    else
     Lugar=int( .5*(Lugar) );
    Dir_Nodo_Padre=( *Dir_Monton ) + Lugar - 1 ;
   }
  }
  return( 2 );
}
//   ------------------------------------------------------------------
void Quita_Monton ( struct Monton  **Dir_Monton,int  *Contador,int  *Dato )

/*    Esta funcion quita el menor elemento del monton.

   Opera de la siguiente manera:

a.- Los nodos se encuentran guardados en un monton en el que    el
   elemento menor se encuentra arriba.
b.- Se intercambian el menor y el mayor, se marca el menor  y se
   reordena el monton.

   Las variables principales que utiliza:

      *Contador: Cuenta el numero de nodos en el monton que faltan
por marcar.
      Lugar: Asigna el lugar en el monton de un nodo.
      Dir_Padre: Direccion del nodo padre.
      Dir_Hijo: Direccion del nodo hijo.                              */


{
 unsigned short int Lugar;                  // Lugar en el monton.
 struct Monton *Dir_Padre, *Dir_Hijo,temp;  // Direcciones del padre e hijo.


 if (  ( *Contador ) > 0 ) {

  *Dato=( *Dir_Monton )->Num_Nodo;

  temp=* ( ( *Dir_Monton ) + ( *Contador ) -1 );
  *( ( *Dir_Monton ) + ( *Contador ) -1)=* ( *Dir_Monton );
  *( *Dir_Monton )=temp;

  ( *Contador )--;

  if ( ( *Contador ) >= 3 ) {                    //  Reordenando monton
     if(((*Dir_Monton)+1)->Dist_Min > ((*Dir_Monton)+2)->Dist_Min)
Lugar=3;
     else  Lugar=2;
  } else {
```

```
    if((*Contador)==2 && (*Dir_Monton)->Dist_Min > ((*Dir_Monton)+1)->Dist_Min)
Lugar=2;
    else  Lugar=1;
  }

  Dir_Padre=*Dir_Monton;
  Dir_Hijo= ( *Dir_Monton )  + Lugar - 1 ;
   //Inician intercambios hasta encontrar la posicion del arco en el monton.
  while ( Dir_Padre->Dist_Min > Dir_Hijo->Dist_Min ) {
   temp=*Dir_Hijo;
   *Dir_Hijo=*Dir_Padre;
   *Dir_Padre=temp;

   Lugar *= 2;
   if ( Lugar + 1 <= ( *Contador ) ) {
    Dir_Padre=Dir_Hijo;
    if(((*Dir_Monton)+Lugar-1)->Dist_Min > ((*Dir_Monton)+Lugar)->Dist_Min)
       Lugar++;
    Dir_Hijo= ( *Dir_Monton )  + Lugar - 1 ;
   }else
    if(Lugar==(*Contador) &&
      ((*Dir_Monton) + Lugar - 1)->Dist_Min<Dir_Hijo->Dist_Min){
     Dir_Padre=Dir_Hijo;
     Dir_Hijo= ( *Dir_Monton )  + Lugar - 1 ;
    }
  }
 }
 return;
}

//  ------------------------------------------------------------------------
int Sucesores( int n,struct Monton **Dir_Monton,int Nodo_Ant_Mon,
    struct Arco *t,float Dist_Min_Ant,int *Contador )

/*    Esta funcion marca de manera temporal los sucesores de un Nodo
 y los mete en un monton.

  Opera de la siguiente manera:

a.- Se  toman los sucesores de un nodo.
b.- Se revisa la Etiqueta de cada uno y si es:
     -1: Se marca temporalmente con cero y se actualiza con la
informacion pertinente y se mete al monton.
      0: Si es necesario se actualiza su informacion y se
actualiza el monton.
      1: Ya esta marcado definitivamente.

  Las variables principales que utiliza son:

     Grafica.Dir_Grafica: Direccion del arreglo de nodos.
```

```
  Las funciones que se utilizan son:

     Monton: Ordena los elementos que le entran en un monton.
     Actualiza_Monton: Actualiza la informacion de un elemento
      del monton.                                          */

{
 int Campo,Ind;
 struct Nodo  *aptns;
 struct Arco  *temporal;

 int Actualiza_Monton ( struct Monton  **,float,int,int * );
 int Monton ( int,struct Monton  **,int,float,int  * );


 temporal=t;

 while ( temporal ) {
  aptns=Grafica.Dir_Grafica;
  while(aptns && aptns->Num_Nodo!=temporal->Nodo_Final)aptns=aptns->Siguiente;

  switch ( aptns->Etiqueta ) {
   case -1:
    aptns->Etiqueta=0;
    aptns->Dist_Min= Dist_Min_Ant + temporal->Peso;
    Campo=Monton ( n,Dir_Monton,aptns->Num_Nodo,aptns->Dist_Min,Contador );
    if ( Campo==0 ) return ( 0 );
    if ( Campo==1 ) return ( 2 );
    aptns->Ant_Ruta_Min=Nodo_Ant_Mon;
    temporal=temporal->Siguiente;
    continue;
   case  0:
    if (  ( aptns->Dist_Min )  >  ( Dist_Min_Ant + temporal->Peso )  ) {
     aptns->Dist_Min=Dist_Min_Ant+temporal->Peso;
     aptns->Ant_Ruta_Min=Nodo_Ant_Mon;
     Ind=Actualiza_Monton(Dir_Monton,aptns->Dist_Min,aptns->Num_Nodo,Contador);
     if ( Ind == 0 ) return ( 2 );
    }
    temporal=temporal->Siguiente;
    continue;
   case  1:
      temporal=temporal->Siguiente;
      continue;
  }
 }
 return ( 1 );
}

//  ------------------------------------------------------------------------
int Actualiza_Monton(struct Monton **Dir_Monton,float Dist_Min,int Num_Nodo,
       int *Contador)
```

```
/*    Esta funcion actualiza un elemento del monton y lo ordena.

  Opera de la siguiente manera:

a.- Se busca el Nodo dado en el monton, y se actualiza.
b.- De no haber estado, se regresa 0.
c.- Se busca el padre que le correspondio al nodo.
d.- Se realiza el intercambio de padre-hijo si es necesario para poner
  en el lugar indicado al nodo.

  Las variables principales que utiliza son:

     *Contador: Indica el numero de nodos que hay en el monton.
     Lugar: Posicion de un nodo en el monton.
     Dir_Nodo_Mon: Direccion de un nodo en el monton.
     Dir_Nodo_Padre: Direccion del nodo padre de un nodo en
    el monton.                                          */

{
 unsigned short int Lugar,i;
 struct Monton *Dir_Nodo_Mon,*Dir_Nodo_Padre,temp;

 for ( i=0;i< ( *Contador );i++ )
  if ( ( ( *Dir_Monton )+i ) ->Num_Nodo==Num_Nodo ) {
   ( ( *Dir_Monton )+i ) ->Dist_Min=Dist_Min;
   if ( i>0 ) {
    Dir_Nodo_Mon= ( *Dir_Monton )  + i ;
    if ( i%2 ) Lugar=int( .5*( i-1 ) );          // Buscando al padre.
    else Lugar=int( .5*i )-1;
    Dir_Nodo_Padre= ( *Dir_Monton )  + Lugar ;

      //Inician intercambios hasta encontrar la posicion del Nodo.
    while(Dir_Nodo_Padre->Dist_Min>Dir_Nodo_Mon->Dist_Min){
     temp=*Dir_Nodo_Mon;
     *Dir_Nodo_Mon=*Dir_Nodo_Padre;
     *Dir_Nodo_Padre=temp;

     if ( Dir_Nodo_Padre !=  ( *Dir_Monton )  ) {
      Dir_Nodo_Mon=Dir_Nodo_Padre;
      if ( Lugar%2 ) Lugar=int( .5*(Lugar - 1) );  //  Buscando el
   // nuevo padre.
      else Lugar=int( .5*(Lugar) ) -1;

      Dir_Nodo_Padre= ( *Dir_Monton )  + Lugar ;
     }
    }
   }
   return ( 1 );
  }
 return ( 0 );
```

```
}

//  ------------------------------------
int Ruta_Minima_Dijkstra ( int n,int Raiz )

/*    Esta funcion encuentra la ruta mininma entre un nodo dado y los
 demas nodos de una digrafica.

  Opera de la siguiente manera:

a.- Se etiqueta la raiz como definitiva.
b.- Se llama a la funcion Sucesores donde se crea el monton y si:
     Campo=0 No se pudo crear el monton.
     Contador=0 La raiz no tiene sucesores por lo que no hay
       solucion.
c.- Se saca un elemento del monton, se marca definitivo y se repite
  el paso b.

  Las variables principales que utiliza son:

     Grafica.Dir_Grafica: Direccion del arreglo de nodos.
     Dir_Monton: Direccion del arreglo para el monton.
     Raiz: Numero del nodo a partir del cual se desea la ruta
  minima.

  Las funciones que se utilizan son:

     Sucesores: Revisa la informacion de los sucesores de un nodo,
       la cambia si es necesario y los mete en un monton.
     Quita_Monton: Quita un elemento del monton.                    */

{
 int Contador=0,Dato=0,Campo;
 struct Nodo  *Dir_Raiz,*Dir_Dato;
 struct Monton *Dir_Monton;

 int Sucesores ( int,struct Monton **,int,struct Arco *,float,int * );
 void Quita_Monton ( struct Monton **,int *,int * );


 Dir_Monton=NULL;

 Dir_Raiz=Grafica.Dir_Grafica;
 while ( Dir_Raiz && Dir_Raiz->Num_Nodo!=Raiz ) Dir_Raiz=Dir_Raiz->Siguiente;

 Dir_Raiz->Etiqueta=1;

 Campo=Sucesores(n,&Dir_Monton,Dir_Raiz->Num_Nodo,Dir_Raiz->Arco,
       Dir_Raiz->Dist_Min,&Contador);

 if ( Campo==0 ) return ( 2 );
```

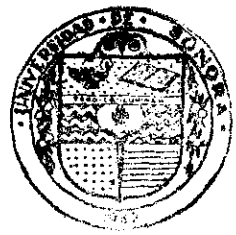
```
 if ( Campo==2 ) return ( 3 );
 if ( Contador==0 )  return ( 0 );
 while ( Contador ) {
  Quita_Monton ( &Dir_Monton,&Contador,&Dato );
  Dir_Dato=Grafica.Dir_Grafica;
  while(Dir_Dato && Dir_Dato->Num_Nodo!=Dato)Dir_Dato=Dir_Dato->Siguiente;

  Dir_Dato->Etiqueta=1;

  Campo=Sucesores(n,&Dir_Monton,Dir_Dato->Num_Nodo,Dir_Dato->Arco,
       Dir_Dato->Dist_Min,&Contador);
  if ( Campo==2 ) return ( 3 );
 }
 free( Dir_Monton );
 return(1);
}

//     --------------------
struct Ciclo Cambia_Ruta ( )

/*    Esta funcion cambia la ruta minima entre un nodo dado y los
 demas nodos de una digrafica encontrada por dijkstra.

  Opera de la siguiente manera:

a.- Se saca un elemento de la pila de arcos candidatos a entrar en
  la solucion.
b.- Se calcula la diferencia entre la distancia minima del arco
  actual y el arco candidato.
c.- Si es mayor la distancia dada por Dijkstra se revisa si el candidato
  forma ciclo al entrar, si es asi el problema no tiene solucion.
d.- Encuentra el arbol que cuelga del nodo final del candidato y
  actualiza la informacion de sus elementos.

  Las variables principales que utiliza son:

     Grafica.Dir_Graf_Sol: Direccion del arreglo de nodos de la digrafica
  solucion.
     *Pila: Direccion del primer elemento de la pila de arcos
   candidatos a entrar en la solucion.
     Hojas: Estructura que contiene el numero del ultimo nodo en
   preorden de un arbol y el antecesor en preorden del primer
   elemento del arbol.
     Elem_Arbol: Direccion del primer elemento de la lista de nodos
que constituyen el arbol calculado.
     Hay_Ciclo: Estructura con la informacion de dos nodos consecutivos
   de un ciclo.

  Las funciones que se utilizan son:

     Ciclo: Revisa si se forma un ciclo entre dos nodos.
```

```
     Arbol: Calcula el arbol que cuelga de un nodo dado.
     Quita_Plato_Preorden: Saca un elemento de la pila para
  preorden.
     Saca_Plato: Saca un elemento de la pila de arcos candidatos
a entrar en la solucion.
     Mete_Plato: Esta funcion mete un elemento a la pila
( para solucion ).                                          */

{
 int flag,Hojita,Difer,Ind,Nodo_Final,para=0,Cuenta=0;
 struct Plato_Preorden  *Elem_Arbol;
 struct Arco  *temp_a;
 struct Nodo  *apt,*Temp,*temp;
 struct Nodo_Solucion *Dir_NF,*Dir_NI;
 struct Plato  *entra;
 struct Plato  *Pila;
 struct Hojas Hojas;
 struct Ciclo Hay_Ciclo;

 int Ciclo ( int,int );
 int Quita_Plato_Preorden ( struct Plato_Preorden  ** );
 int Mete_Plato ( struct Plato  **,int,int,float );
 struct Hojas Arbol ( int,int,float,struct Plato_Preorden  ** );
 struct Plato  *Saca_Plato ( struct Plato  ** );

 Elem_Arbol=NULL;
 Pila=NULL;


 while(!para){
  Cuenta++;

  if ( !Ind )  break;

  if(!Pila) para=1;

   while ( Pila ) {
    entra=Saca_Plato ( &Pila );

    Dir_NI=Grafica.Dir_Graf_Sol;
    while(Dir_NI && Dir_NI->Num_Nodo!=entra->Nodo_Inicial)Dir_NI=Dir_NI->Siguiente;

    Dir_NF=Grafica.Dir_Graf_Sol;
    while(Dir_NF && Dir_NF->Num_Nodo!=entra->Nodo_Final)Dir_NF=Dir_NF->Siguiente;

    Difer= Dir_NF->Dist_Min - Dir_NI->Dist_Min - entra->Peso;
    if ( Difer>0 ) {
     flag=Ciclo ( entra->Nodo_Inicial,entra->Nodo_Final );
     if ( flag==1 ) {
      Hay_Ciclo.Dir_Nodo=Dir_NF;
      Hay_Ciclo.Dir_Antecesor=Dir_NI;
```

```
      return ( Hay_Ciclo );
     }
     Hojas=Arbol ( entra->Nodo_Inicial,entra->Nodo_Final,Difer,&Elem_Arbol );
     if ( (Hojas.Dir_Ultimo)==NULL ) {
      Hay_Ciclo.Dir_Nodo=Grafica.Dir_Graf_Sol;
      Hay_Ciclo.Dir_Antecesor=NULL;
      return ( Hay_Ciclo );
     }

     Dir_NF->Ant_Ruta_Min=entra->Nodo_Inicial;

     Temp=Grafica.Dir_Grafica;
     while ( Temp && Temp->Num_Nodo!=entra->Nodo_Final ) Temp=Temp->Siguiente;
     Temp->Ant_Ruta_Min=entra->Nodo_Inicial;

     ( Hojas.Dir_A_Preorden )-> S_Preorden= ( Hojas.Dir_Ultimo ) ->S_Preorden;
     ( Hojas.Dir_Ultimo )-> S_Preorden=Dir_NI->S_Preorden;
     Dir_NI->S_Preorden=entra->Nodo_Final;

     while ( Elem_Arbol ) {
      Hojita=Quita_Plato_Preorden ( &Elem_Arbol );

      temp=Grafica.Dir_Grafica;
      while ( temp && temp->Num_Nodo != Hojita ) temp=temp->Siguiente;
      temp_a= temp->Arco;

      while ( temp_a ) {
       Nodo_Final=temp_a->Nodo_Final;

       apt=Grafica.Dir_Grafica;
       while ( apt && apt->Num_Nodo!=Nodo_Final ) apt=apt->Siguiente;

       if((apt->Ant_Ruta_Min!=Hojita)&&(apt->Dist_Min>temp->Dist_Min+temp_a->Peso)){
Ind=Mete_Plato(&Pila,temp->Num_Nodo,temp_a->Nodo_Final,temp_a->Peso);
if ( Ind==0 ) {
 Hay_Ciclo.Dir_Nodo=Grafica.Dir_Graf_Sol;
 Hay_Ciclo.Dir_Antecesor=NULL;
 return ( Hay_Ciclo );
}
temp_a=temp_a->Siguiente;
continue;
       }
       temp_a=temp_a->Siguiente;
      }
     }
    }
   }
 }

 cprintf("%d",Cuenta);
```

```
 Hay_Ciclo.Dir_Nodo=NULL;
 Hay_Ciclo.Dir_Antecesor=NULL;

 return ( Hay_Ciclo );
}



//          ------------------------
struct Nodo *Nodo_Agrega( int Nodo )

/*    Esta funcion es para insertar un nodo a la lista.

  Opera de la siguiente manera:

a.- Se pide memoria para el nuevo nodo.
b.- En caso de que haya se llena con la informacion pertinente.
c.- Se agrega ordenadamente en la lista.

  La variable principal que utiliza es:

     *Grafica.Dir_Grafica: Direccion del primer elemento de la lista de nodos.
     Nodo: Numero del nodo por agregar.                                  */


{
 int estancia=0;
 struct Nodo *buscando,*r;

 struct Nodo *Nodo_Busca ( int ,int * );

 buscando=Nodo_Busca( Nodo,&estancia );
 if(estancia<0 && !buscando)  return ( Grafica.Dir_Grafica );
 if(estancia<0 && buscando)  return ( buscando->Siguiente );
 r=(struct Nodo *) malloc ( sizeof (struct Nodo) );

 if(r==NULL) return (NULL);
 r->Num_Nodo=Nodo;
 r->Ant_Ruta_Min=Nodo;
 r->Dist_Min=0;
 r->Etiqueta=-1;
 r->Arco=NULL;

 switch(estancia){
  case 0:
   r->Siguiente=NULL;
   Grafica.Dir_Grafica=r;
   break;
  case 1:
   r->Siguiente=Grafica.Dir_Grafica;
   Grafica.Dir_Grafica=r;
```

```
   break;
  case 2:
   r->Siguiente=buscando->Siguiente;
   buscando->Siguiente=r;
   break;
  case 3:
   r->Siguiente=NULL;
   buscando->Siguiente=r;
   break;
 }
 return (r);
}
//          ---------------------------------------
struct Nodo *Nodo_Busca ( int Nodo,int *Resultado )

/*    Busca un nodo, retorna el anterior, y la posicion que ocupa en la
   lista.

    Las variables principales que se utilizan son:

       Grafica.Dir_Grafica: Direccion al primer elemento de la lista de nodos.
       anterior: Direccion al anterior del nodo buscado.          */

{
 struct Nodo *s,*anterior;

 s=Grafica.Dir_Grafica;

 if(s==NULL){
    *Resultado=0;        /* La lista esta vacia   */
    return(NULL);
 }
 if(Nodo<s->Num_Nodo){
    *Resultado=1;        /* El arco no esta pero          */
    return(NULL);          /*  cabe al principio          */
 }
 if(Nodo==s->Num_Nodo){
    *Resultado=-1;         /* El arco esta al principio   */
    return(NULL);
 }

 anterior=s;
 s=s->Siguiente;

 while(Nodo>s->Num_Nodo && s!=NULL){
    anterior=s;
    s=s->Siguiente;
 }
 if((anterior->Siguiente)->Siguiente==NULL && Nodo==(anterior->Siguiente)->Num_Nodo)
    *Resultado=-3;
 if(anterior->Siguiente==NULL && Nodo>anterior->Num_Nodo)
```



```
    *Resultado=3;
 if(anterior->Siguiente!=NULL && Nodo<(anterior->Siguiente)->Num_Nodo)
    *Resultado=2;
 if((anterior->Siguiente)->Siguiente!=NULL && Nodo==(anterior->Siguiente)->Num_Nodo)
    *Resultado=-2;
 return(anterior);

}
```

B.3 Algoritmo de Floyd

```
/* ========================================================================

       VERANO 93                                                PAREIMM

       NOMBRE DEL ARCHIVO: FLOYD.C

  Este archivo aplica el algoritmo FLOYD.

       FUNCIONES QUE INCLUYE:

  Main: Organiza la informacion de entrada y salida.

   ======================================================================== */

#include <io.h>
#include <math.h>
#include <ctype.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <process.h>

#define         IO_ERR          0
#define         NO_PROBLEM      1
#define         DISCONECTED     2
#define         CICLO           3
#define         NO_MEM          4
#define         TOO_BIG         5
#define         NO_PRIM         6
#define         NO_RUTA         7

#define         EXITO           100

#define         FLOYD           3

#define  TOL            .00001
#define  SI             'S'
#define  NO             'N'

typedef unsigned short unsh;

struct Arco_Suc {                               //Arco Sucesor.
   double Peso;                                 //Peso de ruta.
   double Peso_Arco;                            //Peso del arco.
   struct Nodo *Dir_Nodo_Ini, *Dir_Nodo_Final;//Puntero a nodos extremos.
   struct Arco_Suc *Dir_Siguiente;          //Puntero al siguiente arco sucesor.
```

```
};

struct Arco_Ant {              //  Arco Antecesor.
   double Peso;                //  Peso del arco.
   struct Nodo *Dir_Ext;       //  Puntero al nodo extremo.
   struct Arco_Ant *Dir_Ant;   //  Puntero al anterior arco antecesor.
};

struct Nodo {                          //  Nodo.
   unsh Numero;            //  Numero de nodo.
   struct Arco_Suc *Dir_Arco_Suc;      //  Puntero a su primer arco sucesor.
   struct Arco_Ant *Dir_Arco_Ant;      //  Puntero a su primer arco antecesor.
   struct Nodo *Dir_Siguiente;         //  Puntero al siguiente nodo de la
};                                     // grafica (en la lista de nodos).

struct Nodo *Nodo_Ini = NULL;          //  Puntero al primer nodo.
unsh Nodo_Fuente, Nodo_Destino; //  Para calculo de la ruta.
unsh Num_Nodos = 0;
unsh PAREIMM;

#include "floyddig.h"
#include "floyd.h"

//   ---------------------------
void main(int argc, char *argv[])

/*    Esta funcion sirve para organizar la informacion de entrada y salida
     de la informacion.

      Opera de la siguiente manera:

   a.- Captura la informacion de la grafica.
   b.- Revisa si el numero de datos estan completos, y si el arco ya existe.
   c.- Agrega el arco que no haya estado.
   d.- Se aplica el algoritmo.
   e.- Se analiza si hubo solucion o ciclo negativo.
   f.- Imprime la solucion si la hubo.

      Las variables principales que utiliza son:

 Nodo_Ini: Numero de nodo inicial del arco en captura.
 Nodo_Final: Numero del nodo final en captura.
 Peso: Peso del arco en captura.

      Las funciones que utiliza son:

 Agrega_Arco: Agrega un arco en la grafica.
 Mem_Agot: Revisa si hay memoria disponible.                          */

{
   char  C;
```

```
  char Nomb_Prob[14];

  unsh Estuvo,Tipo_Prob ;      //Indica si se pudo agregar un nodo a la lista.
  unsh Ni, Nf, Destino;        //Nodos inicial y final en captura.
  unsh Num_Arcos_Sol = 0;      //Numero de arcos en la solucion.
  unsh Num_Nodos=0, Num_Arcos=0;

  double   Peso;                //  Peso de arco.
  double   Peso_Sol;            //  Peso de la solucion.

  struct Nodo *Nodo, *NodoP;//Direccion de nodo que se usa en la impresion.
  struct Arco_Suc *Arco, *ArcoP;// Direccion a arco sucesor que se usa en la
//impresion.
  FILE *Arch;

  unsh Agrega_Arco( unsh , unsh , double );
  void Mem_Agot( void );
  void Chk_Nomb_PRB(char *);
  void Chk_Nomb_SOL(char *);

  Nodo_Ini   =  NULL;          //  Direccion al nido inicial de la grafica.

 if ( access("pareimm.tmp", 0) ) {   //  Si no fue llamado por el manager
   PAREIMM = 0;
   if ( argc < 4 ) {
     cputs("\nUso: floyd problema.prb Fuente Destino\r\n\n");
     cputs("La extension .PRB es obligada para el nombre del archivo\r\n");
     cputs("que contiene al problema. Tal archivo debe tener una\r\n");
     cputs("primera linea que solo contenga el numero 0, y una ultima\r\n");
     cputs("linea que contenga 3 ceros (0 0 0).\r\n");
     exit( 1 );
   }
   cputs("\nPAREIMM  Version 1.0,  1992-1993\r\nDEPARTAMENTO DE MATEMATICAS");
   cputs("\r\nUNIVERSIDAD DE SONORA\r\n");
   strcpy(Nomb_Prob, argv[1]);
   Nodo_Fuente  = atoi (argv[2]);
   Nodo_Destino = atoi(argv[3]);
 } else
   PAREIMM = 1;
 strcpy(Nomb_Prob, argv[1]);
 Nodo_Fuente  = atoi (argv[2]);
 Nodo_Destino = atoi(argv[3]);
 Chk_Nomb_PRB( Nomb_Prob );
 if ( access(Nomb_Prob, 0) ) {
   if ( PAREIMM )
     exit( IO_ERR );
   else {
     cprintf("ERROR: Archivo %s no encontrado\r\n", Nomb_Prob);
     exit( 1 );
   }
 }
```

```
  if ( !( Arch = fopen(Nomb_Prob, "rt") ) ) {
    if ( PAREIMM )
      exit( IO_ERR );
    else {
      cprintf("ERROR: Archivo %s no abierto\r\n", Nomb_Prob);
      exit( 1 );
    }
  }
  fscanf(Arch, "%d", &Tipo_Prob);
  if ( Tipo_Prob != 2 ) {
    fclose( Arch );
    if ( PAREIMM )
      exit( NO_PRIM );
    else {
      cprintf("ERROR: Archivo %s no adecuado\r\n", Nomb_Prob);
      exit( 1 );
    }
  }
  do {
    fscanf(Arch, "%d %d %lf", &Ni, &Nf, &Peso);
    if ( Ni == 0  &&  Nf == 0 ) break;
    Estuvo = Agrega_Arco( Ni, Nf, Peso );
    if ( Estuvo == 0 ) {
      fclose( Arch );
      if ( PAREIMM )
exit( NO_MEM );
      else {
cprintf("ERROR: Memoria insuficiente para continuar\r\n", Nomb_Prob);
exit( 1 );
      }
    }
  } while( Ni != 0  ||  Nf != 0 );

  fscanf(Arch, "%d %d %lf\n",&Num_Nodos, &Num_Arcos, &Peso);
  fclose( Arch );
  if (Num_Nodos > 20 ) {
    fclose( Arch );
    if ( PAREIMM )
      exit( TOO_BIG );
    else {
      cprintf("ERROR: Archivo %s demasiado grande\r\n", Nomb_Prob);
      exit( 1 );
    }
  }

  if( !Nodo_Ini ) {
    if ( PAREIMM )
      exit( NO_PROBLEM );
    else {
      cprintf("ERROR: Archivo %s vacio\r\n", Nomb_Prob);
      exit( 1 );
```

```
    }
  }
// ------------------------------------------------------------
A P L I C A N D O   E L   A L G O R I T M O
// ------------------------------------------------------------

   C = Floyd();

// ------------------------------------------------------------

  if ( C == NO ) {
    if ( PAREIMM )
      exit( DISCONECTED );
    else {
      cprintf("ERROR: Grafica disconexa\r\n", Nomb_Prob);
      exit( 1 );
    }
  }
  if ( C != SI ) {
    if ( PAREIMM )
      exit( CICLO );
    else {
      cprintf("ERROR: Ciclo con peso negativo\r\n", Nomb_Prob);
      exit( 1 );
    }
  }

  Chk_Nomb_SOL( Nomb_Prob );
  if ( !( Arch = fopen(Nomb_Prob, "wt") ) ) {
    if ( PAREIMM )
      exit( IO_ERR );
    else {
      cprintf("ERROR: Archivo %s no abierto\r\n", Nomb_Prob);
      exit( 1 );
    }
  }
  if ( fprintf(Arch, "3\n") == EOF ) {
    fclose( Arch );
    unlink( Nomb_Prob );
    if ( PAREIMM )
      exit( IO_ERR );
    else {
      cputs("ERROR: Problemas al escribir la solucion\r\n");
      exit( 1 );
    }
  }
  Nodo = Nodo_Ini;
  while ( Nodo && Nodo->Numero != Nodo_Fuente )
    Nodo = Nodo->Dir_Siguiente;
  if ( !Nodo ) {
    if ( PAREIMM )
```

```
      exit ( NO_RUTA );
    else {
      cputs("ERROR: Nodo Fuente no encontrado\r\n");
      exit( 1 );
    }
  }
  Destino = Nodo_Destino;

  Arco = Nodo->Dir_Arco_Suc;
  while ( Arco && Arco->Dir_Nodo_Final->Numero != Destino )
    Arco = Arco->Dir_Siguiente;
  if ( !Arco ) {
    if ( PAREIMM )
      exit (NO_RUTA );
    else {
      cputs("ERROR: Nodo destino no encontrado\r\n");
      exit( 1 );
    }
  }
  Peso_Sol = Arco->Peso;

  do{
  Arco = Nodo->Dir_Arco_Suc;
    while ( Arco && Arco->Dir_Nodo_Final->Numero != Destino )
      Arco = Arco->Dir_Siguiente;
    if ( !Arco ) {
      if ( PAREIMM ) {
fclose(Arch);
exit (NO_RUTA );
      }
      else {
cputs("ERROR: Nodo destino no encontrado\r\n");
fclose(Arch);
exit( 1 );
      }
    }
    Num_Arcos_Sol++;

    NodoP = Nodo_Ini;          //  Busqueda del peso del arco.
    while (NodoP->Numero != Arco->Dir_Nodo_Ini->Numero)
      NodoP = NodoP->Dir_Siguiente;

    ArcoP = NodoP->Dir_Arco_Suc;
    while ( ArcoP->Dir_Nodo_Final->Numero != Destino)
      ArcoP = ArcoP->Dir_Siguiente;

    fprintf(Arch,"%d %d %lf\n", Arco->Dir_Nodo_Ini->Numero,Destino,
ArcoP->Peso_Arco);
    Destino = Arco->Dir_Nodo_Ini->Numero;
  }while ( Destino != Nodo_Fuente );
```

```
  fprintf (Arch,"0 0 0\n");
  fprintf (Arch, "%d %d %lf\n", Num_Arcos_Sol+1, Num_Arcos_Sol, Peso_Sol);
  fclose(Arch);
  if ( !PAREIMM )  cprintf("\nSolucion grabada en: %s\r\n", Nomb_Prob);
  exit(EXITO);
}

void Chk_Nomb_PRB(char *Nomb)
{
  unsh Pos;
  unsh Pos_Char(char *, char);

  Pos = ( Pos_Char(Nomb, '.') ? Pos_Char(Nomb, '.') - 1 : strlen(Nomb) );

  Nomb[Pos] = '.';
  Nomb[Pos+1] = 'P';
  Nomb[Pos+2] = 'R';
  Nomb[Pos+3] = 'B';
  Nomb[Pos+4] = NULL;
}

void Chk_Nomb_SOL(char *Nomb)
{
  unsh Pos;
  unsh Pos_Char(char *, char);

  Pos = ( Pos_Char(Nomb, '.') ? Pos_Char(Nomb, '.') - 1 : strlen(Nomb) );

  Nomb[Pos] = '.';
  Nomb[Pos+1] = 'S';
  Nomb[Pos+2] = 'O';
  Nomb[Pos+3] = 'L';
  Nomb[Pos+4] = NULL;
}

unsh Pos_Char(char *s, char c)
{
  unsh i=1;

  while ( *s ) {
    if ( *s == c )
      return ( i );
    s++;
    i++;
  }
  return ( 0 );
}
```

```
/* ============================================================================

INVIERNO 92-93                                              PAREIMM

NOMBRE DEL ARCHIVO: FLOYD.H

   Este archivo aplica el algoritmo FLOYD.

FUNCIONES QUE INCLUYE:

   Floyd: Aplica el algoritmo Floyd.
   Agrega_Arco_Suc: Agrega un arco en la lista de sucesores.
   Agrega_Arco_Ant: Agrega un arco en la lista de antecesores.

   ============================================================================ */


//   ---------------
char Floyd ( void )

/*    Esta funcion aplica el algoritmo de Floyd a una digrafica.

      Opera de la siguiente manera:

   a.- Se va haciendo un recorrido sobre los nodos, sobre los arcos sucesores
       y sobre los antecesores.
   b.- Cuando ya se tiene un nodo antecesor i, un nodo sucesor j del nodo k,
       entonces:

 1.- Se busca si existe como arco (i,j).
 2.- Se revisa si el nuevo camino de i a j que pasa por k es de menor
     distancia; si es asi en el nodo j se cambia:
        a) distancia por dist del nuevo camino encontrado.
        b) Nodo antecesor de j va a ser k.
      Si no es menor la distancia del nuevo camino se continua con el
     nodo antecesor de i.
 3.- Si no existe el arco (i,j), se agrega al final de la lista de
     arcos de i y al principio de los de j.
 4.- Si el nodo i y el j son el mismo y con peso negativo hay ciclo
     negativo.
      Se efectua el mismo procedimiento con todos los nodos y arcos
     sucesores y antecesores de la grafica para finalmente obtener la
     ruta minima de cualesquiera dos nodos.

      Las variables principales que utiliza son:

Dir_Arco_Suc: Direccion a arco sucesor.
Dir_Arco_Suc_E: Direccion a arco sucesor para busqueda.
Dir_Arco_Ant: Direccion a arco anterior.
Nodo: Direccion a un nodo.
```

Las funciones que utiliza son:

```
 Agrega_Arco_Suc: Agrega un arco en la lista de sucesores.
 Agrega_Arco_Ant: Agrega un arco en la lista de antecesores.          */
{

 char C;

 struct Arco_Suc *Dir_Arco_Suc, *Dir_Arco_Suc_E, *Dir_Temp_Arco_Suc;
 struct Arco_Ant *Dir_Arco_Ant, *Dir_Temp_Arco_Ant;
 struct Nodo  *Nodo;

 struct Arco_Suc *Agrega_Arco_Suc(struct Nodo *, struct Nodo *,
  struct Nodo *, double);
 char Agrega_Arco_Ant( struct Nodo *, struct Nodo *, double );


 Nodo = Nodo_Ini;
 while ( Nodo ){                                  //  a.-Recorrido Nodo.
  Dir_Arco_Suc = Nodo->Dir_Arco_Suc;

  while ( Dir_Arco_Suc ){                         //  Recorrido Arco_Sucesor.
   Dir_Arco_Ant = Nodo->Dir_Arco_Ant;

   while ( Dir_Arco_Ant ){                //  Recorrido Arco_Antecesor.
    Dir_Arco_Suc_E = Dir_Arco_Ant->Dir_Ext->Dir_Arco_Suc;

    while ( Dir_Arco_Suc_E ) {            //  Paso 1.
     if ( Dir_Arco_Suc_E->Dir_Nodo_Final == Dir_Arco_Suc->Dir_Nodo_Final )
     Dir_Arco_Suc_E = Dir_Arco_Suc_E->Dir_Siguiente;
    }
//  Paso 2.
    if(Dir_Arco_Suc_E && Dir_Arco_Suc_E->Peso>Dir_Arco_Ant->Peso+Dir_Arco_Suc->Peso){
     Dir_Arco_Suc_E->Peso = Dir_Arco_Ant->Peso + Dir_Arco_Suc->Peso;
     Dir_Arco_Suc_E->Dir_Nodo_Ini   = Dir_Arco_Suc->Dir_Nodo_Ini;

     Dir_Temp_Arco_Ant = Dir_Arco_Suc->Dir_Nodo_Final->Dir_Arco_Ant;
     while ( Dir_Temp_Arco_Ant->Dir_Ext != Dir_Arco_Ant->Dir_Ext )
Dir_Temp_Arco_Ant = Dir_Temp_Arco_Ant->Dir_Ant;

     Dir_Temp_Arco_Ant->Peso = Dir_Arco_Ant->Peso + Dir_Arco_Suc->Peso;
    } else {
//  Paso 3.
     if(!Dir_Arco_Suc_E && Dir_Arco_Suc->Dir_Nodo_Final!=Dir_Arco_Ant->Dir_Ext){
      Dir_Temp_Arco_Suc=Agrega_Arco_Suc(Dir_Arco_Ant->Dir_Ext,Nodo,
  Dir_Arco_Suc->Dir_Nodo_Final,Dir_Arco_Ant->Peso+Dir_Arco_Suc->Peso);
      if ( !Dir_Temp_Arco_Suc )   return( NO );
      C = Agrega_Arco_Ant(Dir_Arco_Suc->Dir_Nodo_Final, Dir_Arco_Ant->Dir_Ext,
  Dir_Arco_Ant->Peso + Dir_Arco_Suc->Peso);
      if ( C == NO ) return(NO );
```

```
     }else  //  Paso 4.
      if(!Dir_Arco_Suc_E && Dir_Arco_Ant->Peso+Dir_Arco_Suc->Peso<-TOL)
       return( '-' );
    }
    Dir_Arco_Ant = Dir_Arco_Ant->Dir_Ant;
   }
   Dir_Arco_Suc = Dir_Arco_Suc->Dir_Siguiente;
  }
  Nodo = Nodo->Dir_Siguiente;
 }
 return( SI );
}
//     ---------------------------------------------------------------------
struct Arco_Suc *Agrega_Arco_Suc( struct Nodo *Nodo, struct Nodo *Dir_Nodo_Ini,
  struct Nodo *Dir_Nodo_Final, double Peso )

/*    Esta funcion agrega arcos sucesores cuando no estan en la grafica.

      Opera de la siguiente manera:

   a.- Reserva memoria para el arco que se va agregar
   b.- Se ponen los parametros correspondientes al arco.
   c.- Ajustando apuntadores.

      Las variables principales que utiliza son:

Dir_Arco_Suc_Arco: Direccion del arco por agregar.
Dir_Arco_Suc_A: Para ajuste de apuntadores.                          */


{
   struct Arco_Suc *Dir_Arco_Suc_Arco, *Dir_Arco_Suc_A;


   Dir_Arco_Suc_Arco = new Arco_Suc;              // a.- Reserva de memoria
   if ( !Dir_Arco_Suc_Arco )
      return( NULL );

   Dir_Arco_Suc_Arco->Peso          =Peso; // b.-Poniendo los parametros
   Dir_Arco_Suc_Arco->Dir_Nodo_Final=Dir_Nodo_Final;//correspondientes del arco
   Dir_Arco_Suc_Arco->Dir_Nodo_Ini  =Dir_Nodo_Ini;
   Dir_Arco_Suc_Arco->Dir_Siguiente =NULL;

   Dir_Arco_Suc_A = Nodo->Dir_Arco_Suc;          // c.- Ajustando apuntadores.
   while ( Dir_Arco_Suc_A->Dir_Siguiente )
      Dir_Arco_Suc_A = Dir_Arco_Suc_A->Dir_Siguiente;

   Dir_Arco_Suc_A->Dir_Siguiente = Dir_Arco_Suc_Arco;

      return( Dir_Arco_Suc_Arco );
}
```

```
//  ------------------------------------------------------------------------
char Agrega_Arco_Ant( struct Nodo *Nodo, struct Nodo *Dir_Extr, double Peso )

/*    Esta funcion agrega arcos antecesores en la grafica cuando no estan.

      Opera de la siguiente manera:

   a.- Reserva memoria para el arco que se va agregar.
   b.- Se ponen los parametros correspondientes.
   c.- Se ajustan los apuntadores.

      Las variables principales que utiliza son:
 Dir_Arco_Ant_Arco: Direccion del arco por agregar.
 Dir_Arco_Ant_A: Para ajuste de apuntadores.                          */


{
   struct Arco_Ant *Dir_Arco_Ant_Arco, *Dir_Arco_Ant_A;


   Dir_Arco_Ant_Arco = new Arco_Ant;            //  a.-Reservando memoria
   if ( !Dir_Arco_Ant_Arco )
      return( NO );

   Dir_Arco_Ant_Arco->Peso   = Peso;    //  b.-Poniendo los parametros
   Dir_Arco_Ant_Arco->Dir_Ext= Dir_Extr;// correspondientes al arco
   Dir_Arco_Ant_Arco->Dir_Ant= NULL;

   Dir_Arco_Ant_A = Nodo->Dir_Arco_Ant; //  c.-Ajustando apuntadores
   if ( Dir_Arco_Ant_A->Dir_Ant )
      Dir_Arco_Ant_A = Dir_Arco_Ant_A->Dir_Ant;

   Dir_Arco_Ant_A->Dir_Ant = Dir_Arco_Ant_Arco;

   return( SI );
}
```

```
/* ================================================================================

INVIERNO 92-93                                                    PAREIMM

NOMBRE DEL ARCHIVO: FLOYDDIG.H

   Este archivo crea una digrafica.

FUNCIONES QUE INCLUYE:

   Agrega_Arco: Agrega un arco sucesor y antecesor a la grafica.
  Mem_Agot: Indica si hay memoria.
  Agrega_Nodo: Agrega un nodo a la grafica.

   ================================================================================ */


//              ---------------------------------
unsigned short Agrega_Arco( unsigned short Ni,          //  Nodo inicial
unsigned short Nf,      //  Nodo final
double Peso )

/*    Esta funcion agrega un arco a una grafica, de la siguiente
      manera:

   a.- Busca el arco que se le manda, buscando primero sus nodos extremos
       y de existir ambos, luego el arco.
   b.- Si encuentra el arco con el mismo Peso, se regresa.
   c.- Si lo encuentra con otro peso, actualiza el peso y se regresa.
   d.- Se agrega el nodo que no exista.
   e.- Se agrega el arco en las listas ordenadas de arcos.

       Para los pasos d y e, si no hay memoria disponible tambien se regresa.

       Los valores que devuelve son

0   si no hay memoria,
1   si se agrego con exito,
2   si el arco ya existe

    Las variables principales que utiliza son:

      Dir_Nodo_Ni: Direccion del nodo inicial del arco a agregar.
      Dir_Nodo_Nf: Direccion del nodo final del arco a agregar.
      Dir_ArcoS: Direccion para busqueda del arco en la lista de sucesores.
      Dir_ArcoSuc: Direccion del arco sucesor por agregar.
      Dir_ArcoAnt: Direccion del arco antecesor por agregar.
      Dir_ArcoA: Direccion para busqueda del anrco en la lista de antecesores.

    La funcion que utiliza es:
```

```
 Agrega_Nodo: Agrega un nodo a la grafica.                          */


{
 struct Nodo *Dir_Nodo_Ni, *Dir_Nodo_Nf;        //  Direccion de los extremos.
 struct Arco_Suc *Dir_ArcoS;
 struct Arco_Suc *Dir_ArcoSuc;
 struct Arco_Ant *Dir_ArcoAnt;
 struct Arco_Ant *Dir_ArcoA;

 struct Nodo *Agrega_Nodo( unsigned short );


 Dir_Nodo_Ni = Nodo_Ini;                        //  a.-Se busca el nodo inicial.
 while ( Dir_Nodo_Ni->Numero != Ni  &&  Dir_Nodo_Ni )
  Dir_Nodo_Ni = Dir_Nodo_Ni->Dir_Siguiente;

 Dir_Nodo_Nf = Nodo_Ini;                        //  Se busca el nodo final.
 while ( Dir_Nodo_Nf->Numero != Nf  &&  Dir_Nodo_Nf )
  Dir_Nodo_Nf = Dir_Nodo_Nf->Dir_Siguiente;

 if ( Dir_Nodo_Ni && Dir_Nodo_Nf ) {            //  Buscando el arco.
  Dir_ArcoS = Dir_Nodo_Ni->Dir_Arco_Suc;
  while ( Dir_ArcoS->Dir_Nodo_Final != Dir_Nodo_Nf  &&  Dir_ArcoS )
   Dir_ArcoS = Dir_ArcoS->Dir_Siguiente;

  if ( Dir_ArcoS  &&  abs(Dir_ArcoS->Peso - Peso) > TOL ){//  c.-El arco
   Dir_ArcoS->Peso = Peso;                              // existe con otro peso.
   return( 2 );
  }
 }                              // e.-
 if ( !Dir_Nodo_Ni ) {          //  Agregando el nodo inicial si no existe.
  Dir_Nodo_Ni = Agrega_Nodo(Ni);
  if ( !Dir_Nodo_Ni )
     return( 0 );
 }
 if ( !Dir_Nodo_Nf ) {          //  Agregando el nodo final si no existe.
  Dir_Nodo_Nf = Agrega_Nodo(Nf);
  if ( !Dir_Nodo_Nf )
     return( 0 );
 }

//                               AGREGANDO EL ARCO
//                               ----------------
  //  e.-
 Dir_ArcoSuc = new Arco_Suc;       //  Reservando memoria para el arco sucesor.
 if ( !Dir_ArcoSuc )
  return( 0 );

 Dir_ArcoSuc->Peso          = Peso;//Llenando parametros correspondientes.
```

```
 Dir_ArcoSuc->Peso_Arco     = Peso;
 Dir_ArcoSuc->Dir_Nodo_Final= Dir_Nodo_Nf;
 Dir_ArcoSuc->Dir_Nodo_Ini  = Dir_Nodo_Ni;

 Dir_ArcoS = Dir_Nodo_Ni->Dir_Arco_Suc;
 if ( !Dir_ArcoS ) {                    //  No hay arcos en ese nodo.
  Dir_Nodo_Ni->Dir_Arco_Suc  = Dir_ArcoSuc;
  Dir_ArcoSuc->Dir_Siguiente = NULL;
 } else {                               //  Buscando final de lista.
  while ( Dir_ArcoS->Dir_Siguiente ) {
     Dir_ArcoS = Dir_ArcoS->Dir_Siguiente;
  }
  Dir_ArcoS->Dir_Siguiente   = Dir_ArcoSuc;
  Dir_ArcoSuc->Dir_Siguiente = NULL;
 }

//                      AGREGANDO EL ARCO EN EL OTRO NODO
//                      ---------------------------------

 Dir_ArcoAnt = new Arco_Ant;  //Reservando memoria para el arco antecesor.
 if ( !Dir_ArcoAnt )
    return( 0 );

 Dir_ArcoAnt->Peso       =  Peso;      //  Llenando parametros.
 Dir_ArcoAnt->Dir_Ext    =  Dir_Nodo_Ni;

 Dir_ArcoA = Dir_Nodo_Nf->Dir_Arco_Ant;
  if ( !Dir_ArcoA ) {                   //  No hay arcos en la lista.
  Dir_Nodo_Nf->Dir_Arco_Ant = Dir_ArcoAnt;
  Dir_ArcoAnt->Dir_Ant      = NULL;
 } else {                               //  Buscando final de lista.
  while ( Dir_ArcoA->Dir_Ant ) {
     Dir_ArcoA = Dir_ArcoA->Dir_Ant;
  }
  Dir_ArcoA->Dir_Ant   = Dir_ArcoAnt;
  Dir_ArcoAnt->Dir_Ant = NULL;
 }

 return( 1 );
}

//   ----------
void Mem_Agot()

/*    Esta funcion indica si hubo memoria disponible.                    */
{
   printf( "\nERROR: Memoria agotada.\n" );
   exit( 1 );

}
//          --------------------------------
```

```
struct Nodo *Agrega_Nodo( unsigned short N )

/*     Esta funcion agrega un Nodo a la lista.

   Opera de la siguiente manera:

a.- Se reserva memoria para el nodo que se va a agregar.
b.- Se llenan los parametros correspondientes.
c.- Si la lista de nodos esta vacia, se pone el primer nodo.
d.- Si no es el primer nodo, se busca el final de la lista y se agrega el
   nodo por agregar.
    La funcion regresa los valores de:
     s   si el agregar tuvo exito y
     n   si se agoto la memoria.

   Las variables principales que utiliza son:

      Dir_Nodo_Q: Direccion de nodo para la busqueda del final de lista.
      Dir_Nodo_R: Direccion del nodo por agregar.                        */


{
 struct Nodo  *Dir_Nodo_Q, *Dir_Nodo_R;


 Dir_Nodo_R = new Nodo;      //  a.-Reservando memoria para el nodo.
 if ( !Dir_Nodo_R )
 return( NULL );

 Dir_Nodo_R->Numero        =  N;           //  b.-Se llenan los parametros.
 Dir_Nodo_R->Dir_Arco_Suc  =  NULL;
 Dir_Nodo_R->Dir_Arco_Ant  =  NULL;
 Dir_Nodo_R->Dir_Siguiente =  NULL;

 if ( !Nodo_Ini )                        //  c.-Primer nodo.
  Nodo_Ini = Dir_Nodo_R;
 else {
  Dir_Nodo_Q = Nodo_Ini;
  while ( Dir_Nodo_Q->Dir_Siguiente ) //  d.-Buscando el final.
     Dir_Nodo_Q = Dir_Nodo_Q->Dir_Siguiente;
  Dir_Nodo_Q->Dir_Siguiente = Dir_Nodo_R;
 }
 return( Dir_Nodo_R );
}
```

Bibliografía

[1] Aho, Alfred V.
Estructuras de datos y algoritmos
Addison-Wesley Iberoamericana.

[2] Cisneros Molina, Myriam.
Flujo constante a costo mínimo
Departamento de Matemáticas, Universidad de Sonora.

[3] Hdez. Ayuzo, Ma. del Carmen
Análisis de Redes
Publicaciones del Dpto. de Matemáticas de la Facultad de Ciencias de la UNAM.

[4] Rodríguez Alcántar, Edelmira
Estructras de datos para flujo en redes
Publicaciones internas. Proyecto PAREIMM. Departamento de Matemáticas. Universidad de Sonora.

[5] Smith, Harry F.
Data Structures, form and function
Harcourt Brace Jovanovich, Publishers.