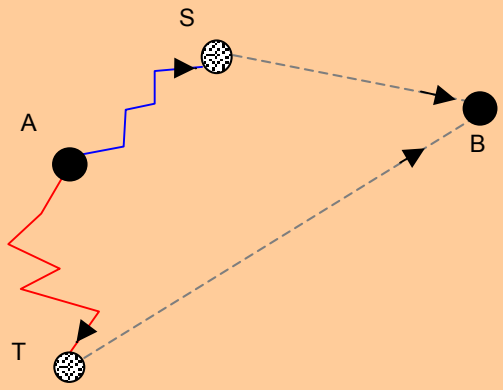
# GRAFOS Y ALGORITMOS

POR

OSCAR MEZA H.

Y

MARUJA ORTEGA F.



# GRAFOS Y ALGORITMOS

POR

OSCAR MEZA H.

Y

MARUJA ORTEGA F.

SEGUNDA EDICIÓN: 2004 (EN PUBLICACIÓN)

PRIMERA EDICIÓN: 1993 EDITORIAL EQUINOCCIO. UNIVERSIDAD SIMÓN BOLÍVAR.
ISBN 980-237-072-X

## *PRÓLOGO*

Este libro surge a raíz de una necesidad que hemos confrontado por años, de tener un material coherente y lo mas completo posible en el tratamiento algorítmico de problemas relacionados con grafos. El libro recoge una serie de conceptos, propiedades y algoritmos que son en gran parte los fundamentos sobre los cuales se apoyan muchas aplicaciones en informática utilizando el modelo de grafos. Por otra parte, existe una necesidad cada vez mayor de unificar el lenguaje y la terminología en castellano; es sabido que la teoría de grafos por ser relativamente reciente, es presentada de diferentes maneras por diferentes autores, llegándose al punto de utilizar un mismo término para definir cosas distintas. Es nuestro propósito incentivar el uso de un vocabulario único entre nuestros colegas y estudiantes. Al final del libro se presenta un glosario bastante completo de la terminología utilizada, colocando los términos equivalentes en inglés y francés.

El libro presenta un enfoque algorítmico del tratamiento de grafos. Se utilizan los grafos en el proceso de abstracción al momento de resolver un problema, es decir, un grafo lo vemos como una herramienta matemática que nos permite especificar formalmente un problema y hallar una solución al problema en términos de una solución algorítmica en el grafo. Por otro lado, se hace un tratamiento minucioso a varios métodos clásicos de búsqueda de caminos en grafos; haciendo un esfuerzo por presentar en forma unificada los algoritmos de búsqueda de caminos en grafos. Los algoritmos clásicos de recorridos en grafos (DFS, BFS) y de búsqueda de caminos mínimos (Dijkstra, Bellman, A*) se presentan en general en la literatura como algoritmos puntuales sin ninguna conexión entre sí; aquí los presentamos dentro del marco de un modelo general de algoritmos denominado modelo de etiquetamiento. Este marco general nos ha permitido hacer un estudio completo de los algoritmos clásicos de recorridos y búsqueda de caminos en grafos, haciendo énfasis en la comparación de los algoritmos, determinando las propiedades comunes entre éstos, evitando así redundancias mediante una presentación coherente y general de la problemática de caminos en grafos. Además, se presentan soluciones algorítmicas eficientes de varios problemas clásicos de grafos y se analiza un tipo particular de grafo denominado árbol, el cual es un modelo de una inmensa utilidad en informática. Cada algoritmo es presentado independientemente de los detalles de implementación, se prueba la "correctitud" y se hace un análisis de la complejidad en tiempo de diferentes implementaciones, sin descuidar la complejidad en espacio.

Una de las características del material que aquí presentamos es el tratamiento simultáneo que se hace de los grafos orientados y no orientados. Utilizamos el término lado para representar tanto a una arista como a un arco. Este tratamiento simultáneo permite condensar muchas propiedades que no dependen de la orientación, así como también permiten extrapolar propiedades y algoritmos en grafos orientados a grafos no orientados. En todo caso, cuando la situación lo amerita, hacemos mención explícita del tipo de grafo con que se trabaja.

Otra característica fundamental del libro es el orden con que las ideas son expuestas a través de los capítulos, tratando de evitar redundancia y dispersión de la información y manteniendo en lo posible la generalidad de los resultados. Por ejemplo, en el capítulo III se hace un estudio bastante amplio de las propiedades de caminos, cadenas, ciclos y circuitos, con el fin de que estas propiedades puedan ser utilizadas más adelante donde se requieran.

Los capítulos se dividen en secciones y éstas a su vez en subsecciones. Los capítulos son numerados en números romanos (I, II, etc.). Las secciones se numeran secuencialmente dentro de cada capítulo concatenando al número del capítulo un punto y el número correspondiente de la sección dentro del capítulo (Por ejemplo: I.1, I.2, etc.). La numeración de las subsecciones es similar, por ejemplo, III.3.2 representa la subsección 2 de la sección 3 del capítulo III. Las propiedades que son demostradas reciben el nombre de "proposiciones" y siguen una numeración análoga a las secciones y subsecciones, secuencialmente partiendo de uno dentro de la subsección donde está ubicada. Las figuras se numeran secuencialmente en cada capítulo.

Para leer el libro se requiere de un curso básico de álgebra (teoría de conjuntos, relaciones, funciones, estructuras algebraicas simples) y de un curso básico en algorítmica y estructuras dinámicas de datos donde se haya

utilizado un lenguaje estructurado de alto nivel como Pascal, JAVA o C. El libro puede ser utilizado en un segundo curso de un semestre sobre algoritmos y estructuras de datos.

El contenido es el siguiente:

El capítulo I pretende motivar al lector presentándole un problema real donde el proceso de abstracción en la búsqueda de una solución al problema nos lleva a modelar el problema en términos de un grafo y proponer una solución algorítmica sobre este modelo. Luego, se presenta el seudo lenguaje, inspirado en Pascal, con el cual presentaremos los algoritmos y finalmente, se definen los términos que utilizaremos en el análisis de la complejidad en tiempo de los algoritmos.

En el capítulo II se entra en materia. Se define lo que es un grafo, se presenta la terminología básica y se definen algunos parámetros importantes de grafos.

El capítulo III presenta los conceptos mas relevantes del libro: cadena, camino, ciclo, circuito, alcance y conectividad. Se dan propiedades fundamentales que serán utilizadas posteriormente.

Los capítulos IV y V comprenden un estudio algorítmico de problemas de recorridos y búsqueda de caminos en grafos y se ofrecen algunas aplicaciones interesantes de los algoritmos. Se parte de un modelo general de algoritmo de etiquetamiento y bajo este marco se analizan en detalle los algoritmos clásicos: búsqueda en profundidad (Depth First Search), búsqueda en amplitud (Breath First Search), Dijkstra, Bellman, A*, etc. Al final del capítulo V se dedica una sección a los grafos de precedencia dada la importancia de esta clase de grafos en Informática.

El capítulo VI es un estudio detallado de los árboles y arborescencias como una clase de grafos. Se estudian sus principales propiedades, las cuales son demostradas a partir de un estudio del rango de los ciclos se un grafo. Este enfoque es poco usual en los textos sobre la materia pero resulta en una forma elegante, sencilla y rápida de llegar a los resultados. Se presentan varias aplicaciones de este modelo en la resolución de problemas: árbol cobertor de costo mínimo, arborescencia de peso máximo, árboles de juego, etc.. Se analizan distintas formas de representación y de recorrido de árboles y arborescencias más adaptados a su particular estructura.

El libro incluye un último capítulo sobre problemas de Optimización en Redes, donde se estudian distintos problemas clásicos de Flujo en Redes bajo un enfoque netamente basado en el modelo de Grafos para justificar la mayoría de los resultados.

En el Anexo 1 presentamos un glosario de términos básicos en teoría de grafos, listados en un orden que obedece al grado de complejidad de la definición correspondiente de forma que todos los términos que aparecen en una nueva definición hayan sido definidos previamente. Esta organización permite que el documento sea leído en forma continua. El Anexo 2 es una lista ordenada alfabéticamente de los mismos términos, en la cual se hace referencia a las definiciones del glosario y donde aparecen las traducciones de cada uno de ellos al ingles y al francés.

En esta segunda edición se ha intentado corregir los errores de la primera edición, aunque es difícil decir que esta nueva edición esté exenta de ellos por lo que agradecemos enormemente al lector nos haga llegar los errores que pueda encontrar. Por otro lado, se han mejorado ciertas demostraciones e incorporado nuevas proposiciones, sobre todo en los capítulos IV y V. Y lo que es más importante, hemos incluido ejercicios al final de cada capítulo.

Queremos agradecer a Julián Aráoz por sus valiosas sugerencias al momento en que comenzamos el trabajo y por su colaboración entusiasta y desinteresada en la revisión del manuscrito. Agradecemos a Edgardo Broner el tiempo que dedicó a la corrección de algunos capítulos. Igualmente, queremos agradecer a Victor Theoktisto, así como a todos los profesores y estudiantes que han utilizado la primera edición del libro y han contribuido a mejorarlo con sus observaciones.

Caracas, enero de 2005.

Oscar Meza (meza@ldc.usb.ve)
Maruja Ortega (mof@ldc.usb.ve)

## *ÍNDICE*

PAG.

PAG.

## *TABLA DE SÍMBOLOS*

| | |
|---|---|
| $\equiv$ | igualdad por definición |
| (x,y) | par ordenado |
| $\{a_1,a_2, \ldots ,a_n\}$ | conjunto formado por los elementos $a_1,a_2, \ldots ,a_n$ |
| $\cong$ | isomorfismo |
| $G^{-1}$ | Grafo simétrico de G |
| $x \leftarrow y$ | a la variable x se le asigna el valor de la variable y |
| $G_S$ | Subgrafo de G inducido por el subconjunto de vértices S |
| G(F) | Subgrafo de G inducido por el subconjunto de arcos F |
| $\gamma$ (G) | Número cromático de G |
| $\alpha$ (G) | Número de estabilidad de G |
| k(G) | Conectividad de G |
| k'(G) | Conectividad fuerte de G |
| $\Omega_G(S)$ | Cociclo de S en G |
| l(C) | Largo del camino C |
| $\|\|$ | Operación de concatenación de caminos |
| Exp(C,v) | Camino resultante de expandir el camino C al vértice v |
| Inv(C) | Cadena inversa de C |
| $Sub_G(C)$ | Subgrafo parcial de G cuyos vértices y arcos son los del circuito C |
| d(v,w) | Distancia de v a w |
| $\cap$ | Intersección de conjuntos |
| $\cup$ | Unión de conjuntos |
| \|A\| | Número de elementos del conjunto A |
| $\delta$ (G) | Grado mínimo del grafo G |
| $\Delta$ (G) | Grado máximo del grafo G |
| V(G) | Conjunto de vértices del grafo G |
| E(G) | Conjunto de aristas (o arcos) del grafo G |
| $A \subseteq B$ | A es subconjunto de B |
| $A \subset B$ | A es subconjunto propio de B |
| $G \cup \{e\}$ | Representa al grafo $(V(G), E(G) \cup \{e\})$ |
| $\log_b(a)$ | Logaritmo en base b de a |
| [x] | representa el mayor entero menor o igual a x. |
| $\Re$ | Números reales |
| $\aleph$ | Números naturales |

## CAPÍTULO I

# *RESOLUCIÓN DE PROBLEMAS MEDIANTE ALGORITMOS*

### INTRODUCCIÓN

Hay que seguir varios pasos en la búsqueda de una solución algorítmica de un problema. Este capítulo presenta, a través de un ejemplo el proceso de abstracción seguido en la formulación de una solución a un problema y se dan ciertas herramientas que permiten analizar la eficiencia de un algoritmo. Finalmente se introduce el seudolenguaje que utilizaremos para presentar los algoritmos; este seudolenguaje está basado en el lenguaje Pascal.

### I.1 PASOS A SEGUIR EN LA RESOLUCIÓN DE UN PROBLEMA

En muchas situaciones reales, al enfrentarnos a un problema, no es evidente en un primer intento conseguir la solución al mismo. La formulación del problema puede venir dada o la podemos describir en lengua natural, la cual es un mecanismo de especificación ambiguo, incompleto y redundante. Si uno busca una solución al problema lo primero que hace es eliminar la incompletitud, la redundancia y la ambigüedad. En otras palabras, se desea una formulación del problema donde todo esté claro. Es aquí donde interviene el proceso de abstracción, que permite modificar el problema a modelos equivalentes más simples, descartando detalles e información que, a priori, no influyen en la solución, y establecer un enunciado preciso del problema. En este proceso el problema deviene más puro y abstracto, hablamos de un enunciado cerrado del problema. Por ejemplo, la forma más general en matemáticas de presentar un enunciado de un problema es: "Encontrar en un conjunto X dado, los elementos x que satisfacen un conjunto de restricciones K(x)".

El primer enunciado cerrado de un problema puede ser modificado para reducir el espacio X, a partir de las restricciones. De esta forma podemos mejorar la especificación del problema. Es a través de una serie de cambios a la especificación, que es finalmente resuelto. El enunciado final representa la solución. Por ejemplo, si nuestro problema es hallar el punto de intersección de dos rectas en un plano cartesiano, para obtener un enunciado cerrado basta con precisar las ecuaciones de las rectas e indicar que se quiere un punto (x,y) que satisfaga las ecuaciones de las dos rectas. Despejando una de las variables en una ecuación y sustituyéndola en la otra, obtenemos una nueva formulación del problema, la cual nos brinda inmediatamente la solución.

Existen variantes en la formulación de un enunciado cerrado. Éstas consisten en presentar una situación inicial (datos), una situación final (resultados esperados) y una serie de operadores que permiten pasar de una situación factible a otra (estados). Una solución del problema consiste en describir una serie de acciones, en términos de los operadores, que permitan pasar de la situación inicial a la final. Es aquí donde interviene el término *solución algorítmica* de un problema. Un *algoritmo* es una descripción de la solución del problema mediante una secuencia finita de acciones elementales que se supone son realizables a priori. Una *acción* es un evento que dura un período de tiempo finito y produce un resultado bien definido y previsto.

Los pasos principales a seguir en la resolución de un problema, mediante un algoritmo, son:

a) Formulación del problema en lengua natural.

b) Especificar el problema mediante un lenguaje preciso, por ejemplo el matemático, eliminando así la redundancia, incompletitud y ambigüedad de la información. En esta etapa se presenta en términos de modelos matemáticos que permiten simplificar el problema al hacer una abstracción de los datos innecesarios, rescatando sólo la información realmente relevante. Se obtiene un enunciado cerrado.

c) Diseño y Análisis de una solución del problema en términos de una solución algorítmica del enunciado cerrado.

d) Refinamiento del algoritmo hasta obtener una versión que pueda ser expresada en el lenguaje de programación que se haya escogido.

En la búsqueda de una solución algorítmica estamos interesados en determinar una que sea eficiente. Por eficiente entendemos una solución que esté compuesta por un número "razonable" de acciones elementales. En la sección I.3 se discutirá mejor qué significa razonable.

En el paso (c) del proceso de resolución de un problema, la solución algorítmica viene expresada en términos de acciones de un alto nivel de abstracción, se manipulan los objetos que intervienen en el enunciado cerrado. El proceso de refinamiento consiste en aplicar el método de *análisis descendente*, el cual posee, entre otras, las siguientes características:

(a) Se intenta utilizar esquemas conocidos de tratamiento de la información.

(b) El método funciona por etapas o niveles. En cada etapa se hace abstracción de detalles, tratando de disminuir la brecha que podría presentarse entre la primera especificación de la solución y la solución algorítmica final. En el nivel siguiente se especifican los detalles dejados de lado en el nivel anterior.

El método de análisis descendente consiste realmente en definir máquinas abstractas, cuyo funcionamiento es descrito por los algoritmos adaptados a cada nivel de abstracción. Se parte de una máquina que trata información de alto nivel. Una vez que esté precisado el tratamiento en un nivel, tomamos cada una de las partes dejadas sin especificar y las describimos en función de máquinas de nivel más bajo, es decir, que tratan información menos compleja. Cuando en este proceso una máquina abstracta coincide con una real (por ejemplo, el lenguaje de programación a utilizar), el análisis se termina.

El análisis descendente permite definir una solución concreta del problema, partiendo de una solución de un alto nivel de abstracción, es decir, una solución descrita por acciones que manipulan los objetos involucrados en el enunciado del problema. Esto permite, desde un primer momento, identificar los tipos de acciones que se efectúan sobre cada tipo de objeto. Esto último, a su vez, permite distinguir la solución algorítmica en sí de la manipulación que hay que hacer a los objetos para llevar a cabo la solución, obteniéndose así una solución que refleja claramente el método de resolución del problema. Por ejemplo, si nuestro problema es determinar en un conjunto de matrices nxn, aquéllas que sean simétricas, la solución sería tomar cada matriz y aplicarle un proceso que determina si es simétrica. Claramente el algoritmo debe reflejar un tratamiento secuencial de las matrices en el conjunto, a cada matriz (objeto) se le aplica un proceso P para determinar si es simétrica. Note que la solución algorítmica dada posee un esquema que serviría para determinar las matrices que cumplen otra propiedad. Basta con cambiar el proceso P. Note que la manipulación de los objetos, que en este caso coincide con el proceso P, no se especifica en la primera etapa de elaboración del algoritmo.

Los comentarios anteriores nos conducen a decir que el método de análisis descendente induce a diferenciar las operaciones sobre los objetos del esquema de solución del problema. Por lo tanto, al concebir un algoritmo debemos destacar el conjunto de operaciones elementales que se realizan sobre cada tipo de objeto, independientemente de como éstas se utilicen para obtener la solución final del problema. Esto permite encapsular aparte las operaciones y tener una descripción única de éstas. Esta noción se conoce con el nombre de *tipo abstracto de dato.*

### TIPOS DE DATOS, ESTRUCTURAS DE DATOS Y TIPOS ABSTRACTOS DE DATOS (TAD)

El tipo de un dato u objeto es lo que lo caracteriza, determinando los valores que pueden tomar y las operaciones que se pueden realizar con él. El tipo de un objeto puede ser uno de los predefinidos por el lenguaje, o un tipo de dato implementado por el programador.

**Definición I.1.1**

Un *Tipo Abstracto de dato (*o *TAD)* es un modelo matemático definido como un par < V, O >, donde:
V es un conjunto de valores
O es un conjunto de operaciones definidas sobre objetos que toman valores en V.

Un ejemplo sencillo de tipo abstracto de dato son Conjuntos de Enteros con las operaciones: UNIÓN, INTERSECCIÓN, DIFERENCIA.

Un permite simular una situación, como el caso de los TAD Cola y Pila.

En un TAD, los operadores pueden tomar valores del mismo TAD y posiblemente de otros tipos básicos , incluso de otros TAD. Igualmente, el resultado de un operador en un TAD no necesariamente es un valor del

mismo; sin embargo, en todos los operadores de un TAD alguno de los operandos o el resultado es un elemento del conjunto V.

Una *implementación* de un tipo abstracto de dato es la traducción de las declaraciones de los valores y de las operaciones a un lenguaje de programación. Para la representación en el computador, de los objetos del tipo definido, se escoge una *estructura de datos* adecuada.

Una estructura de datos se logra a partir de uno o varios tipos de datos básicos del lenguaje, como por ejemplo enteros o caracteres, los cuales se organizan utilizando las distintas facilidades de estructuración de datos que brinda el lenguaje, como pueden ser arreglos, registros y apuntadores.

La característica más resaltante de un TAD es justamente esta separación o independencia entre la definición y su implementación. El usuario de un tipo abstracto de dato tiene la descripción precisa del conjunto de valores y de las operaciones que puede realizar sobre ellos, así como sus resultados, y no necesita para ello conocer ningún detalle sobre la implementación. Esto último garantiza que pueda modificarse la implementación de un TAD sin que ello afecte en absoluto la consistencia del programa que lo utiliza. El acceso a los valores sólo se hace a través de los operadores, cuya implementación también es transparente al usuario.

Podemos considerar dos formas de descripción del conjunto de valores de un TAD:

– por enumeración o extensión: se enumeran las constantes que denotan los valores asociados.

– por estructuración: los valores se definen como colección o estructura compuesta por valores de otros tipos. En dicha estructura es posible manipular de forma individual cada uno de los elementos que la componen, mediante un mecanismo u operador de referencia a cada uno de ellos.

Una razón de peso para definir TAD's diferentes, aun cuando el modelo básico sea el mismo, es si las operaciones van a ser distintas, ya que la implementación más adecuada para un TAD depende sobre todo de estas últimas y no del conjunto de valores V.

Ejemplo de resolución algorítmica de un problema:

A continuación presentamos un ejemplo para ilustrar los pasos en la resolución algorítmica de problemas:

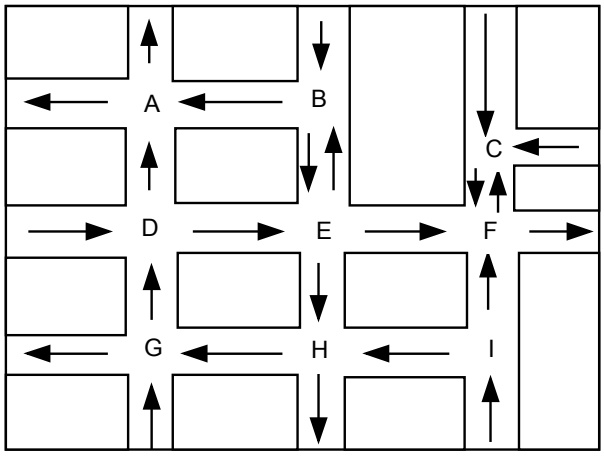



**figura I.1**

Se desea determinar si el flechado de las calles de una urbanización permite a un conductor desplazarse de una intersección a otra sin necesidad de salir de la urbanización. En la figura I.1 un conductor situado en la intersección A no puede ir a ninguna otra intersección.

Podemos modelar el problema mediante una estructura matemática conocida con el nombre de digrafo. Un digrafo G=(V,E) está conformado por un conjunto V de elementos llamados vértices y un conjunto E de elementos llamados arcos que son pares ordenados de vértices. En este caso, cada intersección representa un vértice y habrá un arco de un vértice v a otro w si existe una calle en dirección de v a w, y w es la primera intersección que

conseguimos en la calle que parte de v hacia w. La figura I.2 muestra una representación gráfica del digrafo que modela la situación de la figura I.1.

El digrafo puede ayudarnos a resolver este problema. Una solución del problema, en términos del digrafo consistiría en determinar si entre cada par de vértices (v,w), existe una secuencia $<v_0, e_1, v_1, \ldots, e_n, v_n>$ alternada de vértices y arcos, tal que $\forall i$: $e_i = (v_{i-1},v_i)$, $v_0 = v$, $v_n = w$, para algún n. Este tipo de secuencia se denomina camino de v a w.

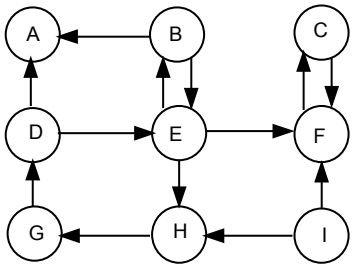



**figura I.2**

Determinar si entre cada par de vértices de un digrafo, existe un camino es un problema clásico en teoría de digrafos al cual se le han dado soluciones muy eficientes. En capítulos posteriores trataremos en más detalle este problema. Por el momento daremos una solución muy sencilla, la cual consiste en determinar, para cada vértice v, los vértices a los cuales se puede llegar a través de un camino que comience en v. El algoritmo va marcando vértices comenzando con v. Mientras exista un arco (s,t) con s marcado y t no marcado se marca t. Si todos los vértices han sido marcados, existe la posibilidad de responder afirmativamente a la pregunta: ¿se puede ir de cualquier intersección a cualquier otra? Si algún vértice no quedó marcado, entonces la respuesta es negativa. Existiendo todavía la posibilidad de responder afirmativamente a la pregunta, se desmarcan todos los vértices y se pasa a considerar otro vértice hasta hacerlo con todos:

```
Determinar si existe un camino entre cada par de vértices en un digrafo (versión 1):
{ Entrada:  Un digrafo G = (V,E).
Salida:     Una variable booleana Existe que será verdadera si existe una camino entre cada par de vértices de
G y será falsa en caso contrario. }
Variable v :vértice;
Comienzo
  Existe ← Verdad;
(1) Para cada vértice v de G hacer:
  Comienzo
  (1.1) Desmarcar todos los vértices de G;
  (1.2) Marcar los vértices que se pueden alcanzar a través de un camino que comience en v;
  (1.3) Si algún vértice no quedó marcado entonces
        Existe ← Falso;
  fin
fin
```

Notemos antes que nada ciertas características del lenguaje utilizado para describir el algoritmo. Todas las palabras reservadas como: Comienzo, fin, Para, Si, entonces, se subrayan. Los comentarios se colocan entre llaves. Las estructuras de control (Para .. hacer, Si .. entonces) tienen la misma semántica de Pascal y la misma sintaxis (una instrucción compuesta se delimita con las palabras Comienzo y fin). La asignación la representamos por una flecha dirigida de derecha a izquierda. Las variables se presentan en tipo de letra cursiva. El punto y coma (;) se utiliza como separador de instrucciones o acciones. Cuando el cuerpo de instrucciones de una estructura de control sólo consta de una instrucción o acción, no es necesario enmarcarlas entre las palabras Comienzo y fin. Las variables de entrada y de salida se consideran externas al algoritmo.

Si deseamos convertir el algoritmo en un programa en Pascal o en Módula 2, debemos refinar aún más las instrucciones. Podemos introducir una mejora al esquema de algoritmo anterior de la siguiente forma: La primera vez que la variable *Existe* sea falsa no es necesario continuar examinando los siguientes vértices de *G*. Por otra

parte, podemos tener un conjunto *M* que contenga los vértices marcados en una iteración. La acción (1.1) del algoritmo consistiría en asignar el conjunto vacío a *M*. La acción (1.2) es menos simple. Una manera de marcar partiendo de un vértice *v* es la siguiente: se marca *v*, luego marcar los vértices *w* que no hayan sido marcados y tales que exista en *G* un arco (*v*,*w*). A cada vértice *w* que se acaba de marcar colocarlo en *M* y aplicarle el mismo proceso aplicado a *v*. Este proceso sugiere mantener un conjunto P con aquellos vértices que se acaban de marcar. En cada iteración se toma un vértice *v* de P, se elimina de P, se marcan los vértices *w* no marcados tales que (*v*,*w* ) ∈ E y se colocan en P y en *M*.

Para representar al tipo abstracto de dato conjunto de vértices utilizaremos una lista lineal. Una lista lineal de elementos de un determinado tipo es una secuencia de elementos del tipo dado terminada por un elemento especial que marca el fin de la secuencia y que denotaremos por NULO. Con esta representación podemos recorrer el conjunto de vértices del digrafo.

La nueva versión más refinada del algoritmo sería:

```
Determinar si existe un camino entre cada par de vértices en un digrafo (versión 2):
{ Entrada: Un digrafo G = (V,E).
  Salida: Una variable booleana Existe que será verdadera si y sólo si existe un camino entre cada par de
vértices. }
Variables v,s,t : vértice;
          P, M : Conjunto de vértices;
Comienzo
   Existe ← Verdad;
   v ← primer vértice de V;
   Mientras (v ≠ NULO) y (Existe = Verdad ) hacer:

      Comienzo
         M ←Ø;
         { marcar vértices }
         P ← Ø;
         M ← M ∪ {v}; P ← P ∪ {v};
         Mientras P ≠ ø hacer:
            Comienzo
               s ← elemento de P ;
               P ← P - { s};
               Para cada arco (s,t) en E hacer:
                  Si t ∉ M entonces
                     Comienzo
                        M ← M ∪ {t}; P ← P ∪ {t};
                     fin
            fin
         Si M ≠ V entonces Existe ← Falso;
         v ← siguiente vértice de V;
      fin
fin
```

Los tipos de datos que manipula nuestro algoritmo son vértices, arcos, digrafos y conjuntos de vértices. Para cada uno de estos objetos habrá que especificar la estructura de datos que lo representará y definir las operaciones elementales efectuadas sobre ellos: unión y diferencia de conjuntos, primer y siguiente vértice de un conjunto, determinar los arcos que salen de un vértice, etc. En esta nueva etapa o nivel de refinamiento, sólo tendremos que ocuparnos de definir tales operadores sin preocuparnos más de la solución algorítmica del problema.

### I.2 EL LENGUAJE ALGORÍTMICO

La sintaxis del seudolenguaje algorítmico que utilizaremos para presentar los algoritmos a lo largo del libro será hecha utilizando la notación Bakus.

<Algoritmo> =
    <Encabezado de algoritmo>
    <Cuerpo de algoritmo>.

<Encabezado de algoritmo> =
  <Nombre>[<Parámetros>]:[<Tipo>]
<Nombre> = Cadena de caracteres.
<Tipo> = Cadena de caracteres.
<Parámetros> = ([VAR]<Nombre>:<Tipo>;...[VAR]
  <Nombre>:<Tipo>)
<Cuerpo de Algoritmo> =
  [<Comentarios>]
  [<Declaración>$_1$;
        .
        .
  <Declaración>$_N$;]
  Comienzo
    <acción>$_1$;
        .
        .
    <acción>$_M$
  [Donde
    [<Encabezado de algoritmo>$_1$:
      <Cuerpo de Algoritmo>];
        .
        .
    [<Encabezado de algoritmo>$_M$:
      <Cuerpo de Algoritmo>]]
fin

Los comentarios y declaraciones:
<comentarios> = { frase en español }
<Declaración> = Constante <Nombres>$_1$=<Valor>$_1$;
                                .
                                .
                  <Nombres>$_L$=<Valor>$_L$
                ⎧ó
          Variable <*Nombres*>$_1$ : <Tipo>$_1$;
                                .
                                .
                  <*Nombres*>$_P$ : <Tipo>$_P$
                ó
                 Tipo    <Nombres>=<Tipo>

<Nombres>= <Nombre>$_1$,...,<Nombre>$_n$.

Los corchetes significan que el item es opcional. Las palabras reservadas serán subrayadas. Los nombres de las variables se escribirán en letra itálica. Los parámetros formales en la lista de parámetros: <Parámetros>, pueden ser de dos tipos, por valor y por referencia. Si el parámetro es por valor, cualquier modificación que se le haga, dentro del algoritmo, no afectará el valor de la variable pasada como parámetro real en una llamada desde otro algoritmo. Si el parámetro es pasado por referencia, entonces la variable pasada como parámetro, en una llamada desde otro algoritmo, será susceptible de las mismas modificaciones del parámetro formal. En cuanto al alcance de los identificadores de variables, tipos, constantes y procedimientos, éstos tendrán significado sólo en el cuerpo del algoritmo donde se definen.

Un algoritmo es un procedimiento o una función. Un procedimiento se diferencia de una función en que la función devuelve un valor de un determinado tipo. El tipo de la función se define en el encabezado del algoritmo.

Dentro del cuerpo de la función se podrá devolver el valor de la función mediante la acción: Devolver(<expresión>), donde <expresión> es una expresión cuyo valor es del mismo tipo que el de la función.

Como una función devuelve un valor, una llamada a la función puede formar parte de una expresión; la llamada se describe de la siguiente forma:

<Nombre de la función>[(<parámetros reales>)]

donde <parámetros reales> es una lista de nombres de variables separados por comas que se corresponde con la lista de parámetros formales declarados en el encabezado de la función.

El tipo de un objeto puede ser especificado usando el lenguaje matemático para especificar los objetos y las operaciones sobre ellos. En este caso, esta especificación se hace fuera del cuerpo del algoritmo.

A continuación damos una lista de los tipos de datos que usaremos con más frecuencia y los consideraremos tipos primitivos del seudolenguaje, es decir, los usaremos sin necesidad de especificarlos separadamente:

**(a) Tipos no estructurados estándar (cuyos valores están predefinidos):**

Entero, Real, Booleano, Carácter, Natural. Las operaciones son las usuales.

**(b) Tipos no estructurados no estándar (hay que definir el conjunto de valores):**

- **Por enumeración**: Se hace una lista de los valores del tipo.

Las operaciones son la prueba de igualdad, asignación de un valor del tipo a un objeto de ese tipo, comparación pues se supone que los valores siguen el orden en que son listados.

Por ejemplo:

Tipo Sexo=(masculino, femenino);

- **Subrango:** es un intervalo finito de un tipo no estructurado. Las operaciones son las mismas del tipo no estructurado usado como base.

Por ejemplo:

Tipo Intervalo = [0..100];

**(c) Tipos Estructurados:**

- **Arreglo:** Los valores son conjuntos de objetos de un determinado tipo donde cada objeto tiene asociado un índice distinto que toma sus valores de un subrango de un tipo no estructurado distinto de los reales. Las operaciones son: Hacer referencia a un elemento del arreglo con índice determinado, asignación de un valor a un elemento con índice determinado, prueba de igualdad y asignación.

Por ejemplo:

Tipo A = Arreglo [0..9] de Real;

- **Registro:** Los valores son tuplas de un producto cartesiano de los conjuntos de valores de objetos de un determinado tipo. Las operaciones son: prueba de igualdad, referencia a una componente de la tupla, asignación tanto de un objeto tipo registro como de una componente del registro.

Por ejemplo:

```
Tipo Persona = Registro
                    Cédula : Natural;
                    Nombre:Arreglo[1..20] de Carácter
                        fin
Variable A:Persona;
```

La variable A es de tipo Persona y está formada por dos componentes, una componente es Cédula y la otra Nombre. Podemos hacer referencia a las componentes de la variable persona de la siguiente forma: A.Cédula, A.Nombre.

**(d) Otros Tipos:** Apuntador a variable de un determinado tipo base. Un apuntador es una referencia a una variable. Por lo tanto, una variable tipo apuntador contiene una referencia a una variable de un determinado tipo.

Por ejemplo:

Tipo Apunt = Apuntador a Real;

Una variable de tipo Apunt contiene una referencia a una variable de tipo Real. En el caso de los lenguajes de computación, es posible crear, al momento de ejecución, una variable de un determinado tipo y colocar su referencia en una variable de tipo Apuntador. Contemplamos esta posibilidad dentro de nuestro seudolenguaje y así tenemos las siguientes operaciones asociadas al tipo Apuntador:

- Prueba de igualdad.
- Referencia: Si A es una variable de tipo apuntador

entonces A↑ representa la variable que es apuntada por A.

- Nuevo(A): Si A es una variable de tipo apuntador, esta

operación crea una variable del tipo base asociado al apuntador y coloca en A la referencia (dirección de memoria en el caso de Pascal) a la variable creada.

Existe un valor especial para el tipo Apuntador que denota una referencia nula y se denota por NULO.

**Acciones:**

Una acción representa una descripción de un proceso finito que conlleva a un resultado bien definido. Por lo tanto, una acción puede ser una frase que describe una manipulación de objetos o variables, para obtener algún resultado. La acción puede ser una acción elemental (una asignación del valor de un objeto a otro objeto) o una acción compuesta la cual puede a su vez ser descrita en términos de una estructura de control (acción estructurada) o ser el nombre de un procedimiento o de una función, cuya descripción se hace a partir de la palabra Donde en el cuerpo del algoritmo. La asignación del valor de una expresión a una variable la representaremos por el símbolo ←. Entonces tenemos:

<Acción> = <Acción elemental> ó <Acción estructurada>
ó <Nombre de Algoritmo>[<parámetros reales>]

El tercer tipo de acción corresponde a la llamada de un procedimiento el cual puede estar descrito entre las palabras Donde y fin del cuerpo del algoritmo que hace la llamada.

Una acción estructurada puede ser una de las siguientes estructuras de control:

Para utilizar con tipo Registro (equivalente a la estructura With de Pascal):
(a) Con <nombre> hacer:
Comienzo
$<Acción>_1$;
.
.
$<Acción>_n$;
fin

Iterativas:

(a) Para cada valor de un conjunto de valores de un tipo no estructurado hacer:
Comienzo
$<Acción>_1$;
.
.
$<Acción>_n$;
fin
(b) Mientras <condición> hacer
Comienzo
$<Acción>_1$;
.
.
$<Acción>_n$;

fin
(c) Repetir
<Acción>$_1$;
.
.
<Acción>$_n$;
Hasta <condición>

Condicionales:
(a) Si <condición> entonces
Comienzo
<Acción>$_1$;
.
.
<Acción>$_n$;
fin
si no
Comienzo
<Acción>$_1$;
.
.
<Acción>$_n$;
fin

(b) La estructura Según (equivalente al CASE de Pascal):
Según <nombre> hacer:
Comienzo
<condición>$_1$: <Acción>$_1$;
<condición>$_2$: <Acción>$_2$;

<condición>$_n$: <Acción>$_n$;
fin

Es <condición> una expresión booleana. Una expresión booleana se evalúa de izquierda a derecha, no siempre llegando a evaluar toda la expresión para determinar su valor de verdad. Es decir, al evaluar (a y b), si a es falsa entonces se concluye que la expresión es falsa sin necesidad de evaluar b, y al evaluar (a ó b), si a es verdadera entonces se concluye que la expresión es verdadera sin necesidad de evaluar b.

En caso de que el cuerpo de una acción estructurada conste de una sola instrucción, no será necesario enmarcarlo entre las palabras Comienzo-fin.

### I.3 COMPLEJIDAD DE UN ALGORITMO

La *complejidad de un algoritmo* es un estimado del número de operaciones o acciones elementales que se efectúan en función del volumen de datos de entrada. Por acción elemental entendemos una acción cuya duración es independiente del volumen de datos de la entrada como, por ejemplo, multiplicar, sumar, dividir, o restar dos números. Para determinar la complejidad de un algoritmo debemos determinar antes que nada un parámetro que indique el orden de magnitud del volumen de datos de entrada; por ejemplo, si queremos buscar un elemento en un conjunto, este parámetro sería el número de elementos del conjunto.

En el libro analizaremos la *complejidad en el peor caso* de los algoritmos, lo cual significa determinar una cota superior del número de operaciones del algoritmo en función del volumen de datos de entrada, para todos los posibles valores de los datos de entrada. Por ejemplo, en la búsqueda de un elemento en un conjunto puede pasar que el elemento buscado sea el primero que tomamos, como también puede pasar que sea el último considerado. En este caso decimos que la complejidad en el peor caso de la búsqueda es del orden del número de elementos del conjunto, pues lo peor que puede pasar es que examinemos todos los elementos antes de dar con el buscado.

Para determinar el número de operaciones en el peor caso sugerimos las reglas siguientes:

1) El número de operaciones elementales de un algoritmo es igual a la suma del número de operaciones de la secuencia de acciones que conforman el algoritmo.

2) El número de operaciones en el peor caso de la acción condicional "Si ... entonces ... si no", es la suma del número de operaciones requeridas para evaluar la condición más el máximo número de operaciones entre el cuerpo del "entonces" y el del "si no".

3) El número de operaciones en el peor caso de una acción iterativa (Para, Mientras, Repetir) es igual a la suma del número de operaciones en cada iteración más el número de operaciones necesitadas para evaluar la condición de parada.

Diremos que el número de operaciones f(n) de un algoritmo es orden g(n), f(n) es O(g(n)), donde n representa el volumen de datos, si existen $c \in R$ y $n_0 \in N$ tal que: $f(n) \leq c.g(n)$, $\forall n \geq n_0$. Por ejemplo, el orden del algoritmo de multiplicación de matrices nxn es $n^3$ pues el número de operaciones que se efectúan es $n^3$ multiplicaciones más $(n-1)n^2$ sumas. Así $f(n)=2n^3-n^2$ y f(n) es menor que $2n^3$ para $n \geq 0$.

Podríamos pensar que un algoritmo es eficiente cuando tiene orden polinomial, mientras que es ineficiente cuando tiene orden exponencial. Esta categorización no se corresponde plenamente con la realidad, pues, aunque una función exponencial crece extremadamente más rápido que una polinomial a medida que aumenta el argumento, para valores pequeños del argumento, puede pasar que el comportamiento de la función exponencial sea mejor que la polinomial. Todo depende de las constantes involucradas en las funciones. Sin embargo, los algoritmos que se presentan en el libro son eficientes, bien sea porque se ha demostrado que son óptimos (no existe algoritmo con menor orden) o porque su orden es lineal o acotado por un polinomio de grado pequeño y las constantes involucradas son pequeñas.

## CAPÍTULO II

# *EL CONCEPTO DE GRAFO Y SUS APLICACIONES.*

### II.1 CONCEPTOS Y TERMINOLOGÍA BÁSICA

Un *grafo* G es una terna (V, E, $\phi$), donde V es un conjunto finito de elementos llamados *vértices* o *nodos* del grafo, E es un conjunto finito de elementos llamados *lados* y $\phi$ es una función que asigna a cada elemento de E un par de elementos de V. Si todos estos pares no son ordenados Decimos que el *grafo es no orientado o no dirigido* y sus lados se llaman *aristas*. Cuando todos los pares {a,b} son ordenados, se denotan (a,b) y se les llama *arcos*. En este caso Decimos que el *grafo es orientado o dirigido* y se le llama también *Digrafo*. Si G tiene n vértices decimos que G es de *orden n*.

Cuando no exista confusión denotaremos un Grafo por el par (V,E) sin incluir la función $\phi$. En este caso, un lado del grafo representará en sí mismo el par que le asociaría la función $\phi$. Escribiremos por ejemplo: e ={a,b}, con $e \in E$.

Cada uno de los nodos que conforman un lado del grafo se llama *extremo* del lado. Si el grafo es dirigido u orientado, entonces si e=(**a**,**b**) es un arco del grafo, el nodo **a** se llama su extremo *inicial* y el nodo **b** su extremo *terminal o final*. A dos nodos que conforman un lado de un grafo se les dice *adyacentes*. Así mismo un lado es *Incidente* en cada uno de sus extremos. Igualmente, dos lados que posean un extremo común. se les dice adyacentes. Si G es dirigido, *los sucesores* de un vértice **a** son los vértices terminales de los arcos cuyo extremo inicial es **a**, *los predecesores* son los vértices iniciales de los arcos con vértice terminal **a**.

Se llama *vecindad o conjunto de adyacencias* de un nodo **a** al conjunto de nodos adyacentes a **a**. Un nodo cuya vecindad es él mismo o el conjunto vacío se llama *vértice aislado*. Un *lazo o bucle* es un lado cuyos dos extremos coinciden.

El *grado* de un vértice **v** de un grafo G, denotado d(**v**), se define como el número de lados incidentes en **v**, donde un lazo se considera doblemente incidente. Si G es un grafo dirigido definimos el *grado interior* de **v**, denotado $d^-(\mathbf{v})$, como el número de arcos de G cuyo extremo terminal es **v**; definimos el *grado exterior* de **v**, denotado $d^+(\mathbf{v})$, como el número de arcos de G cuyo extremo inicial es **v**. Vemos que $d(\mathbf{v}) = d^-(\mathbf{v}) + d^+(\mathbf{v})$.

Con las definiciones previas, ya podemos comenzar a proponer algunas propiedades elementales:

**Proposición II.1.1**

Dado un grafo G=(V,E) se tiene que:

$$|E| = \left( \sum_{x \in V} d(x) \right)/2.$$

Si G es orientado se tiene que:

$$|E| = \sum_{x \in V} d^+(x) = \sum_{x \in V} d^-(x)$$

**Demostración**: En la suma $(\sum_{x \in V} d(x))$ cada lado es contado dos veces, de allí que dividiendo entre dos obtenemos el número total de lados.

Si G es orientado, $d^+(v)$ cuenta el número de arcos que parten de v. Por lo tanto, sumando estas cantidades sobre todos los vértices de G obtendremos el número de arcos de G.

¤

**Proposición II.1.2**

En todo grafo el número de vértices de grado impar es par.

**Demostración**: Particionemos el conjunto V de vértices del grafo en dos subconjuntos: Vi,Vp con $V = Vi \cup Vp$, donde Vp son los vértices de grado par y Vi son los de grado impar. Así:

$$\sum_{x \in V} d(x) = \sum_{x \in Vi} d(x) + \sum_{x \in Vp} d(x).$$

El lado izquierdo de esta igualdad es par (por la proposición II.1.1) y lo mismo ocurre con la segunda sumatoria del lado derecho (nodos en Vp). De esta forma, puesto que suma impar de números impares es impar, la primera sumatoria del lado derecho tendrá que ser sobre un número par de elementos, de donde |Vi| deberá ser par.

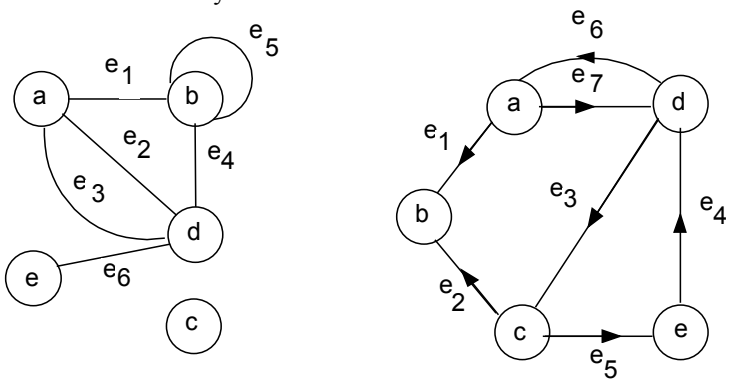
¤

$G=(V,E,\phi_1)$ $\quad\quad$ $D=(W,F,\phi_2)$

$V=\{a,b,c,d,e\}$ $\quad\quad$ $W=\{a,b,c,d,e\}$

$E=\{e_1,e_2,e_3,e_4,e_5\}$ $\quad\quad$ $F=\{e_1,e_2,e_3,e_4,e_5,e_6,e_7\}$

$\phi_1$ y $\phi_2$ se inducen de los dibujos




- G es un Multigrafo, no orientado.
- a,b son extremos de $e_1$
- a,b son adyacentes.
- $e_1$ es incidente en a.
- {b,d} es la vecindad de a.
- c es un nodo aislado.
- $e_5$ es un bucle.
- el grado de b es d(b)=4.
- $e_2$ es una arista múltiple de multiplicidad 2.

- D es un digrafo.
- a es el vértice inicial y d es el vértice terminal de $e_7=(a,d)$.
- a es predecesor de d.
- d es sucesor de a.
- el grado interior de a es $d^-(a)=1$
- D es simple.

**figura II.1**

La *multiplicidad* de un lado e se define como la cardinalidad del conjunto $\{f / \phi(f)=\phi(e)\}$, es decir, es el número de lados que tienen asociado según $\phi$, el mismo par de elementos de V. Un lado cuya multiplicidad es mayor que 1 se llama *lado múltiple*.

Se llama *grafo simple* a un grafo sin bucles ni lados múltiples. Se llama *Multigrafo* a un grafo con al menos un lado múltiple.

En la figura II.1 se ilustran las definiciones dadas hasta ahora, para ello utilizamos una representación gráfica de un grafo.

Un grafo simple no orientado G=(V,E) es un *grafo completo* si todo par a,b de nodos son adyacentes. Se denota $K_n$ al grafo completo no orientado de n nodos. También eecimos que es una *clique* de orden n (en la figura II.2 se presenta a $K_5$ y $K_2$ respectivamente).

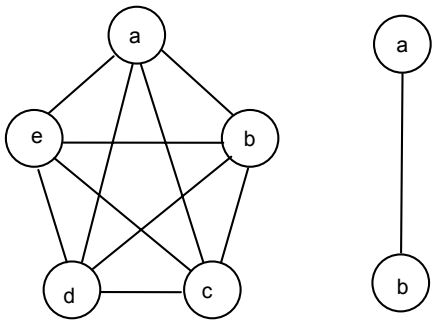



**figura II.2**

**Proposición II.1.3**

El número de aristas de $K_n$ es n(n-1)/2.

**Demostración:** Inmediato según la proposición II.1.1 y que se tiene d(x)=n-1 para todo vértice x de $K_n$.

¤

Otro grafo de interés que podemos asociar a todo grafo simple no orientado G = (V,E) es su *grafo complementario* $\overline{G}$ . Éste se define a partir del primero como $\overline{G} = (V,\overline{E})$ donde $\overline{E} = \{\{a,b\}/\ a,b \in V,\ a \neq b \text{ y } \{a,b\} \notin E\ \}$ (ver figura II.3).

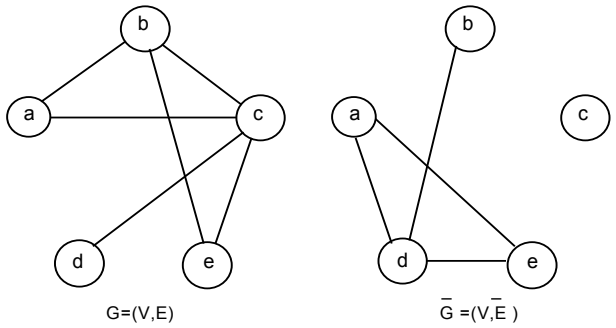



**figura II.3**

A lo largo del libro trataremos simultáneamente los grafos dirigidos y los no dirigidos. Es importante destacar que muchas de las definiciones, propiedades y algoritmos que presentaremos, se aplican tanto para grafos dirigidos como no dirigidos, y no haremos mención explícita de este hecho, asumiendo así que la orientación del grafo no es relevante en estos casos. En caso de que una definición, propiedad o algoritmo se aplique sólo a grafos orientados o sólo a grafos no orientados, haremos mención explícita de este hecho.

Por otra parte, muchas definiciones, propiedades y algoritmos sobre grafos dirigidos pueden ser extendidas a grafos no dirigidos, simplemente reemplazando cada arista del grafo por dos arcos con los mismos extremos de la arista y orientaciones contrarias. Al grafo orientado así obtenido se le denomina *grafo orientado asociado.* Recíprocamente, muchas definiciones, propiedades y algoritmos sobre grafos no dirigidos pueden ser extendidas a grafos dirigidos, asumiendo que se aplican al grafo no dirigido obtenido del grafo dirigido original, eliminando la orientación de sus arcos. El grafo no orientado así obtenido se denomina *grafo subyacente o grafo no orientado asociado*.

## II.2 REPRESENTACIONES DE UN GRAFO

### II.2.1 REPRESENTACIÓN GRÁFICA

A cada grafo se le puede siempre asociar una figura o representación gráfica, la cual viene siendo una herramienta de mucha utilidad para estudiar propiedades del mismo. Esta figura es un dibujo, en donde los nodos aparecen como puntos o círculos en el plano y cada lado se representa mediante una línea que une los dos puntos que corresponden a sus extremos. Cuando el grafo es dirigido, los arcos se representan mediante una flecha que va del nodo inicial al nodo terminal (ver figura II.1).

### II.2.2 REPRESENTACIÓN RELACIONAL

Podemos ver que el concepto de grafo dirigido y más aún el de grafo, generaliza el concepto de relación binaria. Sin embargo, el interés que presentan los grafos no se debe sólo a esta generalización. Las relaciones resultan de la noción de conjunto, mientras que la Teoría de Grafos se fundamenta esencialmente en nociones más bien geométricas, que sugieren un dibujo. Es este último punto de vista el que ha permitido tener un nuevo enfoque de la Teoría de Relaciones y es la fuente de muchos resultados importantes.

**Proposición II.2.2.1**

Sea V un conjunto finito. Existe una biyección entre el conjunto de las relaciones binarias sobre V y el conjunto de los grafos dirigidos sin arcos múltiples y cuyo conjunto de vértices es V.

**Demostración:** A toda relación binaria R sobre V asociemos un grafo orientado G=(V,E) definido de la siguiente manera:

$$(x,y) \in E \iff xRy.$$

Es evidente que G no posee arcos múltiples. Además, la correspondencia dada es una biyección.

¤

### II.2.3 REPRESENTACIÓN MATRICIAL

Las matrices proporcionan otra forma de representar grafos. Este tipo de representación es de gran utilidad en el tratamiento algebraico de grafos y para procesos computacionales.

La *matriz de adyacencias,* |V|x|V|, asociada a un grafo no orientado G=(V,E), es una matriz , A(G)=[$a_{ij}$], donde las filas y las columnas representan elementos de V. El elemento $a_{ij}$ será igual al número de aristas de G, cuyos extremos son los vértices correspondientes a i y j.

**Observación**: toda matriz de adyacencias de un grafo no orientado es simétrica, y si el grafo no tiene bucles, todos los elementos de la diagonal principal son cero.

| | a | b | c | d | e |
|---|---|---|---|---|---|
| a | 0 | 1 | 0 | 1 | 0 |
| b | 0 | 0 | 0 | 0 | 0 |
| c | 0 | 1 | 0 | 0 | 1 |
| d | 1 | 0 | 1 | 0 | 0 |
| e | 0 | 0 | 0 | 1 | 0 |

Matriz de Adyacencias del digrafo D de la figura II.1

**figura II.4**

Si el grafo es dirigido la matriz de adyacencias A(G)=[$a_{ij}$] de G es de orden |V|x|V|, cada fila y cada columna corresponde a un elemento de V y $a_{ij}$ es igual al número de arcos con extremo inicial correspondiente a i y extremo terminal correspondiente a j. Esta matriz no será siempre simétrica (ver figura II.4).

La *Matriz de Incidencias* de un grafo G=(V,E) es una matriz |V|x|E|, donde las filas representan los nodos del grafo y las columnas los lados del mismo. Si el grafo es no orientado, entonces las entradas (i,j) de la matriz tendrán el valor:

1 si el vértice correspondiente a i es extremo del lado correspondiente a j, y j no es un lazo.
2 si el vértice correspondiente a i es extremo del lado correspondiente a j, y j es un lazo.
0 en cualquier otro caso.

| | $e_1$ | $e_2$ | $e_3$ | $e_4$ | $e_5$ | $e_6$ | $e_7$ |
|---|---|---|---|---|---|---|---|
| a | -1 | 0 | 0 | 0 | 0 | 1 | -1 |
| b | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| c | 0 | -1 | 1 | 0 | -1 | 0 | 0 |
| d | 0 | 0 | -1 | 1 | 0 | -1 | 1 |
| e | 0 | 0 | 0 | -1 | 1 | 0 | 0 |

Matriz de incidencias del digrafo D de la figura II.1

**figura II.5**

Si el grafo es dirigido, entonces las entradas de la matriz serán 1, -1 y 0 (ver figura II.5):

1 si el nodo de la fila i es el extremo terminal del arco de la columna j, y éste no es un lazo.
-1 si el nodo de la fila i es el extremo inicial del arco de la columna j, y éste no es un lazo.
0 si el nodo de la fila i no es incidente en el arco de la columna j ó, si el arco de la columna j es un lazo.

**Observación**: la representación de un digrafo por su matriz de incidencia no es significativa si el grafo posee lazos, pues dos grafos diferentes podrían tener la misma matriz de incidencia.

### II.2.4 REPRESENTACIÓN EN EL COMPUTADOR

La estructura de datos utilizada para representar grafos depende de la naturaleza de los datos y de las operaciones que se van a llevar a cabo. Al escoger cuál será la mejor representación para resolver un problema particular se deben considerar factores tales como el número de nodos, densidad o número de lados, si el grafo es o no dirigido; si será necesario agregar o eliminar otros nodos o lados, y cualquier otro tipo de manipulación que pueda ser efectuada. Discutiremos a continuación las representaciones más usuales, sus ventajas y desventajas.

#### II.2.4.1 REPRESENTACIÓN EN ARREGLOS

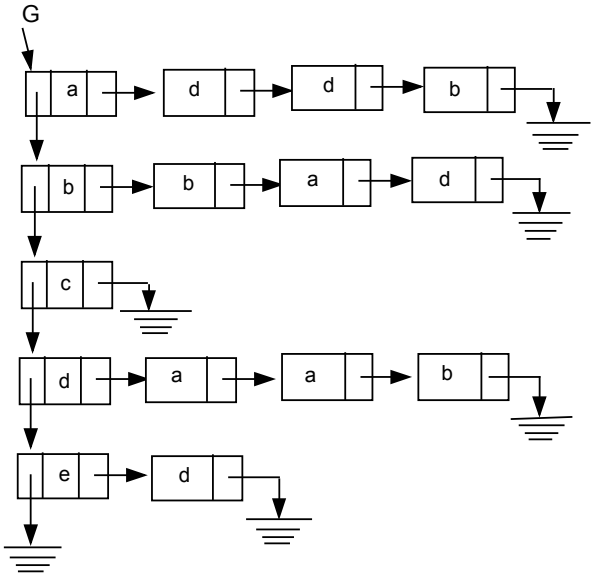
Consiste en representar el grafo mediante su matriz de incidencias o su matriz de adyacencias. Por ejemplo, la declaración de la matriz de adyacencias de un grafo dirigido sería:

```
TIPO  Digrafo= ARREGLO[1..N,1..N] DE ENTEROS
```

Otra forma de representación utilizando arreglos es la conocida como "arreglo de lados". Esta consiste en un arreglo 2x|E|, donde la primera fila corresponde a uno de los extremos del lado, el inicial si el grafo es dirigido, y la segunda fila corresponde al otro extremo del lado, el terminal si el grafo es dirigido.

En general, la representación en forma de arreglos presenta varias desventajas: 1) si el grafo tiene un número pequeño de lados ( $|E| << n^2$) entonces la matriz será poco densa, es decir, con un gran número de ceros, lo que equivale a espacio de memoria desperdiciado; 2) esta forma de representación requiere de una asignación estática de memoria; por lo tanto, si se desconoce el tamaño inicial del grafo o si se requiere agregar nodos durante el proceso,

es necesario declarar los arreglos con más espacio del realmente requerido, para poder permitir estos cambios; 3) si se debe almacenar información acerca de los nodos será necesario utilizar estructuras adicionales. Entre las ventajas de esta representación está la manipulación sencilla y la facilidad de acceso.

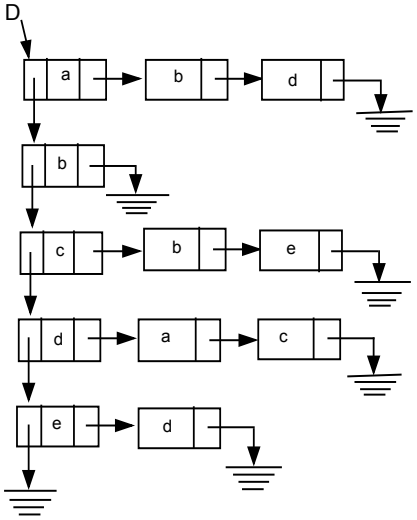



Lista de adyacencias del grafo G de la figura II.1

**figura II.6**

### II.2.4.2 REPRESENTACIÓN CON LISTAS ENLAZADAS

Las listas enlazadas ofrecen una alternativa para representar grafos y cuya ventaja principal es el manejo dinámico de la memoria, lo cual permite en forma más flexible implementar operaciones sobre el grafo tales como agregar vértices, eliminar lados, además de hacer un mejor uso de la memoria que el observado en la representación estática de los arreglos.




Lista de adyacencias del digrafo D de la figura II.1

**figura II.7**

La manera más común de representar un grafo haciendo uso de estructuras dinámicas de datos son las listas de adyacencias. En el caso no dirigido, a cada vértice **v** asociamos la lista de los vértices que son el otro extremo de las aristas incidentes al vértice **v** ( ver figura II.6). En el caso dirigido, a cada vértice **v** asociamos la lista de los vértices que son terminales de los arcos con extremo inicial **v** (ver figura II.7). A cada elemento de la lista, por corresponder a un lado, le podemos asociar información de ese lado.

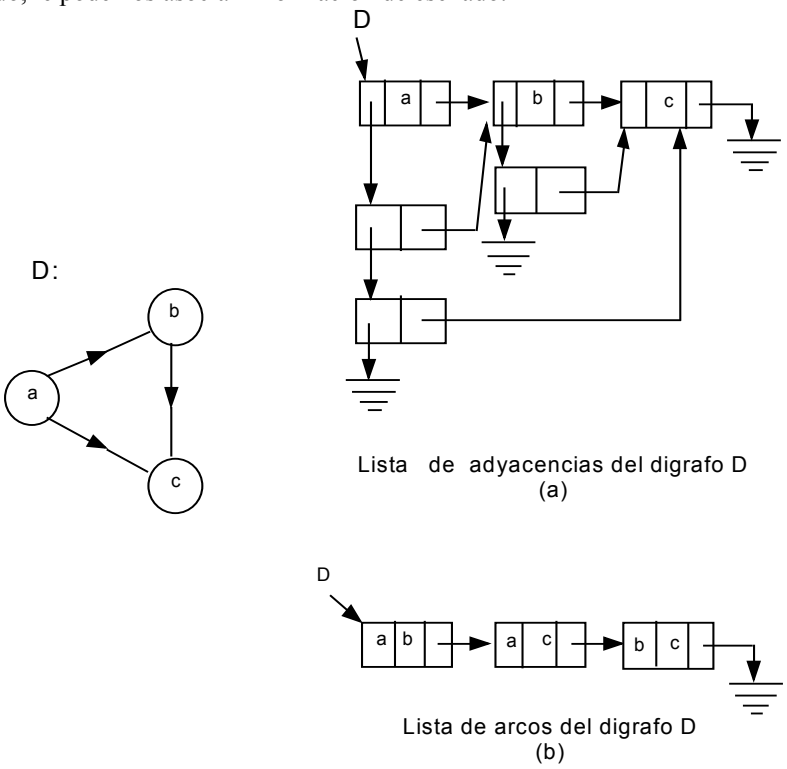



**figura II.8**

Entre las desventajas que podemos citar en la representación por listas es que debemos utilizar una cantidad de memoria directamente proporcional al número de lados del grafo para almacenar los apuntadores.

Otra desventaja para los grafos no orientados es que cada arista aparecería repetida en las listas correspondientes a sus dos extremos. Si se debe manejar información sobre los nodos, y ésta es considerable, entonces generalmente se tiene una lista con los nodos y la información respectiva, y las listas de adyacencias se hacen con apuntadores a los nodos de la lista de nodos (ver figura II.8(a)).

Otra forma alternativa, también utilizando listas enlazadas, es la representación como *Lista de Lados*. Esta consiste en una lista de pares {a,b} de nodos, uno por cada lado del grafo. Si el grafo es dirigido entonces el primer nodo **a** representa el nodo inicial y el segundo **b**, el nodo terminal. (ver figura II.8(b)) Si el grafo no es dirigido, no es necesario imponer ningún orden sobre los pares.

Esta última forma de representación permite almacenar, utilizando la misma estructura, información adicional sobre los lados. Sin embargo, si fuera necesario manejar información sobre los nodos, se requerirían estructuras adicionales.

Veamos un ejemplo sencillo para ilustrar la importancia de escoger la estructura correcta. Supongamos que se desea determinar si un grafo de orden n tiene al menos un lado. Si el grafo está representado por su matriz de adyacencias, es necesario revisar la matriz hasta encontrar un 1; esto puede llevar en el peor de los casos $n^2$ comparaciones. Si el grafo se tiene como lista de adyacencias, será necesario examinar cada nodo hasta encontrar una lista no vacía, en el peor de los casos n comparaciones, una por cada nodo. Si por el contrario tenemos el grafo representado por la lista de arcos, basta hacer una comparación para determinar si la lista es no vacía.

### II.3 SUBGRAFOS DE UN GRAFO

Un grafo $H=(W,F,\phi_H)$ es un *subgrafo parcial* ó simplemente *subgrafo* de $G=(V,E,\phi_G)$ si $W \subseteq V$, $F \subseteq E$ y $\phi_H(e) = \phi_G(e)$ para todo $e \in F$. Si $W=V$, H se llama también *grafo parcial o subgrafo cobertor de G* (ver figura II.9).

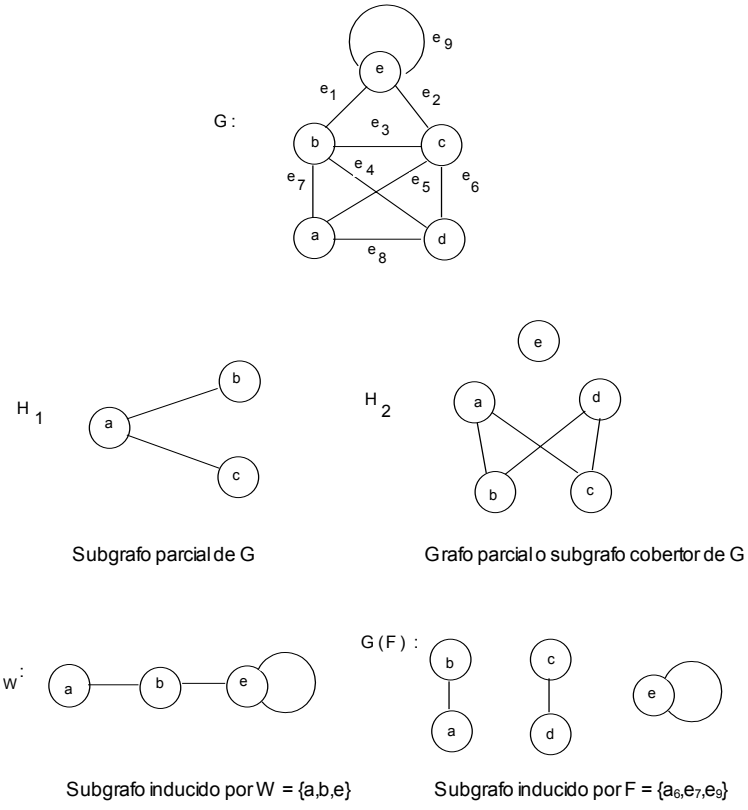



**figura II.9**

Sean W y F subconjuntos de V y E respectivamente en G=(V,E), podemos definir también los subgrafos *engendrados o inducidos* por los subconjuntos W y F de la siguiente manera:

*El subgrafo engendrado o inducido por un subconjunto de nodos* W de V es el subgrafo $G_W=(W,E')$ de G=(V,E), donde E' es el subconjunto de lados de E tales que sus dos extremos están en W.

*El subgrafo engendrado o inducido por un subconjunto de lados* F de E es el subgrafo G(F)=(V',F) de G=(V,E), donde V' es el subconjunto de nodos que son extremos de los lados en F (ver figura II.9).

### II.4 ISOMORFISMOS E INVARIANTES

¿Cuántos grafos simples podemos formar con n vértices?. Sabemos que el número máximo de lados que puede poseer un grafo simple no orientado de orden n es n(n-1)/2. Así, el número de grafos simples con n vértices es igual al número de grafos parciales de $K_n$, es decir $2^{n(n-1)/2}$.

Sin embargo, dos grafos pueden poseer la misma representación gráfica si eliminamos de éstas las etiquetas de los vértices y de los lados. Por ejemplo, los grafos G=({v,x,y},{e}) con e={v,x} y G'=({v',x',y'},{e'}) con e'={x',y'}, poseen la representación gráfica sin etiquetas de la figura II.10

Esto último lo que nos indica en esencia es que las propiedades que pueda poseer uno de los grafos, también las poseerá el otro grafo. De esta forma podemos agrupar los grafos en clases, donde cada clase corresponde a los grafos que comparten una misma representación gráfica. Estudiando las propiedades de un grafo en una clase,

estaremos estudiando las propiedades de cualquier grafo de la clase. Decimos que los grafos de una clase son *Isomorfos* entre sí.

Con la definición de grafos isomorfos, el número de grafos de n vértices no isomorfos entre sí, es considerablemente más pequeño que $2^{n(n-1)/2}$. Como ejercicio, compruebe que el número de clases de grafos isomorfos simples de tres vértices es 4.

Formalmente, dos grafos G1=(V1,E1,$\phi_1$) y G2=(V2,E2,$\phi_2$) son *isomorfos* , y lo denotaremos por G1 $\cong$ G2, si existen dos funciones biyectivas f: V1 $\rightarrow$ V2 y g: E1 $\rightarrow$ E2 tales que, para todo par de nodos a,b en V1, con {a,b} en E1:

Si $\phi_1(e_1)=\{a,b\}$ y $g(e_1)=e_2$ entonces $\phi_2(e_2)=\{f(a),f(b)\}$.

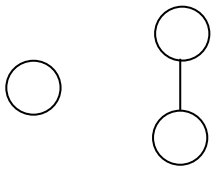


**figura II.10**

No se ha encontrado un procedimiento eficiente que permita decidir si dos grafos son isomorfos. Por procedimiento eficiente entendemos un método que en un número "razonable" de operaciones llevadas a cabo secuencialmente sobre los datos iniciales (por ejemplo, $an^b$ donde n es el número de vértices del grafo y a,b son constantes), se obtenga la respuesta. Note que hay una forma sencilla de establecer si dos grafos son isomorfos: si n es el número de vértices de cada grafo, se deben considerar todas las posibles biyecciones f y g descritas anteriormente. Sin embargo, este procedimiento resulta sumamente ineficiente, sólo para f hay que considerar n! posibilidades; por ejemplo, para grafos con 10 vértices habría que considerar en el peor de los casos 10!=3.628.800 funciones. Incluso en el computador más poderoso, sería imposible resolver el problema para n=50, simplemente porque podría tardar años en dar una solución. Otra manera de atacar el problema es estudiando las características de los dos grafos. Si existe alguna característica que no posean en común, por ejemplo, el número de vértices o el de lados, entonces los grafos no son isomorfos. Hablar de características de un grafo es equivalente, en general, a establecer una serie de parámetros que determinen las propiedades del grafo. Un *parámetro o invariante,* definido sobre una clase de grafos es una función que asigna a cada grafo de la clase un valor o una secuencia de valores numéricos, por lo general enteros. Esta función debe poseer la propiedad de asignar la misma imagen a grafos isomorfos en la clase.

Algunas invariantes de un grafo son: el número de vértices, el número de lados, la secuencia decreciente de los grados de los vértices. En la próxima sección veremos algunos otros ejemplos de invariantes.

### II.5 APLICACIONES

Nuestro interés en esta sección es mostrar varios problemas que pueden ser resueltos utilizando el modelo de grafos. De esta forma, también definiremos algunos parámetros importantes.

Suponga que la municipalidad desea cambiar el sentido de circulación de las calles y avenidas de una urbanización. Entre las preguntas que pueden surgir al momento de escoger una determinada vialidad, se encuentran: ¿la vialidad permite circular de cualquier punto a otro dentro de la urbanización?, ¿cuál es la ruta más corta entre dos puntos? Podemos modelar el problema mediante un grafo dirigido donde los vértices son las intersecciones y los arcos representan las calles con su orientación. A cada arco se le puede asociar el largo de la calle que éste representa.

Vemos que el problema original lo simplificamos utilizando un modelo matemático, sobre el cual podemos determinar propiedades, que permiten dar una respuesta a las interrogantes planteadas más arriba. En los capítulos IV y V tratamos precisamente la búsqueda de "rutas" en grafos.

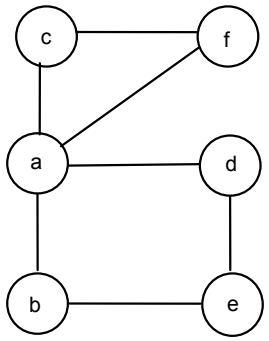
### II.5.1 NÚMERO DE ESTABILIDAD

Queremos determinar el número máximo de reinas que podemos colocar en un tablero de ajedrez de manera que ninguna se vea amenazada por otra.

Consideremos el siguiente grafo no orientado: cada casilla del tablero corresponde a un vértice, y dos vértices son adyacentes si y sólo si podemos en un movimiento de reina ir de una de las casillas, correspondiente a uno de los vértices, a la otra que corresponde al otro vértice. La propiedad importante de este grafo en términos del problema original es que nos permite resolver de una manera sencilla nuestro problema. Si varios vértices no son adyacentes entre sí, entonces podemos colocar reinas en las casillas correspondientes a esos vértices y, viceversa: si colocamos varias reinas en el tablero sin que estén amenazadas, entonces los vértices correspondientes a las casillas donde colocamos a esas reinas no son adyacentes entre sí en el grafo.

Por lo tanto nuestro problema se reduce a determinar un conjunto de mayor cardinalidad entre los conjuntos de vértices que cumplen la siguiente propiedad "los vértices del conjunto no son adyacentes entre sí". Este tipo de conjunto se denomina *conjunto estable o independiente*. La máxima cardinalidad que puede poseer un conjunto estable en un grafo G es un parámetro que recibe el nombre de *número de estabilidad o de independencia* de G y se denota por $\alpha(G)$ (ver figura II.11).

G:




$\alpha(G) = 3$

S = {b,c,d} es un conjunto estable máximo

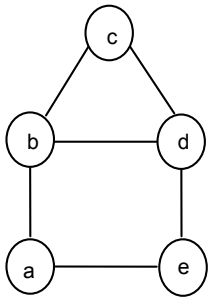
**figura II.11**

Se demuestra que el número de estabilidad del grafo asociado al problema de las reinas del tablero de ajedrez es 8, es decir, el número máximo de reinas que podemos colocar en el tablero sin que estén amenazadas entre sí.

### II.5.2 NÚMERO DE DOMINACIÓN

Queremos colocar el mínimo número de reinas sobre un tablero de ajedrez de forma tal que todas las casillas estén amenazadas.

G:




$\beta(G) = 2$

B = { a, c } es un conjunto dominante

**figura II.12**

Si consideramos el mismo grafo definido en la sección II.5.1 para modelar nuestro problema, encontraremos que, si un conjunto de vértices satisface la propiedad: "todo vértice fuera del conjunto es adyacente a al menos un vértice del conjunto", entonces, al colocar las reinas en las casillas correspondientes a los vértices del conjunto todas las casillas del tablero estarán amenazadas. Recíprocamente, dado un conjunto de casillas que al colocarles reinas todas las demás casillas quedan amenazadas, el conjunto de vértices correspondientes a estas casillas posee la

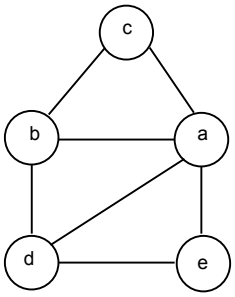
propiedad enunciada más arriba. Un conjunto de vértices de un grafo satisfaciendo esta propiedad se denomina *conjunto dominante*. Podemos concluír entonces que resolver el problema original es equivalente a determinar la cardinalidad mínima de un conjunto dominante del grafo asociado al tablero de ajedrez. La cardinalidad mínima de un conjunto dominante de un grafo es un parámetro denominado *número de dominación* de G y se denota por $\beta(G)$ (ver figura II.12). Para el grafo asociado al tablero de ajedrez este número es 5.

### II.5.3 NÚMERO CROMÁTICO

Se quiere determinar la asignación de horarios a un conjunto de cursos dictados por una empresa de computación. Suponga que hay m cursos y n estudiantes inscritos. Cada estudiante está inscrito en varios cursos, es decir, cada estudiante tiene asociado un conjunto de cursos que son aquellos que él tomó. Cada curso tiene una duración de una hora diaria. El horario de los cursos debe ser tal que en las ocho horas laborales diarias puedan ser dictados todos los cursos y dos cursos no puedan ser dictados a la misma hora si existe al menos un estudiante que haya tomado esos dos cursos.

Para resolver este problema consideramos el grafo no orientado siguiente: los vértices del grafo corresponden a los m cursos y dos vértices son adyacentes si los cursos correspondientes poseen al menos un estudiante en común. La propiedad importante de este grafo en términos del problema es que un conjunto estable de vértices (ver sección II.5.1) corresponde a un conjunto de cursos que pueden ser dictados a la misma hora ya que no tienen estudiantes en común. Ahora bien, al particionar el conjunto de vértices en conjuntos estables, podemos asignar a cada conjunto de la partición una hora del día y así habremos asignado el horario a los cursos. Recíprocamente, una asignación de horarios determina una partición del conjunto de cursos donde cada clase de la partición es el conjunto de cursos que son dictados a la misma hora. Esta partición determina a su vez una partición del conjunto de vértices del grafo asociado al problema en conjuntos estables.

G:




$\gamma$ (G) =3

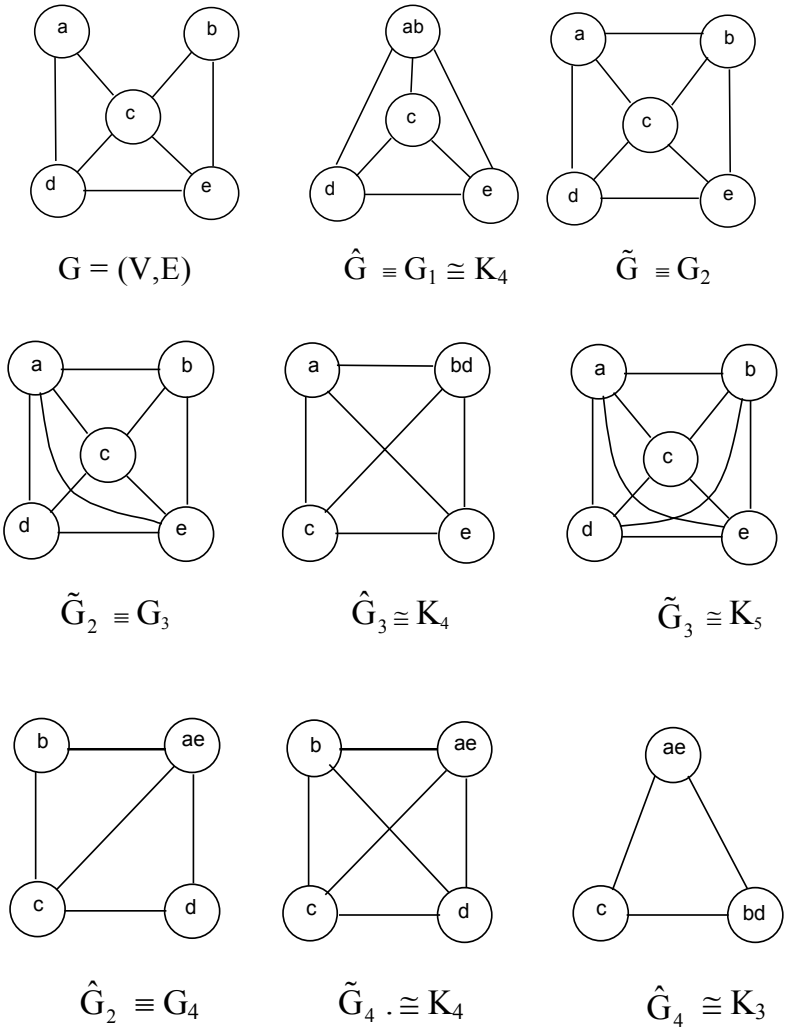
C = { { a } , { c , d }, {b,e} } es una coloración mínima de los vértices de G.

**figura II.13**

Una partición del conjunto de vértices de un grafo en conjuntos estables se denomina *una Coloración* del grafo. Nuestro problema tendrá una solución si existe una coloración formada por a lo sumo 8 conjuntos estables. La existencia de tal coloración la podemos verificar determinando la cardinalidad mínima de una coloración, es decir, el menor número de clases que puede poseer una partición del conjunto de vértices en conjuntos estables. La cardinalidad mínima de una coloración de un grafo G es el parámetro denominado *Número Cromático* de G y se denota por $\gamma(G)$ (ver figura II.13).

Dada la importancia que tiene el número cromático en la resolución de problemas prácticos, veremos un procedimiento que nos permite calcularlo para cualquier grafo. La complejidad de este procedimiento crece exponencialmente en función del número de vértices. Muchos procedimientos se han propuesto para resolver este problema, sin embargo no se conoce un procedimiento eficiente (complejidad polinomial) hasta el momento.

Observación: Hallar la coloración mínima de un grafo cualquiera G es equivalente a hallar la coloración mínima del grafo simple obtenido a partir de G eliminando la orientación de los lados (si G es orientado), eliminando los bucles y dejando un solo lado por cada lado múltiple. Por lo tanto podemos restringirnos a grafos simples cuando hablamos de coloración.




El número cromático del grafo G=(V,E) es: $\gamma(G) = \gamma(K_3) = 3$.

**figura II.14**

Consideremos un grafo simple G=(V,E) y dos nodos **a,b** en V no adyacentes. "Enlazar" estos dos vértices significa colocar una arista entre ellos. Al grafo así obtenido denotémoslo $\tilde{G}$. "Contraer" los dos vértices significa eliminar ambos vértices, **a** y **b**, de G y colocar en su lugar un solo vértice, que podríamos llamar "**ab**" el cual será adyacente a todos los nodos anteriormente adyacentes a **a** ó **b**. Denotaremos por $\hat{G}$ al grafo que se obtiene al contraer dos nodos.

En una coloración mínima de G = (V,E), los nodos **a,b** tienen el mismo color (es una coloración de $\hat{G}$) o tienen colores diferentes (es una coloración de $\tilde{G}$). Consideremos separadamente $\hat{G}$ y $\tilde{G}$, y repitamos en cada uno de ellos los procesos de contracción y enlace separadamente, tantas veces como sea necesario hasta obtener en cada caso grafos completos. El número cromático de G=(V,E) será igual al número de vértices del grafo completo más pequeño obtenido (ver figura II.14).

Un teorema fácil de demostrar y que nos da una cota para $\gamma(G)$, es el siguiente:

**Proposición II.5.3.1**

Para todo grafo simple no orientado G=(V,E): $\gamma(G) \leq \Delta(G) + 1$, donde $\Delta(G) = \max\{d(x)/x \in V\}$.

**Demostración:** Sea x un nodo cualquiera de G=(V,E), los d(x) nodos adyacentes a él estarán coloreados en el peor de los casos con d(x) colores diferentes, por lo que, disponiendo de d(x) + 1 colores, podremos darle uno diferente a x.

Como para todo x se tiene que $d(x) \leq \Delta(G)$, podemos concluir que para colorear todos los nodos del grafo de forma que dos nodos adyacentes tengan colores diferentes, bastarán $\Delta(G) + 1$ colores.

¤

### II.6 EJERCICIOS

**1.** Modele el comportamiento de una máquina que vende chocolates a 650 Bs. La máquina acepta sólo billetes de 500, 100 y 50 bolívares y no devuelve cambio.

**2.** Modele el comportamiento de una máquina similar a la anterior, pero con capacidad de dar vuelto.

**3.** Un sastre trabaja con una máquina de coser que se inspecciona cada año. El sastre, al comienzo de cada año, puede decidir entre reparar o vender la máquina. Los precios de reparación y venta dependen del número de años de uso de la máquina y vienen dados por la tabla:

| Años | Reparar (Bs.) | Vender (Bs.) |
|---|---|---|
| 1 | 700 | 1000 |
| 2 | 300 | 500 |
| 3 | 900 | 200 |
| 4 | - | 0 |

Después de cuatro años de uso, la máquina no puede ser reparada y su precio de venta es despreciable. Si el sastre decide vender su máquina, debe comprarse una nueva cuyo precio es de Bs. 2000. La máquina que posee el sastre actualmente tiene 2 años de uso. Suponga que estamos a comienzo de año. El sastre desea conocer la planificación de cuatro años con todas las posibles acciones que puede tomar.

**4.** Considere el problema de reconocer si una secuencia de caracteres dada pertenece a un lenguaje especificado mediante la siguiente expresión regular:

$a^*b^+c^*d+a^*b^+c^*$

donde:

(a) $V=\{a,b,c,d\}$ es el alfabeto del lenguaje.

(b) $\forall x,y \in V$:

- $x^*$ se interpreta como 0 o más ocurrencias de $x$.
- $x^+$ se interpreta como 1 o más ocurrencias de $x$.
- $x$ se interpreta como una ocurrencia de $x$.
- $x+y$ se interpreta como la ocurrencia de $x$ o la ocurrencia de $y$.

Por ejemplo, el string **aaab** pertenece al lenguaje dado porque:

$\mathbf{aaab}=\mathbf{a}^3\mathbf{b}^1\mathbf{c}^0\mathbf{d}^0$

mientras que el string **aacc** no pertenece al lenguaje ya que:

**aacc**= $\mathbf{a}^2\mathbf{b}^0\mathbf{c}^2\mathbf{d}^0$ y la especificación exige al menos una ocurrencia del símbolo **b**.

Construya un diagrama de estados que permita reconocer una palabra del lenguaje anterior si se lee carácter a caracter secuencialmente de izquierda a derecha. El diagrama de estados viene definido de la siguiente forma:

- Existe un estado inicial que denotaremos por $q_0$ y que representa "No se ha leído ningún carácter de la palabra"
- Uno o varios estados finales que indican que se ha leído toda la palabra secuencialmente de izquierda a derecha y la palabra pertenece al lenguaje especificado.
- Un estado de error $q_e$, al cual se llegará en caso de que la palabra a ser reconocida no se encuentra en el lenguaje.
- Arcos etiquetados $a_k$ de la forma $(q_i, c_k, q_j)$, donde $q_i$ es el vértice inicial del arco, $q_j$ es el vértice terminal del arco y $c_k$ es un carácter del alfabeto del lenguaje. Los vértices $q_i$ y $q_j$ representan estados y el arco en sí representa una transición del estado $q_i$ al estado $q_j$ una vez que se lee secuencialmente el siguiente carácter $c_k$ de la palabra que se está reconociendo. Dos arcos con el mismo nodo inicial no pueden tener la misma etiqueta.

  Ejemplo: el diagrama en la figura II.15 permite reconocer la palabra "ac". Este se interpreta de la siguiente manera: partiendo del estado $q_0$ si el primer carácter de la palabra es una "a" entonces se pasa al estado $q_1$ (este estado se puede interpretar como "se ha leído una a"); si el primer carácter no es "a" entonces se cae en el estado de error $q_e$. La etiqueta *x* representa cualquier carácter salvo las etiquetas de los otros arcos con igual vértice inicial que el arco con etiqueta *x*.

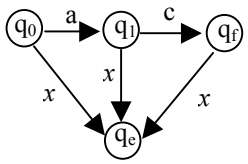



**figura II.15**

**5.** La figura II.16 representa un juego. Una pelota puede entrar por la entrada A o por la entrada B. Los niveles $x_1, x_2, x_3$ pueden hacer que la pelota vaya hacia la derecha o hacia la izquierda. Toda vez que una pelota pasa por un nivel hace que este cambie de sentido, de tal forma que la próxima pelota que pase por allí tomará el camino opuesto. Modele el comportamiento de este juego mediante grafos; donde los vértices representen estados de los niveles $x_1$, $x_2$, $x_3$.

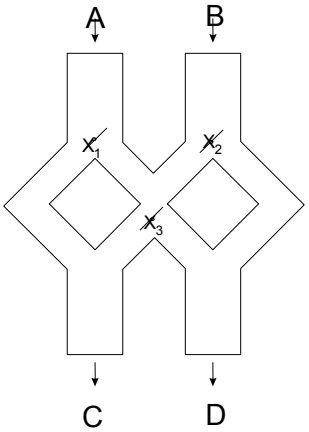



**figura II.16**

**6.** El ladrón de Bagdag se encuentra prisionero en un cuarto en el cual hay tres puertas que conducen a un túnel distinto cada una de ellas, de los cuales sólo uno conduce a la libertad. La duración de los viajes a través de los túneles, así como la probabilidad de escoger cada una de las puertas se indican en la figura II.17.

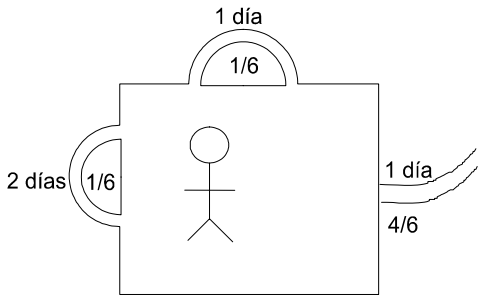



**figura II.17**

Sabiendo que el ladrón tiene muy mala memoria (no recuerda la puerta que tomó la última vez), proponga un grafo que permita calcular la probabilidad de que el ladrón salga al cuarto día (suponiendo que el ladrón no permanece parado en el cuarto.

**7.** Al aproximarse la fecha prevista para el campeonato en el cual intervendrán los equipos A, B, C y D, los organizadores necesitan determinar el número mínimo de días que éste puede durar, sabiendo que:

i. Cada equipo debe jugar una vez contra cada uno de los otros participantes.

ii. Un mismo equipo no puede jugar dos veces el mismo día.

iii. Sólo debe descansar un equipo por día.

Formule el problema planteado en términos de grafos. Construya el grafo que corresponde al problema y diseñe un algoritmo para resolverlo.

**8.** Sea *G* el grafo de la figura II.18.

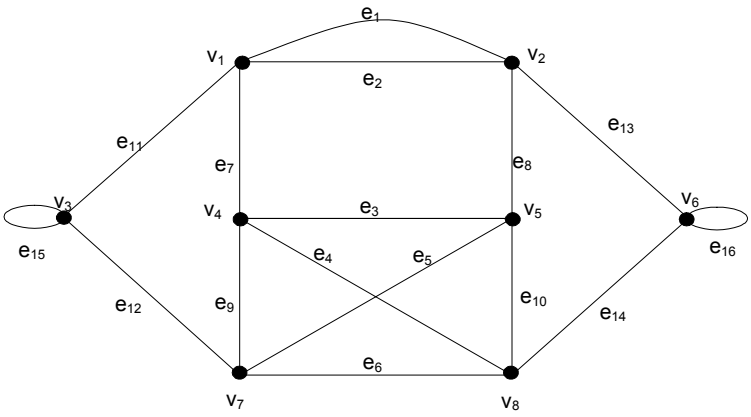



**figura II.18**

Responda las siguientes preguntas, explicando el por qué de sus respuestas

a) ¿*G* es un multigrafo? ¿Es un grafo simple? ¿Es conexo? ¿Es completo?

b) ¿Cuáles son los vértices adyacentes a $v_3$?

c) ¿Cuál es el grado de $v_3$?

d) ¿Cuáles son las aristas incidentes en $v_6$?

e) ¿Cuáles son los extremos de $e_3$?

f) ¿Cuál es la vecindad de $v_6$?

g) ¿Qué multiplicidad tiene $e_1$?

h) Determine dos subgrafos isomorfos en $G$

i) Determine el subgrafo inducido por $W=\{v_1,v_2,v_3,v_8\}$

j) Determine el subgrafo inducido por $F=\{e_2,e_8\}$

**9.** Sea $G$ el grafo dado en la figura II.19.

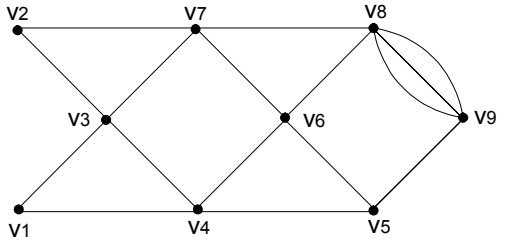



**figura II.19**

Responda a las siguientes preguntas, justificando en cada paso su respuesta:

¿$G$ es un multigrafo?

¿Cuáles son los vértices adyacentes a $v_1$? ¿Cuál es el grado de $v_9$?

¿Cuáles aristas son incidentes en $v_9$?

Determine el subgrafo inducido por $\{v_3, v_5, v_7, v_8, v_9\}$.

**10.** Sea $G$ el grafo de la figura II.20.

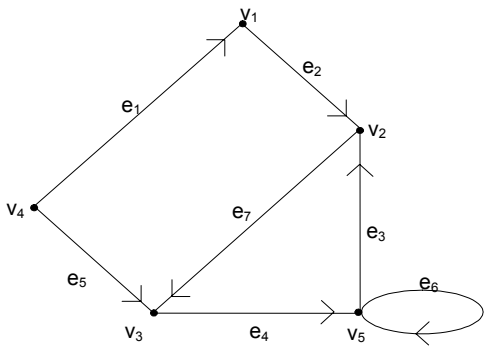



**figura II.20**

Responda las siguientes preguntas, justificando en cada paso su respuesta:

a) ¿Es simple?

b) Calcule los sucesores de $v_4$.

c) Calcule los predecesores de $v_5$.

d) Calcule la vecindad de $v_4$.

e) Calcule $d^-(v_1)$.

f) Calcule $d(v_4)$.

g) Construya la matriz de adyacencias del grafo.

h) Sea $S=\{v_1, v_3, v_4\}$. Construya el subgrafo $Q$ inducido por $S$.

**11.** Pruebe que en un digrafo la suma de los grados de entrada de todos los vértices del grafo es igual a la suma de los grados de salida de todos los vértices, e igual al número de arcos del grafo.

**12.** Dado un grafo dirigido G=(V,E), escriba un procedimiento para construir su grafo simétrico $G^{-1}$, suponiendo que se tiene el grafo G representado mediante:

- Su matriz de adyacencias
- Su matriz de incidencias
- Su lista de adyacencias

**13.** Se quiere organizar una comisión de estudiantes de manera tal que dicha comisión sea lo más pequeña posible y que los que la conforman conozcan la forma de trabajo de todos los demás. Para eso se tiene una lista de los equipos de trabajo que se han formado desde el comienzo de su carrera:

- Año 1: Andrés-Bernardo, Horacio, Daniel, Enrique-Fernando, Carolina-Ana.
- Año 2: Andrés-Fernando, Bernardo-Carolina, Daniel, Enrique,
- Año 3: Andrés-Carolina, Bernardo-Enrique, Daniel, Horacio-Ana, Fernando
- Año 4: Andrés-Enrique, Ana-Fernando, Horacio-Fernando, Daniel, Carolina.

Modele el problema mediante grafos.

**14.** Con motivo de su regreso al país, Horacio desea invitar a su casa a sus antiguos amigos de estudios: Pablo, Diego, Gaspar, Julián y Nicolás. Sin embargo, a pesar de que Horacio mantiene buenas relaciones con cada uno de sus compañeros, entre ellos han surgido diferencias al pasar de los años:

- Nicolás y Gaspar riñeron por cuestiones de trabajo.
- Pablo es ahora derechista mientras que Nicolás sigue siendo izquierdista.
- Julián le debe dinero a Nicolás.
- Pablo, Julián y Diego son fanáticos de distintos equipos de beisbol, por lo que siempre terminan peleando.

Ante esta situación, Horacio se da cuenta de que será necesario hacer más de una reunión para poder verlos a todos sin que se presenten situaciones incómodas, pero por otro lado, la situación económica lo obliga a realizar el mínimo de reuniones posibles.

a) Modele utilizando grafos el problema que se le plantea a Horacio. Indique que representan los vértices y lados de su grafo y especifique si este deberá ser orientado o no.

b) Resuelva el problema, indicando a Horacio tantas soluciones como le sea posible.

c) ¿Es su grafo bipartito? Justifique su respuesta.

**15.** Sea *G=(V,E)* un grafo simple y conexo. Considere el siguiente algoritmo para obtener una buena coloración de *G* (una buena coloración de G es una coloración de los vértices de G donde dos vértices adyacentes tienen colores distintos. Note que el número cromático es una buena coloración con el menor número de colores posible). Los colores a utilizar serán 1, 2, 3, etc:

- Inicialmente ningún vértice está coloreado
- Para cada vértice v de G:

- Colorear v con el menor número posible que no haya sido utilizado para colorear los vértices adyacentes a v.

Muestre que el procedimiento descrito anteriormente encuentra una buena coloración de *G*, independientemente del orden en que se recorran los vértices del grafo. Proponga un invariante del proceso iterativo y demuestre la correctitud del algoritmo.

**16.** Sea *G* un grafo cuyo número cromático $\gamma(G)$ es 3. Diga si es posible que el mismo grafo tenga número de dominancia $\beta(G)$ igual a 1. Justifique su respuesta.

**17.** Sea *G=(V,E)* un grafo y *C* una clique máxima de *G*. Dé una cota inferior de la cardinalidad de una coloración mínima de G en función del orden de *C*.

**18.** Dado el grafo en la figura II.21. Determine su número cromático, su número de estabilidad y dé una coloración a éste.

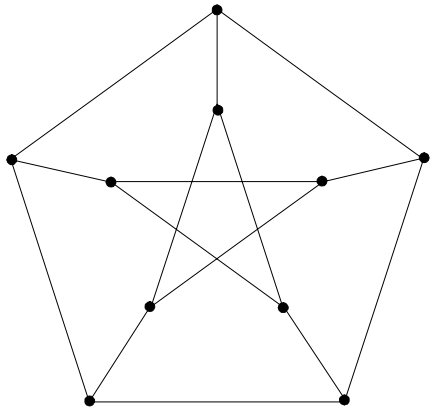


**figura II.21**

**19.** Dé un argumento contundente de por qué los grafos en la figura II.22 no son isomorfos.

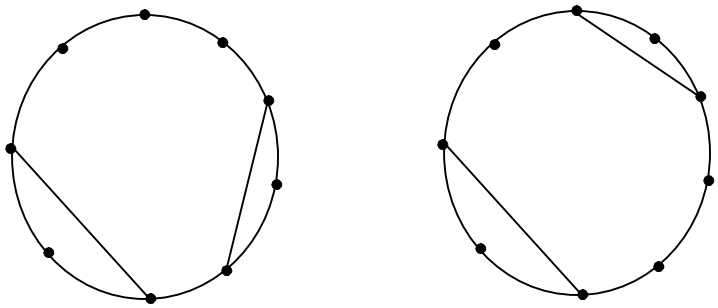


**figura II.22**

**CAPÍTULO III**

# *CONECTIVIDAD EN GRAFOS*

## INTRODUCCIÓN

Una de las aplicaciones del modelo de grafos es la determinación de trayectos o recorridos en una red de transporte, distribución de productos ó itinerarios de viaje. Por ejemplo, el problema del agente viajero se formula como sigue: se desea visitar un conjunto de ciudades sin pasar dos veces por la misma ciudad y recorriendo el menor número de kilómetros posible. Podemos modelar la región mediante un grafo donde los vértices son las ciudades y los lados las vías de comunicación entre ellas. A cada lado asociamos el número de kilómetros que corresponde a la vía. El problema se puede solucionar en términos del grafo, hallando un "recorrido" de los vértices a través de los lados, de forma tal que la suma de los kilómetros asociados a los lados utilizados sea lo más pequeña posible. Este problema y muchos otros pueden plantearse formalmente en términos de las definiciones de cadena, camino, ciclos, circuitos y conectividad, que introduciremos en este capítulo.

## III.1 CADENAS, CAMINOS, CICLOS Y CIRCUITOS

### III.1.1 DEFINICIONES

Sea G=(V,E) un grafo. Una *cadena* C es una secuencia alternada de vértices y lados, C = $<x_0,e_1,x_1,e_2, ... x_{k-1},e_k,x_k>$, tal que para $1\le i\le k$, los extremos de $e_i$ son $x_{i-1}$ y $x_i$. Decimos que C es una *cadena de largo o longitud k* de $x_0$ a $x_k$. Dos cadenas $<v_0,e_1,v_1,e_2,v_2, ... e_k,v_k >$ y $<w_0,f_1,w_1, ... ,f_l,w_l>$ son iguales si k=l, $v_i=w_i$, $e_i=f_i$, $\forall i$, $0\le i\le k$. Si G es orientado y los $e_i$ son arcos con nodo inicial $x_{i-1}$ y nodo terminal $x_i$, $1\le i\le k$, entonces seremos más precisos y diremos que C es *un camino* de $x_0$ a $x_k$ de largo k. Cuando k es positivo y $x_o = x_k$, decimos que C es una *cadena cerrada*. Si los lados $e_1,e_2, ... ,e_k$ son distintos entre sí, se dice que C es una *cadena simple*. Si los vértices $x_0, ... ,x_k$ son distintos entre sí, C es una *cadena elemental* (note que $<x_0>$ es una cadena elemental). Una *cadena pasa por* un vértice o un lado si estos se encuentran en la cadena. Un *ciclo* de G es una cadena cerrada y simple. Cuando G es orientado, un camino cerrado y simple se llama *circuito*. Decimos que un *ciclo es elemental* si los vértices $x_0,x_1, ... ,x_{k-1}$ son distintos entre sí. En particular el largo de un ciclo elemental es igual al número de sus vértices. En la figura III.1 se ilustran estas definiciones:

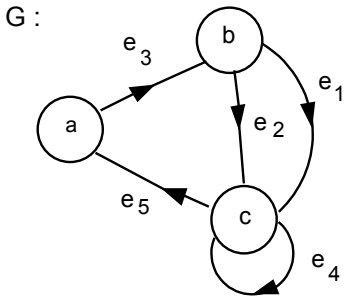



**figura III.1**

- $<b,e_1,c,e_5,a,e_5,c,e_4,c>$ es una cadena de b a c. Si consideramos las orientaciones de los lados vemos que la secuencia resultante no es un camino.

- $<b,e_2,c,e_4,c>$ es un camino de b a c.

- $<b,e_1,c,e_4,c,e_5,a>$ es una cadena simple de b a a. Si consideramos las orientaciones de los lados que intervienen en esta cadena vemos que ésta es también un camino simple de b hasta a.

- $<b,e_1,c,e_5,a>$ es una cadena elemental y considerando la orientación es un camino elemental.

- $<b,e_1,c,e_4,c,e_2,b>$ es un ciclo de G. Si consideramos la orientación de los lados, no es un circuito.

- $<b,e_2,c,e_5,a,e_3,b>$ es un ciclo elemental de G y, considerando la orientación, es un circuito elemental.

**OBSERVACIONES:**

- Las definiciones de cadena y ciclo son válidas en grafos orientados pues en la secuencia de vértices y arcos, lo que importa es que cada arco sea incidente en los vértices anterior y posterior.

- Si el grafo no posee lados múltiples, una cadena o camino está perfectamente determinada por la secuencia de vértices. Así $<x_0,e_1,x_1, \ldots ,e_n,x_n>$ se puede representar por $<x_0,x_1, \ldots ,x_n>$.

- Si n y m son respectivamente el número de vértices y aristas del grafo G=(V,E) entonces:

  . Toda cadena simple es de largo $\leq$m.

  . Toda cadena elemental es de largo $\leq$n-1.

  . Toda cadena (camino) elemental de largo n-1 se llama *cadena (camino) hamiltoniana*: Ella pasa una y sólo una vez por cada vértice del grafo.

  . Toda cadena (camino) simple de largo m se llama *cadena (camino) Euleriana (Euleriano)*

- Todo ciclo es de largo $\leq$m. Un ciclo (circuito) de largo m se llama *ciclo (circuito) Euleriano*.

- Todo ciclo elemental es de largo $\leq$n. Un ciclo (circuito) elemental de largo n se denomina *ciclo (circuito) hamiltoniano*.

### III.1.2 ALGEBRA DE CADENAS Y CAMINOS

Sobre el conjunto de las cadenas de un grafo podemos definir un álgebra lo cual nos permite hacer un tratamiento formal, en este caso el tratamiento algebraico, de la manipulación de cadenas.

Sea G un grafo y $C$ (G) el conjunto de las cadenas de G. Por convención podemos definir dos elementos en $C$, los cuales son la cadena vacía (secuencia vacía) y un elemento distinguido que denotaremos $\infty$ y que simboliza un resultado indefinido de una operación entre dos cadenas. Entre las operaciones que podemos definir sobre $C$ están:

**(1) Largo o Longitud de una cadena:**

$$l : C(G) \rightarrow \aleph$$

$l$(C) es el *largo de la cadena* C, como ya se definió.

**(2) Concatenación de cadenas:**

$$\| : C(G) \times C(G) \rightarrow C(G)$$

Si $C_1$ y $C_2$ son dos cadenas entonces $C_1 \| C_2$ es la cadena obtenida como sigue:

Si $C_1$ ó $C_2$ son iguales a $\infty$ entonces $C_1 \,||\, C_2$ será igual a $\infty$. Si el vértice final de $C_1$ coincide con el inicial de $C_2$ entonces $C_1 \,||\, C_2$ es la secuencia que resulta de concatenar la secuencia $C_1$ con la secuencia $C_2$ identificando con un solo elemento al vértice final de $C_1$ e inicial de $C_2$ si el índice final de $C_1$ no coincide con el vértice inicial de $C_2$ entonces $C_1 \,||\, C_2$ es $\infty$. Si C1 o $C_2$ son vacías entonces $C_1 \,||\, C_2$ es igual a $C_1$ si $C_2$ es vacía o igual a $C_2$ si $C_1$ es vacía. Vemos que si $C_1 \,||\, C_2 \neq \infty$ entonces $l\,(C_1 \,||\, C_2) = l\,(C_1) + l\,(C_2)$. La concatenación es asociativa por lo tanto podemos escribir la concatenación de más de dos cadenas sin paréntesis: $C_1 \,||\, C_2 \,||\, C_3 \,||\, .... \,||\, C_n$. Note que si $C_1 = <v_0,e_1,v_1, \ldots ,e_n,v_n >$ y $C_2 = <v_n >$ entonces $C_1 \,||\, C_2 = C_1$.

Ejemplo: si $C_1 = <v_0,e_1,v_1 >$ y $C_2 = <v_1,e_2,v_2 >$ entonces $C_1 \,||\, C_2 = <v_0,e_1,v_1,e_2,v_2 >$

**(3) Cadena inversa (Inv):**

Inv: $C\,(G) \rightarrow C\,(G)$

Dada $C = <x_0,e_1,x_1, \ldots ,e_k,x_k >$, Inv(C) es la cadena: $<x_k,e_k,x_{k-1}, \ldots ,x_1,e_1,x_0 >$. Inv(C) también es denotada por $C^{-1}$. Note que si C es un camino $C^{-1}$ no es un camino salvo cuando C sea sólo un vértice.

**(4) Subgrafo asociado a una cadena (Sub):**

Sub: $C\,(G) \rightarrow S\,(G)$

$S$ representa el conjunto de todos los subgrafos de G.

Si C es vacía entonces Sub(C) es el subgrafo vacío (aquél con conjunto de vértices vacío). Si C es $\infty$ entonces Sub(C) no está definido. En otro caso, Sub(C) es el subgrafo cuyos vértices y lados son los que aparecen en C. Podemos observar que C y $C^{-1}$ tienen el mismo subgrafo asociado.

**(5) Expandir una cadena:**

Sea G un grafo sin lados múltiples. La expansión de una cadena al sucesor v se puede expresar por la siguiente función:

Exp: $C\,(G) \times V \rightarrow C\,(G)$

Si $C = <v_0,e_1,v_1, \ldots,e_n,v_n >$ y $v \in V$

$$\text{Exp}(C,v) = \begin{cases} <v_0,e_1,v_1, \ldots, v_n, e_{n+1}, v> & \text{si v es adyacente a } v_n \text{ y } e_{n+1}=\{v, v_n\} \\ \infty \text{ en otro caso} & \end{cases}$$

**(6) La *distancia* de un vértice v a un vértice w:**

La distancia de v a w, se denota por d(v,w), y es igual a la longitud más pequeña de una cadena de v a w. Este valor será indefinido en caso de que no exista cadena entre v y w. Si d es la distancia de v a w diremos que w está a distancia d de v. Note que esta operación es conmutativa.

Por lo tanto $A(G) = (C\,(G), S\,(G), \aleph, \,||\,, \text{Exp}, l, \text{Sub}, \text{Inv})$ es un álgebra heterogénea. Para grafos orientados podemos definir un álgebra de caminos similar a la de cadenas, definiendo una operación análoga por cada operación del álgebra de cadenas, salvo la operación "Inv", que no tiene sentido entre caminos. La función "distancia" no sería conmutativa en este caso.

### III.1.3 PROPIEDADES

A continuación presentaremos una serie de propiedades fundamentales de cadenas, ciclos, caminos y circuitos. Las demostraciones de las propiedades ilustran diferentes técnicas de demostración en teoría de grafos.

**Proposición III.1.3.1**

Si C es una cadena de x a y entonces existe una cadena elemental de x a y.

**Primera demostración (Demostración por argumento existencial):**

En el conjunto de las cadenas de x a y (este conjunto no es vacío) tomamos una cadena C* de largo mínimo ($0 \le l(C^*) \le l(C)$ ). Podemos afirmar que esta cadena es elemental pues si no lo fuera existiría un vértice que se repite:

$C^* = <x_0,e_1,x_1, \ldots ,e_j,x_j, \ldots ,e'_j,x_j, \ldots ,e_k,x_k >$, $x_0=x$, $x_k=y$

Por lo tanto $C^*=C_1 \,||\, C_2 \,||\, C_3$ donde $C_1 = <x_0,e_1,\ldots,e_j,x_j >$, $C_2 = <x_j,\ldots,e'_j,x_j >$ y $C_3 = <x_j,\ldots,e_k,x_k >$, con $l(C_2) \ge 1$. Vemos que C1 || C3 es una cadena de x a y de menor longitud que C*, lo cual contradice la escogencia de C*.

**Segunda demostración (Demostración constructiva):**

El procedimiento siguiente nos permite construir una cadena elemental de x a y:

(1) Si C es elemental entonces C es la cadena buscada, parar.

(2) Si C no es elemental entonces se repite un vértice (por lo menos): $C = <x_0,e_1,x_1, \ldots ,e_j,x_j, \ldots ,e'_j,x_j, \ldots ,e_k,x_k > = C_1 \,||\, C_2 \,||\, C_3$, con $x_0=x$, $x_k=y$, $C_1 = <x_0,e_1,x_1, \ldots ,e_j,x_j >$, $C_2 = <x_j, \ldots ,e'_j,x_j >$, $C_3 = <x_j, \ldots ,e_k,x_k >$ y $l(C_2) \ge 1$. Sea $C' = C_1 \,||\, C_3$. El largo $C'$ es menor estricto que el largo de C y $C'$ es una cadena de x a y.

(3) Ir al paso (1) considerando que C es igual a $C'$.

El procedimiento anterior termina cualquiera sea el valor inicial de C pues si $C=C^0,C^1,C^2, \ldots$ es la secuencia de cadenas consideradas en el procedimiento anterior, el paso (2) nos garantiza que $l(C^0) > l(C^1) > l(C^2) > \ldots \ge 0$. Al ser esta secuencia estrictamente decreciente y acotada por cero, ésta debe ser finita. La última cadena de la secuencia es elemental de x a y pues tuvo que satisfacer la condición de parada del procedimiento.

**Tercera demostración (demostración inductiva):**

Aplicamos inducción sobre el número que resulta de sumar, sobre el conjunto de los vértices que forman C, el número de veces que se repite el vértice en C:

$$r(C) = \sum_{v \in C} \text{número de veces que se repite v en C}$$

Si r(C)=0 entonces C es elemental.

Supongamos cierta la proposición III.1.3.1 para los C con $r(C) \le p$.

Demostremos que si r(C)=p+1 entonces C cumple con la proposición:

Como $r(C) = p+1 \ge 1$, tenemos que C posee un vértice que se repite, $C = <x_0,e_1,x_1, \ldots ,e_j,x_j, \ldots ,e'_j,x_j, \ldots ,e_k,x_k >= C_1 \,||\, C_2 \,||\, C_3$, con $x_0=x$, $x_k=y$, $C_1 = <xo,e_1,x_1, \ldots ,e_j,x_j >$, $C_2 = <x_j, \ldots ,e'_j,x_j >$, $C_3 = <x_j, \ldots ,e_k,x_k >$. Por lo tanto $C_1 \,||\, C_3$ es un camino de x a y, con $r(C_1 \,||\, C_3) < r(C)$. Aplicando la hipótesis de inducción, la proposición III.1.3.1. se cumple para $C_1 \,||\, C_3$ y por lo tanto existe una cadena elemental de x a y.

¤

**Proposición III.1.3.2**

Si C es un camino de x a y entonces existe un camino elemental de x a y.

**Demostración:** Los razonamientos son los mismos utilizados en las tres demostraciones de la proposición III.1.3.1.. Basta con reemplazar la palabra cadena por camino.

¤

Diremos que una cadena C' ha sido *extraída de* una cadena C si C' es una concatenación de subsecuencias consecutivas de C. La relación "extraída de" es una relación de orden sobre el conjunto de las cadenas de x a y (supuesto no vacío). Los elementos minimales de esta relación son las cadenas elementales de x a y. Para x=y existe menor elemento y es <x>.

**Proposición III.1.3.3**

Si C es una cadena de x a y entonces existe una cadena elemental C' de x a y que puede ser extraída de C.

**Demostración:**

La segunda y la tercera demostración de la proposición III.1.3.1. nos indican cómo extraer una cadena elemental de C.

¤

**Proposición III.1.3.4**

Si C es un ciclo que contiene el lado e, entonces podemos extraer de C un ciclo elemental que pasa por e.

**Demostración:**

Sea $C= <x_0,e_1,x_1, \ldots ,e_k,x_0 >$. Sin pérdida de generalidad podemos suponer que $e=e_1$. Si e es un lazo, el resultado es evidente. Según la proposición III.1.3.3., se puede extraer una cadena elemental $C'_1$ de $x_1$ a $x_0$ de la cadena $C_1= <x_1,e_2, \ldots ,e_k,x_0 >$. $C'_1$ no contiene a e pues $C_1$ no lo contiene. Por lo tanto $<x_0,e,x_1 > \| C'_1$ es un ciclo elemental extraído de C y que contiene a e.

¤

Suponer que $e_1=e$ en la demostración anterior no implica una pérdida de generalidad en la demostración; basta con construir, a partir del ciclo original que contiene a e, un ciclo donde la primera arista que aparece es e. Por otro lado, si reemplazáramos la hipótesis de que C es un ciclo por C es una cadena cerrada, entonces la proposición no sería verdadera pues el hecho de que C sea simple es imprescindible. Sin embargo, en el caso orientado sí podemos relajar más las hipótesis:

**Proposición III.1.3.5**

Si C es un camino cerrado que contiene el arco e, entonces podemos extraer un circuito elemental que pasa por e.

**Demostración:**

El razonamiento es similar al seguido en la demostración de la proposición III.1.3.4.. Basta sustituir los términos para grafos no orientados por sus correspondientes en grafos orientados. Lo único que se debe mostrar en forma diferente es que $C'_1$ no contiene a e pues éste se puede repetir en C. Si $C'_1$ contuviese a e, esto contradiría el hecho de que $C'_1$ es un camino elemental. Por lo tanto $<x_0,e,x_1 > \| C'_1$ es un circuito elemental extraído de C y que contiene a e.

¤

**Proposición III.1.3.6**

Sea G=(V,E) un grafo no dirigido y C una cadena simple de v a w, $v,w \in V$.
Entonces:
En Sub(C)=(V',E') todo vértice distinto de v y w tiene grado par.
Si v es distinto de w entonces v y w tienen grado impar. Si v es igual a w entonces el grado de v (=w) es par.

**Demostración:**

Sea $C= <v_0,e_1,v_1, \ldots ,e_n,v_n >$, $v_0=v, v_n=w$. Mostremos por inducción sobre n:
Para n=0 ó 1 la proposición es verdadera pues no hay vértices intermedios y si n=0 Sub(C) es un vértice aislado sin bucles y d(v)=d(w). Si n=1 Sub(C) es o un bucle o una arista.
Para n=2

$C=<v,e_1,v_1,e_2,w>$

Si $v_1=v=w$ entonces $d(v_1)=d(v)=d(w)=4$. Si $v_1=v$ y $v\neq w$ entonces $d(v_1)=3$ y $d(w)=1$. Si $v_1\neq v$ y $v_1\neq w$ entonces $d(v_1)=2$. Además si $v\neq w$ $d(v)=d(w)=1$ en Sub(C). Si $v=w$, $d(v)=2$ en Sub(c).

Supongamos que para $n < k$ se cumple la proposición y mostremos que se cumple para $n=k$ $(k\geq 2)$ $C=<v_0,e_1,v_1, \ldots ,e_k,v_k>$

Tenemos que $C'= <v_0,e_1,v_1, \ldots ,e_{k-1},v_{k-1} >$ es de largo menor que k y está bien definida pues $k\geq 2$. Por lo tanto se cumple la proposición en Sub(C'). El hecho que C sea simple implica que $e_k$ es distinto de $e_i$, $1\leq i\leq k-1$. Se presentan dos casos al agregar $e_k$ y $v_k$ a Sub(C'):

(i) $v_0=v_{k-1}$

(ii) $v_0\neq v_{k-1}$

(i) $v_0=v_{k-1}$: entonces todos los vértices de Sub(C') son de grado par.

Si $v_k=v_0$ entonces $e_k$ es un bucle y así el grado de $v_0$ sigue siendo par y los demás grados no se alteran.

Si $v_k\neq v_0$ entonces el grado de $v_k$ es impar en Sub(C): si $v_k$ es un vértice de Sub(C') su grado es par y al colocar la arista $e_k$ entre $v_0$ y $v_k$, su grado se convierte en impar. Cuando $v_k$ no es un vértice de Sub(C') al agregar $e_k$ y $v_k$, el grado de $v_k$ será 1 o sea impar.

(ii) $v_0\neq v_{k-1}$:

Todos los vértices de Sub(C') tienen grado par salvo $v_0$ y $v_k$ que tienen grado impar. Si $v_k$ es igual a $v_{k-1}$, al agregar $e_k$ a Sub(C'), el grado de $v_k(=v_{k-1})$ sigue siendo impar porque $e_k$ es un bucle y los grados de los demás vértices no se alteran. Por lo tanto el grado de $v_0$ y $v_k$ es impar y el de los demás vértices es par en Sub(C).

Si $v_k\neq v_{k-1}$, entonces al agregar $e_k$ y $v_k$ a Sub(C') el grado de $v_{k-1}$ deviene par pues el grado de $v_{k-1}$ es impar en Sub(C'). Si $v_k=v_0$, entonces el grado de $v_0$ se convierte en par pues $e_k$ no es bucle.

Si $v_k\neq v_0$ el grado de $v_0$ no se altera al agregar $e_k$ y $v_k$ , el cual sigue siendo impar y si $v_k$ es un vértice de Sub(C'), como $v_k\neq v_{k-1}$ y $v_k\neq v_0$,el grado de $v_k$ en Sub(C') es par y al agregar $e_k$ se convierte en impar. Si $v_k$ no es vértice de Sub(C') al agregar $e_k$, $v_k$ a Sub(C') el grado de $v_k$ será 1, impar.

¤

**Proposición III.1.3.7**

Sean $C_1$ y $C_2$ dos cadenas simples de v* a w* en un grafo G=(V,E). Sean $Sub(C_1)=(V_1,E_1)$ y $Sub(C_2)=(V_2,E_2)$ los grafos correspondientes a las cadenas $C_1$ y $C_2$. Sea H=(V',E') con $V'=V_1\cup V_2$ y $E'=(E_1-E_2)\cup(E_2-E_1)$. Entonces $\forall v\in V'$, el grado de v es par en H.

**Demostración:**

Sea $v\in V'$ y definamos los conjuntos: $A_v$, $B_v$ y $C_v$ como los conjuntos de lados en E', $E_1$ y $E_2$, respectivamente, incidentes en v y que no sean bucles.

Tenemos que:

$A_v=(B_v-C_v)\cup(C_v-B_v)$

Es un buen ejercicio mostrar la igualdad anterior por propiedades de teoría de conjuntos.

Por lo tanto $|A_v|=|(B_v-C_v)\cup(C_v-B_v)|=|B_v-C_v|+|C_v-B_v|=|B_v|+|C_v|-2|B_v\cap C_v|$

Por lo tanto:

Como el grado de v tiene la misma paridad en $Sub(C_1)$ y $Sub(C_2)$ (ver proposición III.1.3.6.) tenemos que $|A_v|$ es par. Por otro lado, el grado de v en H es $|A_v|$ más dos veces el número de bucles incidentes en v de $(E_1-E_2)\cup(E_2-E_1)$, con lo cual concluimos que el grado de todo vértice es par.

¤

**Proposición III.1.3.8**

Sean $C_1$ y $C_2$ dos cadenas simples de v* a w* en G=(V,E). Si $C_1 \neq C_2$ entonces G posee un ciclo.

**Demostración:**

Sean $C_1=<v_0,e_1,v_1, \ldots ,e_k,v_k>$ y $C_2=<w_0,f_1,w_1, \ldots ,f_l,w_l>$ con $v_0=w_0=v^*$ y $v_k=w_l=w^*$.

(a) Si alguna de las dos cadenas repite un vértice entonces podemos construir un ciclo a partir de esa cadena. Supongamos que existen $v_p$ y $v_q$ en $C_1$ tales que $p<q$ y $v_p=v_q$ entonces: $<v_p,e_{p+1},v_{p+1}, \ldots ,e_q,v_q>$ es una cadena cerrada y simple, o sea, un ciclo.

(b) Supongamos de ahora en adelante que ninguna de las dos cadenas tiene vértices repetidos. Por lo tanto las dos cadenas $C_1$ y $C_2$ son elementales.

Sea i el menor número natural tal que $e_{i+1}\neq f_{i+1}$. El número i está bien definido pues las cadenas son distintas y no repiten vértices (constate este hecho), por lo tanto $0\leq i<\min\{k,l\}$. Consideremos las cadenas:

$C'_1=<v_i,e_{i+1},v_{i+1}, \ldots ,e_k,v_k=w^*>$

$C'_2=<v_i,f_{i+1},w_{i+1}, \ldots ,f_l,w_l=w^*>$

Sea p el menor natural, mayor que i, tal que $v_p$ aparezca en la cadena $C'_2$. Sabemos que p está bien definido pues $v_k=w^*$ está en $C'_2$ y $k\geq i+1$. Sea $w_q$ el vértice en $C'_2$ igual a $v_p$ (hay uno sólo pues $C'_2$ es elemental). Definimos ahora las cadenas siguientes:

$C''_1=<v_i,e_{i+1},v_{i+1}, \ldots ,e_p,v_p>$

$C''_2=<v_i,f_{i+1},w_{i+1}, \ldots ,f_q,w_q=v_p>$

Podemos afirmar que $C_3=C''_1 \,||\, Inv(C''2)$ es un ciclo elemental de G. En efecto:$v_{i+1},v_{i+2}, \ldots ,v_p$ son distintos entre sí; $w_{i+1},w_{i+2}, \ldots ,w_q(=v_p)$ son distintos entre sí, además $v_j$ con $i+1\leq j<p$ no pueden ser iguales a ningún vértice en $w_{i+1},w_{i+2}, \ldots ,v_p$ pues si así fuera entonces j sería incompatible con la definición de p. Por lo tanto: $v_{i+1},v_{i+2}, \ldots , v_p,w_{q-1},w_{q-2}, \ldots ,w_{i+2},w_{i+1},v_i$ son distintos entre sí y $C_3$ es un ciclo elemental.

¤

**Proposición III.1.3.9**

Si G es un grafo tal que el grado mínimo de G, $\delta(G)$, es mayor o igual a 2, entonces G contiene un ciclo.

**Demostración:**

Si G contiene sólo bucles, el resultado es evidente. Si no, consideremos en G una cadena elemental de largo máximo $C=<v_0,e_1,v_1, \ldots ,e_p,v_p>$ con $p\geq 1$ y $v_0\neq v_p$. Como $\delta(G)\geq 2$, tenemos que $d(v_0)\geq 2$. Por lo tanto, existe un lado $e_0\neq e_1$ incidente en $v_0$ (ver figura III.2).

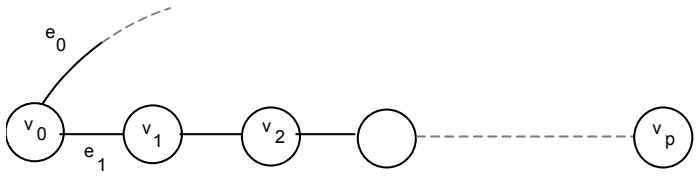



**figura III.2**

Si la otra extremidad de $e_0$, w, no pertenece a C entonces $<w,e_0,v_0> \,||\, C$ sería una cadena elemental de largo p+1, lo que es imposible. Por lo tanto, existe i ($0\leq i\leq p$) tal que $w=v_i$, y así $<v_0,e_1,v_1, \ldots ,e,v_i,e_0,v_0>$ es un ciclo elemental de G.

¤

**Corolario III.1.3.10**

Sea G un grafo con m lados y n vértices. Si $m\geq n$ entonces G contiene un ciclo (por lo tanto, si G no contiene ciclos entonces $m\leq n-1$).

**Demostración:**

Razonemos por el absurdo y supongamos que existe un grafo con m≥n y sin ciclos. Sea $G_0$ un tal grafo el cual posee el menor número de vértices entre todos los que satisfacen: m≥n y sin ciclos. Mostremos que $\delta(G_0)\geq 2$. En efecto, si $G_0$ contiene un vértice $v_0$ tal que $d(v_0)\leq 1$ entonces el grafo $H_0$ obtenido a partir de $G_0$ por eliminación de v0, no posee ciclos y tenemos:

$$|V(H_0)|=|V(G_0)|-1 \quad y \quad |E(G_0)|-1\leq |E(H_0)|\leq |E(G_0)|$$

Sabiendo que $|E(G_0)|\geq|V(G_0)|$, se deduce que:

$$|E(H_0)|\geq|V(H_0)|$$

Conseguimos $H_0$ con menos vértices que $G_0$ y satisfaciendo las mismas condiciones que $G_0$, esto es contradictorio con la escogencia que se hizo de $G_0$. Ahora, según la proposición III.1.3.9., $\delta(G_0)\geq 2$ es incompatible con $G_0$ sin ciclos.

¤

Note que la proposición III.1.3.8 resultaría un corolario de la proposición III.1.3.9 en el caso de que las cadenas $C_1$ y $C_2$ no poseyeran los mismos lados. En este caso el grafo H, definido en la proposición III.1.3.7, poseería al menos un vértice de grado ≥2 y, eliminando los vértices de grado cero, podríamos aplicar la proposición III.1.3.9 al grafo resultante pues su grafo mínimo sería ≥2 y concluir que existe un ciclo.

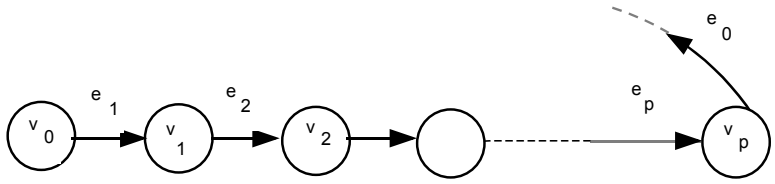



**figura III.3**

**Proposición III.1.3.11**

Si G es un digrafo tal que $\delta^+(G)\geq 1$ ($\delta^-(G)\geq 1$) entonces G posee un circuito.

**Demostración:**

Esta demostración es similar a la de la proposición III.1.3.9. Si G contiene sólo lazos, el resultado es evidente. Si no, sea $C= <v_0,e_1,v_1, ...,e_p,v_p >$ un camino elemental de largo máximo en G, se tiene p≥1 y $v_0\neq v_p$ (Ver figura III.3).

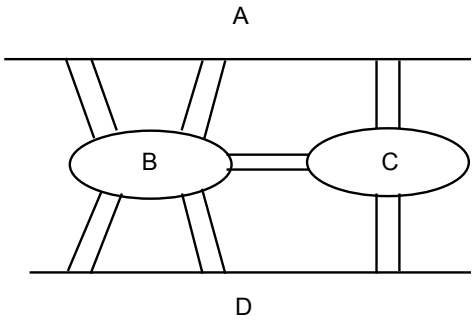
Al ser $\delta^+(G)\geq 1$, tenemos que $d^+(v_p)\geq 1$. Por lo tanto, existe un arco $e_0$ con extremo inicial $v_p$. La otra extremidad de $e_0$,digamos w, debe estar en C. De lo contrario no sería C un camino más largo. Por lo tanto existe i, $0\leq i\leq p$, tal que $w=v_i$ ,y así $C= <v_i,e_{i+1},v_{i+1}, ...,e_p,v_p,e_0,v_i >$ es un circuito elemental de G.

Note que si un digrafo G posee $\delta^-(G)\geq 1$, también contendrá un circuito pues basta aplicar la proposición anterior al grafo $G^{-1}$ obtenido a partir de G invirtiendo sus arcos. Sabemos que $\delta^-(G)=\delta^+(G^{-1})$.

### III.1.4. CICLOS Y CADENAS EULERIANAS

Uno de los primeros problemas que fueron modelados usando grafos fue el que confrontó Leonard Euler (1736). En la ciudad de Kaliningrado (antigua Königsberg) había siete puentes sobre el río Pregel. Uno de los puentes conectaba dos islas entre sí. Una de las islas estaba conectada a una ribera por dos puentes y otros dos

puentes la conectaban con la otra costa. La otra isla poseía un puente hacia cada ribera. Euler se preguntó si sería posible comenzar un paseo desde cualquier punto y atravesar cada puente una y sólo una vez, regresando al punto de partida (ver la figura III.4).




**Puentes de Konigsberg**

**figura III.4**

Estudiando el problema, Euler descubre que no tiene solución, utilizando el modelo de la figura III.5, donde los vértices representan los puntos de tierra y las aristas representan los puentes.

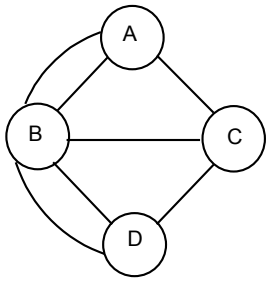



**figura III.5**

**Definición:**

*Un ciclo* (una cadena simple) de un grafo G es *Euleriano* (Euleriana) si en la secuencia que constituye el ciclo (la cadena) aparecen todas las aristas del grafo.

La proposición III.1.3.6. implica directamente el siguiente resultado:

**Proposición III.1.4.1**

Sea G un grafo no orientado,

(a) Si G posee un ciclo Euleriano el grado de todo vértice de G es par.

(b) Si G posee una cadena Euleriana de v a w ($v \neq w$) entonces todo vértice de G distinto de v y w posee grado par y v,w poseen grado impar.

¤

La proposición anterior da la respuesta al problema de Euler pues, en el grafo de la figura III.5., existen vértices de grado impar y por lo tanto no pueden existir ciclos Eulerianos, es decir, no se puede recorrer los puentes una y sólo una vez regresando al punto de partida.

Decimos que un grafo es Euleriano si contiene un ciclo Euleriano. Mostraremos ahora, mediante el método constructivo (un algoritmo), que las condiciones necesarias, dadas en la proposición III.1.4.1., para que exista un

ciclo y una cadena Euleriana son también suficientes en caso de que el grafo posea cadenas entre cada par de vértices.

**Proposición III.1.4.2**

Dados dos ciclos sin aristas comunes:
$C' = <x_0,e_1,x_1, ...,e_n,x_n=x_0 >$
$C'' = <x_0,e'_1,x'_1, ...,e'_m,x_m=x_0 >$

con al menos un vértice común $x_0$, podemos fusionar estos dos ciclos en uno solo $C=C' \,||\, C''$.

**Demostración:**

Que C es un ciclo es una consecuencia directa de la definición de ciclo y que C' y C'' son disjuntos.

¤

Utilizaremos esta proposición para determinar un algoritmo que construye un ciclo Euleriano en un grafo donde necesariamente todos los vértices tienen grado par. De esta forma mostraremos la suficiencia de la proposición III.1.4.1.(a).

Sea G un grafo no orientado donde cada vértice tiene grado par y existe una cadena entre cada par de vértices.

El algoritmo construye el ciclo Euleriano de manera progresiva mediante la concatenación sucesiva de ciclos. Supongamos que $C=C^1$ es un ciclo de G. Si eliminamos de G las aristas que están en el ciclo y luego eliminamos los vértices aislados (vértices de grado cero), en el grafo $G^1$ resultante todo vértice es de grado par ya que el número de aristas en $C^1$ incidentes en un vértice cualquiera de $C^1$ es par. Debe existir un vértice $v^1$ en C que se encuentra en $G^1$, de no ser así no existiría un camino en G entre los vértices de C y los de $G^1$, lo que contradiría la hipótesis. Buscamos otro ciclo $C^2$ en $G^1$ que contenga a $v^1$. Concatenamos $C^1$ y $C^2$ obteniendo un ciclo C. Eliminamos de $G^1$ las aristas de $C^2$, luego eliminamos los vértices aislados obteniendo así un grafo $G^2$. El grafo $G^2$ de nuevo posee todos sus vértices de grado par. Debe existir $v^2$ en C, que se encuentre en $G^2$ pues existen caminos entre los vértices de C y $G^2$. Buscamos un ciclo $C^3$ que contenga a $v^2$, lo concatenamos con C y continuamos el proceso hasta que $G^i$ obtenido de eliminar $C^i$ de $G^{i-1}$ posea sólo vértices aislados. El ciclo producto de todas las concatenaciones será un ciclo Euleriano de G.

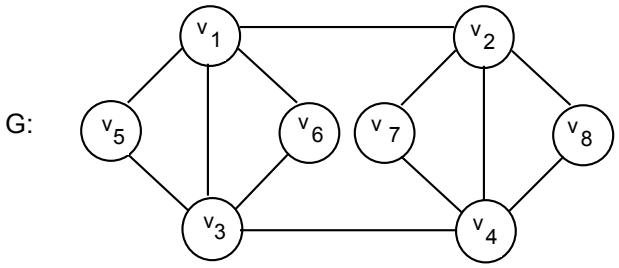



**figura III.6**

La construcción del ciclo $C^i$ en cada iteración del proceso anterior se puede garantizar pues el grado de cada vértice del grafo $G^{i-1}$ es par y mayor que cero. Para construir el ciclo $C^{i+1}$ partimos de un vértice $v^i$ de $G^i$ y construimos la secuencia:

$<v_i,\{v_i,v'_i\}$, vecino de $v_i$ $(=v'_i),\{v'_i,v''_i\}$, vecino de $v'_i(=v''_i)$, etc. >

Siempre que elijamos un vértice podemos escoger un vecino tal que la arista que los une no haya sido recorrida anteriormente pues el grado de cada vértice es par. Como el grafo $G^i$ tiene un número finito de aristas, este recorrido nos llevará de nuevo al vértice $v_i$, obteniendo así un ciclo $C^{i+1}$ de $G^i$.

El algoritmo siempre termina ya que, cada vez que determinamos un ciclo $C^i$ en $G^{i-1}$, el grafo $G^i$ resultante de la eliminación de las aristas de $C^i$ y de los vértices aislados en $G^{i-1}$ posee menos aristas que $G^{i-1}$.

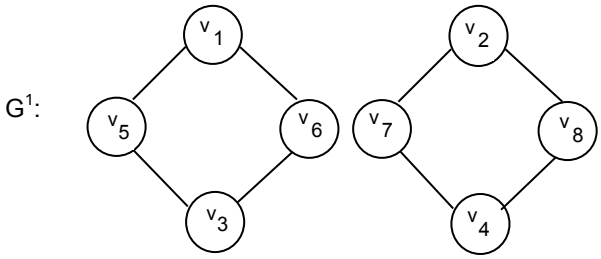



**figura III.7**

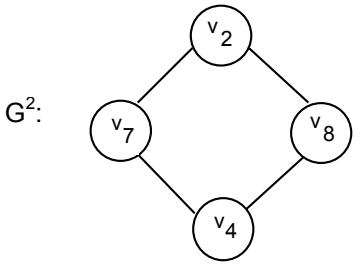



**figura III.8**

Ejemplo:

Sea G el grafo de la figura III.6. Sea $C^1=<v_1,\{v_1,v_2\},v_2,\{v_2,v_4\},v_4,\{v_4,v_3\},v_3,\{v_3,v_1\},v_1>$.

Eliminamos $C^1$ de G (figura III.7).
Cualquier vértice de $C^1$ está en $G^1$.
Buscamos un ciclo $C^2$ en $G^1$ que tenga un vértice en común con $C^1$:
$C^2=<v_1,\{v_1,v_6\},v_6,\{v_6,v_3\},v_3,\{v_3,v_5\},v_5,\{v_5,v_1\},v_1>$

Concatenamos $C^1$ y $C^2$: $C=C^1 \,||\, C^2$
Eliminamos $C^2$ de $G^1$ y los vértices aislados (figura III.8).
Buscamos un ciclo $C^3$ en $G^2$ que tenga un vértice común con C:
$C^3=<v_2,\{v_2,v_8\},v_8,\{v_8,v_4\},v_4,\{v_4,v_7\},v_7,\{v_7,v_2\},v_2>$

Reordenamos C para poderlo concatenar con $C^3$:
$C=<v_2,v_4,v_3,v_1,v_6,v_3,v_5,v_1,v_2>$ (eliminamos las aristas para simplificar la notación).

El nuevo sería :
$C=<v_2,v_4,v_3,v_1,v_6,v_3,v_5,v_1,v_2,v_8,v_4,v_7,v_2>$

Eliminamos $C^3$ de $G^2$. En este caso sólo quedan vértices aislados.

La siguiente implementación del algoritmo va coloreando los vértices del grafo. El vértice por el que se comienza a construir el ciclo se colorea de amarillo. La existencia de un vértice amarillo significa que se puede construir a partir de él un ciclo. Los vértices que pertenecen a un ciclo y que quedarían aislados al eliminar del grafo las aristas de ese ciclo se colorean de rojo. De esta forma, cuando se cierra un ciclo, se parte de un vértice amarillo para construir el ciclo siguiente.

Algoritmo de construcción de un ciclo Euleriano:
{Entrada:
- Un grafo G no orientado con todo vértice de grado par y una cadena entre cada par de vértices.
Salida:
- Un ciclo Euleriano C de G. }
Variables C, C':ciclo; v,w:vértice;

Comienzo
- C ← ∅;
- Inicialmente ningún vértice está coloreado;
- Colorear un vértice cualquiera de amarillo;
- Mientras exista un vértice v amarillo hacer:
Comienzo
- Si C ≠ ∅ entonces reordene C para que comience por v;
- C' ← ∅;
- Mientras d(v)≠0 hacer:
Comienzo
- w ←vecino de v;
- Elimine {v,w};
- C'←C' || <v,{v,w},w>;
- Si d(w) ≤1 entonces colorear w con rojo
si no colorear w con amarillo;
- v ← w;
{ d(v) es impar salvo si v es el primer vértice con el
que se entró al Mientras }
fin
{ v es el mismo que antes de comenzar el Mientras y está coloreado de rojo }
- C←C ||.C';
fin
fin.

En el algoritmo anterior, si el grafo es representado por una lista de adyacencias, el número de operaciones elementales efectuadas en el primer bloque Mientras es $O$ (|E(G)|), siempre y cuando el proceso de reordenamiento de C para que comience en v sea $O$ (1).

Como consecuencia del algoritmo anterior podemos asegurar la proposición siguiente:

**Proposición III.1.4.3**

Sea G un grafo no orientado en el que existe una cadena entre cada par de vértices.

Entonces,

(a) G es Euleriano si y sólo si todo vértice es de grado par.

(b) G posee una cadena Euleriana entre dos vértices v y w (v≠w) si y sólo si d(v) y d(w) son impares y todos los demás vértices son de grado par.

**Demostración:**

(a) Es inmediato según el algoritmo y la proposición III.1.4.1(a).

(b) Para conseguir una cadena Euleriana de v a w, basta con considerar dos casos:
(b.1) $\{v,w\} \in E$
(b.2) $\{v,w\} \notin E$

En el primer caso eliminamos la arista {v,w} de E y aplicamos el algoritmo para hallar un ciclo Euleriano C=<v,...,v> en G'=(V,E-{v,w}). La cadena Euleriana de G sería C || <v,{v,w},w}.

En el otro caso agregamos la arista {v,w} a G y determinamos mediante el algoritmo un ciclo Euleriano C=<v,..,w,{w,v},v> de G"=(V,E∪{{v,w}}). La cadena Euleriana de G es la primera parte de la secuencia que define a C, sin incluir {w,v},v.

¤

## III.2 ALCANCE

### III.2.1 DEFINICIÓN

Hablar de recorridos en grafos lleva implícito y de manera natural, el orden en el cual deben ser "visitados" los vértices y los lados. Tiene sentido entonces tratar sólo con grafos orientados cuando los recorramos.

Al recorrer grafos no orientados, como en principio podemos recorrer las aristas en cualquier sentido, lo que haremos es trabajar con el grafo orientado asociado a éste (ver definición en la sección II.1.). En esta sección trabajaremos con grafos dirigidos sin arcos múltiples. Las definiciones y propiedades pueden ser extendidas a grafos no orientados, considerando el grafo orientado asociado.

Sea G un grafo orientado. Decimos que un vértice w es *alcanzable* desde un vértice v en G si existe un camino de v a w en G. Un ejemplo de utilización de esta definición lo vemos en el siguiente problema: Se desea asignar el flechado a las calles de una urbanización de tal forma que se pueda ir de cualquier punto a otro en automóvil sin tener que salir de la urbanización. Considerando el grafo cuyos vértices representan las encrucijadas de la urbanización y cuyas aristas sean las vías que comunican las encrucijadas, el problema se reduce a dar una orientación a este grafo de forma tal que cualquier encrucijada sea alcanzable desde cualquier otra.

Diremos que un vértice w es *descendiente* de un vértice v, si w es alcanzable desde v; también diremos que v es *ascendiente* de w.

### III.2.2 MATRIZ DE ALCANCE. ALGORITMO DE ROY-WARSHALL

Nuestro objetivo será mostrar un método que permita determinar para cada par de vértices de un grafo si son alcanzables entre sí. Sea G=(V,E) un grafo orientado. Podemos considerar a E como una relación binaria sobre V.

**Proposición III.2.2.1**

Hay un camino de largo k de v a w en G=(V,E) si y sólo si $(v,w) \in E^k$ (Donde $E^k$ es la composición de la relación E, k veces y $E^0$ la relación identidad). Además, $\forall\ k \geq 1$: $E^k \subseteq E \cup E^2 \cup ... \cup E^n \subseteq I \cup E \cup ... \cup E^{n-1}$, donde n=|V|.

**Demostración:**

Haremos la demostración de la primera parte por inducción sobre k.

(1) Si k=0 es obvio que se cumple la proposición. Si k=1 los caminos de largo 1 son los arcos.

(2) Supongamos que para $k \leq i-1$ se cumple la proposición.

(3) Mostremos que se verifica para k=i ($\geq 2$):Sea $<v,e_1,v_1,...,e_i,w>$ un camino de largo i de v a w, tenemos que $<v,e_1,v_1,...,e_{i-1},v_{i-1}>$ es un camino de largo i-1 ($\geq 1$) de v a $v_{i-1}$ y por hipótesis inductiva $(v,v_{i-1}) \in E^{i-1}$. Como $(v_{i-1},w) \in E$ tenemos que $(v,v_i) \in (E^{i-1} o E) = E^i$. Recíprocamente, si $(v,w) \in E^i$, i>1, existe u tal que $(v,u) \in E^{i-1}$ y $(u,w) \in E$. Por hipótesis inductiva existe un camino de largo i-1 de v a u y como $(u,w) \in E$, entonces existe un camino de largo i de v a w.

Para mostrar que $E^k \subseteq E \cup E^2 \cup ... \cup E^n$, vemos que si $(v,w) \in E^k$ entonces existe un camino de v a w de largo k, si $v \neq w$ existe un camino de largo $\leq n-1$ de v a w y si v=w existe un camino cerrado de largo$\leq n$ de v a w. (Proposiciones III.1.3.2 y III.1.3.5). La segunda inclusión es directa ya que un camino de largo n posee un vértice repetido, por lo tanto podemos extraer un camino elemental con los mismos extremos y menor largo.

¤

La matriz $M(E)=(m_{vw})$ *asociada a la relación binaria E sobre el conjunto V* es una matriz cuadrada de orden |V|, donde las filas y las columnas corresponden a los elementos de V y $m_{vw}=1$ si $(v,w) \in E$ y $m_{vw}=0$ si $(v,w) \notin E$. Vemos que, en el caso del grafo G=(V,E), M(E) no es más que la matriz de adyacencias de G.

**Proposición III.2.2.2**

Sea G=(V,E) un grafo dirigido sin arcos múltiples. Las propiedades siguientes son equivalentes:

- w es alcanzable desde v.
- $(v,w)\in E^*=I\cup E\cup E^2\cup \ldots$ . Donde I es la relación identidad sobre V.
- El elemento $m_{vw}$ de la matriz M(E*) es igual a 1.

**Demostración:**

Un vértice w es alcanzable desde v si existe un camino de algún largo de v a w, y de acuerdo a la proposición III.2.2.1 esto es equivalente a decir que $(v,w)\in E^k$ para algún $k\geq 0$. Esto último es equivalente a $(v,w)\in \bigcup_{k\geq 0}E^k$.

¤

Si $A=(a_{ij})$ y $B=(b_{ij})$ son dos matrices cuadradas de igual orden y booleanas entonces $A+B=(a_{ij}\ \mathbf{v}\ b_{ij})$ y $A.B=(m_{ij})=(\mathbf{V}_k(a_{ik}\wedge b_{kj}))$. Es fácil ver que $M(E^*)=M(E^0)+M(E^1)+M(E^2)+....$, y $M(E^i)=(M(E))^i$. A M(E*) se le llama *matriz de alcance* de G. Nótese que M(E*) es simétrica si G es no orientado. Si A es la matriz de adyacencias de G=(V,E), a M(E*) también la denotaremos por A*.

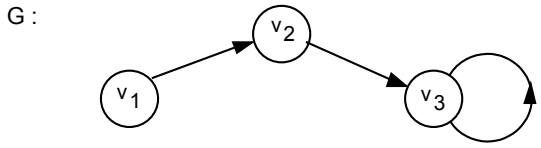



**figura III.9**

**Proposición III.2.2.3**

$(v,w)\in E^*$ si y sólo si $(v,w)\in I\cup E\cup E^2\cup\ldots\cup E^{n-1}$, donde n= |V|.

**Demostración:**

Que $(v,w)\in E^*$ significa que existe un camino de v a w. Según la proposición III.1.3.2 esto equivale a decir que existe un camino elemental de v a w si $v\neq w$ y si v=w sabemos que $(v,w)\in I$. Sea $v\neq w$: que exista un camino elemental de v a w significa que este es de largo $l\leq n-1$ y por lo tanto, según la proposición III.2.2.1, $(v,w)\in E^l$. Esto último implica que $(v,w)\in E^0\cup E^1\cup E^2\cup\ldots\cup E^{n-1}$.

¤

Por lo tanto no es necesario calcular $E^k$ para $k\geq n$, si deseamos determinar E*.

**Corolario III.2.2.1**

$M(E^*)=\sum_{0\leq i\leq n-1}M(E^i)$.

Ilustramos la matriz de alcance del grafo en la figura III.9:

$$M(E^0)=\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}\ M(E^1)=\begin{pmatrix}0&1&0\\0&0&1\\0&0&1\end{pmatrix}\ M(E^2)=\begin{pmatrix}0&0&1\\0&0&1\\0&0&1\end{pmatrix}\ M(E^*)=\begin{pmatrix}1&1&1\\0&1&1\\0&0&1\end{pmatrix}$$

M(E*) nos indica que existe un camino de $v_1$ a $v_1$, $v_2$, $v_3$, de $v_2$ a $v_2$ y $v_3$, de $v_3$ a $v_3$.

Veremos ahora un algoritmo para calcular la matriz de alcance de un grafo. Notemos que intuitivamente el orden de un tal algoritmo debe ser mayor que $n^3$ (n=|V|) pues el algoritmo debe considerar $n^2$ pares de vértices y para cada par debe determinar si existe un camino de un vértice al otro. El algoritmo que presentamos es el

**algoritmo de Roy-Warshall** cuyo orden es $n^3$. Nótese que si calculamos M(E*) a partir de la fórmula en el corolario III.2.2.1, el número de operaciones a efectuar sería del orden $n^4$ pues deberíamos calcular la sucesión de matrices siguiente. Si M es la matriz M(E):

$M_1 = M$

$M_2 = M+M^2 = (I+M)\,M = (I+M_1)\,M$

$M_3 = M+M^2+M^3 = (I+M+M^2)\,M = (I+M_2)\,M$

.

.

$M_{n-1} = M+M^2+ \ldots M^{n-1} = (I+M_{n-2})\,M$

$M(E^*) = I+M_{n-1}$

Para $i \geq 2$, el cálculo de $M_i$ necesita $n^3$ multiplicaciones y $n^2(n-1)+n$ sumas, conociendo $M_{i-1}$. Por lo tanto el cálculo de $M_{n-1}$ necesita $n^3(n-2)$ multiplicaciones y $(n-2)(n^2(n-1)+n)$ sumas. Es decir, el algoritmo es $O(n^4)$.

Podemos acelerar el cálculo de M(E*) observando que por ser las operaciones booleanas, $M(E^*) = (I+M)^{n-1}$ y además, $(I+M)^l = (I+M)^{n-1}$, $\forall l \geq n-1$ según la proposición III.2.2.3. Si q es un entero definido por :

$2^{q-1} < (n-1) \leq 2^q$

es decir,

$q = \lceil \log_2(n-1) \rceil$

considerando la secuencia:

$M_1 = (I+M)$

$M_2 = M_1^2 = (I+M)^2$

.

.

$M_i = M_{i-1}^2 = (I+M)^{2^{i-1}}$

.

.

$M_{q+1} = M_q^2 = (I+M)^{2^q}$

En consecuencia $M(E^*) = M_{q+1}$. Como el producto de dos matrices se efectúa en $O(n^3)$ operaciones, la complejidad del algoritmo anterior es $O(n^3 \log_2 n)$.

En lo que sigue presentamos varios resultados previos que permiten fundamentar el Algoritmo DE Roy-Warshall.

Sea G=(V,E) y $V=\{v_1, v_2, \ldots, v_n\}$. Definamos las relaciones binarias $E_k$ sobre V, k=0,1, ... ,n como sigue:

$(v,w) \in E_k$ si y sólo si existe un camino de v a w cuyo conjunto de vértices intermedios (los que no son extremos) es un subconjunto de $\{v_1, \ldots, v_k\}$.

**Observaciones**:

- Cuando k=0, $\{v_1, \ldots, v_k\}$ representa por convención al conjunto vacío.
- $E_0 = I \cup E$
- $E_{k-1} \subseteq E_k$
- $E_n = E^*$

En el grafo de la figura III.9 tenemos:

$E_0$= {($v_1$,$v_1$),($v_2$,$v_2$),($v_3$,$v_3$),($v_1$,$v_2$),($v_2$,$v_3$)}
$E_1$= $E_0$
$E_2$= $E_1 \cup$ {($v_1$,$v_3$)}
$E_3$=$E_2$

Las siguientes proposiciones nos dicen como calcular $E_k$ a partir de $E_{k-1}$:

**Proposición III.2.2.4**

Sea (v,w) tal que v=$v_k$ o w=$v_k$, 1≤k≤n. Entonces, (v,w)∈$E_{k-1}$ si y sólo si (v,w)∈$E_k$.

**Demostración:**

La condición necesaria es inmediata según la definición.

Supongamos v=$v_k$ y ($v_k$,w)∈$E_k$. Según la definición de $E_k$, existe un camino de $v_k$ a w cuyos vértices intermedios están en {$v_1$, ... ,$v_k$}. Podemos extraer de este camino,un camino de $v_k$ a w cuyos vértices intermedios están en {$v_1$, ... ,$v_{k-1}$}.

En el caso w=$v_k$ se sigue un razonamiento análogo al anterior.

¤

La proposición anterior nos dice que si (v,w)∈$E_k$-$E_{k-1}$ entonces v≠$v_k$ y w≠$v_k$. La proposición siguiente nos indica cómo calcular estos pares.

**Proposición III.2.2.5**

Sea (v,w)∉$E_{k-1}$. Tenemos que (v,w)∈$E_k$ si y sólo si (v,$v_k$)∈$E_{k-1}$ y ($v_k$,w)∈$E_{k-1}$.

**Demostración:**

La condición suficiente es inmediata según la definición de $E_k$.

Mostraremos la condición necesaria. Si (v,w)∈$E_k$ entonces existe un camino de v a w cuyos vértices intermedios están en {$v_1$, ... ,$v_k$}. Como (v,w)∉$E_{k-1}$ este camino incluye a $v_k$ como vértice intermedio, por lo tanto existen caminos de v a $v_k$ y de $v_k$ a w con vértices en {$v_1$, ... ,$v_{n-1}$}, esto según la proposición III.2.2.4. significa que (v,$v_k$)∈$E_{k-1}$ y ($v_k$,w)∈$E_{k-1}$.

¤

Sea M($E_k$)=($e_{ij}^k$) la matriz booleana asociada a $E_k$, 0≤k≤n. El hecho $E_{k-1} \subseteq E_k$ y las proposiciones III.2.2.4 y III.2.2.5. nos dicen como calcular $E_k$ a partir de $E_{k-1}$:

$$e_{ij}^k = \begin{cases} e_{ij}^{k-1} \text{ si i=k ó j=k} \\ \\ e_{ij}^{k-1} \vee ( e_{ik}^{k-1} \wedge e_{kj}^{k-1} ) \text{ si i≠k y j≠k} \end{cases}$$

Como

$$e_{ik}^{k-1} = e_{ik}^{k-1} \vee ( e_{ik}^{k-1} \wedge e_{kk}^{k-1} ) \qquad \text{y} \qquad e_{kj}^{k-1} = e_{kj}^{k-1} \vee ( e_{kk}^{k-1} \wedge e_{kj}^{k-1} )$$

Tenemos,

$$\forall i,j \quad e_{ij}^k = e_{ij}^{k-1} \vee ( e_{ik}^{k-1} \wedge e_{kj}^{k-1} )$$

Como $e_{ik}^{k}$ y $e_{kj}^{k}$ son iguales a $e_{ik}^{k-1}$ y $e_{kj}^{k-1}$ respectivamente, podemos guardar el elemento $e_{ij}^{k}$ en la misma posición de memoria de $e_{ij}^{k-1}$.

Algoritmo de Roy-Warshall: Cálculo de la matriz de alcance de un grafo G=(V,E).
{ Entrada :
- Una variable M que contiene la matriz de adyacencias de G.
- n= |V|
Salida:
- Una variable M* que contiene la matriz de alcance de G. }
Var i,j,k:enteros;
Comienzo
- M*←I+M; {+ representa la suma booleana}
- Para k←1 hasta n, hacer:
  - Para i←1 hasta n, hacer:
    - Si i≠k y M*[i,k]=1 entonces
      - Para j←1 hasta n hacer:
        - M*[i,j]←M*[i,j]+M*[k,j]

fin

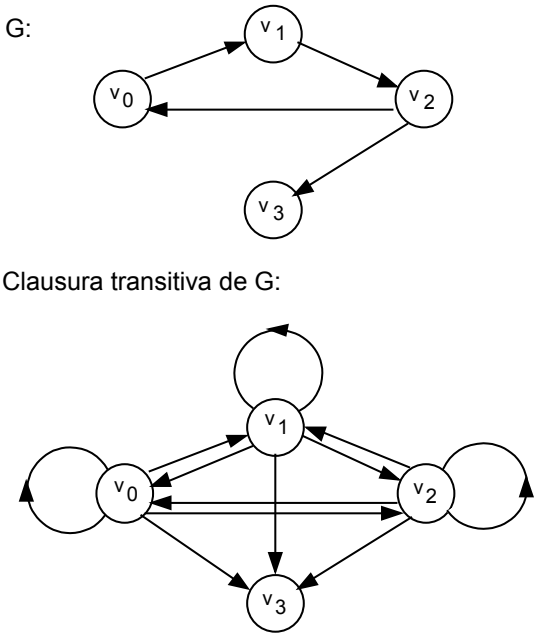



**figura III.10**

### III.2.3. CLAUSURA TRANSITIVA DE UN GRAFO

Una clausura transitiva de un grafo G=(V,E) orientado sin arcos múltiples es un grafo cuyo conjunto de vértices es V y cuyo conjunto de arcos es una relación binaria transitiva minimal que incluye a E; el término minimal significa en este caso que no existe otra relación transitiva que incluya a E y esté contenida en la clausura. Ver figura III.10. Note que G posee al menos una clausura pues (V,VxV) es transitivo.

**Proposición III.2.3.1**

Todo grafo G orientado sin arcos múltiples posee una única clausura transitiva que denotaremos por $G^{+}=(V,E^{+})$.

**Demostración:**

Sean $G_1=(V,E_1)$ y $G_2=(V,E_2)$ dos clausuras transitivas del grafo $G=(E,V)$. Sea $G_3=(V,E_1\cap E_2)$. Tenemos que $E \subseteq E_1\cap E_2$ y $E_1\cap E_2$ es transitiva por lo tanto si $E_1\neq E_2$ tenemos que $E_1$ ó $E_2$ no serían minimales pues $E_1\cap E_2$ estaría incluido estrictamente en uno de los dos conjuntos $E_1$ ó $E_2$.

¤

La proposición anterior nos dice además que dado G=(V,E), si G=(V,E') es un grafo tal que $E \subseteq E'$ y E' es transitiva entonces $E^+ \subseteq E'$.

**Proposición III.2.3.2**

Sea G=(V,E) un grafo y $G^+=(V,E^+)$ su clausura transitiva. Entonces $E^+=E\cup E^2\cup ... \cup E^n$.

**Demostración:**

Es evidente que $E \subseteq E\cup E^2\cup ... \cup E^n$. Mostremos que $E\cup E^2\cup ...\cup E^n$ es transitiva, es decir:
$(E\cup E^2\cup ... \cup E^n)^2 \subseteq E\cup E^2\cup ... \cup E^n$

esta inclusión es consecuencia de la proposición III.2.2.1. Por lo tanto $E^+ \subseteq E\cup E^2\cup ... \cup E^n$.

Por otro lado, para mostrar que $E\cup E^2\cup ... \cup E^n \subseteq E^+$ basta con probar que $\forall\ k\geq 1$, $E^k \subseteq E^+$. Procedamos por inducción sobre k: Para k=1, $E \subseteq E^+$ por definición. Supongamos cierto para valores menores o iguales a k. Tenemos $E^k \subseteq E^+$ y $E \subseteq E^+$, por lo tanto $E^{k+1}=E^k oE \subseteq E^+ oE^+ \subseteq E^+$. La primera contención se debe a una propiedad de composición de las relaciones y la segunda inclusión se debe a la transitividad de $E^+$.

¤

Para calcular $E^+$ podemos aplicar el algoritmo de Roy-Warshall inicializando la variable M* con M en lugar de I+M. La fundamentación de este hecho la exponemos en lo que sigue:

Sea G=(V,E) con $V=\{v_1, ... ,v_n\}$. Sea $E'_k$, $0\leq k\leq n$, la relación binaria sobre V: $(v,w)\in E_k$ si y sólo si existe un camino de v a w de largo mayor o igual a 1, cuyos vértices intermedios estén en $\{v_1, ... ,v_k\}$.

Tenemos que: $E'_0 = E, \quad E'_{k-1} \subseteq E'_k$ y $E'_n = E^+$.

Además, las proposiciones III.2.2.4. y III.2.2.5. siguen siendo válidas si reemplazamos a $E_{k-1}$ y $E_k$ por $E'_{k-1}$ y $E'_k$. Así, podemos aplicar el algoritmo de Roy-Warshall para calcular $E'_n$ $(=E^+)$ con M* igual a M.

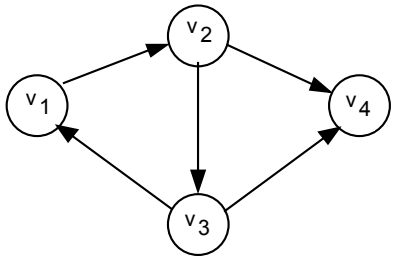
## III.3 CONECTIVIDAD

### III.3.1 DEFINICIONES

La relación "w es alcanzable desde v y v es alcanzable desde w" sobre el conjunto de los vértices V de un grafo G es una relación de equivalencia. Si $V_1,V_2,... ,V_k$ son las clases de equivalencia de esta relación, los grafos $G_{V1},G_{V2}, ... ,G_{Vk}$ engendrados por $V_1,V_2,... ,V_k$ se denominan *componentes fuertemente conexas de G*. Si k=1 decimos que *G es fuertemente conexo* (ver figura III.11(a)).

Cuando G es no orientado, los subgrafos de G engendrados por las clases de equivalencia de la relación anterior con respecto al grafo orientado asociado a G, se denominan simplemente *componentes conexas de G*. Dos vértices estarán en la misma componente conexa si y sólo si existe una cadena de uno al otro en G. Cuando el número de componentes conexas de G es igual a 1 decimos que G es un *grafo conexo*. Decimos que un grafo orientado es conexo, si el grafo no orientado asociado es conexo (ver figura III.11(b) ).

G:




Componentes fuertemente conexas de G:

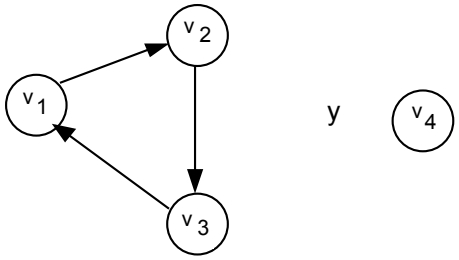
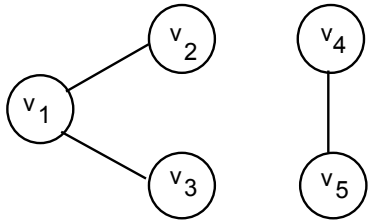



**figura III.11(a)**

G':




Componentes conexas de G':

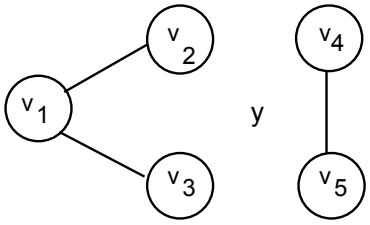



**figura III.11(b)**

Normalmente, en la literatura, el concepto de conectividad de un grafo no orientado no se define a partir del grafo orientado asociado como lo hemos hecho anteriormente. Se razona independientemente definiendo la relación de equivalencia siguiente, sobre el conjunto de vértices: "existe una cadena entre v y w". Los grafos inducidos por las clases de equivalencia de esta relación son las componentes conexas del grafo. Si sólo hay una componente conexa, se dice que el grafo es conexo.

Entonces, decir que un grafo es fuertemente conexo es equivalente a decir que existe un camino entre todo par de vértices. De igual forma, decir que un grafo es conexo es equivalente a decir que existe una cadena entre todo par de vértices.

A continuación daremos algunas propiedades de las componentes fuertemente conexas de un grafo.

**Proposición III.3.1.1**

Sea G un grafo dirigido, $G_{V_i}$ una componente fuertemente conexa de G y v, w dos vértices de $V_i$. C es un camino de v a w en G si y sólo si C es un camino de v a w en $G_{V_i}$.

**Demostración:**

Demostremos sólo la condición necesaria pues la suficiente es obvia.

Sea C un camino de v a w en G. Hay que mostrar que todos los vértices intermedios de C están en $V_i$, o sea, para todo vértice w' en el camino C se debe cumplir que existe un camino de v a w' en G y un camino de w' a v en G.

Sea C' un camino de w a v en G. C||C' es un camino de v a v que pasa por w'. Podemos entonces extraer un camino de v a w' y otro de w' a v.

¤

Llamaremos *grafo reducido* de un grafo G al grafo GR sin arcos múltiples ni bucles cuyos vértices son los conjuntos de vértices de las componentes fuertemente conexas $G_{V1}$, $G_{V2}$,..., $G_{Vk}$ de G. Para $i \neq j$, el arco $(V_i,V_j) \in E(GR)$ si y sólo si existe un arco $(v_i,v_j)$ en G con $v_i \in V_i$ y $v_j \in V_j$.

**Proposición III.3.1.2**

El grafo reducido GR de un grafo dirigido G no posee circuitos.

**Demostración:**

Supongamos que existe un circuito C de largo mayor o igual que 2 en GR:

$C=<V_i, \ldots, V_j, \ldots, V_i>$, $V_i \neq V_j$.

A partir de C podemos construir un circuito C' en G de acuerdo a las definiciones de componente fuertemente conexa y GR:

$C'= <v_i, \ldots, v_j, \ldots, v_i >$ con $v_i \in V_i$, $v_j \in V_j$.

Por lo tanto existe un camino de $v_i$ a $v_j$ y otro de $v_j$ a $v_i$ en G, lo que significa que $v_i$ y $v_j$ deben estar en la misma componente fuertemente conexa, lo que se contradice con $V_i \neq V_j$.

¤

**Proposición III.3.1.3**

Sea G=(V,E) un digrafo y $v \in V$. Si denotamos por D(v) y A(v) los conjuntos de descendientes y ascendientes de v entonces, el conjunto de vértices de la componente fuertemente conexa que contiene a v, C(v), es $C(v)=D(v) \cap A(v)$.

**Demostración:**

Sea $w \in C(v)$. Por definición w es ascendiente y descendiente de v así $w \in D(v) \cap A(v)$.

Por otro lado por propiedades de las clases de equivalencia todo elemento fuera de una clase no está relacionado con los de esa clase. En nuestro caso si $w \in D(v) \cap A(v)$ entonces w debe pertenecer a C(v) pues, si no, esto implicaría que w no está relacionado con ningún elemento de C(v), en particular con v. Como la relación es "x es descendiente y ascendiente de y", esto sería una contradicción con nuestra hipótesis.

¤

**Observación:** Cuando G no es dirigido D(v)=A(v)=C(v).

### III.3.2 UN PRIMER ALGORITMO PARA DETERMINAR LAS COMPONENTES (FUERTEMENTE) CONEXAS

Para calcular las componentes fuertemente conexas de un digrafo G, basta con conocer los descendientes y los ascendientes de cada vértice del grafo. Si D(v) y A(v) denotan los conjuntos de descendientes y ascendientes de $v \in V(G)$ entonces, la componente fuertemente conexa de v, C(v) será según la proposición III.3.1.1:

$$C(v)=D(v) \cap A(v)$$

Para el cálculo de D(v) podemos determinar la matriz de alcance $A_G^*$ del grafo G (donde $A_G$ representa la matriz de adyacencias de G) y tomar la fila que corresponde a v, esa fila corresponde al vector característico de D(v) (el vértice w estará en D(v) si y sólo si la casilla correspondiente a (v,w) en $A_G^*$ es igual a 1). Para calcular A(v) basta con tomar la columna de $A_G^*$ correspondiente a v; esta columna es el vector característico de A(v). Cada elemento del vector característico de C(v) se obtiene haciendo el producto de los elementos correspondientes en A(v) y D(v).

De todos estos comentarios podemos observar las propiedades siguientes:

Denotemos por $G^{-1}$ el grafo obtenido al invertir el sentido de todos los arcos de G. Tenemos evidentemente que v es un descendiente de w en G si y sólo si v es un ascendiente de w en $G^{-1}$, además $(G^{-1})^{-1}= G$. Calcular A(v) en G, es equivalente a calcular D(v) en $G^{-1}$. La matriz de alcance de $G^{-1}$ proporciona, en la fila correspondiente a cada vértice, sus ascendientes en G; por lo tanto si $A_G^*$ y $A_{G^{-1}}^*$ son las matrices de alcance de G y $G^{-1}$ respectivamente, tendremos que cada fila de la matriz $A_G^* \oplus A_{G^{-1}}^*$ (donde $\oplus$ representa el producto elemento a elemento) es el vector característico del conjunto de vértices de la componente fuertemente conexa que contiene el vértice correspondiente a la fila.

**Proposición III.3.2.1**

Sea A la matriz de adyacencias de un digrafo G=(V,E). Sea $G^{-1}$ definido anteriormente. Entonces $A_{G^{-1}}^* = (A^t)^* = (A^*)^t$.

Es decir la matriz de alcance de $G^{-1}$, $A_{G^{-1}}^*$, es igual a la matriz $(A^t)^*$ obtenida de transponer A y calcular la matriz de alcance asociada a esa transpuesta, y esta matriz a su vez es igual a la transpuesta de la matriz de alcance $A^*$ del digrafo G.

**Demostración:**

$A^t$ representa la matriz de adyacencias de $G^{-1}$, por lo tanto $A_{G^{-1}}^* = (A^t)^*$. Por otro lado el elemento (v,w) de $A_{G^{-1}}^*$ es igual a uno si y sólo si existe un camino de v a w en $G^{-1}$. Esto ocurre si y sólo si existe un camino de w a v en G, o lo que es lo mismo, el elemento (w,v) es igual a 1 en $A^*$. Por lo tanto $(v,w) \in (A^*)^t$.

¤

En el capítulo IV veremos un algoritmo *O* (max(|V|,|E|)) para calcular las componentes fuertemente conexas. Por los momentos daremos un algoritmo basado en los comentarios anteriores.

Algoritmo para calcular las componentes fuertemente conexas de un grafo:

{ Entrada:
- Matriz de adyacencias A o lista de adyacencias de G=(V,E)

Salida:
- Variable COMPCON que contendrá los conjuntos de vértices de cada componente fuertemente conexa }

Variables RESTO, COMP: conjunto de vértices; v,w:vértice;
Comienzo
(1) Calcular la matriz de alcance A* del grafo G;

RESTO←V; COMPCON ← ∅;

(2) Mientras RESTO ≠ ∅ hacer:

Comienzo

v←elemento de RESTO;

COMP ← ∅;

(3) Para cada w alcanzable desde v hacer:

Si v es alcanzable desde w entonces COMP←COMP∪{w};

COMPCON←COMPCON∪{COMP};

RESTO←RESTO-COMP

fin

fin.

**Observaciones:**

- Si G no es orientado no hace falta hacer la pregunta ¿v es alcanzable desde w?, cuando se asigna w a COMP.
- El orden de este algoritmo es $O(|V|^3)$ pues (1) es $O(|V|^3)$, (3) es $O(|V|)$ y (2) es O(Número de componentes conexas $* |V|) = O(|V|^2)$, siempre y cuando el orden de las operaciones entre conjuntos sea constante.

### III.3.3 CONJUNTOS DE ARTICULACIÓN

Sea G=(V,E) un grafo orientado. Diremos que G es *disconexo* cuando G no es fuertemente conexo. Cuando G es no orientado, si G es disconexo es porque el grafo orientado asociado a G no es fuertemente conexo, esto último es equivalente a decir que G *no es conexo*. Si G es un grafo orientado fuertemente conexo, diremos que un subconjunto propio A de V(G) *desconecta* a G si $G_{V-A}$ es disconexo.

Un subconjunto propio A de vértices de un grafo no orientado G=(V,E) se llama *conjunto de articulación* si el grafo $G_{V-A}$ posee al menos una componente conexa más que G, es decir, si al eliminar de G todos los vértices en A y todas las aristas incidentes en vértices de A, el grafo resultante posee al menos una componente conexa más que G. Cuando $|A|=1$, el vértice en A lo llamamos *punto de articulación* (ver fig. III.12).

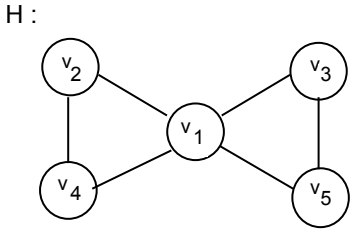



**figura III.12**

Note que si G es conexo y A es un conjunto de articulación de G, entonces se dice que A *desconecta* a G.

Dado un grafo orientado G, se define *la conectividad de* G, k(G), como el mínimo número de vértices cuya eliminación desconecta a G o lo reduce a un solo vértice. Nótese que k(G)=0 significa que G es disconexo. De los grafos no orientados sabemos que su grafo orientado asociado es fuertemente conexo, por lo tanto extenderemos el concepto de conectividad a los grafos no orientados, entendiéndose que trabajamos sobre el grafo orientado asociado. Diremos que un grafo G es *h-conexo* si su conectividad k(G) es mayor o igual a h, es decir, ningún conjunto de cardinalidad h-1 desconecta a G. Si G=(V,E) es conexo, no orientado y no completo entonces existen dos vértices no

adyacentes a,b $\in$ V y V-{a,b} es un conjunto de articulación de G y por lo tanto k(G)$\leq$|V-{a,b}|=|V|-2. Si G es no orientado y completo entonces k(G)=|V|-1.

Por lo tanto, decir que un grafo no orientado y completo es h-conexo es equivalente a decir que se tiene simultáneamente: h$\leq$|V|-1 y no existe conjunto de articulación A con |A| <h.

De igual forma definiremos la *conectividad respecto a los lados* de un grafo G como el mínimo número de lados cuya eliminación desconecta a G. Un *istmo* es un lado cuya eliminación aumenta el número de componentes conexas de G.

### III.3.4 PROPIEDADES

**Proposición III.3.4.1**

Sea G=(V,E) un grafo. G es (fuertemente) conexo si y sólo si M(E*) es igual a la matriz cuyos elementos son todos iguales a 1.

¤

Sea S un conjunto de vértices de un grafo G=(V,E). Llamaremos *cociclo asociado a* S y lo denotaremos por $\Omega$(S), al conjunto de lados que poseen exactamente una extremidad en S, es decir, e$\in\Omega$(S) si y sólo si e posee una extremidad en S y la otra en $\bar{S}$. Un subconjunto F de lados se llama cociclo de G si existe S $\subseteq$ V tal que $\Omega$(S)=F. Se puede mostrar que un istmo es un cociclo que tiene cardinal uno.

Cuando G es orientado el cociclo asociado a S, $\Omega$(S), admite una partición en dos subconjuntos (ver figura III.13):

$\Omega^+$(S)={e$\in$E/ la extremidad inicial de e$\in$S}

$\Omega^-$ (S)={e$\in$E/ la extremidad final de e$\in$S}

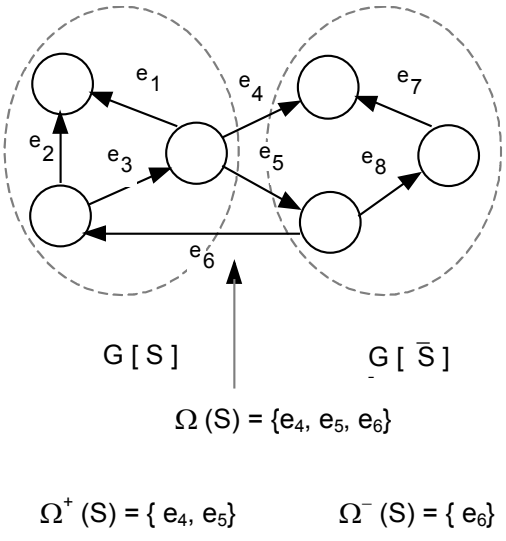



**figura III.13**

**Proposición III.3.4.2**

Sea G=(V,E) un grafo. Una condición necesaria y suficiente para que G sea (fuertemente) conexo es que, para todo subconjunto propio S de V, se tenga $\Omega$(S) $\neq \varnothing$ ($\Omega^+$(S) $\neq \varnothing$).

**Demostración:**

Sea S un subconjunto propio de vértices. Sea v$\in$S y w$\in$ V-S. Como G es (fuertemente) conexo, existe una cadena elemental (camino elemental) de v a w: $<x_0,e_1,x_1, \ldots ,x_{p-1},e_p,x_p>$, $x_o$=v y $x_p$=w. Sea q el entero más grande tal que $x_q\in$S. Es evidente que q$\leq$p-1 de donde $x_{q+1} \in$V-S, y $e_{q+1}\in\Omega$(S) ( $\in \Omega^+$(S) ).

Para mostrar la condición suficiente supongamos que G es no conexo (es no fuertemente conexo). Existen dos vértices v y w, v≠w tales que no existe una cadena de v a w (un camino de v a w). Sea S = {v} ∪ {x/ existe una cadena (camino) de v a x}. Tenemos que S ≠ ∅, S≠V y $\Omega(S) = \emptyset$ ($\Omega^+(S) = \emptyset$), lo que contradice las hipótesis.

¤

**Proposición III.3.4.3**

Todo grafo conexo G=(V,E) con n vértices (n≥2) posee al menos dos vértices que no son puntos de articulación.

**Demostración:**

Como G es conexo y n≥2, G contiene una cadena elemental más larga de largo mayor o igual a 1: C= $<x_0,e_1,x_1, \ldots ,x_{p-1},e_p,x_p>$, $x_0 \neq x_p$. Mostremos que $x_0$ y $x_p$ no son puntos de articulación. Supongamos lo contrario, que $x_0$ es un punto de articulación, es decir, $H=G_{V-\{x0\}}$ no es conexo (ver figura III.14).

Consideremos en H el conjunto S de vértices de la componente conexa que contiene a $x_p$. Tenemos que:

- $T=V-S-\{x_0\} \neq \emptyset$

- $\{x1,x2, \ldots , xp\} \subseteq S$

Por otro lado, no existen aristas en H con una extremidad en S y otra en T. Como G es conexo, sí existe al menos una arista $e_0$ en G entre $x_0$ y T, por lo tanto esta arista debe conectar $x_0$ con $y_0 \in T$. Pero entonces $<y_0,e_0,x_0>$ ‖ C será de largo p+1, lo que contradice la elección de C.

El mismo razonamiento anterior sirve para probar que $x_p$ no es punto de articulación.

¤

**Proposición III.3.4.4**

Todo grafo conexo de orden n posee al menos n-1 lados.

**Demostración:**

Utilizaremos un razonamiento inductivo sobre el número de vértices. Para n=1, es evidente. Supongamos cierta la propiedad para los grafos de orden n-1. Sea G=(V,E) un grafo de orden n≥2. Según la proposición III.3.4.3, existe al menos un vértice v∈V que no es punto de articulación. El grafo obtenido de G al eliminar v es conexo de orden n-1 y por hipótesis de inducción posee a menos n-2 lados. Como v es adyacente al menos con un nodo de V-{v}, entonces el número de lados de G debe ser al menos n-1.

¤

**Proposición III.3.4.5**

Sea G=(V,E) un grafo y v∈V. v es un punto de articulación de G si y sólo si existen dos nodos x e y, distintos de v y distintos entre sí, tales que toda cadena de x a y pasa por v.

**Demostración:**

Supongamos que v es un punto de articulación. Sea G' la componente conexa de G que contiene a v. Al eliminar v de G', éste se desconecta en varias componentes conexas $G'_1$, $G'_2$, ... , $G'_k$ (k≥2). Sean x∈V($G'_1$), y∈V($G'_2$). Si una cadena de x a y no contuviera a v, entonces, al eliminar v de G, x e y pertenecerían a la misma componente conexa del grafo resultante. Pero x∈V($G'_1$) e y∈V($G'_2$), de donde toda cadena de x a y pasa por v.

Recíprocamente, es evidente que al eliminar a v de G, no existirá cadena de x a y, lo cual implica que x e y pertenecerán a componentes conexas diferentes, aumentando así el número de componentes conexas.

¤

**Proposición III.3.4.6**

Si G=(V,E) es un grafo entonces e∈E es un istmo si y sólo si e no pertenece a ningún ciclo de G.

**Demostración:**

Sea e={v,w} un istmo de G y supongamos que pertenece a un ciclo de G. De acuerdo a la proposición III.1.3.4., existe un ciclo elemental que pasa por e y por lo tanto existe una cadena de v a w que no pasa por e. De esto último vemos que la componente conexa de G que contiene a v y w no se desconecta al eliminar la arista e, lo que contradice el hecho de suponer que e es un istmo. De donde e no puede pertenecer a un ciclo de G.

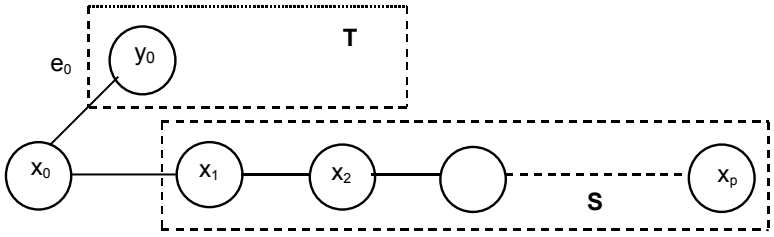



**figura III.14**

Recíprocamente, si e no pertenece a ningún ciclo de G entonces toda cadena de v a w contiene a e ( para mostrar este hecho basta con suponer lo contrario y llegar a una contradicción). Por lo tanto al eliminar e de G, aumenta el número de componentes conexas de G pues v y w no pertenecerían a la misma componente conexa.

¤

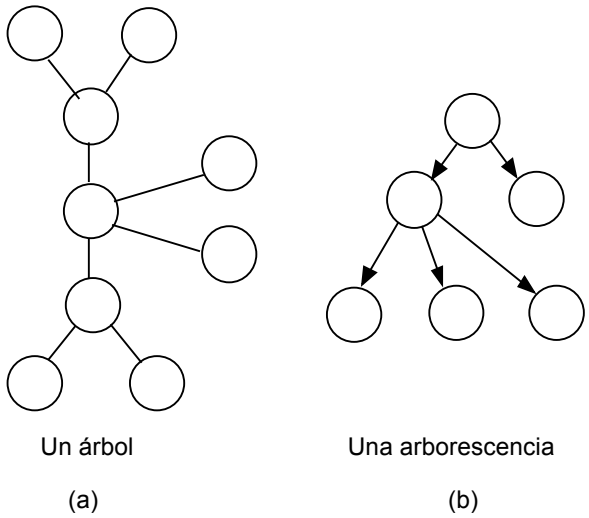



**figura III.15**

Un grafo conexo y sin ciclos lo llamamos *árbol* (ver figura III.15(a)). Vemos, según la proposición III.3.4.6 que todo lado de un árbol es un istmo y las proposiciones III.1.3.10 y III.3.4.4 nos dicen que el número de lados es igual al número de vértices menos uno. Un árbol orientado que posee un vértice r, tal que todo vértice es alcanzable desde r se llama *arborescencia* y r se llama *raíz* de la arborescencia (ver figura III.15(b)). En el capítulo VI nos avocaremos al estudio de estas clases de grafos las cuales son de mucha utilidad en computación. Por los momentos podemos dar las siguientes propiedades fundamentales de árboles y arborescencias.

Un grafo dirigido o no, cuyas componentes conexas sean árboles, lo llamaremos *bosque*.

**Proposición III.3.4.7**

Si G=(V,E) es un árbol entonces existe una y sólo una cadena simple entre cada par de vértices (por lo tanto una y sólo una cadena elemental entre cada par de vértices).

**Demostración:**

Esta proposición es consecuencia directa de la proposición III.1.3.8.

¤

**Proposición III.3.4.8**

Sea G=(V,E) una arborescencia con raíz r. Entonces: $\forall v \in V$, $v \neq r$: $d^-(v)=1$ y $d^-(r)=0$. Además, existe uno y sólo un camino de r a cada vértice de G.

**Demostración:**

Si $d^-(r)$ fuera distinto de cero, tendríamos $v^* \in V$ con $e^*=(v^*,r) \in E$. Como G es una arborescencia existe un camino C* de r a v*. Del camino cerrado C* || <v*,e*,r> podemos extraer un circuito y por lo tanto un ciclo en G. Así, $d^-(r)=0$, lo que contradice la hipótesis.

Por otra parte, sea $v \neq r$ entonces $d^-(v)$ no puede ser cero pues existe un camino de r a v. Si $d^-(v)$ fuera mayor que 1 tendríamos al menos dos vértices $v_1$, $v_2$, $v_1 \neq v_2$, tales que $(v_1,v), (v_2,v) \in E$. Tendríamos entonces dos caminos distintos de r a v y por lo tanto podemos construir dos cadenas simples distintas de r a v, pero según la proposición III.3.4.7 esto es imposible ya que el grafo no orientado asociado a G es un árbol.

La definición de arborescencia garantiza la existencia de un camino desde r a cada vértice de G y la unicidad la garantiza la proposición III.3.4.7.

## III.4 EJERCICIOS

**1.** Si en un grafo *G=(V,E)* todos los vértices tienen grado 2, entonces $|E| = |V|$.

**2.** Sea *G* el dígrafo de la figura III.16.

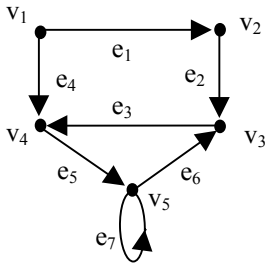



**figura III.16**

Determine cadenas simples pero no elementales, cadenas elementales, caminos simples pero no elementales, ciclos y circuitos elementales.

**3.** Sea A la matriz de adyacencias de un grafo dirigido G sin arcos múltiples. Demuestre que G no tiene circuitos si y sólo si existe $k \in \aleph$ tal que $\forall m > k : A^m$ es la matriz nula (es decir, todos sus elementos son iguales a cero).

**4.** Pruebe que si un grafo simple *G=(V,E)* con $E \neq \varnothing$ no tiene ciclos, entonces su número cromático es 2.

**5.** Mostrar que en el grafo de precedencia inmediata de la relación de orden $(P(A), \subseteq)$, *A* un conjunto infinito, todos los caminos de un vértice x a un vértice y tienen la misma longitud.

**6.** Sea *G=(V,E)* un grafo simple no dirigido conexo pero no completo. Demuestre por inducción sobre $|V|$ que *G* tiene 3 vértices *u*, *v* y *w* tales que $\{u, v\}, \{v, w\} \in E$ pero $\{u, w\} \notin E$.

**7.** Explique por qué el grafo de la figura III.17 no tiene ciclos hamiltonianos.

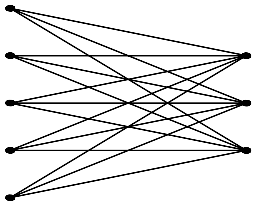


**figura III.17**

**8.** ¿El grafo de la figura III.18 posee un camino hamiltoniano?:

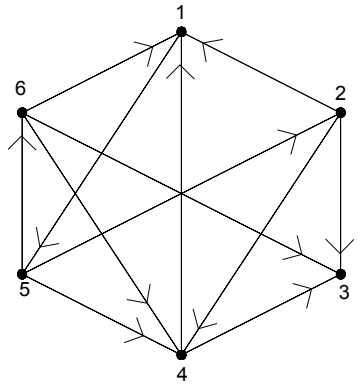



**figura III.18**

**9.** Se dice que un grafo $G=(V,E)$ es hipohamiltoniano si $G$ no es hamiltoniano, pero $G_{V-\{v\}}$ es hamiltoniano para todo vértice $v$ en $V$. Muestre que el grafo en la figura III.19 (conocido como el grafo de Petersen) es hipohamiltoniano:

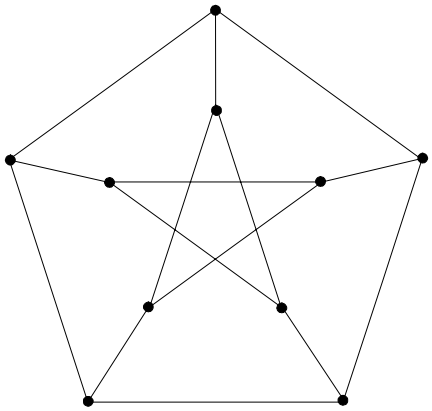


**figura III.19**

**10.** Sea $G$ un grafo completo y $G'$ un grafo dirigido obtenido al dar una orientación a cada una de las aristas de $G$.

a) Demuestre que $G'$ tiene un camino que incluye cada vértice de $G$ una y sólo una vez.

b) Escriba un algoritmo que calcule el camino en (a).

c) Diga que estructuras de datos consideraría apropiadas para implementar la parte (b).

**11.** Se define un **apareamiento** en un grafo $G=(V,E)$ como un conjunto de lados sin vértices comunes entre sí. Se define una **cadena de aumento con respecto a un apareamiento *A*** a una cadena cuyos lados se encuentran alternadamente en $A$ y en $E$-$A$ y tal que los vértices inicial y terminal de la cadena no son incidentes a lados en $A$. Se sabe que un apareamiento $A$ tiene máxima cardinalidad si no es posible encontrar ninguna cadena de aumento con respecto a $A$. Dado un grafo bipartito $G=(V_1 \cup V_2,E)$, se desea encontrar el número máximo de parejas que se pueden formar con elementos en $V_1$ y $V_2$. Escriba un algoritmo para encontrar cadenas de aumento en un grafo bipartito y utilícelo en otro algoritmo que determine un apareamiento de máxima cardinalidad. Aplique el algoritmo al grafo en la figura III.17.

**12.** Dado el grafo:

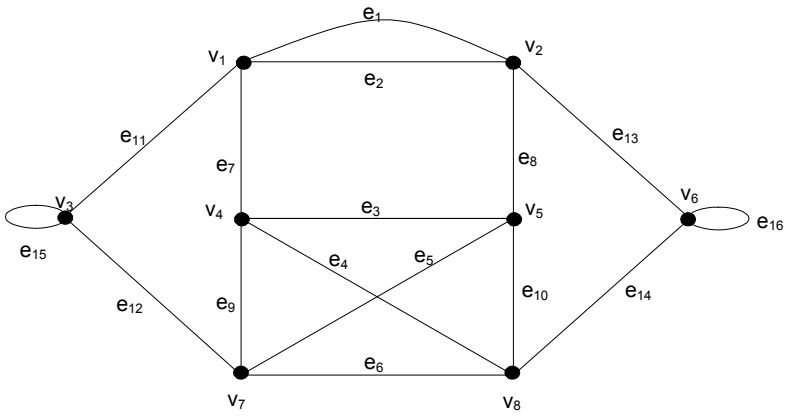



**figura III.20**

a) Diga si se puede determinar un ciclo euleriano en el grafo. ¿Por qué?. En caso afirmativo, determínelo utilizando el algoritmo de la sección III.1.4.

b) Diga, en general, cómo a partir de un ciclo euleriano se puede dar una orientación a las aristas de forma tal que $d^+(v)=d^-(v)$ para todo vértice.

c) Dé una orientación al subgrafo inducido por $\{v_4,v_5,v_6,v_7,v_8\}$ de forma que este subgrafo inducido y con esa orientación sea igual, si es posible, a su clausura transitiva.

d) ¿En que difieren el conjunto de arcos de la clausura transitiva del digrafo hallado en (c) y del grafo inducido por la relación de alcance $E^*$ de ese mismo grafo?.

**13.** ¿Cuál de los grafos en la figura III.21 posee un ciclo o una cadena euleriana? ¿Por qué?

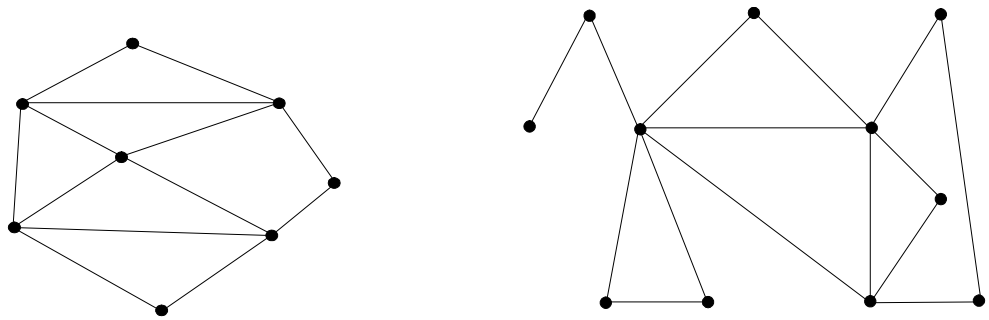


**figura III.21**

**14.** Sea $G=(V,E)$ un grafo no orientado. Demuestre que:

a) Si $G$ es conexo con $2k$ vértices de grado impar entonces $E$ puede particionarse en $k$ cadenas de forma tal que cada lado se encuentra en una sola de estas cadenas.

b) Si $G$ es conexo y $v$, $w$ son los únicos vértices de grado impar entonces $G$ tiene una cadena euleriana entre $v$ y $w$.

c) *G* no tiene vértices aislados y tiene una cadena euleriana entre v y w si y sólo si *G* es conexo y v, w son los únicos dos vértices de grado impar de G.

d) Demuestre que si todos los vértices de un grafo $G=(V,E)$ tienen grado par, entonces existen ciclos $C_1=(V_1,E_1)$, …, $C_m=(V_m,E_m)$ tales que $E_1$, …, $E_m$ conforman una partición de $E$.

e) Demuestre que una condición necesaria y suficiente para que un grafo dirigido sin vértices aislados admita un circuito euleriano es que sea conexo y $d^+(v) = d^-(v)$ para todo vértice v del grafo. Dé un algoritmo para calcular un circuito euleriano en un grafo.

**15.** Se dispone del grafo en la figura III.22 que modela las calles de una ciudad. Los vértices representan las intersecciones y las aristas las calles. El servicio de aseo urbano de esta ciudad está planificando el recorrido que debe hacer el único camión que posee para recolectar la basura en todas las calles. Existen dos intersecciones de la ciudad que corresponden al sitio de estacionamiento del camión y al relleno sanitario de la ciudad.

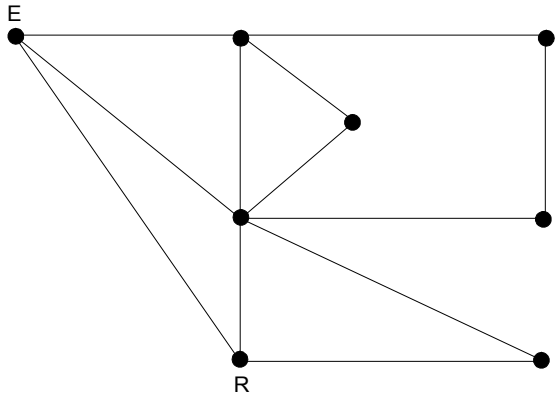



**figura III.22**

a) Debido a las molestias que causa el proceso de recolección se desea saber si es posible encontrar un recorrido que no pase por ninguna calle más de una vez y en caso afirmativo halle el recorrido.

b) Para efectos de recolección de productos reciclables, la compañía ha decidido colocar el mínimo número de centros de reciclaje en las intersecciones, de forma tal que todo vecino tenga disponible uno a menos de una cuadra. Modele el problema en términos de la representación planteada.

**16.** ¿Cómo utilizaría el algoritmo de cálculo de un ciclo euleriano en un grafo y el algoritmo de cálculo de una cadena de longitud mínima entre dos vértices dados, para diseñar un algoritmo polinomial que determine una cadena cerrada con el menor número de lados en un grafo conexo que pase por todas las aristas del grafo?. Justifique su respuesta. (Ayuda: recuerde que el número de vértices de grado impar de un grafo es par. Por otro lado, si se tiene una cadena entre dos vértices distintos de grado impar y se agrega una arista múltiple por cada lado presente en la cadena, la paridad de los vértices internos de la cadena no se altera y los grados de los vértices extremos de las cadenas serían pares).

**17.** Resuelva mediante grafos si es posible colocar las 28 piezas de un dominó en forma circular de manera que las dos mitades de cualquiera dos piezas consecutivas contenga el mismo número. (Ayuda: trate de buscar un ciclo euleriano en un grafo).

**18.** La superficie de un tambor rotatorio se divide en 16 sectores como se muestra en la figura III.23. La información posicional del tambor se representa por señales digitales binarias **a**, **b**, **c** y **d**, como se muestra en la figura, donde el área conductora (sombreada) y el área no conductora (blanca) son utilizadas para construir los sectores. Dependiendo de la posición del tambor, los terminales **a**, **b**, **c** y **d** estarán conectados a tierra o no. Por ejemplo, cuando la posición del tambor es como la que se muestra en la figura, los terminales **a** y **d** están conectados a tierra, mientras que los terminales **b** y **c** no lo están. Para que las 16 diferentes posiciones del tambor sean representadas unívocamente por las señales binarias de los terminales, los sectores deben ser

configurados de forma tal que los patrones de cuatro sectores consecutivos no sean iguales. El problema es determinar si tal arreglo de sectores conductores y no conductores existe, y si existe, determinarlo. (Ayuda: este problema se puede plantear como hallar un circuito euleriano en un grafo)

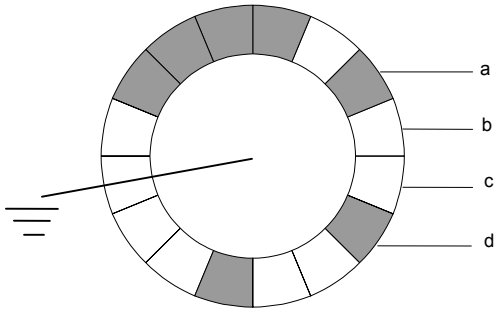



**figura III.23**

**19.** Sea *G* el digrafo en la figura III.24.

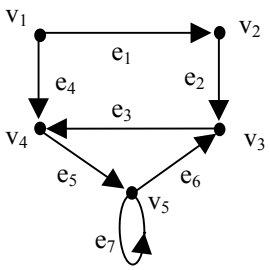



**figura III.24**

- Calcule la matriz de alcance de G.
- Calcule el grafo clausura transitiva de G.
- Diga si G es fuertemente conexo.

**20.** Sea *G* un grafo dirigido cualquiera sin arcos múltiples. Sea *G'* un grafo arco minimal respecto a la propiedad "la clausura transitiva de *G'* es igual a la de *G*".

a) Demuestre que si (*v*, *w*) es un arco de *G'* entonces el único camino de largo al menos 1 de *v* a *w* es *<v,(v,w) w>*.

b) Determine *G'* para el siguiente grafo *G*:

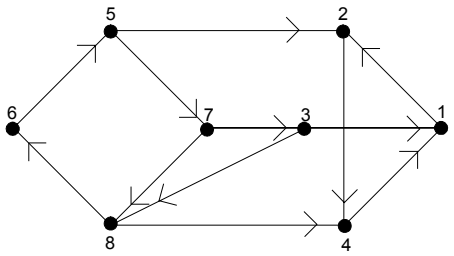



**figura III.25**

c) Determine un algoritmo polinomial que calcule un *G'*. Calcule la complejidad del su algoritmo en función del número de vértices y arcos de *G*.

**21.** El siguiente grafo representa un plano del nuevo flechado de una urbanización, donde los vértices representan puntos de interés clave. Se desea saber si con este flechado todos los puntos estarán intercomunicados y en caso contrario, cuales puntos lo estarán entre sí. Plantee el problema mediante grafos y resuélvalo.

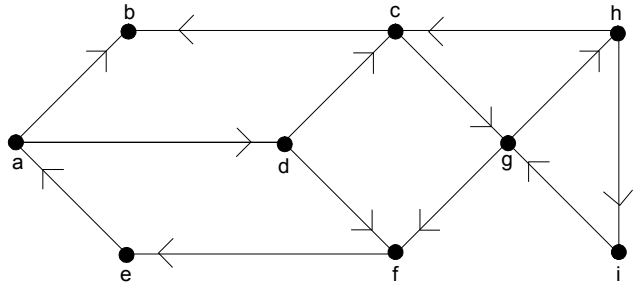



**figura III.26**

**22.** Se tiene una red de computadoras interconectadas de acuerdo a un esquema representado por un digrafo *G=(V,E)* donde *V* representa las computadoras y *E* las interconexiones en la red. No todas las interconexiones son en ambos sentidos. En esta red se ha presentado una falla en un conjunto *S* de *k* computadoras, y se desea determinar si sigue habiendo comunicación en ambos sentidos entre los vértices *x* e *y* (*x* e *y* dados). Escriba un algoritmo que resuelva el problema anterior (se recomienda que su algoritmo esté basado o sea una modificación de alguno visto en clase). Discuta la complejidad de su algoritmo.

**23.** Sea *G=(V,E)* un grafo no orientado y conexo. Demuestre que se puede dar una orientación a cada una de sus aristas de forma que el digrafo resultante sea fuertemente conexo.

**24.** Demuestre que si *G=(V,E)* tiene *p* componentes conexas, entonces $|E| \geq |V| - p$. Además, si $|E| > |V| - p$, entonces *G* tiene al menos un ciclo.

**25.** Sea *G=(V,E)* un grafo simple no dirigido de orden ≥ 2. Demuestre que si $|E| > \binom{|V|-1}{2}$ entonces *G* es conexo.

**26.** Demuestre que todo grafo conexo *G=(V,E)* con $|V| \geq 2$ tiene al menos dos vértices que no son puntos de articulación.

**27.** Sea *G=(V,E)* un grafo no orientado. Demuestre que si *G* es un árbol entonces *G* tiene al menos dos vértices de grado 1.

**28.** Pruebe que si para cada par de vértices *x*, *y* en un grafo no orientado *G=(V,E)*, existe exactamente una cadena que los une, entonces *G* es un árbol.

**29.** Pruebe que todo árbol con exactamente dos vértices de grado 1 es el grafo inducido por una cadena.

**30.** Un grafo dirigido en el que existe exactamente un vértice v con $d^-(v)=0$ y para todos los demás vértices $x \in V$ del grafo $d^-(x)=1$ no necesariamente es una arborescencia.

a) Dé un ejemplo que confirme lo dicho arriba.

b) Demuestre que *A* es una arborescencia si y sólo si *A* es un grafo conexo y existe un sólo vértice *r* tal que $d^-(r)=0$ y $\forall x \in V, x \neq r: d^-(x)=1$.

**31.** Demuestre que si *G* es una arborescencia con raíz *r* y $C=\langle x_0, e_1, x_1, \ldots, e_m, x_m \rangle$ es un camino de longitud máxima en *G*, entonces $x_0 = r$.

**32.** ¿Cuál es el máximo número de hojas en una arborescencia con grado máximo exterior igual a 2 y longitud del camino más largo igual a *a*?

**33.** Demuestre que una arborescencia $A$ sólo puede tener un vértice raíz.

**34.** ¿Cuál es el número máximo de vértices que puede tener una arborescencia $A$ con máximo grado de salida $d^+$ igual a $d$ y longitud (número de arcos) del camino más largo igual a $k$?

**35.** Sea $G=(V,E)$ conexo y no orientado. Demuestre ó refute mediante un contraejemplo cada una de las proposiciones siguientes:

a) Si G tiene un istmo y $|V| > 2$, entonces $G$ tiene al menos un punto de articulación.

b) Dé un ejemplo que muestre que el recíproco de a) no siempre es cierto.

c) Si todos los vértices de $G$ tienen grado par, entonces $G$ no tiene puntos de articulación.

d) Si e es un lado de $G$ tal que e está en todo árbol cobertor de $G$ entonces e es un istmo.

e) Sea $v \in V$. Si $v$ es un punto de articulación de $G$ entonces el grado de $v$ en todo árbol cobertor de $G$ es mayor o igual a 2.

f) Si todos los vértices de $G$ tienen grado par entonces $G$ no posee istmo.

**36.** Demuestre o refute mediante un contraejemplo, cada una de las siguientes proposiciones:

a) Un conjunto estable en un grafo $G$ es un conjunto dominante en el grafo complementario de $G$.

b) Si un vértice $y$ es alcanzable desde otro vértice $x$ entonces $x$ e $y$ pertenecen a la misma componente fuertemente conexa.

**37.** Sea $G$ un grafo dirigido. Demuestre que si $\delta^-(x) \geq 1$ entonces $G$ posee un circuito.

**38.** a) Muestre con un contraejemplo que la siguiente proposición es falsa: "Si $G$ posee una cadena cerrada entonces $G$ posee un ciclo".

b) Demuestre que si $G$ posee un camino cerrado entonces posee un circuito.

**39.** Demuestre que un dígrafo $G$ es fuertemente conexo si y sólo si $G$ es conexo y todo arco está en un circuito de $G$.

**40.** Determine todos los puntos de articulación de los grafos de la figura III.21.

**CAPÍTULO IV**

# *RECORRIDOS EN GRAFOS*

## INTRODUCCIÓN

En este capítulo y en el siguiente trataremos uno de los problemas más importantes y clásicos de la teoría de grafos y sus aplicaciones como es la búsqueda de caminos en grafos. La teoría de grafos nace en 1736 cuando Euler se plantea el problema de los puentes de Köninsberg. Este problema consistía en hacer un paseo recorriendo siete puentes partiendo de un lugar y regresando a éste sin haber pasado dos veces por el mismo puente (ver figura IV.1(a)).

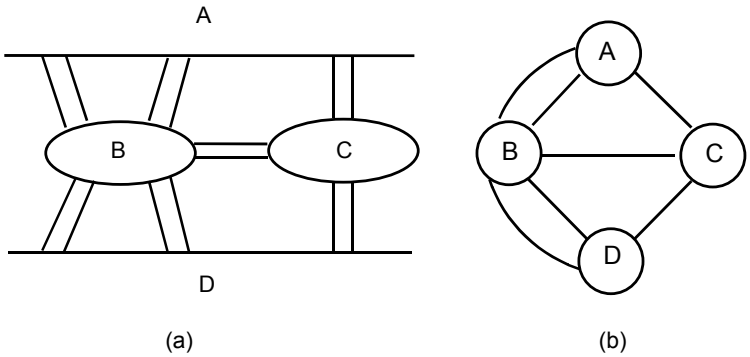



**figura IV.1**

Haciendo un estudio sistemático de la situación, Euler llega a la conclusión de que el problema no tenía solución. Modelando el problema mediante el grafo de la figura IV.1(b), resolverlo era equivalente a hallar un ciclo de largo igual al número de lados. A este tipo de ciclo lo hemos llamado ciclo Euleriano en el Capítulo III. Como vimos en dicho capítulo, Euler muestra que para que un grafo posea un tal ciclo es necesario y suficiente que cada vértice del grafo tenga grado par y que el grafo sea conexo, lo cual no sucede con el grafo de la figura IV.1(b).

Los grafos generalizan el concepto de Relación Binaria sobre un conjunto y han permitido, a través de los nuevos conceptos introducidos sobre esta estructura, por una parte, ahondar más en el estudio de propiedades de relaciones binarias y por la otra, presentar un método de análisis sobre una estructura matemática basado muchas veces en un dibujo. Representar una relación entre pares de elementos mediante un dibujo nos puede proporcionar más información que el puro tratamiento algebraico (¡Es más natural y simple pensar sobre un dibujo que sobre un conjunto de ecuaciones algebraicas!).

En este capítulo abordaremos varios problemas de búsqueda de caminos en un grafo y algunas aplicaciones de los resultados obtenidos como son la determinación de las componentes conexas, las componentes fuertemente conexas, las componentes biconexas y los puntos de articulación de un grafo.

El tratamiento de varios problemas clásicos sobre caminos y recorridos en grafos (recorrer un grafo significa ir de un vértice a otro a través de un camino), lo haremos mediante un enfoque unificador que consiste en considerar los algoritmos tradicionales de búsqueda de caminos en grafos, como un caso particular de un modelo general de algoritmos conocido en la literatura como Modelo de Etiquetamiento (Clovis C. Gonzaga, CPPE -UFRJ). De esta forma, algoritmos clásicos como la búsqueda en profundidad (Depth First Search), búsqueda en amplitud (Breadth

First Search), cálculo de caminos de costo mínimo de Dijkstra, ordenamiento topológico y caminos de costo máximo en grafos de precedencia, etc., son presentados como algoritmos particulares de etiquetamiento. Por otra parte, la presentación de los algoritmos estará disociada de consideraciones de implementación y para cada uno discutiremos su complejidad en función de alguna implementación adecuada.

Por último, queremos mencionar que todos los grafos considerados en el capítulo son dirigidos sin arcos múltiples, a menos que se especifique lo contrario. Además, los algoritmos que se presentan pueden aplicarse a grafos no dirigidos considerando el grafo dirigido asociado, es decir, el que se obtiene reemplazando cada arista por arcos dirigidos en sentidos opuestos, salvo los bucles que simplemente se les da una orientación.

## IV.1 MODELO GENERAL DE ALGORITMOS DE ETIQUETAMIENTO.

### IV.1.1 PRESENTACIÓN DEL PROBLEMA GENERAL.

Dado un grafo dirigido G=(V,E) sin arcos múltiples deseamos conseguir caminos desde un vértice s hasta cada vértice de un conjunto $D \subseteq V$. Estos caminos tienen que satisfacer ciertas condiciones (el conjunto de condiciones puede ser vacío).

En el caso que nos ocupa, el conjunto de condiciones puede ser de dos tipos. Por una parte, estamos interesados en saber cuáles vértices de D son alcanzables desde s en cuyo caso el conjunto de condiciones será vacío. Por otra parte, si definimos una función de costos sobre E, c: $E \rightarrow \Re$. Definimos el costo C(P) de un camino P de v a w como la suma de los costos de los arcos del camino, entonces nos interesa hallar un camino de costo mínimo de s a cada vértice de D. Note que si la función de costos es igual a uno para todo arco, entonces el problema consistirá en encontrar caminos de largo mínimo de s a D.

Definimos cmin(v,w) = inf{C(P)/ P es un camino de v a w} con la convención: inf ø=+∞. (inf significa ínfimo).

Ver ejemplos en la figura IV.2.

Ahora nuestro problema general lo podemos definir como sigue:

**Problema de búsqueda de caminos:**

Encontrar, si existe, un camino de s a cada vértice w en D, tal que el camino P de s a w satisfaga ciertas condiciones.

### IV.1.2 EL MODELO GENERAL DE ETIQUETAMIENTO

La idea detrás de los algoritmos de etiquetamiento es ir determinando caminos que partan de s, colocándolos en una lista de caminos e ir escogiendo los que sean solución a nuestro problema. Un primer algoritmo, aunque muy ineficiente, sería: "liste todos los caminos posibles y escoja los que se ajusten a la solución".

El modelo que proponemos es un proceso iterativo que va guardando en una lista los caminos generados. El primer camino de la lista será <s>. En cada iteración se escoge un camino de la lista que no haya sido expandido (ver sección III.1.2) y lo expande a cada sucesor del nodo terminal. A cada camino en la lista posee una etiqueta con la inscripción siguiente:

Abierto: cuando todavía no haya sido expandido.

Cerrado: cuando ya fue expandido.

Algunos de los caminos generados por la expansión se incorporan a la lista de abiertos y el resto es desechado. Este proceso se denomina rutina de eliminación y se lleva a cabo comparando los atributos de los caminos recién expandidos con los de los caminos listados (un atributo puede ser por ejemplo, el costo del camino).

El modelo general de algoritmo de etiquetamiento no especifica condiciones de parada ni reglas de elección del próximo camino abierto a ser expandido: son estos puntos los que caracterizan cada algoritmo particular. Los casos particulares que esudiaremos evitan en todo momento, mediante la rutina de eliminación de caminos, la presencia de dos caminos listados (abierto o cerrado) con el mismo nodo terminal

Abrir un camino significa colocarlo en la lista con la etiqueta "abierto". De igual forma cerrar un camino significa colocarlo en la lista con la etiqueta "cerrado". Diremos que un vértice v está "abierto" ("cerrado") si existe

un camino en la lista con la etiqueta "abierto" ("cerrado") y nodo terminal v.

Cuando hablamos en términos de "lista" su acepción es la comúnmente utilizada en computación, es decir, una secuencia de objetos donde podemos distinguir el primer elemento y el resto.

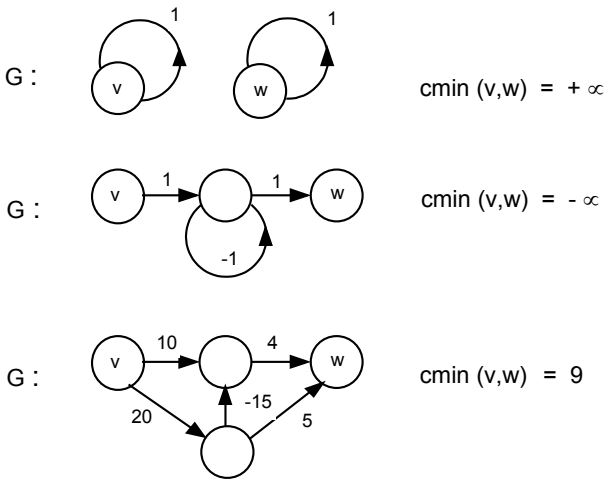



**figura IV.2**

Modelo general de algoritmo:
{ Entrada: *G=(V,E)* un grafo dirigido sin arcos múltiples y $s \in V$.
Salida: Una lista de caminos cerrados que parten de s.}
Comienzo
(0) Inicialmente la lista de caminos está vacía.
Calcular los atributos del camino <*s* > y luego abrirlo;
Mientras "Existan caminos abiertos" hacer:
Comienzo
(1) Escoger un camino abierto $P_j$=<*s* , ... ,$n_j$>;
(2) Cerrar $P_j$ (cambiarle la etiqueta "abierto" por "cerrado");
(3) Obtener los sucesores de $n_j$: $n^1$, ... ,$n^q$ ;
(4) Construir-caminos: $P^i$=Exp($P_j$,$n^i$ )=$P_j$ || <$n_j$,e,$n^i$ >
donde e=($n_j$ ,$n^i$ );
(5) Calcular los atributos de cada $P^i$;
(6) Para cada camino $P^i$ hacer:
Ejecutar la rutina de eliminación de caminos a $P^i$
fin
fin.

**Observaciones:**

(1) Dependiendo de las reglas de elección que se den en el paso (1), de los atributos que se calculen en el paso (5) y de la rutina de eliminación del paso (6), obtendremos diferentes algoritmos.

(2) En el paso (4), Exp($P_j$,$n^i$) es la expansión de $P_j$ a $n^i$ (ver sección III.1.2).

(3) El paso (6) consiste en examinar la lista de caminos y en caso de que exista un camino (abierto o cerrado) cuyo vértice terminal sea igual al de $P^i$, dependiendo de las reglas de eliminación se decidirá reemplazar o no el camino en la lista por $P^i$. Si $P^i$ no sustituye al camino en la lista, se desecha. Si $P^i$ sustituye al camino de la lista, se elimina este último de la lista, y se abre $P^i$.

Si al examinar la lista de caminos en el paso (6), no existe un camino en la lista con igual nodo terminal al de $P^i$ entonces se abre $P^i$.

(4) Note que, en la lista de caminos no es siempre necesario tener por cada camino toda la sucesión de vértices que lo define. Según el paso (4) basta con representar al camino $P^i$ por el nodo terminal $n^i$ del camino y una referencia (o apuntador) al camino $P_j$. De esta manera, al implementar el algoritmo se ahorrará considerablemente la memoria. Así un camino $P^i$ se representa por una terna: ($n^i$,atributos,j), donde $n^i$ es el nodo terminal de $P^i$, "atributos" son los atributos asociados al camino que permitirán hacer la comparación de éste con otros caminos listados en la rutina de eliminación, y j es la referencia al camino $P_j$ con $P^i = Exp(P_j,n^i)$, es decir, $P^i$ se obtiene de la expansión de $P_j$ a $n^i$ .

El problema que podría presentarse con esta representación recursiva de caminos es que se pierdan trechos de caminos pues los caminos cerrados podrían ser eliminados por la regla de eliminación. Sin embargo, para los algoritmos particulares que trataremos más adelante, esto no pasará. Demos un ejemplo para ilustrar la situación:

Supongamos que se han listado, aplicando el modelo de algoritmo, los siguientes caminos del grafo de la figura IV.3 (recordemos que el primer camino que se abre es $P_0$=(s,xxx,-)) :

$P_0$=(s,xxx,-) cerrado.

$P_1$=(b,xxx,0) cerrado.

$P_2$=(d,xxx,1) abierto.

$P_3$=(c,xxx,0) abierto.

Note que $P_2$ puede ser reconstruído a partir de la terna (d,xxx,1) que lo representa. Según la terna, $P_2=Exp(P_1,d)= Exp(Exp(P_0,b),d) = Exp(Exp(<s>,b),d) = Exp(<s,b>,d) = <s,b,d>$.

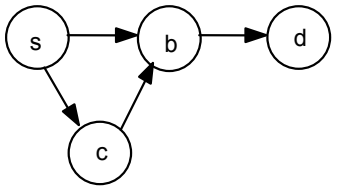



**figura IV.3**

Si en la siguiente iteración del algoritmo se expande $P_3$ se genera el camino $P_4$=(b,xxx,3) el cual debe ser comparado con $P_1$ en el paso (6). Si $P_4$ elimina a $P_1$ esto significa que desaparece $P_1$ de la lista y se coloca a $P_4$. Tendríamos la configuración siguiente de la lista:

$P_0$=(s,xxx, -) cerrado.

$P_4$=(b,xxx,3) abierto.

$P_2$=(d,xxx,1) abierto.

$P_3$=(c,xxx,0) abierto.

De esta forma se ha perdido un trecho del camino $P_2$=(d,xxx,1) pues $P_1$ no aparece en la lista.

Por otro lado, es possible obtener a partir del modelo un algoritmo cuya ejecución no termina. Por ejemplo, si el criterio de la rutina de eliminación fuera "colocar en la lista de caminos abiertos al camino considerado $P^i$ y de existir otro camino listado (abierto o cerrado) con igual nodo terminal eliminar este ultimo de la lista", entonces con el grafo siguiente el algoritmo no pararía:

En las secciones siguientes estudiaremos lo que pasa en cada algoritmo particular que trataremos.

(5) Obsérvese que la situación anterior no sucedería si los caminos estuvieran representados por la secuencia de vértices que lo conforman pues $P_2$ estaría representado por <s,b,d> antes y después de colocar a $P_4$ en la lista.

(6) El modelo general no precisa condiciones de parada en función del conjunto D (el conjunto definido en la presentación del modelo general, sección IV.1.1). Estas condiciones serán precisadas en cada algoritmo particular.

(7) La lista de caminos abiertos y cerrados puede ser dividida en dos listas. Una lista de caminos abiertos y otra de caminos cerrados. De esta forma no hablaríamos de caminos etiquetados pues sabremos que un camino está abierto o cerrado dependiendo de la lista en la cual se encuentre. Con esta modalidad el modelo general sería:

Modelo general de algoritmo:
{Entrada: $G=(V,E)$ un grafo dirigido sin arcos múltiples y $s \in V$.
Salida: Una lista de caminos cerrados que parten de s.}
Comienzo
(0) Inicialmente las listas de abiertos y cerrados están vacías. Calcular los atributos del camino $<s>$ y colocarlo en la lista de abiertos;
Mientras la lista de abiertos no esté vacía hacer:
Comienzo
(1) Escoger un camino $P_j=<s, \ldots, n_j>$ de la lista de abiertos;
(2) Eliminar a $P_j$ de la lista de abiertos y colocarlo en la lista de cerrados;
(3) Obtener los sucesores de $n_j$: $n^1, \ldots, n^q$ ;
(4) Construir los caminos:
$P^i=Exp(P_j,n^i)=P_j \,||<n_j, e, n^i>$ donde $e=(n_j, n^i)$ ;
(5) Calcular los atributos de cada $P^i$;
(6) Para cada camino $P^i$ hacer:
Ejecutar la rutina de eliminación a $P^i$;
fin
fin.

(8) Un vértice cualquiera n no puede ser nodo terminal de dos caminos listados (abiertos o cerrados) y desde la primera vez que se coloca en la lista un camino con nodo terminal n, siempre existirá en la lista un camino hasta ese nodo:

Suponga que, en un momento dado, un nodo es terminal de dos caminos listados. Si consideramos la primera iteración en la cual esto ocurre, en esa iteración uno de los dos caminos fue el resultado de la expansión a n y, por lo tanto, en la rutina de eliminación uno de los caminos es eliminado, no pudiendo así coexistir dos caminos en la lista con el mismo nodo terminal. Por otro lado, siempre que un camino con nodo terminal n es eliminado de la lista, éste es reemplazado por otro camino con nodo terminal n.

## IV.2 RECORRIDOS EN GRAFOS

En esta sección presentamos dos técnicas que nos permiten "visitar" los vértices de un grafo a través de caminos. Estas técnicas de recorrido son de gran utilidad en la resolución de muchos problemas en grafos. Su importancia se deriva del hecho que los caminos que se generan en el recorrido determinan una estructura muy particular del grafo. Las técnicas se denominan Búsqueda en Profundidad (Depth First Search) y Búsqueda en Amplitud (Breadth First Search). Haremos una presentación de estas técnicas como algoritmos particulares del modelo general de etiquetamiento.

### IV.2.1 BÚSQUEDA EN PROFUNDIDAD

Este caso particular del modelo general consiste en generar un camino desde s hasta cada vértice de G alcanzable desde s; no asocia atributos a los caminos. Los caminos que se generan dependen de la regla de elección de los abiertos a cerrar. En cada iteración, si el camino escogido se expande, es decir, posee sucesores, entonces en la fase de aplicación de la regla de eliminación cada camino C resultante de la expansión se compara con los caminos listados: si existe un camino listado "cerrado" con nodo terminal igual al nodo terminal de C, entonces el camino C es desechado; si existe una camino listado "abierto" con nodo terminal igual al nodo terminal de C, entonces se elimina el camino abierto de la lista y se abre el camino C; si no existe un camino listado con nodo terminal igual al nodo terminal de C entonces se abre C. En la siguiente iteración será escogido para expandir (y cerrar), uno de los caminos abiertos resultantes de la última expansión que se haya realizado. Por lo tanto, el criterio de elección del abierto a ser cerrado, es: escoja el último abierto colocado en la lista (último en entrar, primero en salir).

A continuación presentamos el algoritmo:

Búsqueda en Profundidad (Versión 1):
{ Entrada: Un grafo dirigido G=(V,E) sin arcos múltiples y s∈V.
Salida: Una lista de caminos que parten de s hacia cada vértice alcanzable desde s. }
Comienzo
(0) Abrir el camino $P_0$=<s>;
(1) Mientras Existan caminos abiertos hacer :
Comienzo
(1.1) Escoja el último camino abierto $P_j$=<s,...,$n_j$> colocado en la lista;
(1.2) Cierre $P_j$;
(1.3) Obtenga los sucesores de $n_j$:
$n^1, n^2, \ldots, n^q$;
(1.4) Construya los caminos expandidos:
$P^i$=Exp($P_j$,$n^i$), i=1,...,q;
(1.5) Aplique la rutina de eliminación de caminos;
fin

Donde
Rutina de eliminación:
Comienzo
Para cada $P^i$, i=1,...,q, hacer:
Comienzo
Si no existe un $P_k$ en la lista con nodo terminal igual a $n^i$ entonces Abrir $P^i$
si no
Si $P_k$ es abierto entonces
Comienzo
Eliminar a $P_k$ de la lista;
Abrir $P^i$
fin;
fin
fin
fin.

Ejemplo: Sea G=(V,E) el grafo de la figura IV.4 con s=1.

Inicialmente $P_0$=<s>.

Al expandir a $P_0$ obtendremos en la primera iteración la lista siguiente (supondremos que los sucesores de un vértice son considerados en orden creciente del número asociado a él):

$P_0$=<1> cerrado.
$P_1$=<1,2> abierto.
$P_2$=<1,3> abierto.
$P_3$=<1,4> abierto.

En la próxima iteración el abierto a expandir es $P_3$. Al finalizar la iteración tendremos:

$P_0$=<1> cerrado.
$P_1$=<1,2> abierto.
$P_2$=<1,3> abierto.
$P_3$=<1,4> cerrado.
$P_4$ =<1,4,6> abierto.

En la próxima iteración escogemos para expandir a $P_4$ (último abierto colocado en la lista). Al finalizar la iteración tendremos:

$P_0$=<1> cerrado.
$P_1$=<1,2> abierto.
$P_2$=<1,3> abierto.
$P_3$=<1,4> cerrado.
$P_4$ =<1,4,6> cerrado.

Note que, como el vértice 6 no tiene sucesores, no se generan nuevos caminos.
En la próxima iteración el último abierto colocado en la lista es $P_2$ y al final de esa iteración tendremos:

$P_0$=<1> cerrado.
$P_2$=<1,3> cerrado.
$P_3$=<1,4> cerrado.
$P_4$ =<1,4,6> cerrado.
$P_5$=<1,3,2> abierto.

En este caso, en la rutina de eliminación el camino $P_5$ reemplazó al camino $P_1$ y el camino expandido <1,3,3> obtenido de la expansión de $P_2$ fue desechado pues $P_2$ estaba cerrado.
Ahora expandemos a $P_5$ y obtenemos al final de la iteración:

$P_0$=<1> cerrado.
$P_2$=<1,3> cerrado.
$P_3$=<1,4> cerrado.
$P_4$ =<1,4,6> cerrado.
$P_5$=<1,3,2> cerrado.
$P_6$=<1,3,2,5> abierto.

En esta iteración el camino <1,3,2,6> fue desechado y $P_6$ fue abierto. Al escoger a $P_6$, obtenemos al final de la iteración:

$P_0$=<1> cerrado.
$P_2$=<1,3> cerrado.
$P_3$=<1,4> cerrado.
$P_4$ =<1,4,6> cerrado.
$P_5$=<1,3,2> cerrado.
$P_6$=<1,3,2,5> cerrado.

Por lo tanto el algoritmo termina.

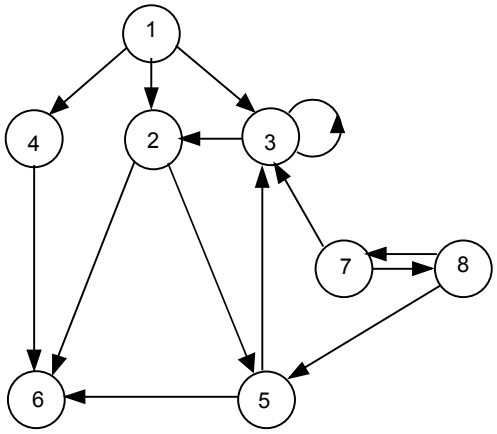



**figura IV.4**

Note que el camino $P_1$ fue reemplazado por $P_5$. $P_5$ es un camino de mayor "profundidad" (más largo) que $P_1$ llegando al nodo 2.

Al final del algoritmo obtenemos un único camino desde s=1 hasta cada vértice alcanzable desde 1. De acuerdo a la proposición III.3.4.8, el conjunto formado por la unión de los arcos de los caminos genera en G una arborescencia con raíz s; ver figura IV.5.

Propiedades del algoritmo de búsqueda en profundidad:

(1) Una manera de implementar el criterio de elección: "tome el último abierto colocado en la lista", sería considerar dos listas, una de caminos cerrados y otra de abiertos. La lista de abiertos puede ser manipulada como una pila (último que entra, primero que sale). Al comienzo de cada iteración, se escoge el tope de la pila y cada nuevo camino abierto se coloca en el tope de la pila. Sin embargo el orden en que son colocados en la lista de abiertos los caminos resultantes de la expansión en el paso (1.5) es arbitrario (depende del orden en que son considerados los sucesores de cada vértice).

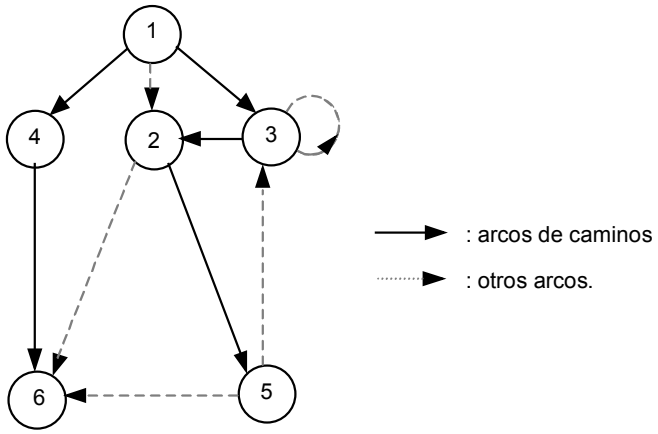



**figura IV.5**

(2) Un vértice cualquiera no puede ser nodo teminal de dos caminos listados (propiedad (8) del modelo general).

(3) Un camino cerrado nunca puede ser eliminado: Esto es una consecuencia de la rutina de eliminación.

(4) Si $P=<s,e_1,x_1,...,e_n,x_n>$, $(n\geq 1)$, es un camino listado, entonces los caminos $<s,e_1,x_1,...,e_i,x_i>$, i=0,...,n-1, son todos caminos listados cerrados (para i=0, el camino es <s>):

En la iteración donde P se obtuvo de una expansión, se tiene que $P=Exp(<s,e_1,x_1,...,e_{n-1},x_{n-1}>,x_n)$, con $<s,e_1,x_1,...,e_{n-1},x_{n-1}>$ listado y cerrado. Ahora bien, como este último camino está listado, aplicando inductivamente el razonamiento anterior y la propiedad (3), llegamos a que $<s,e_1,x_1,...,e_i,x_i>$, i=0,...,n-1, son todos caminos listados cerrados (para i=0, el camino es <s>).

(5) No pueden existir dos caminos listados conteniendo un nodo en común n y tal que el nodo predecesor de n en uno de los caminos sea diferente al predecesor de n en el otro camino:

Supongamos que $P=<s, e_1, x_1,..., e_i, n,..., e_p, x_p>$ y $Q = <s,e'_1,x'_1, \dots ,e'_j, n, \dots ,e'_q,x'_q>$ sean dos caminos listados. Según la propiedad (4) los caminos $<s, e_1, x_1,..., e_i, n>$ y $<s,e'_1,x'_1, \dots ,e'_j, n >$ son caminos listados diferentes con nodo terminal n. Esto último contradice la propiedad (2).

(6) El algoritmo de búsqueda en profundidad termina en un número de iteraciones igual al número de nodos alcanzables desde s. En cada iteración el número de operaciones viene dado por el grado del vértice terminal del camino cerrado en esa iteración, multiplicado por un factor generado por la búsqueda en la rutina de eliminaciones.

Más adelante veremos que este factor de la búsqueda puede ser reducido a orden 1, lo cual nos permite concluir que el orden del algoritmo de búsqueda en profundidad puede ser O(max(número de arcos, número de vértices)) con una implementación adecuada. Cuando el algoritmo termina hay un camino cerrado partiendo de s hacia cada nodo alcanzable desde s; además, el grafo H formado por los vértices alcanzables desde s y los arcos de los caminos cerrados al final del algoritmo, es una arborescencia con raíz s, la cual llamaremos *arborescencia de la búsqueda en profundidad*:

Veamos formalmente que el número de iteraciones del algoritmo es igual al número de vértices alcanzables desde s y que al concluir el algoritmo existirá un camino cerrado hasta cada nodo alcanzable desde s :

Como los caminos cerrados no son eliminados y en cada iteración los nodos terminales de los caminos listados son todos distintos entre sí, podemos concluir que el número de iteraciones del algoritmo será a lo sumo igual al número de vértices alcanzables desde s.

Ahora bien, sea n un nodo alcanzable desde s y $<n_0,e_1,n_1,...,e_k,n_k>$, con $s=n_0$ y $n=n_k$ , un camino de s a n en G. Si al comienzo de una iteración cualquiera no existe un camino cerrado con nodo terminal n, entonces existirá un camino abierto con nodo terminal $n_i$ para algún i entre 0 y k. Esta última afirmación, que demostraremos más adelante, nos garantiza que al concluir el algoritmo, por cada vértice alcanzable desde s existirá un camino cerrado con nodo terminal igual a n, ya que al concluir el algoritmo no existirán caminos abiertos y, por un proceso inductivo, podemos decir que existirán entonces caminos cerrados con nodos terminales $n_1$, $n_2$, ... , $n_k$ (=n).

Veamos entonces que si al comienzo de una iteración cualquiera no existe un camino cerrado con nodo terminal n, entonces existirá un abierto con nodo terminal $n_i$, para algún i entre 0 y k :

En la primera iteración es cierto pues al comienzo de ésta el camino <s> está abierto.

En una iteración j distinta a la primera, supongamos que no hay cerrado con nodo terminal n. Sea i el mayor índice tal que exista un camino $P_i$ cerrado con nodo terminal $n_i$. Este i existe pues es al menos 0. En la iteración en que fue cerrado $P_i$, este camino se expandió a $n_{i+1}$ ($P_{i+1} = Exp(P_i,n_{i+1})$). En la rutina de eliminación, como no existía cerrado con nodo terminal $n_{i+1}$, se tuvo que abrir $P_{i+1}$ y este camino permanece abierto o es reemplazado por otro abierto con nodo terminal $n_{i+1}$, por la elección que se hizo de i.

(7) Consideraciones de implementación:
Ya vimos en la propiedad (1) que la lista de caminos puede ser dividida en dos listas: una de caminos cerrados y otra de caminos abiertos, ésta última considerada como una pila (último en entrar primero en salir). Los caminos listados pueden ser representados en un formato recursivo, tal como lo sugerimos en la observación (4) del modelo general (sección IV.1.2):
El elemento $P_i=(n,j)$ representa un camino listado donde:
n:es el vértice terminal del camino.
j: es un apuntador a otro elemento listado , con $P_i=Exp(Pj,n)$.

Obsérvese que en el algoritmo de búsqueda en profundidad no se pierden trechos de caminos (lo cual puede pasar en el modelo general) pues los caminos cerrados no son eliminados.

Con esta forma de representación se ve más claro que al terminar el algoritmo, el conjunto de caminos representa una arborescencia pues el único predecesor de n es el vértice terminal de $P_j$.

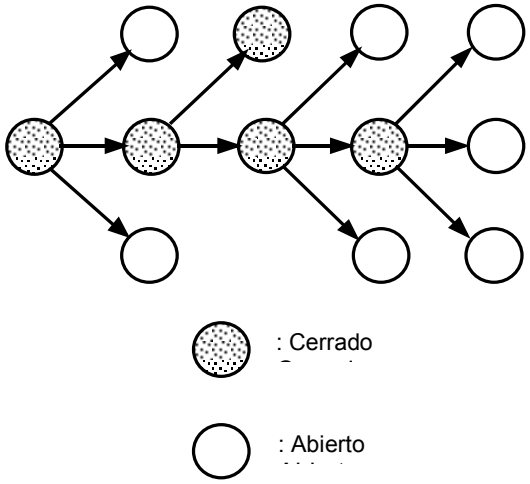
Al terminar el algoritmo podemos recuperar recursivamente los caminos hasta cada vértice, a través de los apuntadores.

Búsqueda en profundidad (versión 2):
{ Entrada: Un grafo dirigido sin arcos múltiples G=(V,E) y $s \in V$
Salida: una lista de caminos de s a cada vértice alcanzable desde s. }
Comienzo:
(0) Abra $P_0=(s,0)$;
(1) Mientras Existan elementos abiertos hacer
Comienzo

(1.1) Escoja el último elemento abierto $P_j=(n_j,k)$;
(1.2) Cierre $P_j$;
(1.3) Obtenga los sucesores de $n_j$: $n^1,n^2, \ldots ,n^q$;
(1.4) Construya los elementos $P^i=(n^i,j)$, i=1, ... ,q;
(1.5) Aplique la rutina de eliminación de elementos
fin
(2) Construya recursivamente los caminos a partir de los elementos listados
fin

(8) Como un vértice tiene asociado a él a lo sumo un camino listado, podemos pensar en la búsqueda en profundidad como un procedimiento particular para "*visitar"* los vértices de un grafo. Un vértice es visitado cuando se cierra el camino listado que llega a ese vértice. De aquí el término *recorridos* en grafos: damos un método para visitar los vértices del grafo recorriéndolo a través de caminos.

(9) Si definimos *la profundidad* de un vértice terminal de un camino listado como el largo del camino, entonces en cada iteración se escoge un camino abierto con vértice terminal de profundidad máxima, de aquí el término "búsqueda en profundidad".




Organización de los elementos abiertos en una iteración cualquiera

**figura IV.6**

Considere que utilizamos la implementación en (7). Al concluir la primera iteración, todos los abiertos que se generan apuntan a s y el único abierto al comienzo de esa iteración es <s> y por lo tanto es de profundidad máxima entre todos los abiertos. Tomemos como hipótesis inductive que en la iteración k (≥ 1) se cierra un elemento abierto P de máxima profundidad entre todos los elementos abiertos y sea $P_0$ , $P_1$, ... , $P_q$ la cadena de elementos cerrados obtenidos siguiendo los apuntadores a partir de P, $P_0=(s,0)$ y $P_q=P$, entonces al finalizar esa iteración, todos los elementos abiertos apuntan a elementos de la cadena $P_0$, $P_1$, ... , $P_q$ . Note que los abiertos que se generan en esa iteración serán de máxima profundidad entre todos los abiertos pues P era de máxima profundidad. Al comienzo de la iteración k+1, al cerrar P' (el último abierto generado), éste será de máxima profundidad entre todos los abiertos pues, por hipótesis inductiva, en la última iteración donde se generaron abiertos, éstos eran de máxima profundidad entre todos los abiertos. Ahora bien, en la última iteración en donde se generaron abiertos, sea $P_0$, $P_1$, ... , $P_q$ la cadena de elementos obtenidos siguiendo los apuntadores del camino P que se cerró en esa iteración. Tenemos que P' apunta a $P_q$ . Por lo tanto, al finalizar la iteración k+1, los abiertos que se generan apuntarán a P' y serán de máxima profundidad. Note que si antes de aplicar la rutina de eliminación en la iteración k+1, después de cerrar P', existía un abierto con vértice terminal igual al vértice terminal de un camino expandido a partir de P', entonces ese

abierto será reemplazado por este camino expandido. Cualquier otro abierto que no fue reemplazado apunta a $P_0$ , $P_1$, ... , $P_q$ .

En conclusión, en cada iteración del algoritmo sucede lo siguiente:

(a) Se cierra un camino de máxima profundidad entre todos los abiertos.

(b) Si P es el elemento que se cierra en esa iteración y $P_0$, $P_1$,…, $P_q$ son los elementos que se obtienen a partir de P, siguiendo los apuntadores ($P_0$=(s,0), $P_q$=P), entonces al finalizar la iteración todo elemento abierto apunta a algún elemento de $P_0$, $P_1$, ... , $P_q$ .

(c) Los abiertos que se generan en esa iteración serán de máxima profundidad entre todos los abiertos.

(d) Dos elementos abiertos con igual profundidad apuntan al mismo elemento.

Este formato particular nos permite probar que si el largo máximo de un camino elemental partiendo de s es *L*, entonces el número de abiertos en cualquier iteración no puede superar $(\Delta(G)-1).L$ +1 (vea la figura IV.6).

### IV.2.2 ARBORESCENCIA DE LA BÚSQUEDA EN PROFUNDIDAD

Sea G=(V,E) un grafo dirigido sin arcos múltiples, sea $s \in V$ y aplicamos el algoritmo de búsqueda en profundidad partiendo de s. En el punto (6) de la sección IV.2.1 definimos la arborescencia de la búsqueda en profundidad como el subgrafo engendrado por los arcos de los caminos cerrados resultantes del algoritmo. Podríamos decir que estos son los arcos recorridos por el algoritmo. Allí mismo mostramos que efectivamente el subgrafo descrito es una arborescencia.

En esta sección haremos una clasificación de los arcos del grafo, inducida por el algoritmo de búsqueda en profundidad.

Note que es en el momento de cerrar un camino en el algoritmo de búsqueda en profundidad, que el arco terminal de dicho camino pasa a formar parte de la arborescencia de la búsqueda en profundidad. Si el grafo es fuertemente conexo la arborescencia contendrá necesariamente todos los vértices del grafo G pues según el punto (6) de la sección IV.2.1, al final del algoritmo habrá un camino cerrado hasta cada vértice alcanzable desde s.

Si el grafo no es fuertemente conexo los vértices no alcanzables desde s no formarán parte de la arborescencia.

Diremos que un vértice v ha sido *visitado* por el algoritmo de búsqueda en profundidad si v es vértice terminal de algún camino cerrado generado por el algoritmo. Diremos que v *se visita* al momento en que se cierra el camino con vértice terminal v. Para visitar todos los vértices del grafo G podemos aplicar el algoritmo de búsqueda en profundidad de la siguiente forma:

Visita de vértices de G aplicando búsqueda en profundidad:
Comienzo
(1) Inicialmente ningún vértice ha sido visitado.
(2) Para cada vértice v de G hacer:
    Si v no ha sido visitado entonces aplicar la búsqueda en profundidad a partir de v.
fin

Cada vértice será visitado una sola vez y esto permite efectuar algún proceso a cada vértice de G. El algoritmo anterior efectúa varias llamadas al algoritmo de búsqueda en profundidad y podemos suponer que la lista de caminos abiertos y cerrados es una sola pues los caminos cerrados no son reemplazados. Por lo tanto, la arborescencia que se genera en cada llamada no podrá ser modificada por las llamadas posteriores. De esta forma, al finalizar el algoritmo se habrá visitado todos los vértices del grafo G. En cuanto a la complejidad de este algoritmo vemos que el número de iteraciones en cada llamada al algoritmo de búsqueda es igual al número de vértices visitados en esa llamada estos vértices son los alcanzables desde v. Así el número total de iteraciones producidas por las sucesivas llamadas al algoritmo de búsqueda será igual al número de vértices del grafo pues cada vértice se visita una sola vez. En cada iteración del algoritmo de búsqueda en profundidad el número de operaciones efectuadas es proporcional al grado de salida ($d^+$) del vértice visitado en esa iteración pues al expandir el camino se trata a cada sucesor del vértice (veremos más adelante que la rutina de eliminación puede ser $O$ ($d^+$) con una implementación adecuada). Se puede concluir entonces que el número total de operaciones del algoritmo de visita de los vértices de G es $O$ (max(|V(G)|,|E(G)|)).

Sea A el conjunto de los arcos de los caminos cerrados resultantes al terminar el algoritmo, el digrafo

(V(G),A) es un bosque cobertor de G. Cada componente conexa del bosque es una arborescencia. Los arcos de G se pueden clasificar de la siguiente manera, una vez aplicado el algoritmo de visita de vértices:

(a) Los *arcos del bosque*.

(b) Los *arcos de ida*: Son los que parten de un vértice a un descendiente del vértice en el bosque y no son arcos del bosque.

(c) Los *arcos de vuelta o de retorno*: Son los que parten de un vértice a uno de sus ascendientes en el bosque.

(d) Los *arcos cruzados*: Son los arcos cuyo vértice inicial no es ni ascendiente ni descendiente del vértice terminal en el bosque.

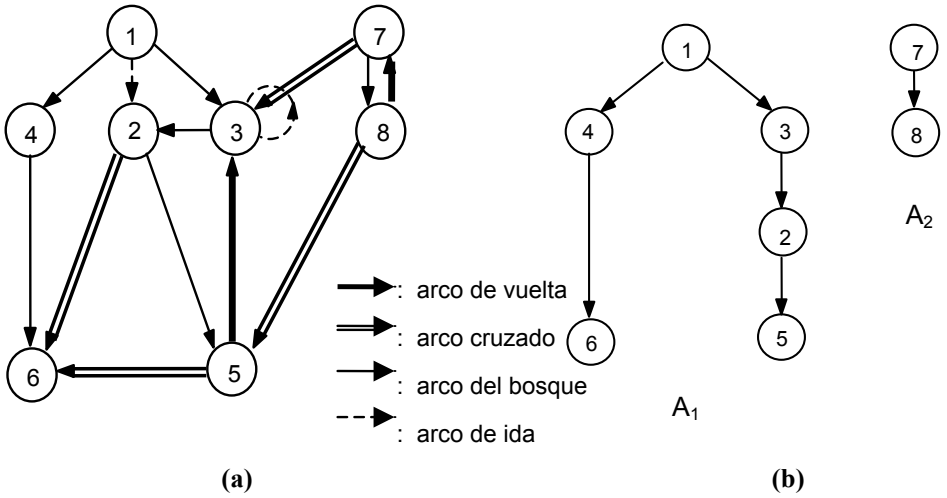



**figura IV.7**

Los bucles serán considerados tanto arcos de ida como de vuelta pues todo vértice es ascendiente y descendiente de sí mismo. Si el grafo no posee bucles, la clasificación anterior constituye una partición de los arcos de G. La figura IV.7(a) muestra la clasificación de los arcos resultante de aplicar el algoritmo de visita de vértices al grafo de la figura IV.7. El bucle se ha podido catalogar como arco de vuelta. La figura IV.7(b) muestra el bosque generado, donde $A_1$ es la arborescencia generada en la primera llamada al algoritmo de búsqueda y $A_2$ es la generada en la segunda llamada.

Daremos a continuación algunas propiedades de los diferentes tipos de arcos. Supongamos que a cada vértice del grafo G, le asociamos el *número de búsqueda* el cual nos indica el orden en que fueron visitados los vértices, la numeración comenzando desde 1. El número de búsqueda asociado a un vértice v lo denotamos por *NumBusq(v)*.

En el ejemplo de la figura IV.7 tenemos:

| Vértice: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| NumBusq: | 1 | 5 | 4 | 2 | 6 | 3 | 7 | 8 |

**Proposición IV.2.2.1**

Sea B el bosque generado por el algoritmo de visita de vértices aplicado al grafo G. Para todo descendiente w de un vértice v en B tenemos que: $NumBusq(v) \leq NumBusq(w)$.

**Demostración:**

Al momento en que se cierra el camino $<w_0,e_1,w_1,..,e_m,w_m=w>$ con vértice terminal w, sabemos que todos los caminos $<w_0,e_1,w_1,..,e_i,w_i>$, i=0,1,2,..., están cerrados y uno de ellos posee como vértice terminal a v pues w es descendiente de v en B. El camino hasta v fue cerrado antes y así v fue visitado antes, lo que significa que $NumBusq(v) \leq NumBusq(w)$.

¤

**Proposición IV.2.2.2**

Sea e=(x,y) un arco de G y B el bosque generado por el algoritmo de visita de vértices aplicado a G:

(a) Si e es una arco del bosque o un arco de ida, entonces NumBusq(x) ≤ NumBusq(y).

(b) Si e es un arco de vuelta o cruzado, entonces NumBusq(x) ≥ NumBusq(y). Si e es cruzado entonces la desigualdad es estricta.

**Demostración:**

La parte (a) es consecuencia de la proposición IV.2.2.1.

Para mostrar la parte (b) basta con probar que si NumBusq(x) ≤ NumBusq(y) entonces e es un arco del bosque o un arco de ida.

Si NumBusq(x) < NumBusq(y), entonces x fue visitado antes que y. En la iteración en que se visita x, sea C el camino que se cierra con vértice terminal x. En esa iteración se abre el camino $C_1$ = C || <x, (x,y), y> . Ahora bien, en una iteración posterior puede pasar una de las dos situaciones siguientes:

(a) Se cierra $C_1$, en cuyo caso (x,y) es un arco del bosque .

(b) Se reemplaza a $C_1$ por otro abierto. En este caso, mostremos que el primer camino $C_2$ que reemplaza a $C_1$ como abierto, necesariamente pasa por x. De esta forma, por un razonamiento inductivo, podemos probar que en las siguientes iteraciones, los sucesivos caminos abiertos que se generan, con vértice terminal y, necesariamente pasan por x; con lo cual, como al final del algoritmo no existen caminos abiertos, el camino cerrado con vértice terminal y pasa por x, resultando y descendiente de x en el bosque, es decir, (x,y) es un arco de ida. Veamos pues que $C_2$ necesariamente pasa por x. En la iteración donde $C_1$ es reemplazado por $C_2$, tenemos que al comienzo de esta iteración $C_1$ está abierto y, por lo tanto, de acuerdo con la propiedad (9) caso (b) sección IV.2.1, el camino $C_3$ que se cierra en esa iteración pasa por x, es decir, es de la forma <s, ... , x, ... , t>. Como $C_2$ reemplaza a $C_1$ en esa iteración, $C_2$ es igual a $C_3$ || <t,(t,y),y>. En conclusión, $C_2$ pasa por x.

¤

Sean $A_1$, $A_2$, ... , $A_p$ Las arborescencias (componentes conexas) del bosque B generado por el algoritmo de visita de vértices aplicado a un grafo G; $A_i$ fue generada en la i-ésima llamada al algoritmo de búsqueda en profundidad. De la proposición IV.2.2.2 podemos asegurar que si existe un arco entre un vértice v de $A_i$ y uno w de $A_j$, con i<j, entonces este arco es (w,v) y es un arco cruzado. También esto nos dice que ningún vértice en $A_{i+1},...,A_p$ es alcanzable desde un vértice en $A_1,...,A_i$, $\forall i: 1 \leq i \leq p-1$.

**Proposición IV.2.2.3**

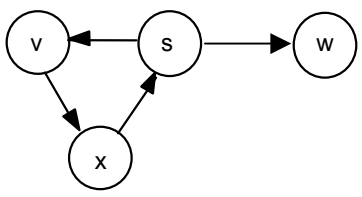
Sean v, w vértices de un digrafo G sin arcos múltiples. Si al momento de visitar v existe un camino de v a w en el cual ninguno de los vértices ha sido visitado, entonces w será descendiente de v en el bosque generado por el algoritmo de búsqueda en profundidad.

**Demostración:**

Esto se debe a que al momento de visitar v, el vértice w es alcanzable desde v en el subgrafo inducido por los vértices no visitados. En efecto, el bosque generado por el algoritmo de búsqueda en profundidad eliminando los vertices visitados y comenzando por v es un subgrafo del bosque original

¤

Note que si existe un camino de v a w en un digrafo G sin arcos múltiples y al aplicar el algoritmo de visita de vértices a G el vértice v es visitado antes que w, entonces w no necesariamente es un descendiente de v en el bosque generado por el algoritmo. Ejemplo:



### IV.2.3 BÚSQUEDA EN PROFUNDIDAD EN GRAFOS NO ORIENTADOS

Si G es un grafo no orientado sin aristas múltiples, para recorrer este grafo mediante el algoritmo de búsqueda en profundidad, lo que hacemos es aplicar el algoritmo al grafo dirigido asociado a G. Al aplicar el algoritmo de visita de vértices al grafo dirigido asociado a G se obtienen los siguientes resultados:

**Proposición IV.2.3.1**

Al aplicar el algoritmo de visita de vértices al grafo dirigido asociado a un grafo no dirigido G sin aristas múltiples se tiene:

(a) No habrá arcos cruzados.

(b) Para cada arco e=(x,y), $x \neq y$, de ida, existirá el arco e'=(y,x) de vuelta y e será "recorrida" primero que e' por el algoritmo, es decir, el algoritmo genera primero el camino con arco terminal e y luego genera el camino con arco terminal e'.

(c) Por cada arco e=(x,y) del bosque, existe el arco de vuelta e'=(y,x).

**Demostración:**

Parte (a):
Supongamos que existe un arco cruzado e=(x,y). Según la proposición IV.2.2.2, NumBusq(x) > NumBusq(y). Como existe el arco e'=(y,x), éste es cruzado y debe satisfacer NumBusq(y) > NumBusq(x), lo cual contradice la primera desigualdad.

Parte (b):
Si e=(x,y) es un arco de ida sabemos que existe e'=(y,x) y por lo tanto será un arco de vuelta. Como x es visitado antes que y, al momento de expandir el camino cerrado C con vértice terminal x, se abre el camino $C_1 = C \,||\, \langle x,e,y\rangle$. En una iteración posterior se cierra un camino con vértice terminal y; sea $C_2$ este camino. Es esa iteración ya $C_1$ ha sido eliminado ($C_1$ es distinto de $C_2$ por ser (x,y) un arco de ida y no pueden haber dos caminos listados con igual vértice terminal y ) y $C_2$ es expandido hasta x (es decir, se considera el camino $C_2 \,||\, \langle y,e',x\rangle$)

Parte (c): Es inmediata.

¤

La propiedad IV.2.3.1(a) indica que si $A_1,...,A_p$ son las arborescencias generadas por el algoritmo, entonces $G_{V(A_1)}, \ldots, G_{V(A_p)}$ son las componentes conexas de G.

En caso de que G no sea orientado, el algoritmo de visita de vértices lo podemos ver como un algoritmo de búsqueda de cadenas donde el conjunto de las cadenas obtenidas al final del algoritmo conforman un bosque del grafo, los vértices de cada componente conexa del bosque determinan una componente conexa del grafo y el orden en que son visitados los vértices viene dado por el número NumBusq. Cada arista del grafo e={x,y} será recorrida dos veces: al expandir los caminos con vértice terminal x e y respectivamente.

A continuación damos la versión del algoritmo de búsqueda en profundidad para grafos no orientados sin tener que pasar por el grafo orientado asociado:

Búsqueda en Profundidad para grafos no dirigidos:
{ Entrada: Un grafo no dirigido G=(V,E) sin aristas múltiples y $s \in V$.
Salida: Una lista de cadenas que parten de s hacia cada vértice alcanzable desde s. }
Comienzo
(0) Abrir la cadena $P_0=\langle s\rangle$;
(1) Mientras Existan cadenas abiertas hacer :
Comienzo
(1.1) Escoja la última cadena abierta $P_j=\langle s,...,n_j\rangle$ de la lista;
(1.2) Cierre $P_j$;
(1.3) Obtenga los vecinos de $n_j$: $n^1, n^2, \ldots, n^q$;
(1.4) Construya las cadenas expandidas:

$P^i$=Exp($P_j$,$n^i$), i=1,...,q;

(1.5) Aplique la rutina de eliminación de cadenas;

fin

fin

La rutina de eliminación de cadenas es idéntica a la versión orientada, sólo cambia el término "camino" por "cadena".

Si aplicamos a un grafo no orientado G, el algoritmo anterior, en virtud de la proposición IV.2.3.1, las aristas del grafo quedarán clasificadas en dos categorías:

(a) Aristas del bosque.

(b) Aristas de Ida.

Las aristas de ida las hemos podido llamar aristas de vuelta, sin embargo el término "ida" lo utilizamos para enfatizar que la primera vez que se recorre la arista es a través del arco de ida de la versión dirigida(ver proposición IV.2.3.1(b)).

### IV.2.4 DETECCIÓN DE LOS DIFERENTES TIPOS DE ARCOS

En las versiones dirigida y no dirigida del algoritmo de búsqueda en profundidad, podemos modificar los pasos siguientes para permitir enumerar los vértices según el orden de visita:

- Inicializar contador:

Visita de vértices de G aplicando búsqueda en profundidad:
Comienzo
(1) Inicialmente ningún vértice ha sido visitado;
(2) *Contador* ← 0;
(3) Para cada vértice v de G hacer:
Si v no ha sido visitado entonces aplicar la búsqueda en profundidad a partir de v.
fin

- El paso (1.2) del algortitmo de búsqueda en profundidad de la sección IV.2.1, lo sustituimos por:
"Cerrar $P_j$;
*Contador* ←*Contador* +1;
NumBusq($n_j$) ← *Contador* ;"

Con estas versiones modificadas podemos indicar en qué lugar de los algoritmos de búsqueda en profundidad se detectan los distintos tipos de arcos:

**Versión dirigida:**

**- Arcos del Bosque:**

Al cerrar $P_j$=<s,...,$n_j^{\leftarrow}$ ,$e_j$, $n_j$> en el paso (1.2) del algoritmo, el arco ($n_j^{\leftarrow}$ ,$n_j$) es un nuevo arco del bosque.

**- Arcos de ida:**

En el cuerpo del entonces de la segunda instrucción "Si ..." de la rutina de eliminación, el camino abierto $P_k$=<s, … , $n_k^{'}$ , $e_k$, $n_k$> con $n_k = n^i$, es eliminado por $P^i$ y NumBusq( $n_k^{'}$ ) ≤ NumBusq($n^i$) pues $n^i$ todavía no ha sido visitado, mientras $n_k^{'}$ ya ha sido visitado. Por lo tant, ( $n_k^{'}$ ,$n^i$) es un arco de ida, ya que al eliminarse $P_k$, él nunca podrá ser cerrado y así ( $n_k^{'}$ ,$n^i$) no podrá ser un arco del árbol.

**- Arcos de vuelta y cruzados:**

Entre la palabra Comienzo y la primera instrucción Si ... de la rutina de eliminación, podemos agregar la siguiente instrucción que detecta los arcos de vuelta y cruzados:

"Si existe un cerrado $P_k$ con vértice terminal igual a $n^i$ entonces
Comienzo
{ Tenemos que NumBusq($n_j$) ≥ NumBusq($n^i$) pues $n^i$ ya fue visitado y $n_j$ es el último vértice visitado.
De donde, ($n_j$,$n^i$) es un arco cruzado o de vuelta}

Si $n^i$ está en el camino $P_j$
entonces {$(n_j,n^i)$ es un arco de vuelta}
si no {$(n_j,n^i)$ es un arco cruzado}."

Vemos que los bucles son detectados como arcos de vuelta por el algoritmo.

**Versión no dirigida:**
Las aristas del bosque y las aristas de ida se detectan en los mismos puntos que en la versión dirigida.

### IV.2.5 IMPLEMENTACIÓN RECURSIVA

Hasta ahora hemos visto que el algoritmo de búsqueda en profundidad nos permite determinar una arborescencia con raíz s, cuyos vértices son los alcanzables desde s. Obtenemos así un único camino desde s a cada vértice alcanzable desde s. Esto nos permitió definir la expresión "vértice visitado". Cuando cerramos un camino decimos que visitamos el vértice terminal de este camino.

La estrategia de visita de vértices (elección de caminos a cerrar) consiste en:
- Visitar un vértice.
- Por cada sucesor del vértice visitado, que no haya sido visitado, aplique la misma estrategia.

Esto nos sugiere un esquema recursivo de visita de vértices en donde ahora la generación de caminos es una consecuencia del proceso de visita de vértices: antes de aplicar la misma estrategia al sucesor w de un vértice visitado v, asociamos a w un apuntador al vértice v. Estos apuntadores permitirán recuperar recursivamente los caminos desde s a cada vértice alcanzable desde él.

Versión recursiva de la visita de vértices:
{ Entrada: Un grafo dirigido G=(V,E) sin arcos múltiples.
Salida: Un apuntador por cada vértice, que apunta al vértice anterior en el camino.}
Var *v* : vértice;
Comienzo
(0) Inicialmente ningún vértice de G ha sido visitado (esto lo podemos indicar asociando a cada vértice una variable booleana que indique si el vértice está o no visitado).
(1) Para cada vértice *v* de G hacer:
Si *v* no ha sido visitado entonces
Comienzo
Colocar apuntador de *v* en nulo (no apunta a nada);
Búsqueda en profundidad(*v* );
fin;
(2) Recuperar caminos
Donde
Búsqueda en Profundidad(*v* :vértice):
Var *w* :vértice;
Comienzo
(0) Marcar a *v* como visitado;
(1) Para cada vértice *w* sucesor de *v* hacer:
Si *w* no ha sido visitado entonces
Comienzo
Asociar a *w* un apuntador hacia *v* ;
Búsqueda en Profundidad(*w* );
fin
fin
fin

Es evidente la sencillez de esta implementación con respecto a la original. Esto se debe a la equivalencia entre la estrategia de elección de caminos abiertos: "Cerrar en la próxima iteración el último camino abierto", y aplicar, a cada vértice no visitado, el mismo proceso. Por otro lado, al suprimir la rutina de eliminación de caminos, el orden del algoritmo es $O(\max(|V(G)|,|E(G)|)$. La versión no orientada es idéntica a la orientada.

Para detectar los diferentes tipos de arcos tenemos la versión siguiente:

Búsqueda en Profundidad(*v* :vértice):

```
Var w :vértice;
Comienzo
(0) Marcar a v como visitado; Contador ← Contador + 1; NumBusq(v )←Contador ;
(1) Para cada vértice w sucesor de v hacer:
    Si w no ha sido visitado entonces
        Comienzo
            {(v,w) es un arco del bosque}
            Asociar a w un apuntador hacia v;
            Búsqueda en Profundidad(w );
        fin
        si no
        Comienzo
        Si NumBusq(v ) ≤ NumBusq(w ) entonces
                {(v,w ) es un arco de ida};
        Si NumBusq(v ) > NumBusq(w ) entonces
                Si w se encuentra en el camino hasta v
                    entonces {(v,w ) es un arco de vuelta}
                    si no {(v,w ) es un arco cruzado}
        fin
fin
```

La variable *Contador* debe ser inicializada en cero antes de la primera llamada a Búsqueda en Profundidad(*v*).

Al aplicar la versión recursiva del algoritmo de visita de vértices al grafo dirigido asociado a un grafo no dirigido, cada arco de vuelta es recorrido primero que el arco paralelo de ida. Por lo tanto en la versión no orientada podemos hablar de aristas del bosque y aristas de vuelta, en contraste con la versión no recursiva.

### IV.2.6 APLICACIONES

Una de las aplicaciones más importantes del algoritmo de búsqueda en profundidad (D.F.S.) es la técnica de "backtracking", la cual es muy utilizada en problemas de enumeración implícita; estos son problemas de combinatoria donde se requiere recorrer el espacio de soluciones de un problema dado, dicho espacio podría tener una cardinalidad muy alta o infinita. Por ejemplo, podemos resolver el problema de las 8 reinas del tablero de ajedrez mediante D.F.S.; este problema consiste en determinar si se puede colocar 8 reinas en un tablero de ajedrez sin que se amenacen entre ellas. Este tipo de aplicaciones del D.F.S. podrá encontrarlas en [4].

#### IV.2.6.1 ALCANCE

Podemos utilizar al algoritmo de búsqueda en profundidad para determinar la matriz de alcance M* de un grafo:

```
Matriz de Alcance:
{ Entrada: Un grafo dirigido G sin arcos múltiples.
  Salida: La matriz de alcance M* de G. }
Var v :vértice;
Comienzo
(0) Inicialice M* con todos sus elementos iguales a cero;
(1) Para cada vértice v de G hacer:
        Alcanzables(v );
Donde
  Alcanzables(w : vértice):
  Var z:vértice;
  Comienzo
    M*[v,w] ← 1;
    Para cada vértice z sucesor de w hacer:
        Si M*[v,z ]=0 entonces Alcanzables(z )
  fin
fin.
```

La rutina Alcanzables es una modificación del algoritmo de búsqueda en profundidad, que permite ir calculando M*.

**Observaciones:**

- La variable *v* es una variable global y el pasaje de parámetros se hace por valor.

- El número de operaciones en cada iteración del paso (1) es *O* (número de arcos del subgrafo inducido por los vértices alcanzables desde *v*), en el peor caso esto sería *O* (|E(G)|). Por lo tanto el algoritmo Matriz de Alcance es *O* (|V|x|E|).

- Si comparamos el orden del algoritmo anterior con el orden del algoritmo de Roy-Warshal (sección II.2.2) que es *O* ($|V|^3$), el algoritmo anterior es más eficiente en tiempo aunque no en espacio pues cada llamada recursiva crea nuevos lugares de almacenamiento.

- Si G es no orientado, al aplicar la búsqueda en profundidad a partir de un vértice v de G (versión recursiva), los vértices que resultan marcados son todos los que están en la misma componente conexa de v. En cada llamada al algoritmo de búsqueda se marca una nueva componente conexa de G.

### IV.2.6.2 DETECCIÓN DE CIRCUITOS Y CICLOS

**Proposición IV.2.6.2.1**

Sea G un digrafo sin arcos múltiples. G no posee circuitos si y sólo si al aplicar el algoritmo de visita de vértices al grafo G no se generan arcos de vuelta.

**Demostración:**

Si G no posee circuitos no se generarán arcos de vuelta pues si se generara un arco de vuelta e=(x,y), tomando el trecho de camino en el bosque que va de y a x en el bosque, y concatenándolo con <x,e,y>, obtenemos un circuito.

Para demostrar que es suficiente apliquemos un argumento por contraposición. Supongamos que G posee un circuito C. Sea v el primer vértice visitado por el algoritmo entre todos los vértices de C . Sea (u,v) un arco en el circuito C. Por la Proposición IV.2.2.3, el vértice u debe ser descendiente de v en el bosque generado por el algoritmo . Por lo tanto (u,v) es un arco de vuelta.

¤

De igual forma podemos detectar un ciclo en un grafo no orientado: existirá un ciclo en un grafo no orientado si y sólo si al aplicar el algoritmo de visita de vértices al grafo, se generan aristas de vuelta.

### IV.2.6.3 PUNTOS DE ARTICULACIÓN Y COMPONENTES BICONEXAS

En esta sección los grafos considerados son no orientados y simples.

Un punto de articulación de un grafo G es un vértice que al eliminarlo de G, el grafo resultante posee al menos una componente conexa más que G. Un *grafo biconexo* es un grafo que no posee puntos de articulación. Un *bloque* de G es un subconjunto de vértices A tal que $G_A$ (el subgrafo inducido por A) es conexo sin puntos de articulación y maximal con respecto a esta propiedad, es decir, no existe otro conjunto de vértices B tal que $A \subset B$ y B sea un bloque. Al grafo $G_A$ generado por un bloque A se le llama *componente biconexa* de G.

En el grafo de la figura IV.8 tenemos que $A_1=\{v_1,v_2\}$, $A_2=\{v_2, v_3, v_4\}$, $A_3=\{v_5\}$ y $A_4=\{v_6, v_7\}$ son los bloques de G. Si A es un bloque, el subgrafo $G_A$ es 2-conexo (si |A|>2), un istmo (si |A|=2) o un vértice aislado de G (si |A|=1).

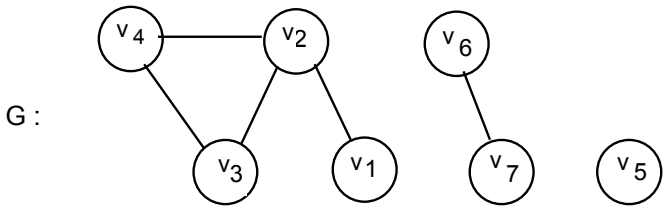



**figura IV.8**

**Proposición IV.2.6.3.1**

Sean G un grafo simple, A y B (A≠B) dos bloques de G. Se tiene que $|A \cap B| \leq 1$.

**Demostración:**

Haremos la demostración por el absurdo, suponiendo que $|A \cap B| \geq 2$.

Mostraremos que $G_{A \cup B}$ es un grafo conexo sin puntos de articulación. Esto nos llevará a una contradicción pues $A \subset A \cup B$ y por lo tanto A no es maximal con respecto a la propiedad: $G_A$ es conexo sin puntos de articulación. Lo mismo sucede con B, ni A ni B podrían ser bloques de G.

Sean v,w (v≠w) dos vértices en $A \cap B$. Como $G_A$ y $G_B$ son conexos y poseen al menos un vértice en común (v ó w) entonces $G_{A \cup B}$ es conexo. Para mostrar que $G_{A \cup B}$ no posee puntos de articulación, como es conexo basta mostrar, según la proposición III.3.4.5, que:

Para todo vértice t de $G_{A \cup B}$, si x,z son vértices de $G_{A \cup B}$, distintos entre sí y distintos de t, entonces existe una cadena de x a z que no pasa por t.

Sean $x,z,t \in A \cup B$:

Si x,z son vértices de $G_A$ entonces existe una cadena de x a z que no pasa por t, pues t no es un punto de articulación de $G_A$. Esta cadena es una cadena de $G_{A \cup B}$ de x a z que no pasa por t. El mismo razonamiento anterior se puede aplicar al caso en que x, z sean vértices de $G_B$.

Sea $x \in V(G_A)$ y $z \in V(G_B)$. Si t≠v entonces existe una cadena $C_1$ de x a v en $G_A$ y una cadena $C_2$ de v a z en $G_B$ tales que $C_1$ y $C_2$ no pasan por t y así $C_1 \,||\, C_2$ es una cadena de $G_{A \cup B}$ que no pasa por t. Si t=v entonces t≠w y podemos aplicar el mismo razonamiento anterior sustituyendo a v por w.

En conclusión, $G_{A \cup B}$ no posee puntos de articulación y ni A ni B pueden ser bloques de G.

¤

Si e={x,y} es una arista de G, entonces e debe pertenecer a alguna componente biconexa. Esto se debe a que el grafo H=({x,y},{e}) es conexo sin puntos de articulación. De donde, {x,y} es un bloque de G o está contenido en un bloque de G.

La proposición IV.2.6.3.1 junto con la observación anterior implican que los conjuntos de las aristas de cada componente biconexa particionan al conjunto de aristas del grafo.

Otra observación interesante la podemos resumir en la proposición siguiente:

**Proposición IV.2.6.3.2**

Sea G un grafo simple y C un ciclo elemental del G. Las aristas de C pertenecen todas a una misma componente biconexa de G.

**Demostración:**

Sea A el conjunto de vértices de C. Tenemos que $G_A$ es un grafo conexo sin puntos de articulación ($G_A$ contiene a C). De donde, A es un bloque o está contenido en un bloque de G.

¤

Se puede mostrar que la relación de equivalencia R sobre el conjunto de las aristas del grafo G, inducida por la partición de E(G) en los conjuntos de las aristas de cada componente biconexa de G es:

$\forall$ $e_1,e_2 \in E(G)$: $(e_1,e_2)\in R$ si y sólo si $e_1=e_2$ ó existe un ciclo elemental en G que contiene a $e_1$ y $e_2$.

Otro resultado interesante que relaciona a los puntos de articulación con las componentes biconexas de un grafo es:

**Proposición IV.2.6.3.3**

Sea G un grafo simple. Un vértice t es un punto de articulación de G si y sólo si existen dos bloques A y B ($A\neq B$) de G tales que $A\cap B=\{ t \}$.

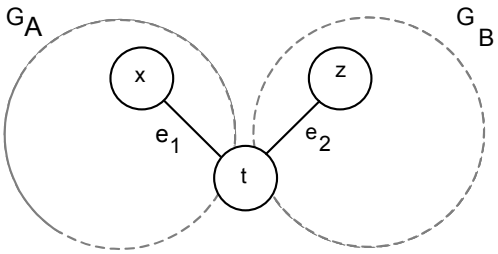



**figura IV.9**

**Demostración:**

Supongamos que t es un punto de articulación de G. Entonces existen x,z vértices distintos entre sí y distintos de t, tales que toda cadena elemental de x a z pasa por t. Sea $C=<x,...,x_1,e_1,t,e_2,x_2,...,z>$ una de esas cadenas. Tenemos que $x_1,t,x_2$ son distintos entre sí. Por lo tanto $e_1$ y $e_2$ son distintas. Las aristas $e_1$ y $e_2$ no pueden estar en una misma componente biconexa de G pues si así fuera, $x_1$, t y $x_2$ estarían en esa componente biconexa y entonces debería existir una cadena $C_1$ de $x_1$ a $x_2$ en esa componente, que no pasa por t y así $C_x \,||\, C_1 \,||\, C_z$ sería una cadena de x a z que no pasa por t, con $C_x$ la subcadena de C que va de x a $x_1$ y $C_z$ la subcadena de C que va de $x_2$ a z. De donde $e_1$ y $e_2$ están en componentes biconexas distintas. Sean A y B los bloques asociados a las componentes biconexas que contienen a $e_1$ y $e_2$ respectivamente. Entonces, $t\in A\cap B$ y según la proposición IV.2.6.3.1, $A\cap B=\{t\}$.

Recíprocamente, sean A y B dos bloques de G tales que $A\cap B=\{t\}$. Como $A\in B$ y $G_A$ y $G_B$ son conexos, entonces existen $x\in A$ y $z\in B$ tales que $e_1=\{x,t\}$ es una arista de $G_A$ y $e_2=\{z,t\}$ es una arista de $G_B$. Los vértices x,t,z son distintos entre sí. Mostremos que toda cadena de x a z pasa por t (ver figura IV.9).

Supongamos que existe una cadena en G de x a z que no pasa por t. Tomemos una cadena elemental C de x a z que no pasa por t. Se tiene que $C \,||\, <x,e_1,t,e_2,z>$ es un ciclo elemental de G. De acuerdo a la proposición IV.2.6.3.2, $e_1$ y $e_2$ deben pertenecer a la misma componente biconexa de G lo cual contradice la suposición de que $e_1$ y $e_2$ pertenecían a diferentes componentes biconexas de G. Por lo tanto toda cadena de x a z en G pasa por t y así t es un punto de articulación de G.

¤

Podemos aplicar búsqueda en profundidad para hallar los puntos de articulación y las componentes biconexas de G. El algoritmo fue desarrollado por Hopcroft y Tarjan en 1973. Veamos con un ejemplo en qué consiste el algoritmo. Apliquemos el algoritmo de búsqueda en profundidad al grafo de la figura IV.10(a) obteniendo así el árbol de la búsqueda en profundidad en la figura IV.10(b). Note que si la raíz del árbol posee más de un sucesor en el árbol, como no hay arcos cruzados, la raíz debe ser punto de articulación. En nuestro ejemplo el vértice 1 es un punto de articulación.

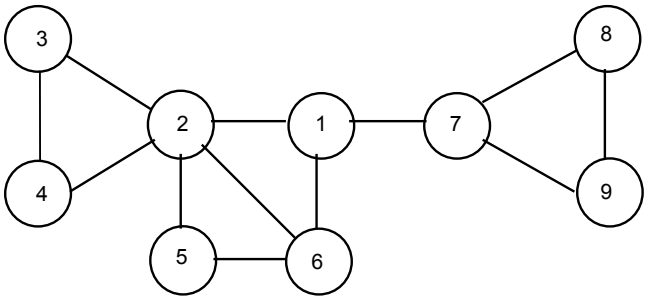



**figura IV.10(a)**

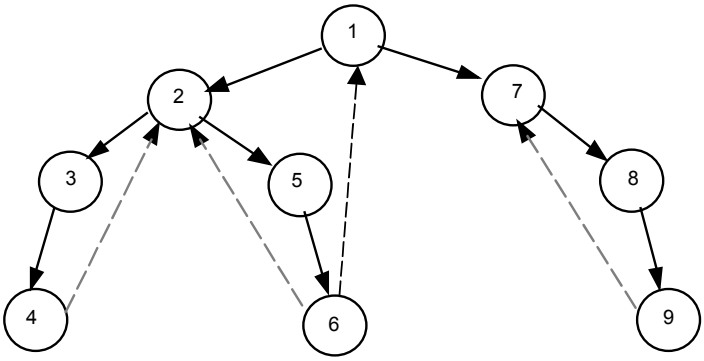



**figura IV.10(b)**

Para cualquier vértice v que no sea la raíz, v sera un punto de articulación si y sólo si las aristas de retorno que parten desde cualquier descendiente de algún sucesor del vértice v en el árbol, no conectan a este descendiente con un ascendiente propio de v. Si v posee un sucesor w tal que todos sus descendientes no están conectados con un ascendiente propio de v entonces v desconecta a w y todos sus descendientes del resto del grafo. Recíprocamente, si algún descendiente de cualquier sucesor w de v, está conectado con un ascendiente propio de v a través de una arista de retorno, entonces al eliminar v de G, w sigue conectado al resto del grafo a través de uno de sus descendientes.

Por lo tanto, para determinar los puntos de articulación, basta con calcular para cada vértice v del árbol el NumBusq(v) más pequeño entre todos ascendientes de v que sean extremo de una arista de retorno recorrida partiendo de un descendiente de v. Llamemos a este número MasBajo(v). Si en el recorrido de la búsqueda en profundidad v posee un sucesor w con MasBajo(w) mayor o igual a NumBusq(v), entonces v es un punto de articulación de G, si v no es la raíz. Si v es la raíz, v será un punto de articulación si posee al menos dos sucesores en el árbol.

Supongamos que vamos guardando en una pila las aristas (del árbol o de retorno) a medida que son recorridas por el algoritmo de búsqueda en profundidad. En el instante en que se termina de visitar todos los descendientes de un vértice w, si el padre v de w (el predecesor de w en el árbol) es un punto de articulación (MasBajo(w) $\geq$ NumBusq(v)) entonces sacamos de la pila todas las aristas hasta la {v,w} y las colocamos en un conjunto. Cada uno de estos conjuntos que se van generando es el conjunto de aristas de cada componente biconexa de G.

El número MasBajo de cada vértice puede ser calculado a medida que se recorre el grafo, de manera que al terminar de recorrer todos los descendientes de un vértice, se haya calculado su número MasBajo. Esto lo permite llevar a cabo la siguiente fórmula recursiva:

MasBajo(v)=Min A

donde A = {NumBusq(v)} $\cup$ {NumBusq(w)/ existe una arista de retorno {u,w} tal que u es un descendiente de v y w es un ascendiente de v en el árbol}=

{NumBusq(v)} $\cup$ {MasBajo(w)/ w es sucesor de v en el árbol} $\cup$ {NumBusq(w)/ {v,w} es una arista de retorno}.

Algoritmo para determinar las componentes biconexas y los puntos de articulación (basado en el algoritmo de visita de vértices):

```
{ Entrada: Un grafo simple no orientado G.
  Salida: Conjunto P de puntos de articulación y conjunto B de conjuntos de aristas de cada componente
  biconexa de G}

Variable Contador :Entero;
         Aristas :Pila de aristas;
         v : vértice;
Comienzo
(0) P ←∅;
    B ←∅;
    Aristas ←∅;
    Inicialmente ningún vértice de G ha sido visitado;
    Contador ←0;
(1) Para cada vértice v de G hacer:
(2) Si v no ha sido visitado entonces
              Comienzo
    (3) Búsqueda en profundidad modificada(v);
              Si v posee al menos dos sucesores en el bosque entonces Insertar v en P ;
              fin;

Donde

  Búsqueda en profundidad modificada(v):
  Variable w :vértice;
  Comienzo
  (0) Marcar a v como visitado;
      Contador←Contador +1;
      NumBusq(v)←Contador;
      MasBajo(v)←NumBusq(v);
  (1) Para cada vértice w adyacente a v hacer:
  (2)   Si w no ha sido visitado entonces
            Comienzo
  (3)         Colocar {v,w } en la pila Aristas ;
  (4)         Búsqueda en profundidad modificada(w );
  (5)         Si MasBajo(w ) ≥NumBusq(v ) entonces
                Comienzo
  (6)             Si v no es raíz entonces Insertar v en P ;
  (7)             Sacar de la pila Aristas todas las aristas hasta la {v,w }
                    inclusive. Insertar este conjunto de aristas en B ;
                fin;
  (8)         MasBajo(v )←Min(MasBajo(v ),MasBajo(w ));
            fin
  (9)   si no
        {Aunque no existan aristas de cruce, w puede ser el padre de v en el bosque}
  (10)      Si {v,w } es una arista de retorno entonces
              Comienzo
  (11)          Insertar la arista {v,w } en la pila Aristas ;
  (12)          MasBajo(v )←Min(MasBajo(v ),NumBusq(w ));
              fin
  fin
fin.
```

El algoritmo anterior supone que el grafo no tiene vértices aislados. En caso de existir vértices aislados, cada vértice aislado es una componente biconexa, se eliminan del grafo y se aplica el algoritmo para determinar las otras

componentes biconexas.

Detalles de implementación:

Si a cada vértice del grafo asociamos un apuntador al padre en el bosque, colocando un apuntador nulo a la raíz, podemos identificar:

- Un vértice raíz;
- Los sucesores en el bosque de un vértice dado.
- Si una arista es de retorno.

Las modificaciones que habría que hacer al algoritmo anterior serían:

(a) En el algoritmo principal:

Reemplazar el cuerpo del entonces (instrucciones (3)) por:

Comienzo
Colocar apuntador de *v* en nulo;
Búsqueda en profundidad modificada(*v* );
fin

(b)En el algoritmo de búsqueda en profundidad modificada:

Colocar la siguiente instrucción entre la palabra Comienzo, del cuerpo del entonces de la instrucción (2), y la instrucción (3):

Asociar a *w* un apuntador hacia *v*;

Reemplazar la condición del Si de la instrucción (5) por:

Si (MasBajo(*w* )≥NumBusq(*v* ) y el apuntador asociado a *v* no apunta a nulo) ó (el apuntador asociado a *v* es nulo y (al menos dos vértices apuntan a *v* ó *v* posee un sucesor no visitado)) entonces ...

Reemplazar la condición del Si de la instrucción (10) por:

Si NumBusq(*v* )>NumBusq(*w* ) y el apuntador asociado a *v* no apunta a *w* entonces ...

### IV.2.6.4 COMPONENTES FUERTEMENTE CONEXAS

En esta sección presentaremos un algoritmo de Tarjan para determinar las componentes fuertemente conexas de un grafo dirigido G. Los grafos que consideraremos serán sin arcos múltiples.

Sea R la relación de equivalencia siguiente sobre el conjunto de los vértices V de un grafo G=(V,E):

$\forall$ v,w $\in$ V: (v,w)$\in$R si y sólo si existe un camino de v a w y un camino de w a v en G.

Si $V_1$,$V_2$, ... ,$V_k$ son las clases de equivalencia de R entonces $G_{V_1}$, $G_{V_2}$, ... ,$G_{V_k}$ son las componentes fuertemente conexas de G. Si k=1 decimos que G es fuertemente conexo.

El algoritmo de Tarjan utiliza el algoritmo de visita de vértices mediante la búsqueda en profundidad. Antes veamos una serie de propiedades que permitirán justificar la correctitud del algoritmo.

Aplicando el algoritmo de visita de vértices al grafo G (ver sección IV.2.5) obtenemos un bosque expandido B. Si $V_i$ es el conjunto de vértices de una componente fuertemente conexa de G mostraremos seguidamente que el subgrafo inducido por $V_i$ en B es conexo y por lo tanto es una subarborescencia de B (es decir, un subgrafo de B que es una arborescencia). Sin embargo, no toda subarborescencia de B determina una componente fuertemente conexa. Ver ejemplo en la figuras IV.11 y IV.12.

Aplicando el algoritmo de visita de vértices a G a partir de un vértice arbitrario, por ejemplo $v_1$, obtenemos:

**Proposición IV.2.6.4.1**

Sea $G_{V_i}$ una componente fuertemente conexa de un grafo dirigido G=(V,E) sin arcos múltiples y B=(V,T) el bosque obtenido al aplicar a G el algoritmo de visita de vértices por búsqueda en profundidad. Entonces $B_{V_i}$, el subgrafo inducido por $V_i$ en B, es una subarborescencia de B.

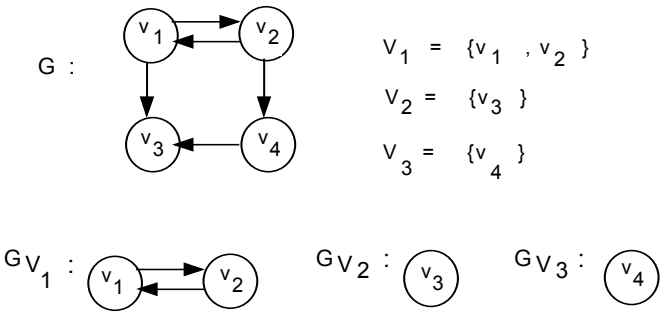



**figura IV.11**

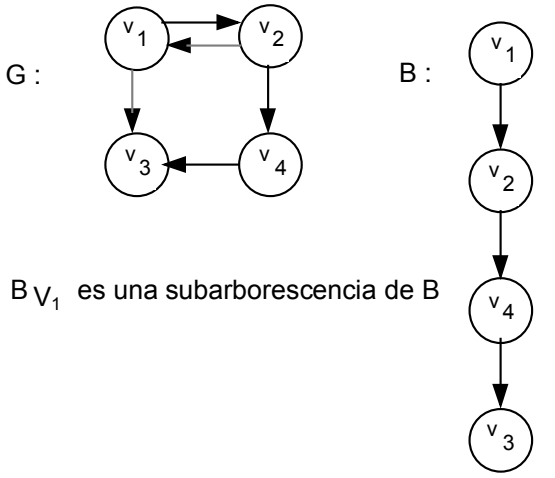



**figura IV.12**

**Demostración:**

Sea $v \in V_i$ el primer vértice de $V_i$ visitado por el algoritmo. NumBusq(v) es el menor entre todos los NumBusq de los vértices de $V_i$.

Como para todo vértice w de $V_i$existe un camino de v a w en G entonces la proposición IV.2.2.3 nos dice que w es descendiente de v en B y la proposición III.3.1.1 dice que no puede existir un camino de v a w en B con un vértice intermedio fuera de $V_i$.

¤

En el capítulo V veremos que si un digrafo G=(V,E), con $V=\{v_1,v_2,...,v_n\}$, no posee circuitos, entonces podemos asignar un orden lineal a los vértices de G: $v_{i_1} \leq v_{i_2} \leq \ldots \leq v_{i_n}$, de forma que si $(v_i,v_j) \in E$ entonces $v_i \leq v_j$.

Sea GR el grafo reducido de G (sección III.3.1), es decir, si $G_{V_1}$, $G_{V_2}$, ... ,$G_{V_k}$ son las componentes fuertemente conexas de G entonces los vértices de GR son $V_1,V_2$, ... ,$V_k$ y $(V_i,V_j) \in E(GR)$ si y sólo si existe un arco $(v_i,v_j)$ en E(G) con $v_i \in V_i$ y $v_j \in V_j$. Como GR no posee circuitos podemos asignar un orden lineal $V_{i_1} \leq V_{i_2} \leq \ldots \leq V_{i_k}$ a los vértices de GR de forma que si $(V_i,V_j) \in E(GR)$ entonces $V_i \leq V_j$. Si aplicamos el algoritmo de visita de vértices a G tomando primero un vértice de $V_{i_k}$, luego un vértice de $V_{i_{k-1}}$, ... luego un vértice de $V_{i_1}$, entonces cada arborescencia maximal del bosque B coincide con un $B_{V_i}$ para algún i. La proposición IV.2.6.4.1 nos

dice que cada $B_{V_i}$ es una arborescencia. Esta arborescencia debe ser maximal en el sentido que no existe otra arborescencia A en B tal que $B_{V_i}$ sea una subarborescencia propia de A. Esto último se debe a lo siguiente: inicialmente cuando aplicamos la búsqueda en profundidad comenzando con un vértice de $V_{i_k}$ se visitarán todos los vértices de $V_{i_k}$(proposición IV.2.6.4.1) y no se visitarán otros vértices pues no hay arcos que vayan de un vértice de $V_{i_k}$ a otro fuera de él. Por lo tanto el algoritmo de búsqueda en profundidad sólo visita, en su primera llamada, a los vértices de $V_{i_k}$. Luego al tomar un vértice de $V_{i_{k-1}}$ no se habrá todavía visitado ningún otro vértice de $V_{i_{k-1}}$. Al aplicar el algoritmo de búsqueda en profundidad a partir de ese vértice, se visitarán todos los vértices de $V_{i_{k-1}}$ (proposición IV.2.6.4.1) y no otros pues no existen arcos que partan de un vértice en $V_{i_{k-1}}$ hacia otro fuera de $V_{i_k} \cup V_{i_{k-1}}$. En general, en una llamada cualquiera al algoritmo de búsqueda en profundidad comenzando en un vértice de $V_{i_j}$, al momento de la llamada se han visitado los vértices de $V_{i_k} \cup V_{i_{k-1}} \cup ... \cup V_{i_{j-1}}$. Por lo tanto en esa llamada se visitarán los vértices de $V_{i_j}$ y no otros pues no existe arco desde un vértice de $V_{i_j}$ a otro fuera de $V_{i_k} \cup V_{i_{k-1}} \cup ... \cup V_{i_{j-1}}$. Veamos la situación con el ejemplo de la figura IV.13.

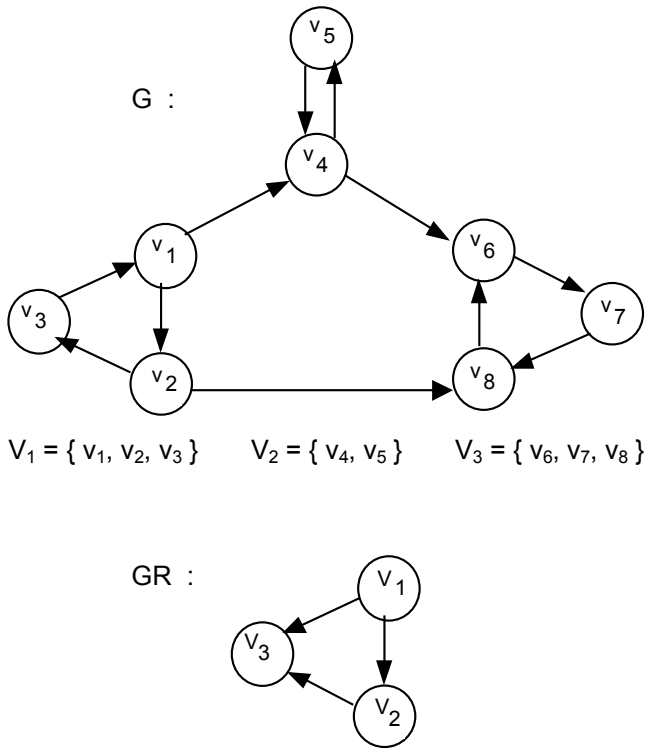



**figura IV.13**

Orden lineal que preserva el sentido de los arcos:

$$V_1 \leq V_2 \leq V_3$$

Aplicamos el algoritmo de visita de vértices comenzando con un vértice de $V_3$, luego uno de $V_2$ y finalmente uno de $V_1$. Obtenemos el bosque de la figura IV.14.

Vemos que los vértices de cada arborescencia maximal de B coinciden con los vértices de algún $V_i$.

En lo que sigue determinaremos un método que nos permita aplicar el algoritmo de visita de vértices de modo que en la llamada j-ésima al algoritmo de búsqueda en profundidad se comience con un vértice de $V_{i_{k-j+1}}$

$(V_{i_1} \leq V_{i_2} \leq \ldots \leq V_{i_k})$

Sea G' el grafo simétrico de G, obtenido a partir de G invirtiendo la orientación de todos los arcos.(también denotado por $G^{-1}$) Se tiene:

$G_{V_i}$ es una componente fuertemente conexa de G si y sólo si $G'_{V_i}$ es una componente conexa de G'.

Lo anterior es lo mismo que decir: $V_1,...,V_k$ son los conjuntos de vértices de las componentes fuertemente conexas de G si y sólo si $V_1,...,V_k$ son los conjuntos de vértices de las componentes conexas de G'.

Apliquemos el algoritmo de visita de vértices a G' y numeremos los vértices de G' por orden de finalización de la visita a los descendientes de cada vértice, es decir, asignar 1 al primer vértice al cual se haya visitado todos sus descendientes, asignar 2 al siguiente vértice al cual se haya visitado todos sus descendientes y así sucesivamente. Llamaremos a este número NumComp(v) (número de finalización de la búsqueda) para cada $v \in V(G')$.

**Proposición IV.2.6.4.2**

Sean $G'_{V_i}$ y $G'_{V_j}$ dos componentes fuertemente conexas de G', tales que:

$(V_i,V_j) \in E(G'R)$

Apliquemos el algoritmo de visita de vértices a G' asignando a cada vértice v de G' el NumComp(v). Sea B el bosque generado por el algoritmo.

Si $v_i$ es la raíz de $B_{V_i}$ y $v_j$ es la raíz de $B_{V_j}$ entonces:

$NumComp(v_i) > NumComp(v_j)$.

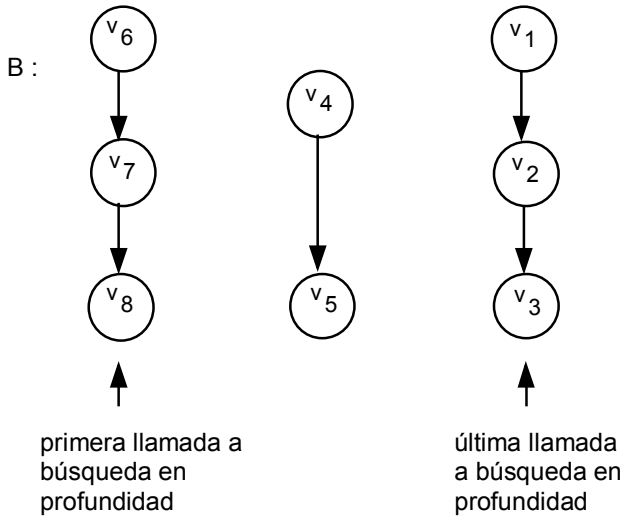



**figura IV.14**

**Demostración:**

Hay dos casos que considerar:

(a) $v_j$ es visitado antes que $v_i$.

(b) $v_i$ es visitado antes que $v_j$.

(a) Cuando se visita $v_j$ ningún descendiente en B de $v_j$ puede ser $v_i$ pues existiría un camino de $v_j$ a $v_i$ en G' lo cual implica que existiría un camino de $V_j$ a $V_i$ en G'R. Esto último no puede pasar porque G'R no posee circuitos y $(V_i,V_j)$ es un arco de G'R. Por lo tanto $v_i$ no es descendiente de $v_j$. Al terminar de visitar todos los descendientes de $v_j$ todavía no se habrá visitado a $v_i$ por lo tanto $NumComp(v_i) > NumComp(v_j)$.

(b) Cuando se visita $v_i$ ninguno de los vértices en $V_iUV_j$ han sido visitados y como existe camino en $G_{ViUVj}$ de $v_i$ a $v_j$ entonces por la Proposición IV.2.2.3, $v_j$ debe ser descendiente de $v_i$ en B. Por lo tanto, antes de visitar todos los descendientes de $v_i$ se deben haber visitado todos los descendientes de $v_j$ y así NumComp($v_i$) > NumComp($v_j$).

¤

La proposición anterior nos permite efectuar el siguiente proceso para determinar las componentes fuertemente conexas de un grafo G:

(1) Aplicar el algoritmo de visita de vértices al grafo G' obtenido a partir de G invirtiendo la dirección de todos sus arcos. Numerar los vértices de G' según el NumComp.

(2) Aplicar el algoritmo de visita de vértices al grafo G considerando los vértices no marcados, al llamar a la rutina de búsqueda en profundidad, en orden decreciente del NumComp. Es decir, cada vez que se llame al algoritmo de búsqueda, el vértice de partida es aquel que tenga el mayor NumComp entre los vértices no marcados. Los vértices visitados en cada llamada al algoritmo de búsqueda corresponden a los vértices de una componente fuertemente conexa de G.

Nos queda justificar el comentario final del paso (2) del proceso anterior.

Aplicando transitividad al resultado de la proposición IV.2.6.4.2 tenemos que si existe un camino de $V_i$ a $V_j$ en G'R entonces cualquiera sea el bosque B generado por una aplicación del algoritmo de visita de vértices a G', se tiene que NumComp($r_i$) de la raíz $r_i$ de $B_{V_i}$ es mayor que NumComp($r_j$) de la raíz $r_j$ de $B_{V_j}$. Si invertimos la dirección de los arcos de G'R, el grafo que obtendremos es GR. El vértice $V_i$ en GR tal que la raíz de $B_{V_i}$ posee el mayor valor de NumComp tiene la propiedad de no poseer sucesores en GR y por lo tanto $V_i$ no posee arcos que salen de él hacia V-$V_i$ en G. En efecto, si $V_i$ poseyera un sucesor $V_j$ en GR entonces $V_i$ poseería un predecesor $V_j$ en G'R y por lo tanto el NumComp de la raíz de $B_{V_j}$ sería mayor que el NumComp de la raíz de $B_{V_i}$ lo que contradice la manera como hemos escogido $V_i$.

Al aplicar la rutina de búsqueda en profundidad partiendo del vértice con mayor NumComp, los vértices que se visitan en esta llamada a la rutina son los de $V_i$.

Para obtener los siguientes $V_l$ ($V_l \neq V_i$) notemos que por un razonamiento similar al anterior, al eliminar el vértice $V_i$ de GR, en el grafo resultante el $V_j$ tal que la raíz de $B_{V_j}$ posee el mayor valor de NumComp no posee sucesores en ese grafo. Por lo tanto, si en la segunda llamada al algoritmo de búsqueda en profundidad escogemos como vértice de partida a aquél no visitado con mayor NumComp, entonces los vértices que se visitan en esa llamada son los de $V_j$. En general, lo que hacemos es visitar los vértices de G de derecha a izquierda en un ordenamiento lineal:

$$V_{i_1} \leq V_{i_2} \leq \ldots \leq V_{i_k}.$$

tal que si $(V_i,V_j) \in E(GR)$ entonces $V_i \leq V_j$.

Los comentarios anteriores justifican la siguiente proposición:

**Proposición IV.2.6.4.3**

Si aplicamos el algoritmo de visita de nodos mediante la búsqueda en profundidad a un grafo dirigido G, donde cada vez que se llama al algoritmo de búsqueda se toma como vértice de partida aquel que tenga mayor NumComp, entonces en cada llamada al algoritmo de búsqueda los vértices visitados corresponden a los vértices de una componente fuertemente conexa de G.

¤

Para cada vértice v el NumComp(v) se calcula aplicando el algoritmo de visita de nodos al grafo G' obtenido de G invirtiendo la dirección de los arcos y asignando el siguiente número (partiendo de 1) a un vértice, una vez se hayan visitado todos sus descendientes.

Algoritmo para determinar las componentes fuertemente conexas:
{ Entrada: Un grafo dirigido *G* sin arcos múltiples.
Salida: Una partición $\Pi$ de *V(G)*, donde cada conjunto $V_i$ de la partición corresponde a los vértices de una componente fuertemente conexa de *G*. }

```
Variable G' : digrafo;
Comienzo
    G' ← el grafo obtenido a partir de G invirtiendo la dirección de los arcos;
    Calcular los NumComp de cada vértice en G' ;
    Π←∅;
    A partir de G y los NumComp calcular los V_i e insertarlos en Π; .
Donde
    Calcular los NumComp a partir de G ':
    Variable Contador :Entero;
                    v :vértice;
    Comienzo
        Inicialmente ningún vértice de G' ha sido visitado;
        Contador ←0;
        Para cada vértice v de G' hacer:
            Si v no ha sido visitado entonces Búsqueda en profundidad(v );
    Donde
        Búsqueda en profundidad(v ):
        Variable w :vértice;
        Comienzo
            Marcar v como visitado;
            Para cada vértice w sucesor de v hacer
                Si w no ha sido visitado entonces Búsqueda en profundidad(w );
            Contador ←Contador + 1;
            NumComp(v ) ←Contador;
        fin
      fin;
  A partir de G y los NumComp calcular los V_i e insertarlos en Π:.
    Variable VI :conjunto de vértices;
    Comienzo
        Inicialmente ningún vértice de G ha sido visitado;
        Mientras Existan vértices no visitados hacer:
            Comienzo
            Escoger el vértice v no visitado con mayor
            NumComp;
            VI← ∅;
            Segunda búsqueda en profundidad(v );
            Π←Π∪{VI };
            fin
    Donde
        Segunda búsqueda en profundidad(v ):
        Variable w :vértice;
        Comienzo
            Marcar v como visitado;
            VI ←VI ∪ {v };
            Para cada vértice w sucesor de v hacer:
                Si w no ha sido visitado entonces Segunda búsqueda en profundidad ( w );
        fin
    fin
fin.
```

Ejemplo: Sea G el grafo de la figura IV.15 y G' el grafo obtenido a partir de G invirtiendo el sentido de todos sus arcos.

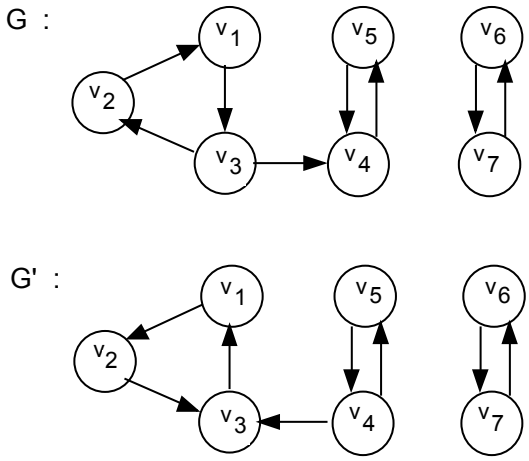



**figura IV.15**

Aplicamos el algoritmo de visita de nodos a G' para calcular los NumComp (ver figura IV.16):

Aplicamos el algoritmo de visita de nodos a G para calcular los $V_i$ tales que $G_{V_i}$ es una componente fuertemente conexa de G:

- Mayor NumComp = 7. Se llama a "Segunda búsqueda en profundidad" y se obtiene $V_1 = \{v_6,v_7\}$.
- Mayor NumComp entre los no visitados = 5. Se llama a "Segunda búsqueda en profundidad" y se obtiene $V_2 = \{v_4,v_5\}$.

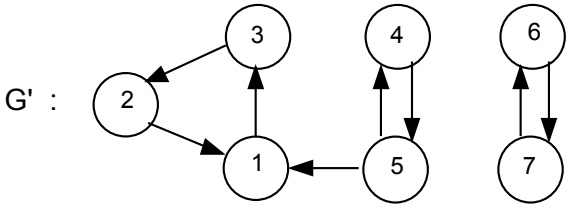



**figura IV.16**

- Mayor NumComp entre los no visitados = 3. Se llama a "Segunda búsqueda en profundidad" y se obtiene $V_3 = \{v_1,v_2,v_3\}$.

Finalmente:

$$\Pi = \{ \{v_6,v_7\},\{v_4,v_5\},\{v_1,v_2,v_3\} \}$$

Se puede conseguir una implementación donde el orden del algoritmo sea $O(\max(|V(G)|, |E(G)|))$

### IV.2.7 BÚSQUEDA EN AMPLITUD

La búsqueda en amplitud es un caso particular del modelo general de etiquetamiento y consiste en generar un camino desde un vértice s hasta cada vértice de un grafo G, sin asociar atributos adicionales al camino que se desea encontrar. Los caminos que se generan dependen de la regla de elección del próximo camino a ser cerrado. En cada iteración, si el vértice terminal del camino que se acaba de cerrar posee sucesores, entonces se expande el camino a esos sucesores y en la regla de eliminación cada camino expandido se compara con los caminos listados. Si existe un camino listado, abierto o cerrado, con nodo terminal igual al del camino expandido que se esté considerando, entonces ese camino expandido es eliminado o desechado. Si no existe un camino listado con nodo terminal igual al

del camino expandido considerado, entonces se abre el camino expandido. Una vez se hayan considerado todos los caminos expandidos, en la siguiente iteración será escogido para ser cerrado el primer camino abierto que se colocó en la lista entre todos los abiertos en la lista. Por lo tanto el criterio de elección de un abierto sería: "escoger el primer abierto que se colocó en la lista" (primero en entrar, primero en salir).

A continuación presentamos el algoritmo:

Búsqueda en amplitud (versión 1):

{ Entrada: Un digrafo $G=(V,E)$ sin arcos múltiples y $s \in V$.
Salida: Una lista de caminos de $s$ a cada vértice alcanzable desde $s$ }
Comienzo
(0) Abra el camino $P_0=<s>$;
(1) Mientras Existan caminos abiertos hacer:
Comienzo
(1.1) Escoger el primer camino abierto colocado en la lista entre todos los abiertos de la lista. Sea $P_j = <s,...,n_j>$ este camino;
(1.2) Cerrar $P_j$;
(1.3) Obtener los sucesores de $n_j$ : $n^1,n^2,...,n^q$;
(1.4) Construir los caminos expandidos:
$P^i = Exp(P_j,n^i)$, i=1,2,...,q;
(1.5) Aplicar la rutina de eliminación de caminos;
fin;
Donde
Rutina de eliminación:
Comienzo
Para cada $P^i$, $1 \le i \le q$, hacer:
Si no existe un $P_k$ en la lista con igual nodo terminal que $P^i$ entonces abra $P^i$
fin
fin.

Ejemplo: Sea G=(V,E) el grafo de la figura IV.17 con s=1.

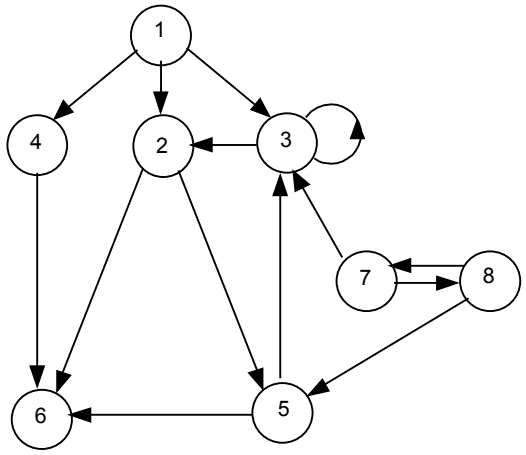



**figura IV.17**

Supondremos que el orden en que serán considerados los sucesores de cada vértice será en orden creciente de los números que los identifican.

Inicialmente $P_0$ = <1>.
En la primera iteración del algoritmo, al expandir $P_0$ obtenemos la lista:

$P_0$= <1> cerrado.
$P_1$= <1,2> abierto.
$P_2$= <1,3> abierto.
$P_3$= <1,4> abierto.

En la siguiente iteración el abierto a expandir sería $P_1$. Al final de la iteración se tendría la lista:

$P_0$= <1> cerrado.
$P_1$= <1,2> cerrado.
$P_2$= <1,3> abierto.
$P_3$= <1,4> abierto.
$P_4$= <1,2,5> abierto.
$P_5$= <1,2,6> abierto.

En la próxima iteración se expandirá $P_2$. Al final de la iteración se tendrá:

$P_0$= <1> cerrado.
$P_1$= <1,2> cerrado.
$P_2$= <1,3> cerrado.
$P_3$= <1,4> abierto.
$P_4$= <1,2,5> abierto.
$P_5$= <1,2,6> abierto.

Note que no se insertó un nuevo camino en la lista debido a que los caminos expandidos <1,3,3> y <1,3,2> fueron descartados en la rutina de eliminación.

En la próxima iteración se expande a $P_3$:

$P_0$= <1> cerrado.
$P_1$= <1,2> cerrado.
$P_2$= <1,3> cerrado.
$P_3$= <1,4> cerrado.
$P_4$= <1,2,5> abierto.
$P_5$= <1,2,6> abierto.

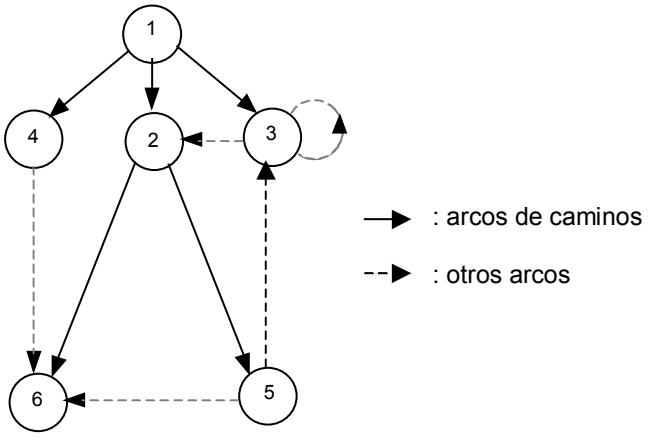



**figura IV.18**

De nuevo no se insertan nuevos caminos.

En las dos iteraciones siguientes se cierran $P_4$ y $P_5$ sin insertar nuevos caminos en la lista, con lo cual el algoritmo termina. La arborescencia con raíz s generada por el algoritmo se muestra en la figura IV.18

**Propiedades y observaciones:**

(1) Una manera de implementar el criterio de elección del próximo abierto a ser cerrado: "Tomar el primer abierto colocado en la lista", es considerar que se tienen dos listas, una de caminos cerrados y otra de abiertos. La lista de caminos abiertos puede ser manipulada como una cola (primero que entra, primero que sale). En cada iteración se escoge el abierto que esté en la cabeza de la cola y los caminos expandidos que se abren se colocan al final de la cola. Sin embargo, el orden en que son colocados en la cola estos últimos caminos es arbitrario, y dependerá del orden en que son considerados los sucesores de cada vértice.

(2) Una vez que un camino es listado, abierto o cerrado, éste no podrá ser eliminado. Este hecho es consecuencia del criterio de eliminación. La rutina de eliminación desecha caminos pero nunca elimina caminos listados. Los caminos desechados son los que tienen nodo terminal igual al nodo terminal de algún camino listado, sea abierto o cerrado. Este hecho nos dice que todo vértice es nodo terminal de, a lo sumo, un camino listado.

(3) Si $P=<s,e_1,x_1,...,e_n,x_n>$, $n\geq 1$, es un camino listado entonces los caminos $<s,e_1,x_1,...,e_i,x_i>$, $i=0,1,...,n-1$, son todos caminos listados cerrados (para $i=0$ el camino es $<s>$). La demostración de este hecho es similar a la dada en la propiedad (4) del algoritmo de búsqueda en profundidad. Se puede concluir que no puede haber dos caminos listados pasando por un mismo nodo n y tales que los predecesores de n en cada camino sean distintos entre sí. Por lo tanto, el conjunto de arcos de los caminos generados por la búsqueda en amplitud induce una arborescencia en el grafo. Se puede mostrar fácilmente que los nodos de esta arborescencia son los vértices alcanzables desde s.

(4) El algoritmo de búsqueda en amplitud termina después de un número de iteraciones igual al número de vértices alcanzables desde s pues en cada iteración se cierra un camino por cada vértice alcanzable desde s. En cada iteración el número de operaciones que se realizan es de orden el grado de salida del vértice terminal del camino que se cierra en esa iteración multiplicado por el número de caminos listados en esa iteración. Se puede lograr una implementación donde en cada iteración el número de operaciones sea igual al grado de salida del vértice terminal del camino que se cerró, es decir, podemos obtener una implementación que sea O(max(número de vértices alcanzables desde s, número de arcos del subgrafo inducido por el conjunto de vértices alcanzables desde s))

(5) Al implementar el algoritmo según la observación (1) notamos que en cada iteración del algoritmo el largo de los caminos abiertos difieren entre sí en a lo sumo 1. Como veremos en la proposición siguiente, podemos decir aún más de los caminos que genera el algoritmo.

**Proposición IV.2.7.1**

Al comienzo de una iteración cualquiera del algoritmo de búsqueda en amplitud aplicado a un digrafo G sin arcos múltiples, a partir de un nodo s, se cumplen las propiedades siguientes:

(a) Los caminos abiertos poseen largo k ó k+1, para algún $k\geq 0$; además, los primeros abiertos (o ninguno) tendrán longitud k y los últimos abiertos (o ninguno) tendrán longitud k+1. *Nota*: los primeros abiertos son los que están más cerca de la cabeza de la cola.

(b) Sea k el largo mínimo entre los largos de los caminos abiertos. Para todo vértice v a distancia j de s, con j menor que k, existe un camino cerrado de largo j con nodo terminal v.

**Demostración:**

Haremos una demostración por inducción sobre el número de iteraciones.

Al comenzar la primera iteración, en la cola de abiertos está únicamente el camino $<s>$ de largo cero. Evidentemente se satisfacen las dos propiedades.

Supongamos que al comienzo de la iteración i se satisfacen las dos propiedades. Mostremos que al comienzo de la iteración i+1 también se satisfacerán:

Si el primer camino abierto al comienzo de la iteración i tiene largo k, entonces al cerrar y expandir el primer abierto, los caminos expandidos que se abren después de aplicar la rutina de eliminación son todos de largo k+1. Además, estos últimos caminos son colocados al final de la cola de abiertos con lo cual se satisfacerá la propiedad (a) al comienzo de la iteración i+1. Por otro lado, si al cerrar el primer camino abierto al comienzo de la iteración i quedan caminos abiertos de largo k, entonces al comienzo de la iteración i+1 se satisfacerá la propiedad (b) por hipótesis de inducción. Supongamos ahora que al cerrar el primer camino abierto al comienzo de la iteración i no quedan caminos de largo k, es decir que todos los abiertos son de largo k+1. Sea $v_k$ un vértice a distancia k ($\geq 1$) de s, debemos probar que existe un camino cerrado de largo k y nodo terminal $v_k$. Sea $C=<s,e_1,v_1,...,v_{k-1},e_k,v_k>$ un camino de largo k de s a $v_k$. Sabemos que existe un camino cerrado de largo k-1 con nodo terminal $v_{k-1}$, pues por

hipótesis de inducción al comienzo de la iteración i el largo mínimo de los caminos abiertos es k y existe un camino cerrado de largo k-1 con nodo terminal v, para todo v a distancia k-1 de s. En la iteración donde se cerró el camino con nodo terminal $v_{k-1}$, se tuvo que haber expandido este camino a $v_k$. En la rutina de eliminación pasó uno de los siguientes hechos: o se descartó el camino expandido a $v_k$ porque ya existía un camino abierto con nodo terminal $v_k$, en cuyo caso ese camino era de largo k (en esa iteración sólo hay caminos de largo k-1 y k), o se abrió el camino expandido hasta $v_k$. En cualquiera de los casos anteriores, al finalizar esa iteración, en la lista de abiertos habría un camino de largo k con nodo terminal $v_k$. Por lo tanto, este camino tuvo que ser cerrado antes de comenzar a cerrar los caminos de largo k+1, de acuerdo al criterio de elección y la propiedad IV.2.7.1(a). Así, al comienzo de la iteración i+1, el largo mínimo de los abiertos es k+1 y existe un camino cerrado hasta cada nodo a distancia k de s.

¤

La proposición anterior es el fundamento del corolario siguiente:

**Corolario IV.2.7.2**

Sea A la arborescencia inducida por los arcos de los caminos cerrados resultantes de aplicar el algoritmo de búsqueda en amplitud a un grafo dirigido G sin arcos múltiples y con vértice inicial s. Para todo vértice v en V(A) se tiene que el camino de s a v en A (el cual es el camino cerrado por el algoritmo con nodo terminal v) es un camino de largo mínimo en G entre todos los caminos de s a v. En conclusión, A es una arborescencia de caminos de largo mínimo desde s hasta cada nodo alcanzable desde s en G.

(6) Al igual que en la propiedad (7) del algoritmo de búsqueda en profundidad, los caminos listados pueden ser representados en un formato recursivo. El elemento $P_i=(n,j)$ representa un camino listado donde:

- n es el nodo terminal del camino.
- j es un apuntador a otro elemento listado con $P_i=Exp(P_j,n)$.

En el algoritmo de búsqueda en amplitud no se pierden trechos de caminos pues los caminos cerrados no son eliminados.

Al terminar el algoritmo podemos recuperar los caminos a través de los apuntadores.

Búsqueda en amplitud (versión 2):

{ Entrada: Un digrafo $G=(V,E)$ sin arcos múltiples y $s \in V$.
Salida: Una lista de caminos de $s$ a cada vértice alcanzable desde $s$ }
Comienzo
(0) Abra el camino $P_0=(s,0)$;
(1) Mientras Existan elementos abiertos hacer:
Comienzo
(1.1) Escoger el primer abierto $P_j=(n_j,k)$ colocado en la lista entre todos los abiertos de la lista;
(1.2) Cerrar $P_j$;
(1.3) Obtener los sucesores de $n_j$ : $n^1,n^2,...,n^q$;
(1.4) Construir los elementos expandidos:
$P^i=(n^i,j)$, i=1,2,...,q;
(1.5) Aplicar la rutina de eliminación de elementos;
(2) Construir recursivamente los caminos a partir de los elementos listados.
Donde
Rutina de eliminación:
Comienzo
Para cada $P^i$, $1 \le i \le q$, hacer:
Si no existe un $P_k$ en la lista con primera coordenada igual a la de $P^i$ entonces abra $P^i$
fin
fin.

Como a cada nodo v alcanzable desde s corresponderá un camino cerrado con nodo terminal v, podemos decir que *visitamos* cada nodo cuando se cierra el camino correspondiente. Por lo tanto, si estamos interesados en efectuar un proceso a cada nodo visitado y no en generar caminos, podemos utilizar la versión siguiente del algoritmo la cual utiliza la misma terminología introducida en la sección IV.2.2 y IV.2.5. Cada vértice tendrá asociado un apuntador que apunta al vértice anterior del camino.

```
Visita de nodos aplicando búsqueda en amplitud:
{ Entrada: Un digrafo G=(V,E) sin arcos múltiples.
  Salida: Un apuntador por cada vértice v, el cual apunta al
      vértice anterior en el camino que va de s a v. }
Variable v :vértice.
Comienzo
(1) Inicialmente ningún vértice ha sido visitado;
(2) Para cada vértice v de G hacer:
          Si v no ha sido visitado entonces Búsqueda en
               amplitud(v );
(3) Recuperar caminos a través de los apuntadores
Donde
    Búsqueda en amplitud(v ):
    Variable Q: cola de vértices;
                  x,y : vértice;
    Comienzo
        Visitar v ;
        Vaciar la cola Q ;
        Colocar nulo el apuntador asociado a v ;
        Colocar v de último en la cola Q;
        Mientras Q no sea vacía hacer:
        Comienzo
            x ← primer vértice de Q;
            Eliminar primer vértice de Q;
            Para cada vértice y sucesor de x hacer:
                Si y no ha sido visitado entonces
                    Comienzo
                        Visitar y ;
                        Asociar a y un apuntador hacia x ;
                        Colocar y de último en la cola Q;
                    fin
        fin
    fin
fin.
```

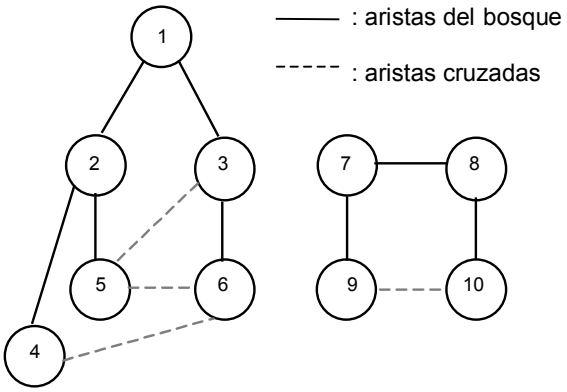



**figura IV.19**

La complejidad del algoritmo anterior es $O(\max(|V(G)|,|E(G)|))$ pues el algoritmo de búsqueda en amplitud es esencialmente el mismo de la versión 2, la diferencia fundamental es que al asociar el indicador "visitado" ó "no visitado" a cada vértice, eliminamos la rutina de eliminación de caminos y con ello el proceso de búsqueda de caminos con igual nodo terminal en cada iteración.

(7) Al igual que en el algoritmo de búsqueda en profundidad, los arcos de un grafo G, al cual se le aplica el algoritmo de búsqueda en amplitud para visitar sus vértices, pueden ser particionados en: arcos del bosque (aquéllos

pertenecientes a los caminos cerrados), arcos de retorno (aquéllos cuyo vértice inicial es descendiente del vértice terminal) y arcos cruzados (cuyos extremos no son ni descendientes ni ascendientes el uno del otro). Sin embargo, arcos de ida (cuyo vértice inicial es ascendiente del terminal), distintos de un bucle, no existen. Supongamos que $(v_1,v_2)$ es un arco de ida distinto de un bucle. Esto significa que $d(v_1,v_2)=1$ pero según las propiedades del algoritmo $d(v_1,v_2)$ es igual al número de arcos de un camino en el bosque de $v_1$ a $v_2$ y este número es mayor que 1. Los arcos cruzados (x,y) entre arborescencias cumplen con NumBusq(x) > Numbusq(y), sin embargo esto ultimo no se cumple para arcos cruzados en una misma arborescencia.

### IV.2.8 BÚSQUEDA EN AMPLITUD EN GRAFOS NO ORIENTADOS

Si G es un grafo no orientado sin aristas múltiples, recorrer G mediante la búsqueda en amplitud significa recorrer el grafo orientado asociado a G mediante la búsqueda en amplitud. Note que en este caso no existirán arcos de ida ni de vuelta pues por cada arco de vuelta habría uno de ida y estos últimos no pueden existir. Por lo tanto, podemos decir que las aristas de G quedan particionadas en aristas del bosque y aristas cruzadas. Además, entre dos árboles del bosque generado por el algoritmo de visita de nodos, no puede haber aristas. Esto último nos dice que los nodos visitados en cada llamada a la rutina de búsqueda en amplitud corresponden a los nodos de cada componente conexa de G (ver figura IV.19).

Note que si $\{v_1,v_2\}$ es una arista de cruce y $v_1$ es visitado antes que $v_2$ entonces en la arborescencia la distancia desde la raíz a $v_1$ es menor o igual que la distancia desde la raíz a $v_2$.

```
Visita de nodos de un grafo no orientado aplicando búsqueda en amplitud:
{ Entrada: Un grafo G no orientado sin aristas múltiples.
  Salida: Conjunto B de aristas del bosque. y conjunto C de
     aristas cruzadas.}
Variable v : vértice;
Comienzo
(1) Inicialmente ningún nodo ha sido visitado;
      C ← ∅ ; B ← ∅ ;
(2) Para cada vértice v de G hacer:
         Si v no ha sido visitado entonces Búsqueda en amplitud(v );
Donde
    Búsqueda en amplitud(v ):
    Variable Q : cola de vértices;
                 x , y : vértice;
    Comienzo
        Visitar v ;  Vaciar la cola Q ;
        Colocar v de último en la cola Q ;
        Mientras Q no sea vacía hacer:
        Comienzo
            x ←primer vértice de Q ;
            Eliminar primer vértice de Q ;
            Para cada vértice y adyacente a x hacer:
               Si y no ha sido visitado entonces
                   Comienzo
                      B ←B ∪{{x , y }};
                      Visitar y ;
                      Colocar y  de último en la cola Q ;
                   fin
               si no C ←C ∪ {{x , y }}
        fin
    fin
fin.
```

### IV.2.9 APLICACIONES

Hemos visto que una de las aplicaciones del algoritmo de búsqueda en amplitud consiste en hallar la distancia desde un nodo s hasta cada nodo alcanzable desde s, sea el grafo dirigido o no. Si el grafo es no dirigido, se puede

determinar las componentes conexas. La presencia de aristas cruzadas significa que el grafo tiene ciclos.

### IV.3 EJERCICIOS

**1.** Aplique el modelo general de etiquetamiento al grafo en la figura IV.20.

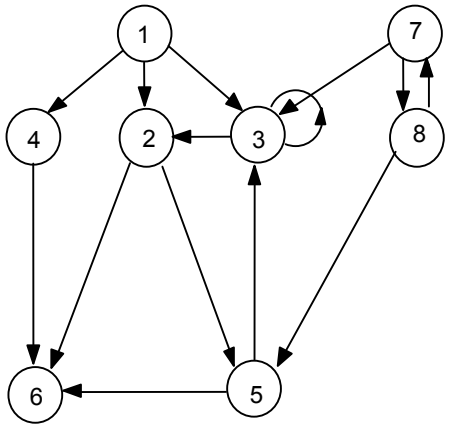



**figura IV.20**

Utilizando los siguientes criterios:

a) DFS (considerando los vertices en orden inverso) y BFS.

b) Partiendo del vértice 1, tomando como próximo abierto a cerrar el último colocado en la lista de abiertos y como rutina de eliminación "descartar el camino considerado $P^i$ si ya existe un camino listado abierto o cerrado con igual nodo terminal al de $P^i$".

**2.** Diseñe un algoritmo basado en una pequeña modificación del modelo general de etiquetamiento que genere todos los caminos elementales que parten de un vértice dado.

**3.** Proponga algoritmo basado en una pequeña modificación del modelo general de etiquetamiento, cuya entrada sea un digrafo *G=(V,E)* sin arcos múltiples y un vértice *x* de *V*, y cuya salida sea una lista de todos los caminos simples con vértice inicial *x*. Estime la complejidad en espacio y tiempo de su algoritmo (considere el peor caso).

**4.** Se define *r*(*G*,*x*), donde *G=(V,E)* es un grafo dirigido y *x* un vértice cualquiera de *V*, como el conjunto de vértices de *G* que son alcanzables desde *x* pasando exactamente por *r* arcos. Desarrolle un algoritmo que reciba un grafo y devuelva el conjunto *r(G,x)* para un vértice *x* dado y un valor de *r* dado como dato.

**5.** En el siguiente grafo halle un árbol cobertor usando DFS.

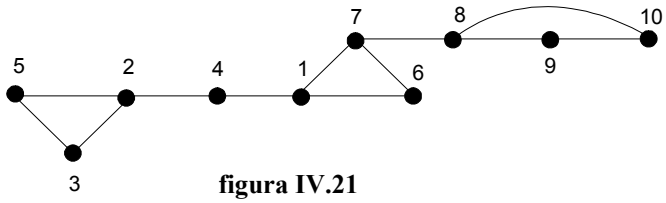



**figura IV.21**

**6.** Modifique DFS para obtener un algoritmo que tenga como entrada a un digrafo y verifique si este tiene circuitos.

**7.** Modifique DFS para obtener un algoritmo que halle las componentes conexas de un grafo *G=(V,E)*.

**8.** Sea *G* un grafo dirigido cualquiera sin arcos múltiples. Sea *G'* un grafo arco minimal respecto a la propiedad "la clausura transitiva de *G'* es igual a la de *G*". Determine un algoritmo polinomial que calcule un *G'*. Calcule la complejidad del su algoritmo en función del número de vértices y arcos de *G*.

**9.** Se define vértice **sumidero** como aquel vértice *x* donde $d^+(x)=0$ y *x* es alcanzable desde todos los demás vértices. ¿Cómo utilizaría DFS para determinar si un digrafo tiene un vértice sumidero?.

**10.** Sea *G* un digrafo conexo sin circuitos. Hacer un algoritmo que encuentre el número de caminos desde un vértice dado s hasta cada uno de los demás vértices.

**11.** Dada una arborescencia *A* con raíz *r*, modifique DFS para hallar el camino de largo mínimo de la raíz a las hojas y listarlo.

**12.** Sea *G* un digrafo representado mediante lista de adyacencias. Usando DFS encuentre un vértice de *G* cuyo grado interior sea mínimo.

**13.** Diseñe un algoritmo para determinar si un grafo es acíclico.

**14.** Diseñe un algoritmo basado en DFS para encontrar un conjunto estable maximal de un grafo *G*.

**15.** Diseñe un algoritmo que dado un digrafo sin circuitos encuentre un ordenamiento < de los vértices tal que *v* < *w* si existe un camino de *v* a *w*.

**16.** Sea *G=(V,E)* un grafo no orientado y *C* una clique (subgrafo completo) de *G*. Muestre que en el bosque del DFS todos los vértices de *C* estarán en un mismo camino desde la raíz. ¿Aparecen necesariamente en forma consecutiva?

**17.** Dado un ciclo en un grafo no orientado y una arborescencia generada al aplicar DFS al grafo. ¿Es verdad que los vértices del ciclo se encuentran todos en un camino que va de la raíz a una hoja de la arborescencia?.

**18.** Dar un contraejemplo a la conjetura siguiente: si existe un camino de v a u en un grafo dirigido G y si v es visitado antes que u en un recorrido de DFS entonces u es descendiente de v en la arborescencia del DFS.

**19.** Escriba un procedimiento que permita determinar si un DAG (grafo sin circuitos) tiene algún vértice raíz.

**20.** Considere el laberinto en la figura IV.22. Use DFS para encontrar un camino de *E* a *S* (esta es una aplicación de la técnica de "backtracking" [4]):

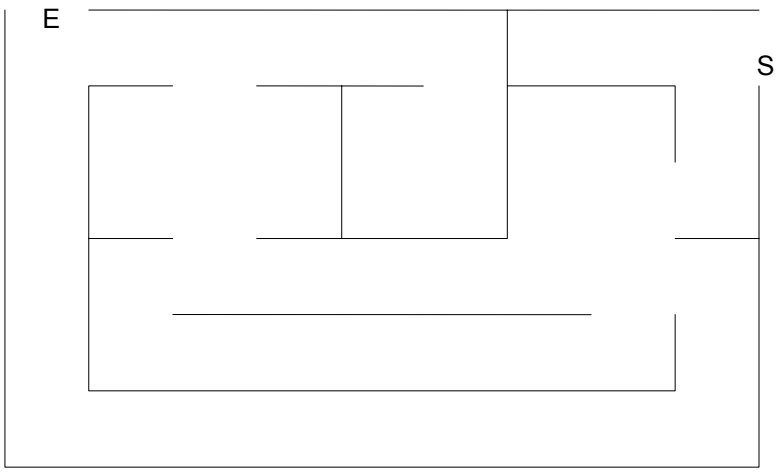



**figura IV.22**

**21.** En el siguiente grafo se desea hallar un camino entre los vértices *a* y *b*. Para ello se usa DFS, tomando como vértice de partida el vértice *a* y tomando el camino que termina en *b*. El algoritmo escoge siempre la arista de menor costo entre las que salen del vértice visitado. Asigne pesos a las aristas para que ningún camino entre *a* y *b* sea más costoso que el obtenido por DFS.

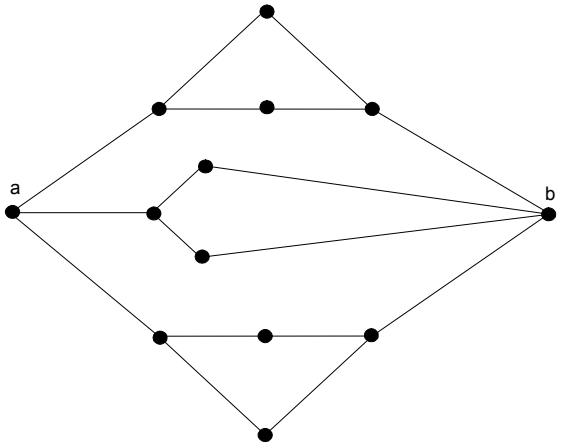



**figura IV.23**

**22.** Aplique DFS y BFS a los siguientes grafos para encontrar un árbol cobertor. Comience en el vértice 1 y considere los sucesores en orden creciente del número de vértice. Compare los resultados de ambos algoritmos.

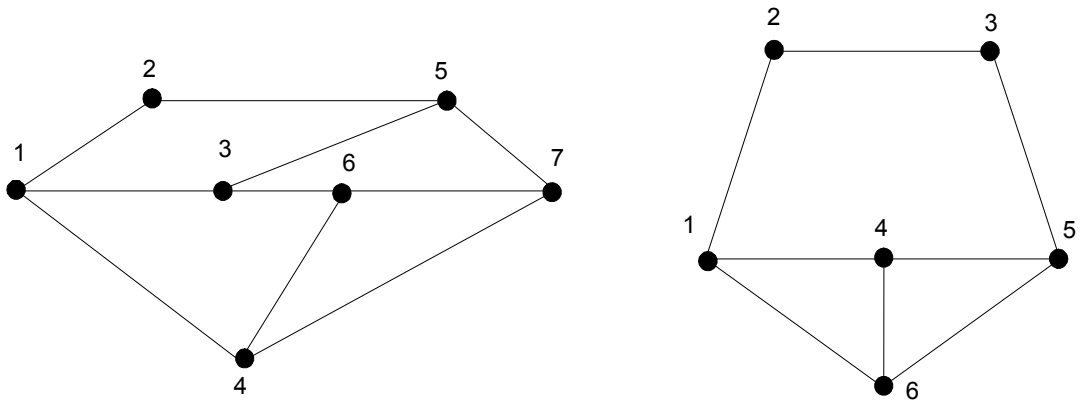



**figura IV.24**

**23.** Diseñe un algoritmo basado en BFS, para encontrar un conjunto dominante minimal en un grafo *G*.

**24.** La gerencia general de AVION AIR ha decidido agregar nuevas rutas al servicio que ofrece actualmente; para ello ha determinado que las nuevas rutas serán aquellas que unan ciudades para las cuales la conexión actual incluye más de una escala en el vuelo. Además, los aviones que adquirirá AVION AIR para cubrir las nuevas rutas utilizan un tipo de combustible especial, por lo que la empresa una vez establecida las rutas, deberá determinar los puntos donde se surtirá de combustible a estos aviones. Indique en términos de grafos como plantearía y resolvería el problema. AVION AIR le puede proporcionar un plano de las ciudades y las rutas actuales.

**25.** Dado un grafo dirigido *G=(V,E)*, un vértice *s* y una función de costos en las aristas $c:E\rightarrow\Re$, modifique el algoritmo BFS para encontrar para cada vértice v alcanzable desde *s*, un camino de menor costo de *s* a *v* entre todos los caminos de menor longitud de *s* a *v*.

**26.** Encontrar los puntos de articulación y las componentes biconexas del grafo en la figura IV.25. Utilice el algoritmo basado en DFS.

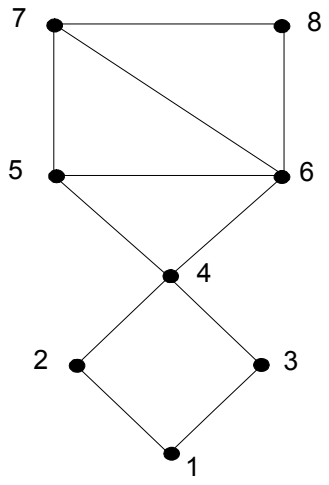



**figura IV.25**

**27.** Dado un grafo *G* con un punto de articulación *z*, escriba un algoritmo para encontrar dos vértices *x*, *y* en *G* tales que toda cadena entre *x* e *y* pasa por *z*.

**28.** Implemente el siguiente algoritmo para encontrar puntos de articulación en un grafo *G=(V,E)* no orientado:

- Aplicar DFS a G para calcular NumBusq(x) para cada vértice x de G.
- Para cada vértice x calcule el valor Low(x), donde Low(x) es el mínimo entre NumBusq(x) y NumBusq(z) para todo vértice w alcanzable desde x siguiendo un camino en el árbol de DFS y luego un lado de ida-vuelta hasta z.

En otras palabras, para todo vértice *x*:

Low(*x*) = min (min{NumBusq(z): existe un lado ida-vuelta {x,z}}, min{Low(y): y descenciente de x en el árbol del D.F.S.}

Los puntos de articulación son:

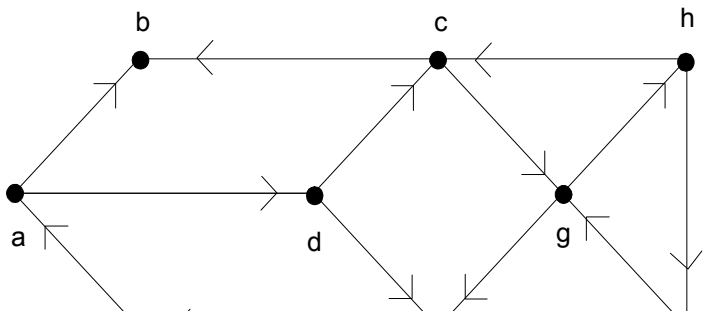
a) La raíz del árbol si y sólo si tiene más de un descendiente en el árbol.

b) Cualquier otro vértice *v* distinto de la raíz es un punto de articulación si y sólo si tiene un sucesor directo *w* en el árbol con *Low*(*w*) ≥ *NumBusq(v*).

Observación: Nótese que *Low*(*w*) ≥ *NumBusq*(*v*) para todo descendiente directo *w* de *v*, lo cual implica que no existe forma de comunicar los sucesores de v con sus antecesores que no sea pasando por *v*.

**29.** Sea *G=(V,E)* un grafo no dirigido sin aristas múltiples y *B* el bosque generado por el algoritmo de visita de vértices por búsqueda en profundidad en *G*. Sea *e*={*u, v*} ∈ *E* y *u* el extremo que es visitado primero en el algoritmo. Demuestre que *e* es un istmo de *G* si y sólo si no hay aristas de retorno desde ningún descendiente de *v* en *B* a un ancestro propio de *v* en *B*.

**30.** Aplicar el algoritmo visto en clases para determinar las componentes fuertemente conexas del siguiente grafo, partiendo en el DFS del vértice a. Aplique de nuevo el algororitmo pero partiendo esta vez del vértice b. Considere los sucesores de un vértice en orden alfabético.



**CAPITULO V**

# *CAMINOS DE COSTO MÍNIMO Y GRAFOS DE PRECEDENCIA*

## INTRODUCCIÓN

En este capítulo trataremos el segundo problema que nos ocupa, presentado en las secciones IV.1.1 y IV.2.2: Dado un grafo dirigido G=(V,E) sin arcos múltiples y D el conjunto de vértices alcanzables desde s, encontrar, si existe, un camino $P_w$ de s a cada vértice w en D tal que el camino de s a w satisfaga la condición $C(P_w)=cmin(s,w)$, donde cmin(s,w) es el costo de un camino mínimo de s a w, entendiéndose por costo C(P) de un camino P a la suma de los costos asociados a los arcos del camino.

En este capítulo se presentan varios algoritmos de búsqueda de caminos de costo mínimo basados en el modelo general de algoritmos de etiquetamiento (capítulo IV). Entre estos algoritmos tenemos el de Dijkstra, la técnica de Programación Dinámica y la de ramificación y acotamiento (Branch-and-Bound). Al final del capítulo estudiamos los grafos de precedencia por su importancia y utilidad en los problemas de caminos. Los algoritmos que presentaremos difieren unos de otros en el criterio de elección del próximo abierto a ser cerrado. Para garantizar que el algoritmo termine, cualquiera sea el criterio de elección, debemos tener la siguiente hipótesis:

$\forall v \in D: cmin(s,v) > -\infty$

Esta hipótesis excluye la posibilidad de tener circuitos de costo negativo en el subgrafo inducido por los vértices alcanzables desde s, si hay circuitos negativos entonces el problema no tendría solución ya que para cualquier N negativo, bastaría recorrer esos circuitos un número finito de veces para obtener C(P) menor que N.

El algoritmo de cálculo de caminos mínimos que deduciremos del modelo general de etiquetamiento se fundamenta en el Principio de Optimalidad que daremos a continuación. Este principio permite tener en la lista a lo sumo un camino hasta cada vértice del grafo de forma tal, que el conjunto de los arcos de estos caminos induzca una arborescencia con raíz s.

**Proposición V.1 (Principio de Optimalidad):**

Si $C(<n_0,e_1,n_1, ...,e_n,n_p>)=cmin(n_0,n_p)$ entonces $C(<n_i,e_{i+1},n_{i+1}, ... ,n_j>)=cmin(n_i,n_j)$ para $0\le i\le j\le p$.

**Demostración:**

Basta mostrar que $C(<n_i,e_{i+1},n_{i+1}, ... ,n_j>)\le cmin(n_i,n_j)$.

Supongamos que no:

$$C(<n_i,e_{i+1},n_{i+1}, ... ,n_j>) > cmin(n_i,n_j)$$

Por definición de ínfimo existe un camino $P=<m_1,e'_2,m_2, ...,e'_k,m_k>$ con $m_1=n_i$, $m_k=n_j$ y $C(P)< C(<n_i,e_{i+1},n_{i+1}, ... ,n_j>)$

Por lo tanto

$C(<n_0,e_1, ... ,n_{i-1},e_i,m_1,e'_2, ... ,e'_k,m_k,e_{j+1},n_{j+1}, ... ,n_p>)$

$< C(<n_0,e_1,n_1, ... ,e_p,n_p>)=cmin(n_0,n_p)$

lo que contradice la definición de camino mínimo.

¤

El principio de optimalidad asegura que todo trecho de un camino mínimo es mínimo. Una consecuencia inmediata de la proposición anterior es la siguiente:

**Corolario V.2**

Si C(<$n_0$,$e_1$,$n_1$, ...,$e_n$,$n_p$>)=cmin($n_0$,$n_p$) entonces cmin($n_0$,$n_k$) + cmin($n_k$,$n_p$) = cmin($n_0$,$n_p$) para k=0,1,...,p.

### V.1 MODELO DE ALGORITMO DE BÚSQUEDA DE CAMINOS DE COSTO MÍNIMO

Presentamos ahora el modelo general de algoritmo de búsqueda de caminos mínimos basado en el modelo de la sección IV.1.2.

Modelo de algoritmo de búsqueda de caminos de costo mínimo (versión 1):

{ Entrada: Un grafo dirigido *G* sin arcos múltiples, una función de costos *c* de *E(G)* a $\Re$ y un vértice *s*.
Salida: Un conjunto de caminos de costo mínimo desde *s* hasta cada vértice alcanzable desde *s* . }
Comienzo
(0) Abra el camino <*s* > con costo cero;
(1) Mientras Existan caminos abiertos hacer:
Comienzo
(1.1) Escoger un camino abierto $P_j$=<*s* ,...*nj* >;
(1.2) Cerrar $P_j$;
(1.3) Obtener los sucesores de *nj* : $n^1$, $n^2$,...,$n^q$;
(1.4) Construir los caminos $P^i$= Exp($P_j$,$n^i$ );
(1.5) A cada camino $P^i$ asocie el costo:

$$c^i = c_j + c\ (n_j\ ,n^i\ );$$

(1.6) Ejecutar la rutina de eliminación;
fin
Donde
Rutina de eliminación:
Comienzo
Para cada $P^i$, $1 \le i \le q$, hacer:
Si existe un camino listado $P_k$ (abierto ó cerrado) con igual vértice terminal que $P^i$
entonces
Comienzo
Si el costo de $P^i$ es menor que el costo de $P_k$ entonces eliminar a $P_k$ de la lista y abrir $P^i$
fin
si no Abrir $P^i$
fin
fin.

**Observaciones:**

(1) Obtendremos diferentes algoritmos especificando el paso (1.1).

(2) En la lista de caminos no es necesario almacenar por cada camino la secuencia de vértices que lo definen. Podemos representar a un camino $P_j$ por una terna ($n_j$,$c_j$,$i_j$) donde:

- $n_j$ es el vértice terminal del camino $P_j$.
- $c_j$ es el costo del camino.
- $i_j$ es un apuntador hacia otro elemento listado tal que $P_j$=Exp($P_{i_j}$,$n_j$).

El inconveniente de esta representación recursiva de caminos es la posibilidad que existe de que se pierdan trechos de caminos a medida que el algoritmo avanza pues los cerrados pueden ser eliminados de la lista por la rutina de eliminación, tal como se ilustró en la sección IV.1.2 propiedad (4). Sin embargo, los criterios de elección de los abiertos a ser cerrados, que discutiremos mas adelante, nos garantizarán que en esos casos los caminos cerrados no podrán ser eliminados. Veamos la versión del modelo de algoritmo con esta representación de caminos:

Modelo de algoritmo de búsqueda de caminos de costo mínimo (versión 2):

{ Entrada: Un grafo dirigido *G* sin arcos múltiples, una función de costos *c* de *E(G)* a $\Re$ y un vértice *s*.
Salida: Un conjunto de caminos de costo mínimo desde *s* hasta cada vértice alcanzable desde *s* . }

Comienzo

(0) Abra el elemento $P_0=(s,0,*)$;

(1) Mientras Existan elementos abiertos hacer:

Comienzo

(1.1) Escoger un elemento abierto $P_j=(n_j,c_j,i_j)$;

(1.2) Cerrar $P_j$;

(1.3) Obtener los sucesores de $nj$ : $n^1, n^2,...,n^q$ ;

(1.4) Construir los elementos $P^i=(n^i,c^i,j)$ donde:

$$c^i = c_j + c\ (n_j ,n^i\ );$$

(1.5) Ejecutar la rutina de eliminación;

fin

Donde

Rutina de eliminación:

Comienzo

Para cada $P^i$, $1 \le i \le q$, hacer:

Si existe un elemento listado $P_{k=}(n_k,c_k,i_k)$ (abierto ó cerrado) con $n_k=n^i$ entonces

Comienzo

Si el costo de $P^i$ es menor que el costo de $P_k$ entonces eliminar a $P_k$ de la lista y abrir $P^i$

fin

si no Abrir $P^i$

fin

fin

(3) Daremos varias propiedades del modelo de algoritmo anterior, siendo la más importante el hecho de que el modelo resuelve el problema de la arborescencia de caminos mínimos, es decir, el modelo garantiza que termina y al terminar, los caminos cerrados constituyen caminos de costo mínimo desde s a cada vértice alcanzable desde s y el conjunto de los arcos de estos caminos induce una arborescencia en el grafo. A pesar de que se pueden perder trechos de caminos en la versión 2 del algoritmo, veremos que nunca se perderán trechos de caminos óptimos.

(4) En adelante trabajaremos con la versión 2 del algoritmo.

**Proposición V.1.1**

Un vértice no puede ser vértice terminal en dos elementos listados.

**Demostración:**

Supongamos por el absurdo que existen dos elementos listados con un mismo vértice en una iteración cualquiera del algoritmo. Considere la primera iteración al final de la cual un vértice n está en dos elementos listados P y $\overline{P}$ En esta iteración P ó $\overline{P}$ fue obtenido por expansión del camino que se cerró en esta iteración y por lo tanto tuvo que ser comparado con el otro camino en la rutina de eliminación: la contradicción se obtiene de saber que la rutina de eliminación dejará sólo uno de los dos caminos en la lista.

Por lo tanto, la lista siempre contendrá una representación de caminos, ternas, cada uno terminando en un vértice diferente.

¤

Veamos ahora que, aunque se pueden perder trechos de caminos con la representación dada por las ternas, nunca se perderán trechos de caminos óptimos.

Decimos que un elemento listado P=(n,c,p) es mínimo si c=cmin(s,n).

**Proposición V.1.2**

Un elemento listado mínimo nunca puede ser eliminado y un sucesor mínimo (resultante de la expansión de un camino cerrado) sólo puede ser eliminado por un elemento listado mínimo.

**Demostración:**

Para eliminar un elemento listado (n,cmin(s,n),.), sería necesario un sucesor (n,c,p) con $c<cmin(s,n)$. Por definición de cmin(s,n), obligatoriamente $c \ge cmin(s,n)$, ya que c es el costo de un camino de s a n. Si c=cmin(s,n)

vemos que el sucesor es mínimo y es eliminado por un elemento listado mínimo.

¤

**Proposición V.1.3**

Sea $P_j=(n_j,c_j,p)$ mínimo, apuntando hacia $P_p=(n_p,c_p,.)$. Entonces $P_p$ es mínimo.

**Demostración:**

Por el principio de optimalidad, corolario V.2, se tiene que $cmin(s,n_p) + cmin(n_p,n_j) = cmin(s,n_j)$. Como $c_j=cmin(s,n_j)$, entonces $c(n_p,n_j)=cmin(n_p,n_j)$ y $c_j=c(n_p,n_j) + c_p$ entonces $c_p= cmin(s,n_p)$.

¤

**Proposición V.1.4**

Si el algoritmo coloca en la lista un elemento mínimo, el camino correspondiente se puede construir siguiendo los apuntadores.

**Demostración:**

La proposición V.1.3 nos dice que los caminos obtenidos siguiendo apuntadores son mínimos y de acuerdo a la proposición V.1.2 ellos nunca son eliminados.

¤

La proposición siguiente garantiza que se encontrarán caminos mínimos hasta cada vértice alcanzable desde s pues de no haberse encontrado todavía en una iteración cualquiera un camino mínimo hasta un vértice, todos los caminos mínimos hasta ese vértice están siendo construidos por el algoritmo.

**Proposición V.1.5**

Considere el algoritmo al iniciar el paso (1.1), en una iteración cualquiera. Sea n un vértice cualquiera y $<n^0,e^1,n^1,...,e^p,n^p>$, $n^0=s$, $n^p=n$, un camino mínimo cualquiera de s a n (que suponemos existente). Entonces se tiene el invariante:

Si no existe un cerrado de la forma $(n,cmin(s,n),.)$ entonces existe un abierto de la forma $(n^k,cmin(s,n^k),.)$, con $0\leq k\leq p$.

**Demostración:**

En la primera iteración (s,0,*) está abierto.

En una iteración cualquiera distinta de la primera, supongamos que no existe un cerrado $(n,cmin(s,n),.)$. Sea k-1 el índice más alto tal que $(n^{k-1},cmin(s,n^{k-1}),.)$ está cerrado (k-1, con $0\leq k-1<p$, siempre existe pues al menos (s,0,*) está cerrado).

Como $(n^{k-1},cmin(s,n^{k-1}),.)$ está cerrado, él fue expandido en una iteración anterior, generando un sucesor mínimo $(n^k,cmin(s,n^k),.)$ pues $cmin(s,n^k)=cmin(s,n^{k-1}) + c(n^{k-1},n^k)$. Si este sucesor no fue eliminado entonces fue abierto y si fue eliminado tuvo que ser eliminado por un abierto mínimo $(n^k,cmin(s,n^k),p^k)$. En cualquier caso fue listado como abierto y no puede ser eliminado según la proposición V.1.2. Por lo tanto permanece abierto completando así la demostración.

¤

Finalmente damos la proposición que garantiza que el algoritmo termina en un número finito de iteraciones. Al terminar habrá en la lista un camino mínimo hasta cada vértice alcanzable desde s y los arcos de estos caminos inducen una arborescencia.

**Proposición V.1.6**

Si G es un grafo finito y satisface la hipótesis: $\forall$v alcanzable desde s $[cmin(s,v) > -\infty]$, entonces:

(a) El algoritmo termina en un número finito de iteraciones.

(b) Cuando el algoritmo termina, hay un elemento cerrado mínimo para cada vértice alcanzable desde s y por otro lado, los arcos de los caminos obtenidos a partir de los elementos listados, inducen una arborescencia de

caminos mínimos en G, cuya raíz es s.

**Demostración:**

Primero observamos que un camino correspondiente a un elemento listado es elemental: Sabemos que si hay un camino listado con vértice terminal n, otro camino con vértice terminal n sólo podrá ser abierto (reemplazando al anterior) si tiene costo menor estricto que el del elemento listado. Por lo tanto si el algoritmo expande un camino con vértice terminal n y pasando por n, P=<s,...,n,...,n>, sabemos que el costo c' del primer trozo <s,...,n> del camino P debe ser menor o igual que el costo de P pues <n,...,n> (el segundo trozo de P) no puede tener costo negativo por hipótesis.

(a) Cada elemento listado representa un camino elemental. El mismo camino no puede ser listado dos veces. El número de caminos elementales en un grafo finito es finito ($\leq |V(G)|!$) y por lo tanto el número de iteraciones es finito pues en cada iteración se cierra un abierto y los caminos cerrados no se vuelven a abrir.

(b) Sea n alcanzable desde s. Si (n,cmin(s,n),.) no está cerrado, la proposición V.1.5 garantiza que hay abiertos y que el algoritmo no puede parar. Por lo tanto, cuando el algoritmo termina, cada cerrado corresponde a un camino mínimo que puede ser recuperado por los apuntadores, es decir, se obtiene una arborescencia.

¤

(5) Si en el subgrafo generado por los vértices alcanzables desde s hay circuitos de costo negativo, este hecho puede ser detectado por el algoritmo de la manera siguiente: cuando en la rutina de eliminación conseguimos que un camino expandido elimina a uno listado, en ese momento verificamos si el vértice terminal del camino expandido aparece dos veces en el camino. Si este es el caso, entonces el trozo del camino expandido entre las dos ocurrencias del vértice es un circuito de costo negativo.

## V.2 ALGORITMOS PARTICULARES DE BÚSQUEDA DE CAMINOS DE COSTO MÍNIMO

En la sección anterior presentamos un modelo general de algoritmo de búsqueda de caminos mínimos basado en el modelo de etiquetamiento. En esta sección presentamos casos particulares de este modelo. Cada algoritmo estará caracterizado por la regla de elección del próximo abierto a ser cerrado. Entre los algoritmos que veremos están: el de Dijkstra, la técnica Branch and Bound y el algoritmo de Bellman (técnica de la Programación Dinámica) para clases de grafos particulares. En cada caso discutiremos la complejidad en tiempo y consideraciones de espacio de almacenamiento. Por último presentamos el algoritmo de Floyd para el cálculo de los caminos mínimos entre cada par de vértices; este algoritmo es una generalización del algoritmo de Roy-Warshall utilizado en el capítulo III para determinar la matriz de alcance.

Muchas veces interesa determinar el mejor camino desde un vértice s hasta algún vértice del conjunto O que llamaremos *conjunto objetivo*. Si éste es el caso, podríamos agregar una condición de parada a nuestro modelo general, de tal forma que cuando entre los cerrados exista uno con vértice terminal en O entonces termine la ejecución del algoritmo. Podemos suponer que cuando O es vacío, estamos en el problema original de búsqueda de caminos a todos los alcanzables desde el vértice de partida.

Consideraremos que tenemos en esta sección un conjunto objetivo O.

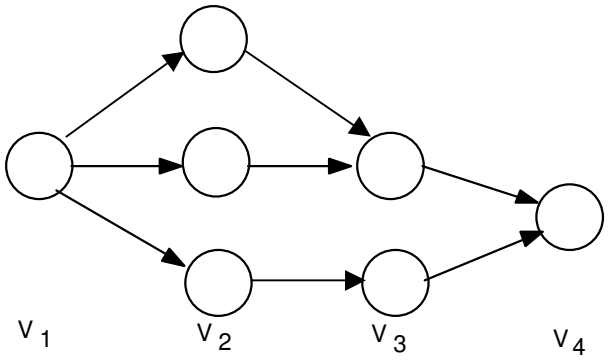
Hablamos de *algoritmos de búsqueda no informados* cuando éstos no utilizan información contenida en los elementos listados con el fin de alcanzar más rápido el objetivo, es decir, ninguna información en cuanto a la proximidad al objetivo es utilizada para guiar la elección de los elementos a expandir (y cerrar) en cada iteración.

Los algoritmos no informados más conocidos son los algoritmos de búsqueda en amplitud (técnica de Programación Dinámica Progresiva), la búsqueda en profundidad (técnica de enumeración implícita no informada o "Backtracking") y el algoritmo de Dijkstra que se utiliza en grafos con costos no negativos.

### V.2.1 ALGORITMO DE BÚSQUEDA EN AMPLITUD PARA CAMINOS DE COSTO MÍNIMO

Este algoritmo se obtiene del modelo general aplicando la regla siguiente para escoger el próximo camino a cerrar: Escoja el primer abierto que fue colocado en la lista entre todos los abiertos. Como vimos en la sección IV.2.7, a pesar de que la regla de eliminación varía con respecto a la presentada en esa sección, en la lista los abiertos siguen poseyendo en cada iteración largo k ó k+1 para algún k. Esto se puede interpretar diciendo que el algoritmo explora el grafo por etapas, en una etapa se consideran todos los caminos de un mismo largo k, en la etapa siguiente se consideran los caminos de largo k+1 y así sucesivamente.

Decimos que un grafo G=(V,E) es *en camadas* si V puede ser particionado en conjuntos $V_i$, i=1,...,q, de modo que si $v \in V_i$ entonces los sucesores de V están en $V_{i+1}$ (ver figura V.1).




Grafo en Camadas

**figura V.1**

El algoritmo de búsqueda en amplitud aplicado a un grafo en camadas se convierte en una técnica muy conocida de búsqueda de estrategias óptimas en problemas de decisiones secuenciales; esta técnica se denomina Programación Dinámica Progresiva. Las características del algoritmo aplicado a este tipo de grafos, son las siguientes:

- Se cierran todos los vértices de una camada antes de comenzar a cerrar los de la camada siguiente. Al terminar de cerrar los de una camada, están abiertos todos los vértices de la siguiente.

- Los cerrados no son eliminados. Esto garantiza que el número de iteraciones del algoritmo es igual al número de vértices alcanzables desde s. Se podría implementar el algoritmo de manera que en cada iteración el número de operaciones realizables sea de orden $\Delta^+(G)$ (grado máximo exterior de G); bastaría con hacer algunas modificaciones al algoritmo de visita de vértices por búsqueda en amplitud presentado en la sección IV.2.7.

- Se tendrá en memoria sólo los elementos en la camada que se está expandiendo y la camada resultante de la expansión.

### V.2.2 ALGORITMO DE BÚSQUEDA EN PROFUNDIDAD PARA CAMINOS DE COSTO MÍNIMO

Este algoritmo se obtiene del modelo general, aplicando la regla siguiente de elección del próximo abierto a ser cerrado: escoger para cerrar y expandir al último camino que se haya abierto (último que entra primero que sale).

Como vimos en las propiedades (8) y (9) de la sección IV.2.1, en cada iteración se cierra un abierto de largo o profundidad máxima. Si se abre alguno de los caminos acabados de expandir en una iteración, en la siguiente uno de estos caminos expandidos será cerrado por el algoritmo, continuando de esta forma hasta alcanzar un vértice que no posee sucesores o que todos los caminos resultantes de la expansión hayan sido eliminados por la rutina de eliminación. En cualquiera de los casos el siguiente camino a ser cerrado será el de largo máximo entre los abiertos.

En cada iteración la lista tiene un formato muy particular. Si P es el camino a expandir en una iteración cualquiera y $P_0,P_1,...,P_p$ es la secuencia de caminos cerrados a partir de P siguiendo los apuntadores, entonces todos los abiertos apuntan a uno de los caminos en la secuencia (ver figura IV.2.1.3). Por lo tanto, si el largo máximo de un camino elemental de G es p, el número de abiertos no supera a $(\Delta^+(G)-1)p+1$.

En problemas con objetivo, si el objetivo es un conjunto numeroso, el algoritmo de búsqueda en profundidad podría ser utilizado para conseguir rápidamente soluciones sub-óptimas. Estas soluciones pueden ofrecer información para la construcción de algoritmos informados. Esta última técnica se conoce en la literatura como algoritmos de "Branch-and-Bound".

### V.2.3 ALGORITMO DE DIJKSTRA

Los dos algoritmos estudiados anteriormente tienen la ventaja de utilizar "poca" memoria. Sin embargo son

poco eficientes pues no toman en consideración los costos y exploran indiscriminadamente todos los caminos que parten de s. En el problema de la arborescencia de caminos mínimos, los caminos que más prometen ser óptimos entre los abiertos son aquéllos de costo mínimo. El algoritmo de Dijkstra se obtiene estableciendo el siguiente criterio de elección de abiertos a cerrar: escoger el abierto de menor costo, de esta manera la exploración del grafo será hecha en camadas de caminos de costo similar salvo en el caso de que existan costos negativos.

Cuando los costos son no negativos, este criterio de elección implicará resultados importantes.

Sea G un grafo orientado donde todo arco tiene asociado un costo no negativo.

**Proposición V.2.1**

Suponga que un algoritmo cualquiera, según el modelo, es aplicado a G a partir de s. Al comienzo de cualquier iteración si $P=(n^*,c^*,p^*)$ es un abierto con $c^*=\min\{c/(n,c,p)$ es un abierto$\}$ entonces $c^*=cmin(s,n^*)$.

**Demostración:**

Supongamos por el absurdo que $c^* > cmin(s,n^*)$. Sabemos que no existe un cerrado con vértice terminal $n^*$ y, por la proposición V.1.5, existe un abierto (n,cmin(s,n),.) donde n es un vértice intermedio de un camino óptimo de s a $n^*$.

Por otro lado, $cmin(s,n) \leq cmin(s,n^*)$ pues los costos son no negativos. Así $cmin(s,n) < c^*$ lo que contradice la definición de $c^*$.

¤

Las consecuencias inmediatas de esta proposición son las siguientes:

(1) Cuando el algoritmo de Dijkstra es utilizado con un grafo de costos no negativos, cualquier cerrado P=(n,c,p) satisface c=cmin(s,n). Esto implica que los cerrados no son eliminados y así el número de iteraciones del algoritmo es igual al número de vértices alcanzables desde s.

(2) Los costos de los elementos cerrados por el algoritmo de Dijkstra, en grafos con costos no negativos, crecen con las iteraciones del algoritmo. Por lo tanto, si el problema incluye un conjunto objetivo O, el primer camino cerrado con vértice terminal en el objetivo es una solución del problema, es decir, es el camino de menor costo desde s a cualquier vértice del objetivo.

Note que el algoritmo de búsqueda en amplitud no es más que un caso particular del algoritmo de Dijkstra, escogiendo para cerrar al abierto de longitud mínima. En este caso el costo de cada arco es 1.

Como dos caminos listados no pueden tener igual vértice terminal, podemos hacer una implementación del algoritmo de Dijkstra de manera que a cada vértice del grafo le asociamos: la etiqueta "cerrado" o "abierto", un apuntador al vértice predecesor en el camino y el costo del camino. La diferencia entre esta implementación y las ternas (n,c,p) es que p ya no apunta a otra terna sino a un vértice al cual podemos tener acceso inmediato, lo cual nos permite reducir el número de operaciones de la rutina de eliminación a $O(\Delta^+(G))$.

A continuación damos la implementación del algoritmo de Dijkstra según los comentarios anteriores.

Algoritmo de Dijkstra:

{ Entrada: Un digrafo *G* sin arcos múltiples. Una función de costos $c: E(G) \rightarrow \Re$, con $c(e) \geq 0$ $\forall e \in E(G)$. Y *s* un vértice de *G*.

Salida: Un apuntador (o referencia a un vértice) y un costo asociados a cada vértice alcanzable desde *s*. El camino de costo mínimo hasta un determinado vértice puede ser reconstruido siguiendo los apuntadores. Los vértices que resulten con costos $+\infty$ son los no alcanzables desde *s* }

Variable *n,m* : vértice;
Comienzo
(1) Inicialmente *s* está abierto, tiene costo cero y apuntador nulo. Todos los demás vértices están abiertos y tienen costo $+\infty$ (un valor muy grande);
(2) Mientras Existan vértices abiertos hacer:
Comienzo
(2.1) Escoger un vértice abierto *n* con el menor costo;
(2.2) Cerrar *n*;
(2.3) Para cada vértice abierto *m* sucesor de *n* hacer:
Si el costo de $n + c(n,m)$ es menor que el costo de *m* entonces

asociar a *m* un apuntador hacia *n* y reemplazar el costo de *m* por el costo de $n + c(n,m)$

fin

fin

Por convención el resultado de las sumas con algún operando igual a $+\infty$ seguirán siendo $+\infty$. Note que el conjunto de vértices abiertos puede ser representado por una cola de prioridades (ver VII.4.1) y el grafo por una lista de adyacencias. De esta forma el paso (2.1) es O(1), el paso (2.2), que significa eliminar el primer vértice de la cola de prioridades, es $O(\log_2$ (número de elementos en la cola)), si la cola se implementa como un "heap" binario en un arreglo.

En el paso (2.3) se efectuarán $d^+(n)$ iteraciones. En cada iteración, si el costo de m varía entonces se debe reestructurar la cola de prioridades y esta operación es $O(\log_2$(número de elementos en la cola)). El número de iteraciones de (2) es igual al número de vértices del grafo y, en cada iteración, el número de elementos de la cola de prioridades se reduce en una unidad. Por lo tanto la complejidad en tiempo del algoritmo de Dijkstra sería:

$$\sum_{v \in V(G)} (c_1 * \log_2(|V(G)|) + d^+(v) * c_2 * \log_2(|V(G)|)) \leq O((|V(G)| + |E(G)|) * \log_2(|V(G)|))$$

Así conseguimos una implementación $O((|V|+|E|).\log_2|V|)$ del algoritmo de Dijkstra.

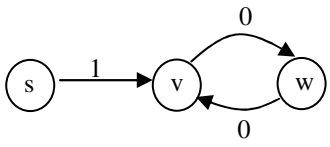
### V.2.4 ALGORITMO DE BELLMAN

El algoritmo de Bellman permite calcular todos los caminos mínimos que parten de un vértice s. Se supone que el vértice s es raíz del grafo y que el grafo no posee circuitos. Este algoritmo se puede ver como una generalización del algoritmo de búsqueda en amplitud en grafos en camadas y puede verse como una generalización de la técnica de Programación Dinámica Progresiva para el cálculo de caminos mínimos desde s. Esta técnica está basada en la fórmula recursiva siguiente:

Si C(v) representa el costo de un camino mínimo desde s hasta v:

$C(v) = \min \{C(w) + c(w,v) / w$ es predecesor de $v\}$

Note que si el grafo tiene circuitos entonces la fórmula recursiva anterior no funcionaría, como se puede comprobar con en el grafo siguiente:




El algoritmo de Bellman es también un caso particular del modelo de algoritmo de búsqueda de caminos mínimos, bajo la suposición de que s sea una raíz de G. La regla de elección del próximo abierto a ser cerrado es : "Escoger un abierto tal que haya un camino listado cerrado hasta cada predecesor del vértice terminal de ese camino abierto". Obsérvese que la hipótesis del modelo general: $cmin(s,v) > -\infty$, $\forall v \in V(G)$, se satisface para grafos sin circuitos. La regla de elección garantiza que un camino cerrado no es eliminado y que en cada iteración exista un abierto con todos sus predecesores como vértices terminales de caminos cerrados. Esto se debe a que si $m_1,...,m_p$ son los predecesores de n entonces todo camino de s a $m_i$ , $\forall i$, no pasa por n ni por ningún ascendiente de n por ser G sin circuitos.

Si s no es raíz de G podremos aplicar el algoritmo de Bellman de la siguiente forma: se determinan los vértices alcanzables desde s y la regla de elección sería “escoger un abierto tal que exista un camino listado cerrado hasta cada predecesor, alcanzable desde s, del vértice terminal del abierto”.

Una implementación elegante y simplificada del algoritmo de Bellman es la siguiente:

Suponga que s es raíz de G. G viene representado por una lista de adyacencias. A cada vértice v asociamos su grado interior o de entrada *grado(v)* $=d^-(v)$, el costo *costo(v)* del camino mínimo de s a v y un apuntador al predecesor en el camino. Inicialmente T={s}, costo(s)=0, costo(v)=$+\infty$ $\forall v \neq s$, *Apuntador(s)* =NULO.

En cada iteración se toma un elemento n de T y se elimina de T. Para cada sucesor m de n se efectúa lo siguiente:

- grado(m)←grado(m)-1.
- Si grado(m)=0 entonces T←T∪{m}
- Si costo(m) > costo(n) + c(n,m) entonces
  costo(m)←costo(n) + c(n,m) y Apuntador(m)←n

Algoritmo de Bellman:

{ Entrada: Un digrafo *G* sin circuitos y una raíz *s*.
Salida: Para cada vértice v de G un apuntador al predecesor en un camino mínimo de s a v y el costo de ese camino hasta v.}

Variable *T* : conjunto de vértices;
*s*,*v*,*n*,*m* : vértice;

Comienzo
(1) *T* ←{*s* };
(2) costo(*s* )←0; Apuntador(*s* )←NULO;
(3) $\forall v \in V(G)$, $v \neq s$: costo(*v* )←$+\infty$;
(4) Mientras *T*≠ø hacer:
Comienzo
Tomar un vértice *n* de *T* ;
*T* ← *T* - {*n* };
Para cada sucesor *m* de *n* hacer:
Comienzo
grado(*m* )←grado(*m* )-1;
Si grado(*m* )=0 entonces *T* ← *T* ∪{*m* };
Si costo(*m* )>costo(*n* )+c(*n*,*m*) entonces
Comienzo
costo(*m* )←costo(*n* )+c(*n*,*m*);
Apuntador(*m* )← *n*;
fin
fin
fin
fin.

El cuerpo del paso (4) se realiza una sola vez por cada vértice de G y en cada iteración el número de operaciones es O(max(1,$d^+$(n)). Por lo tanto este algoritmo es O(max(|V(G)|, |E(G)|).

Los costos de los arcos pueden ser negativos al aplicar el algoritmo de Bellman, debido a que el grafo no posee circuitos y en estos grafos se cumple la hipótesis cmin(s,v) > $-\infty$ ,$\forall$ v∈V(G). Esto nos sugiere la posibilidad de hallar con el mismo algoritmo un camino de costo máximo; basta con cambiar el signo a los costos de los arcos.

### V.3 CAMINOS MÍNIMOS ENTRE CADA PAR DE VÉRTICES

En esta sección suponemos que todo circuito tiene costo no negativo. Sean $v_1,v_2,...,v_n$ los vértices de un digrafo G. Supongamos que conocemos un camino mínimo desde cualquier vértice v a cualquier vértice w de G entre todos los caminos de v a w cuyos vértices interiores (es decir, vértices por donde pasa el camino y distintos de v y w) están en $\{v_1,...,v_i\}$. Podemos suponer que estos caminos son elementales pues todo circuito tiene costo no negativo. Un camino elemental de costo mínimo P de v a w, entre todos los caminos de v a w cuyos vértices interiores están en $\{v_1,...,v_{i+1}\}$ cumple con una de las dos propiedades siguientes:

(a) Ningún vértice interior de P es igual a $v_{i+1}$. En cuyo caso P es un camino mínimo de v a w entre todos los caminos de v a w cuyos vértices interiores están en $\{v_1,...,v_i\}$.

(b) Si $v_{i+1}$ es un vértice interior de P, P=<v,...,$v_{i+1}$,...,w>, sean Q=<v,...,$v_{i+1}$> y R=<$v_{i+1}$,...,w> con P=Q || R. Entonces los vértices interiores de Q y R están en $\{v_1,...,v_i\}$ pues P es elemental. Q (R) es un camino mínimo de v a $v_{i+1}$ (de $v_{i+1}$ a w) entre todos los caminos de v a $v_{i+1}$ (de $v_{i+1}$ a w) cuyos vértices interiores están en $\{v_1,...,v_i\}$. Esto último se debe al principio de optimalidad (Proposición V.1).

Las dos propiedades anteriores indican que un camino mínimo P de v a w, entre todos los caminos de v a w con vertices interiores en $\{v_1,...,v_{i+1}\}$, se puede construir a partir de los caminos mínimos con vértices interiores en $\{v_1,...,v_i\}$. Lo hacemos comparando el costo de un camino mínimo S de v a w con vértices interiores en $\{v_1,...,v_i\}$ y el costo de un camino T de v a w pasando por $v_{i+1}$. Este último costo es la suma del costo de un camino mínimo Q de v a $v_{i+1}$ con vértices interiores en $\{v_1,...,v_i\}$ más el costo de un camino mínimo R de $v_{i+1}$ a w con vértices interiores en $\{v_1,...,v_i\}$. Si costo(S) < costo(Q)+costo(R) entonces P=S, si no P=Q || R. Note que costo(P) es igual al costo de un camino mínimo pasando por $\{v_1,...,v_{i+1}\}$.

Hemos así conseguido un método recursivo para calcular un camino mínimo de v a w, $\forall v,w \in V(G)$. Cuando $i=|V(G)|$ habremos conseguido un camino mínimo de v a w, $\forall v,w \in V(G)$. Este algoritmo se conoce como Algoritmo de Floyd. El método es similar al algoritmo de Roy-Warshall para calcular la matriz de alcance (capítulo III).

Inicialmente cuando i=0, es decir el conjunto de vértices interiores es vacío, los caminos iniciales son los arcos del grafo. Por lo tanto podemos partir de una matriz $C=(c_{ij})$, $|V(G)|x|V(G)|$, donde $c_{ij}=c(i,j)$ si $(i,j)\in E(G)$ ó $c_{ij}=+\infty$ si $(i,j)\notin E(G)$.

Haremos una implementación del algoritmo donde sólo calcularemos los costos de los caminos mínimos entre cada par de vértices, sin calcular los caminos en sí. El cálculo de los caminos puede hacerse sin necesidad de guardar para cada par de vértices la secuencia de vértices que lo define, basta tener una matriz A, $|V(G)|x|V(G)|$, de índices, donde el elemento $a_{ij}$ puede contener tres tipos de valores:

- un índice de un vértice intermedio del camino de i a j.
- un indicador de que (i,j) es un arco.
- un indicador de que no hay camino de i a j.

Recursivamente se podrá construir el camino de i a j pues este será:

- (El camino de i a $a_{ij}$) || (El camino de $a_{ij}$ a j) ó
- <i,(i,j),j> si (i,j) es un arco ó
- No hay camino de i a j.

Algoritmo de Floyd:

{ Entrada: Un digrafo *G* sin arcos múltiples. Una función de costos no negativos $c$ :$E(G)\rightarrow\Re^{\geq 0}$. El número $n$ de vértices del grafo. Una matriz $C=(c_{ij})$ con $c_{ij}=c(i,j)$ =costo del arco *(i,j )*. Si no existe arco de $i$ a $j$, $c_{ij}$ es $+\infty$, si $i=j$ entonces $c_{ij}=0$.

Salida: Matriz $C=(c_{ij})$ con $c_{ij}$ =costo de un camino de costo mínimo de $i$ a $j$ o $+\infty$ si no existe camino.}

Variables *i,j,k* : índice de vértice;
Comienzo
  Para $k \leftarrow 1$ a $n$ hacer:
    Para $i \leftarrow 1$ a $n$ hacer:
      Si $(C[i,k] < +\infty)$ y $i \neq k$ entonces
        Para $j \leftarrow 1$ a $n$ hacer:
          Si $(j \neq k) \wedge (i \neq j) \wedge (C[k,j] < +\infty)$ entonces $C[i,j] \leftarrow \min(C[i,j], C[i,k]+C[k,j])$
fin.

### V.4 ALGORITMOS INFORMADOS DE BÚSQUEDA DE CAMINOS MÍNIMOS

En esta sección continuamos tratando el problema de encontrar un camino de costo mínimo desde un vértice s, de un grafo dirigido sin arcos múltiples, hasta un conjunto O de vértices que hemos llamado conjunto objetivo. Cada arco (v,w) tendrá un costo asociado c(v,w).

**Definición V.4.1**

Si A y B son dos conjuntos de vértices de un grafo orientado G=(V,E), definimos *el costo mínimo para ir de* A *a* B como:

h(A,B) = ínfimo {C(P) / P es un camino con extremo inicial en A, extremo final en B, y cualquier otro

vértice distinto de los extremos de P no está ni en A ni en B}.

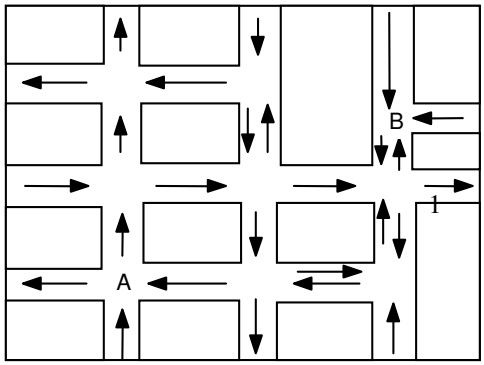
donde C(P) es el costo del camino P. Por convención ínfimo($\emptyset$) = $+\infty$.

Dado un conjunto objetivo O y un conjunto de vértices A, definimos:

h(A) = h(A,O) y g(A) = h({s},A). Si A = {n} escribimos h(n) y g(n) para simplificar la notación..

El problema de búsqueda con conjunto objetivo se formula de la siguiente forma:

Encontrar un vértice $n_0 \in O$ y un camino P de s a $n_0$ tal que C(P) = h(s) (= g(O)). Si O es vacío supondremos que el problema es determinar la arborescencia de caminos mínimos que parten de A.

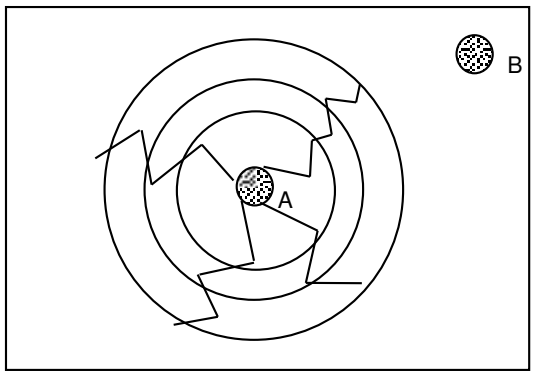



Mapa de la Ciudad

A: lugar de llegada B: lugar de partida

**figura V.2**

LLamaremos *algoritmos informados* a aquellos que poseen una información adicional a la topología del grafo, que permita alcanzar más rápido el objetivo O con un camino mínimo de s a O. Esta información adicional podría ser un estimado de h(n), $\forall n \in V(G)$. En la sección anterior todos los algoritmos vistos eran no informados pues los criterios de elección de caminos a expandir no toman en cuenta para nada la proximidad del vértice terminal del camino hasta el objetivo.




El algoritmo de Dijkstra considera los caminos
por "ondas" de costo similar

**figura V.3**

Un algoritmo informado permitiría llegar más rápido al objetivo, como lo muestra el ejemplo siguiente: suponga que usted se encuentra en un lugar A de una ciudad y desea desplazarse a otro lugar B de la ciudad. Usted

cuenta con un plano de la ciudad donde aparece por cada calle y avenida, la cantidad de metros entre intersecciones consecutivas. Además el mapa está cuadriculado de forma que podemos conocer las coordenadas de cada lugar en el mapa. En la figura V.2 se ilustra un mapa. Si usted desea conocer cuál es el camino a menor distancia para llegar a su objetivo podría emplear por ejemplo el algoritmo de Dijkstra. Este algoritmo avanza hacia el objetivo por etapas calculando todos los caminos de costo homogéneo hasta llegar al objetivo (ver figura V.3). Sin embargo, intuitivamente vemos que se puede mejorar la búsqueda tomando en cuenta la distancia euclideana entre el lugar de partida y el objetivo; es natural expandir primero aquellos caminos donde la suma del costo del camino con la distancia euclideana entre el vértice terminal del camino y el objetivo sea mínima. En la figura V.4 es preferible expandir el camino S antes que T pues S es más prometedor que T.

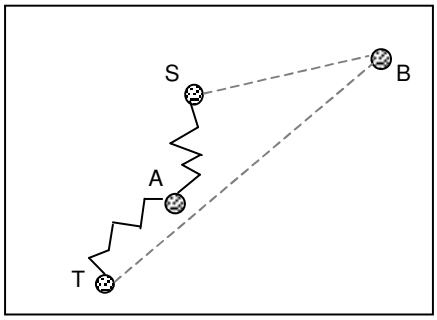



**figura V.4**

Supongamos que conocemos alguna información sobre h(n). Esta información puede ser utilizada de la manera siguiente:

(1) Si se conoce algún camino C de s a O, su costo puede ser utilizado como una cota superior en la búsqueda de nuevos caminos. Si el costo de un camino listado P con vértice terminal n, sumado a un subestimado(cota inferior) de h(n), supera al costo del camino C, P no puede ser parte del un camino mínimo hasta el objetivo y puede ser eliminado. Este método genera una familia de técnicas conocidas como "Branch-and-Bound" (en [4] se encuentra más información sobre la técnica “brach-and-bound”).

(2) Si conocemos un estimado de h(n), $\forall n \in V(G)$, esta estimativa puede ser utilizada para establecer las prioridades al momento de escoger un abierto a cerrar en el paso (1.1) del modelo general de algoritmo presentado en la sección V.1. El método resultante se conoce como *algoritmo A**.

En lo que sigue describiremos el algoritmo A* y sus propiedades, en función de las propiedades de una estimativa de h(.).

En esta sección suponemos que los grafos no poseen circuitos de costo negativo, es decir, $cmin(s,n) > -\infty$, $\forall n \in V(G)$. Supongamos que al grafo está asociada una función :

$$\hat{h} : V(G) \rightarrow \Re$$

Utilizaremos la versión 2 del modelo general de algoritmo de búsqueda de caminos mínimos que representa los caminos por ternas P=(n,c,p) y definimos:

$$\hat{f}(P) = c + \hat{h}(n)$$

Si $\hat{h}$ es una estimativa de h, entonces $\hat{f}$ es una estimativa del costo de un camino de costo mínimo hasta el objetivo O, restringido a tener su parte inicial igual a P. Así, $\hat{f}(P)$ estima lo que se puede esperar de la expansión del camino P. La regla de elección del próximo abierto a cerrar y expandir: "escoger un abierto P con menor $\hat{f}(P)$", da preferencia a los caminos abiertos más prometedores con la información que se dispone.

A continuación presentamos el algoritmo A*.

Algoritmo A*:

{ Entrada: Un digrafo *G* sin arcos múltiples, con costos en los arcos y sin circuitos de costo negativo. Un vértice *s* de G. $\forall n \in V(G)$ [cmin(s,n)>-∞]. Una función estimativa $\hat{h}$ de h. Un conjunto objetivo *O*.

Salida: Un camino P de *s* a *O* (si existe) que satisface el criterio de parada. Si no existe tal camino, P

contendrá la secuencia vacía.}
Comienzo

(0) Abrir el elemento $P_0=(s,0,*)$ y asociarle $\hat{f}(P_0) = \hat{h}(s)$. Inicialmente no se ha satisfecho el criterio de parada y P es la secuencia vacía;

(1) Mientras Existan elementos abiertos y no se haya satisfecho el criterio de parada hacer:

Comienzo

(1.1) Escoger un elemento abierto $P_j=(n_j,c_j,p_j)$ tal que $\hat{f}(Pj)$ sea mínimo entre los abiertos;`

(1.2) Cerrar $P_j$;

(1.3) Si $n_j \in O$

entonces Si $P_j$ satisface el criterio de parada entonces asignar a P el camino $P_j$

si no

Comienzo

- Obtener los sucesores de $n_j$: $n^1,...,n^q$;
- Construir los caminos expandidos $P^i=(n^i,c^i,j)$, i=1,2,...,q,, de $P_j$ a cada sucesor de $n_j$ donde $c^i=c_j+c(n_j,n^i)$;
- Asociar a cada camino $P^i$ el valor: $\hat{f}(P^i) = c^i + \hat{h}(n^i)$;
- Aplicar la rutina de eliminación a los $P^i$ (la versión 2 del modelo general de la sección V.1);

fin

fin

fin.

**Observaciones:**

- El criterio de parada debe cumplir con la restricción de que sólo puede ser satisfecho cuando se cierra algún camino (el que permite satisfacer el criterio de parada) cuyo vértice final esté en el objetivo O. Por ejemplo, un criterio de parada podría ser "el primer camino que se cierre y que llegue al objetivo" , otro criterio podría ser "cuando el costo del camino que se cierre y llega al objetivo, sea menor que una cierta cantidad preestablecida".
- Si el conjunto objetivo no es alcanzable desde s entonces el algoritmo A* termina con una arbor4escencia de caminos mínimos hasta los vértices alcanzables desde s y P será la secuencia nula.

La proposición V.1.6 garantiza que A* termina con una arborescencia de caminos mínimos hasta los vértices alcanzables desde s si no hay criterio de parada ni conjunto objetivo.

**Definición V.4.2**

Una estimativa $\hat{h}$ se llama *admisible* si $\forall n \in V(G)$: $\hat{h}(n) \leq h(n)$, es decir, $\hat{h}$ es una subestimativa de h.

**Proposición V.4.1**

Supongamos que $\hat{h}$ es admisible y el objetivo es alcanzable desde s.

Si al comienzo de una iteración cualquiera no se ha cerrado todavía un camino con vértice terminal en O entonces el algoritmo A* cierra un $P_j$ con $\hat{f}(P_j) \leq h(s)$.

**Demostración:**

Supongamos por el absurdo que en una iteración el algoritmo cierra $P_j$ con $\hat{f}(P_j) > h(s)$.

La proposición V.1.5 garantiza la existencia de un abierto mínimo P=(n,cmin(s,n),.). Se tiene que:

$\hat{f}(P) = g(n) + \hat{h}(n) = cmin(s,n) + \hat{h}(n) \leq$

$cmin(s,n) + h(n)$ (pues $\hat{h}$ es admisible)

$= h(s)$ (pues P es mínimo)

$< \hat{f}(P_j)$

lo que contradice la elección de $P_j$ y completa la demostración.

¤

La proposición siguiente nos dice que el primer camino que encuentre el algoritmo con vértice terminal en O,

es un camino mínimo de s a O, si $\hat{h}$ es admisible. De esta forma el criterio de parada será simplemente: $n_j \in O$, en el algoritmo.

**Proposición V.4.2**

Sea $\hat{h}$ admisible en el algoritmo. Si $P_j$ es el camino escogido en el paso (1.1) la primera vez que $n_j \in O$, entonces $P_j$ es un camino de costo mínimo de s a O, $C(P_j)=h(s)$.

**Demostración:**

Por la proposición V.4.1:

$\hat{f}(P_j) = c_j + \hat{h}(n_j) \leq h(s)$

Como $\hat{h}(n_j) = 0$ entonces $c_j \leq h(s)$. Por lo que $P_j$ es de costo mínimo de s a O

¤

Trataremos de ver como conseguir propiedades semejantes a las del algoritmo de Dijkstra. En particular, en que casos A* tiene un comportamiento polinomial en función del número de vértices.

**Definición V.4.3**

Decimos que una estimativa $\hat{h}$ es *consistente* si es admisible y $\forall n,m \in V(G)$: $\hat{h}(n) \leq h(n,m) + \hat{h}(m)$. Esto es, la estimativa tiene un comportamiento similar a la distancia entre vértices de un grafo.

**Proposición V.4.3**

Sea $\hat{h}$ una estimativa consistente. Entonces en cualquier iteración se tiene que $\hat{f}(P_j) \leq \hat{f}(P^i)$, i=1,...,q, donde los $P^i$ son los elementos expandidos a partir del elemento $P_j$ escogido en el paso (1.1).

**Demostración:**

Sea $P^i=(n^i,c^i,j)$ un elemento expandido a partir de $P_j$:

$$\begin{aligned}\hat{f}(P^i) &= c^i + \hat{h}(n^i)\\ &= c_j + c(n_j,n^i) + \hat{h}(n^i)\\ &\geq c_j + h(n_j,n^i) + \hat{h}(n^i) \text{ (por definición de } h(n_j,n^i))\\ &\geq c_j + \hat{h}(n_j) \text{ (pues } \hat{h}(n_j) \leq h(n_j,n^i) + \hat{h}(n^i))\\ &= \hat{f}(P_j)\end{aligned}$$

¤

De la proposición anterior se concluye que $\hat{f}(P_j)$ crece en cada iteración. El algoritmo de Dijkstra es un caso particular de A* tomando $\hat{h} = 0$, la cual es consistente cuando los costos son no negativos.

**Proposición V.4.4**

Sea $\hat{h}$ consistente. Todo elemento cerrado P=(n,c,p) satisface:

$c=g(n)=cmin(s,n)$

Es decir, P es de costo mínimo.

**Demostración:**

Suponga que $\bar{P} = (\bar{n}, \bar{c}, \bar{p})$ es cerrado en alguna iteración con $\bar{c} > g(\bar{n})$.

De acuerdo a la proposición V.1.5, al comienzo de esa iteración existe un abierto P=(n,c,p) en esa iteración tal que:

$c=g(n),\ c+h(n,\bar{n}) = g(\bar{n})$

Por lo tanto, $\hat{f}(P) = g(n) + \hat{h}(n)$

$$\begin{aligned}&\leq g(n) + h(n,\bar{n}) + \hat{h}(\bar{n}) \text{ (pues } \hat{h} \text{ es consistente)}\\ &= g(\bar{n}) + \hat{h}(\bar{n})\\ &< \bar{c} + \hat{h}(n)\end{aligned}$$

$= \hat{f}(\bar{P})$  (pues $\bar{c} > g(\bar{n})$ )

lo que contradice la elección de $\bar{P}$ en el paso (1.1).

¤

Las dos últimas proposiciones garantizan que el número de iteraciones del algoritmo A* es de orden lineal en función del número de vértices del grafo cuando la estimativa es consistente. Note que este resultado es independiente del signo de los costos de los arcos.

### V.5 GRAFOS DE PRECEDENCIA.

Un *grafo de precedencia* es un grafo dirigido sin circuitos. Muchas son las aplicaciones donde encontramos grafos de precedencia. El concepto de orden entre los elementos de un conjunto aparece con frecuencia en Computación. Veamos algunos ejemplos:

(1) Podemos representar una expresión aritmética:

((a+b)*c + ((a+b)+e) * (e+f)) * ((a+b) * c)

mediante el digrafo sin circuitos de la figura V.5.

(2) El grafo reducido de un grafo dirigido (sección III.3.1).

(3) El grafo de una relación de orden parcial (eliminando los bucles).

(4) El grafo que representa un proceso de decisión secuencial como el de la figura V.6.

(5) Planificación de proyectos: un proyecto se puede dividir en actividades que deben realizarse en un cierto orden. Tal es el caso de un pensum de estudios de una carrera universitaria, donde ciertas materias son requisitos de otras. En este caso podemos representar las materias por vértices de un grafo donde los arcos son los requisitos inmediatos entre materias.

En este capítulo veremos algunas propiedades y algoritmos importantes en grafos de precedencia.

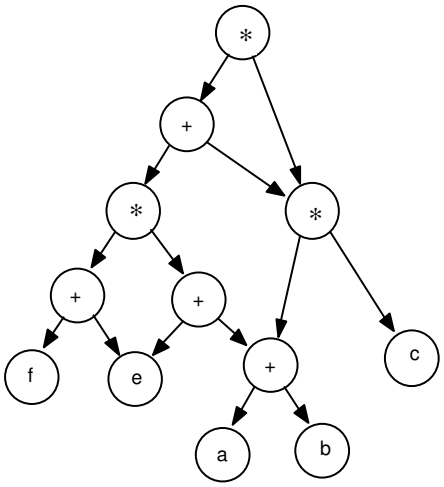



**figura V.5**

#### V.5.1 PROPIEDADES

Empezamos por dar dos propiedades sencillas pero importantes de grafos sin circuitos:

(a) Un grafo G no posee circuitos si y sólo sí todo subgrafo de G no posee circuitos.

(b) Principio de dualidad de grafos sin circuitos:

Un grafo G no posee circuitos si y sólo si el *grafo inverso* de G, $G^{-1}$, obtenido a partir de G invirtiendo la orientación de todos los arcos, no posee circuitos.

El estudio de grafos sin circuitos se fundamenta esencialmente en las propiedades siguientes:

**Proposición V.5.1.1**

Sea G un digrafo sin bucles. G no posee circuitos si y sólo si toda componente fuertemente conexa es un vértice aislado.

**Demostración:**

Al haber una componente fuertemente conexa con más de un vértice es evidente que habrá un circuito. Recíprocamente, si G posee un circuito, éste estará en una componente fuertemente conexa.

¤

**Proposición V.5.1.2**

Un digrafo G es de precedencia si y sólo si todo camino es elemental.

**Demostración:**

Que exista un camino no elemental es equivalente a decir que existe un camino cerrado. La proposición III.1.3.5 nos dice que existe un camino cerrado si y sólo si existe un circuito.

¤

Un vértice *fuente* de un digrafo es un vértice con grado interior igual a cero, es decir, el vértice es extremo inicial de todos los arcos incidentes en él. Un vértice *sumidero* es un vértice con grado exterior igual a cero, es decir, el vértice es extremo terminal de todo arco incidente en él.

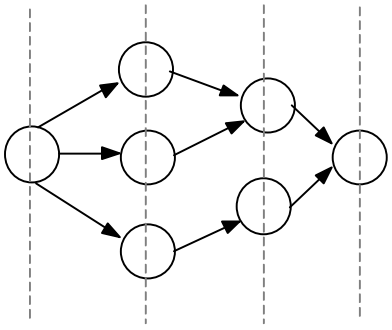


**figura V.6**

**Proposición V.5.1.3**

Todo grafo de precedencia G contiene un vértice fuente y un vértice sumidero.

**Demostración:**

Supongamos que G no contiene un vértice fuente. Entonces $\delta^-(G) \geq 1$ y la proposición III.1.3.11 nos dice que G posee un circuito.

Por otro lado, si G es de precedencia, entonces el grafo inverso de G, $G^{-1}$, es de precedencia y por lo tanto $G^{-1}$ posee una fuente, la cual es sumidero de G.

¤

A continuación resaltaremos algunas características estructurales de los grafos de precedencia mediante dos números que asociaremos a cada vértice del grafo.

A todo vértice v de un digrafo cualquiera podemos asociar dos números enteros:

– $\eta(v)$ = nivel de v = largo máximo de un camino elemental terminando en v.
- h(v) = altura de v = largo máximo de un camino elemental comenzando en v.

En grafos de precedencia podemos eliminar el término "elemental" de estas definiciones. Note que el nivel de un vértice en G es la altura de ese vértice en $G^{-1}$, por lo que nivel y altura son conceptos duales.

**Proposición V.5.1.4**

En un grafo de precedencia, para todo vértice v del grafo, el vértice inicial de un camino de largo máximo terminando en v es una fuente.

**Demostración:**

Si v es una fuente, el resultado es evidente. Si v no es fuente, existe un camino de largo $\geq 1$ comenzando en w ($\neq$v) y terminando en v. Si este camino es de largo máximo entonces cualquier predecesor de w debe estar en el camino. Como el grafo no posee circuitos, el vértice w no puede poseer predecesores, es decir, w es una fuente.

¤

**Proposición V.5.1.5**

En un grafo de precedencia, para todo vértice v del grafo, el vértice terminal de un camino de largo máximo comenzando en v es un sumidero.

**Demostración:**

Aplicar el principio de dualidad y la proposición V.5.1.4.

¤

La estructura simple que posee un grafo sin circuitos es puesta de manifiesto por el resultado siguiente:

**Proposición V.5.1.6**

Un digrafo G=(V,E) es de precedencia si y sólo si V se puede particionar en conjuntos $V_0$, $V_1$,..., $V_p$ tal que $\forall$i, $0 \leq i \leq p$, se tiene: $v \in V_i$ si y sólo si i es el largo de un camino más largo terminando en v.

**Demostración:**

Mostremos la condición necesaria.
Si G no posee circuitos, consideremos la secuencia, bien definida según la proposición V.5.1.3 y la propiedad (a):

$V_0$ = Conjunto de vértices fuentes de G.
$V_1$ = Conjunto de vértices fuentes de $G_{V-V_0}$
....
$V_i$=Conjunto de los vértices fuentes de $G_{V-V_0-V_1-..-V_{i-1}}$

La secuencia $V_0$, $V_1$, ... , $V_p$ (con $V_p$ tal que $G_{V_p}$ contiene sólo vértices aislados) es una partición de V. Basta mostrar ahora que si $v \in V_i$ entonces el camino más largo terminando en v es de largo i. Razonemos por inducción sobre i. Para i=0 es evidente. Supongamos que para $i \leq k$ se cumple que si $v \in V_i$ entonces el camino más largo terminado en v es de longitud i. Sea $v \in V_{k+1}$. Existe $w \in V_k$ tal que $(w,v) \in E$ y por hipótesis de inducción existe entonces un camino de largo k+1 hasta v. No puede haber un camino hasta v de largo mayor que k+1 pues el vértice w predecesor de v en ese camino está en $V_0 \cup ... \cup V_k$, lo cual implica que un camino más largo hasta w es de largo $\leq$ k, contradiciendo la existencia de tal camino.

Es importante notar que $V_i$, $0 \leq i \leq p$, es un conjunto estable de G.

Condición suficiente:

De acuerdo a la hipótesis, todo camino en G es de largo finito. Esto significa que todo camino de largo > 0 es elemental. La proposición V.5.1.2 nos dice que G debe ser de precedencia.

¤

De acuerdo a la proposición anterior si G es de precedencia entonces la partición $V_o$,...,$V_p$, se llama *partición*

*en niveles pues* se tiene que $v \in V_i$ si y sólo si $\eta(v)=i$ (ver figura V.7).

### V.5.1.1 RELACIONES DE ORDEN Y GRAFOS DE PRECEDENCIA

Un digrafo de precedencia sin arcos múltiples G=(V,E) induce un orden parcial estricto sobre los elementos de V. Basta tomar la clausura transitiva $E^+$ de la relación binaria E. Decimos que E es una *relación de precedencia inmediata* si $\forall (v,w) \in E$ no existe un camino en G de largo mayor que 1 de v a w; también decimos que G es de precedencia inmediata.

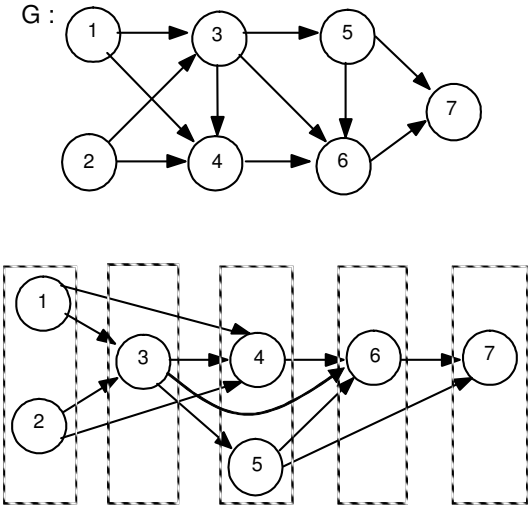
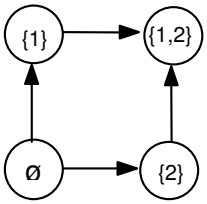



**figura V.7**

En una relación de orden R al eliminar los bucles y todas las *transitividades* , es decir, eliminar los pares (a,b) tales que existe c ($\neq$a,b) con $(a,c) \in R$ y $(c,b) \in R$, la relación resultante es de precedencia inmediata (el grafo asociado es de precedencia inmediata). Es fácil ver que al eliminar cualquier subconjunto de transitividades de R, la relación resultante posee la misma clausura transitiva de R. Debido a esta propiedad, podemos decir que la relación de precedencia inmediata extraída de R conserva la topología de R y muchas veces es más fácil trabajar con ella pues con menos información (pares) se sabe todo sobre R. Es lo que sucede cuando representamos el orden $(P(A),\subseteq)$, donde P(A) es el conjunto de las partes de un conjunto A, mediante un $|A|$-cubo (ver figura V.8).




2 - cubo asociado a P(A)

A = { 1, 2 }

**figura V.8**

Mostraremos ahora que cualquiera sea E *una relación de precedencia* sobre un conjunto V, es decir, G=(V,E) es un grafo de precedencia, el grafo G'=(V,E') donde E' es la relación de precedencia inmediata obtenida a partir de E eliminando las transitividades, es un grafo parcial de todo grafo parcial H=(V,F) de G tal que $E^+=F^+$. Esto último indica que cualquier grafo parcial de G cuya clausura transitiva coincide con la de G es obtenido a partir de G' agregándole arcos de G. A G' también se le llama *esqueleto de un grafo de precedencia o diagrama de Hasse*.

**Proposición V.5.1.7**

Sea G=(V,E) un grafo de precedencia sin arcos múltiples y $T=\{F / F \subseteq E \text{ y } F^+=E^+ \}$. Entonces $(T,\subseteq)$ es un

orden parcial con menor elemento E' y mayor elemento E donde E' es la relación de precedencia inmediata de E.

**Demostración:**

Es evidente que E es el mayor elemento.

Mostremos que $\forall F \in T$ y $(v,w) \in E$ tal que no existe un camino de largo $\geq 2$ de v a w, se cumple que $(v,w) \in F$:

Si $(v,w) \notin F$ entonces no existe camino de v a w en F esto último significa que $(v,w) \notin F^+$, pero $F^+=E^+$, lo cual es una contradicción. Así $E' \subseteq F$ y E' es el menor elemento de $(T,\subseteq)$.

¤

**Proposición V.5.1.8**

Sea $V_0,V_1,...,V_p$ la partición en niveles de V, con G=(V,E) un grafo de precedencia sin arcos múltiples. Si $\exists i$, $1 \leq i \leq p$, tal que $v \in V_{i-1}$, $w \in V_i$ y $(v,w) \in E$ entonces el vértice v precede inmediatamente a w, es decir, no existe en G un camino de longitud mayor que 1 de v a w.

**Demostración:**

Si $v \in V_{i-1}$ y $w \in V_i$ entonces $\eta(v)+1=\eta(w)$. Si existiera x descendiente de v y ascendiente de w entonces $\eta(v)<\eta(x)<\eta(w)$ lo que implicaría $\eta(w)-\eta(v) \geq 2$, lo cual es contradictorio. Así, v precede inmediatamente a w.

¤

Note que un vértice puede preceder inmediatamente a otro y no estar en niveles consecutivos.

La demostración de la proposición V.5.1.6 nos proporciona un método sencillo para calcular la partición en niveles, aún más, el método permite verificar si el grafo no posee circuitos. En cada iteración se detectan los vértices fuentes del grafo y se eliminan de éste.

Algoritmo de partición en niveles y reconocimiento de grafos de precedencia (versión 1):

{ Entrada: Un digrafo $G=(V,E)$ sin arcos múltiples.
Salida: Una variable booleana *circuito* que será verdadera si el grafo posee circuitos y falsa en caso contrario. Cuando *circuito* sea falsa, el algoritmo proporciona una partición $V_0,V_1,...,V_p$ de $V$ en niveles.}

```
Variables i : entero;
Comienzo
    i ←0; circuito ← falso;
    Mientras G posea vértices y circuito =falso hacer:
    Comienzo
        V_i ← vértices fuentes de G ;
        Si V_i = ∅ entonces circuito ← verdad
            si no
            Comienzo
                Eliminar de G los vértices en V_i ;
                I ← i + 1
            fin
    fin
fin.
```

Una implementación de este algoritmo podría ser la siguiente: supongamos que G viene representado por la lista de adyacencias LA. LA es un arreglo de 1 a $|V|$ donde LA[i] es la lista de los sucesores del vértice i. En lugar de ir eliminando los vértices de G, tendremos asociado a cada vértice v el grado interior $d^-(v)$. Estos valores pueden ser calculados a partir de la lista de adyacencias mediante un algoritmo $O(|V| + |E|)$ y pueden ser almacenados en un arreglo GI de 1 a $|V|$. Sea N un arreglo tal que N[i] contendrá los vértices del nivel i. Cuando un vértice es colocado en N[i], indicaremos su eliminación de G, disminuyendo en una unidad el grado interior de cada sucesor del vértice.

Algoritmo de partición en niveles y reconocimiento de grafos de precedencia (versión 2):

{ Entrada: La lista de adyacencias *LA* de un digrafo *G=(V,E)* sin arcos múltiples. Un arreglo *GI* con el grado interior de cada vértice.

Salida: Una variable booleana *circuito* que será verdadera si existe un circuito en el grafo. Si *circuito* es falsa en el arreglo *N* el elemento *N[i]* contendrá los vértices del nivel i.}

```
Variables i ,nvert,x,y,nivel : entero;
Comienzo
    Para i ← 0 a |V|-1 hacer: N[i] ← ∅;
    nvert ← 0; nivel ← 0; circuito ← falso;
    {Buscar las fuentes de G}
    Para I ← 1 a |V| hacer:
        Si GI[i]=0 entonces
        Comienzo
            Insertar i en el conjunto N[nivel] ;
            nvert ← nvert + 1
        fin
    Mientras nvert < |V| y N[nivel] ≠ ∅ hacer:
    Comienzo
        Para cada x en N[nivel] hacer:
            Para cada y en LA[x] hacer:
            Comienzo
                GI[y] ← GI[y] - 1;
                Si GI[y] = 0 entonces
                Comienzo
                    Insertar y en N[nivel + 1] ;
                    nvert ← nvert + 1
                fin
            fin;
        nivel ← nivel + 1
    fin
    circuito ← (nvert < |V|)
fin.
```

La complejidad en tiempo del algoritmo anterior es la siguiente:

- El cálculo del arreglo GI es O(|V| + |E|).
- La inicialización de N y la determinación de las fuentes de G es O(|V|).
- En el cuerpo del Mientras cada arco es examinado una sola vez, por lo tanto el número de operaciones del Mientras es O(max(|V|,|E|)).

Por lo tanto el algoritmo es O(max(|V|,|E|)).

### V.5.2 ORDENAMIENTO TOPOLÓGICO

Un *ordenamiento topológico* de un digrafo G=(V,E) es cualquier función inyectiva f: V→ ℵ tal que:

$$\forall v,w \in V: \text{Si } (v,w) \in E \text{ entonces } f(v) < f(w).$$

**Proposición V.5.2.1**

Un grafo G es de precedencia si y sólo si G admite un ordenamiento topológico.

**Demostración:**

Condición suficiente:
Si G admitiera un circuito $C=<x_1,x_2,...,x_p,x_1>$, tendríamos:

$$f(x_1) < f(x_2) <...< f(x_p) < f(x_1).$$

lo cual es una contradicción.

Condición necesaria:
Sea $V_0,V_1,...,V_p$ la partición en niveles de G. Numeremos los vértices de $V_0$ con los enteros en el intervalo

$[1,|V_0|]$. Este es un ordenamiento topológico de los vértices de $V_0$ por ser $V_0$ un estable. Para numerar los elementos de $V_i$, utilizaremos los enteros en el intervalo $[\sum_{j<i}|V_j|+1,\ \sum_{j\le i}|V_j|]$.

La numeración corresponde a un orden topológico de G pues si $(v,w)\in E(G)$ entonces $\eta(v) < \eta(w)$, por lo tanto v fue numerado con un número menor que el de w.

¤

La proposición anterior proporciona un método para hallar un ordenamiento topológico de un grafo. Incluso, utilizando el algoritmo O(max(|V|,|E|)) para calcular la partición en niveles de G (sección V.5.1,versión 2), podemos determinar un orden topológico de los vértices de G. Basta con asignar el siguiente entero (comenzando en 1) cada vez que un vértice es insertado en el nivel correspondiente. El contador nvert puede ser utilizado para tal fin.

En el capítulo IV, sección IV.2.6.4, dimos un algoritmo basado en el de búsqueda en profundidad para determinar un ordenamiento topológico inverso del grafo reducido de un grafo. El algoritmo consiste en lo siguiente: aplicamos el algoritmo de visita de vértices a G y numeramos sus vértices por orden de terminación de la visita a todos sus descendientes. Llamemos NumComp(v) al número que se asigna a cada vértice v de G.

**Proposición V.5.2.2 (similar a la Proposición IV.2.6.4.2)**

Sea G un grafo de precedencia sin arcos múltiples y $(v,w)\in E(G)$. Aplicamos el algoritmo de visita de vértices por búsqueda en profundidad al grafo G, asignando a cada vértice v el NumComp(v). Entonces:

NumComp(v) > NumComp(w).

**Demostración:**

Hay dos casos a considerar:

(a) w es visitado antes que v.

(b) v es visitado antes que w.

(a) Cuando se visita w, ningún descendiente de w en B puede ser v, pues existiría un camino de w a v en G, pero como $(v,w) \in E(G)$, esto implicaría que G posee un circuito. Por lo tanto v no es descendiente de w. Al terminar de visitar los descendientes de w, todavía no se habrá visitado v, así NumComp(v) > NumComp(w).

(b) La proposición IV.2.2.3 nos dice que w debe ser descendiente de v. Por lo tanto, antes de visitar todos los descendientes de v se habrán visitado todos los de w (que también son descendientes de v), así NumComp(v) > NumComp(w).

¤

NumComp es un ordenamiento topológico inverso de G. Un ordenamiento topológico de G sería:

f(v) = |V|-NumComp(v)+1.

Algoritmo de ordenamiento topológico:
{ Entrada: Un grafo *G=(V,E)* sin circuitos y sin arcos múltiples.
Salida: Un ordenamiento topológico f de los vértices de *G*. }
Variables *Contador* : entero;
*v* : vértice;
Comienzo
Inicialmente ningún vértice de *G* ha sido visitado;
*Contador* ← |*V* | + 1;
Para cada vértice *v* de G hacer:
Si *v* no ha sido visitado entonces Búsqueda en profundidad(*v* );
Donde
Búsqueda en profundidad(*v* :vértice):
Variables *w* : vértice;
Comienzo
Marcar *v* como visitado;
Para cada sucesor *w* de *v* hacer:
Si *w* no ha sido visitado entonces Búsqueda en profundidad(*w* );

*Contador* ← *Contador* - 1;
f (*v* ) ← *Contador*
fin
fin.

### V.5.3 CAMINOS DE COSTO MÍNIMO Y COSTO MÁXIMO

El algoritmo de Bellman (Programación Dinámica Progresiva) presentado en la sección V.2, puede ser utilizado para calcular caminos de costo mínimo y máximo en un grafo sin circuitos. El algoritmo en efecto calcula los caminos de costo mínimo desde un vértice raíz del grafo (por lo tanto fuente) hasta cada vértice alcanzable desde la raíz. Los caminos se van cerrando siguiendo un orden topológico. La implementación que se dio es similar al algoritmo de cálculo de niveles. En ese algoritmo, el cálculo de los caminos mínimos desde un vértice s se hace considerando los vértices según un orden topológico. Al momento de calcular el camino mínimo hasta un vértice v ya tendremos calculados los caminos mínimos hasta cada predecesor de v y el criterio de Bellman nos dice:

$cmin(s,v) = \min \{cmin(s,w) + c(w,v) / w \text{ es predecesor de } v\}$.

Presentaremos a continuación una variante del algoritmo, conocida como *Programación Dinámica Regresiva.* En este caso se calcula el camino mínimo desde un vértice $v_i$ a uno $v_j$ partiendo del vértice destino $v_j$ y aplicando el principio de dualidad de Bellman:

$cmin(v_i,v_j) = \min \{ c(v_i,w) + cmin(w,v_j) / w \text{ es sucesor de } v_i\}$

Avoquémonos a la presentación del algoritmo.

El algoritmo determinará un camino mínimo desde un vértice s a uno t de un grafo de precedencia G=(V,E) con función de costos c en los arcos. Supondremos que si $(v,w) \notin E$ entonces $c(v,w)=+\infty$.

Sea f un ordenamiento topológico de G:

$f(v_1) < f(v_2) < ... < f(v_n)$ con $V=\{v_1, v_2, ... , v_n\}$.

Para todos i,j con i<j se tiene que todos los posibles caminos de $v_i$ a $v_j$ contienen sólo vértices $v_k$ con $i \leq k \leq j$. Por lo tanto si conocemos un camino de menor costo desde $v_{k-1}$ a $v_j$, éste tiene que ser un camino de menor costo entre los caminos:

$<v_{k-1},v_l> \,||\, P_l$, donde $v_l$ es un sucesor de $v_{k-1}$ con $l \leq j$ y $P_l$ es un camino de menor costo de $v_l$ a $v_j$.

El algoritmo de visita de vértices por búsqueda en profundidad permite considerar los vértices por orden inverso a un orden topológico. Utilizaremos este algoritmo como esqueleto de nuestro algoritmo.

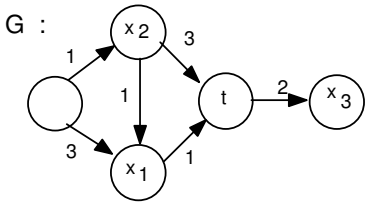
La diferencia del método con los algoritmos hasta ahora presentados es que estos últimos calculan caminos partiendo del vértice origen s, mientras que el algoritmo que presentaremos calcula un camino mínimo desde cada vértice del grafo, presente en un camino de s a t, hasta el vértice sumidero t. Por lo tanto debemos conocer el vértice t al cual se quiere llegar desde s. A cada vértice v del grafo se asocia el valor cmin(v) que representa el costo de un camino mínimo de v a t. En caso de no existir camino se tendrá $cmin(v) = +\infty$; también asociaremos a v un apuntador al vértice sucesor de v en un camino mínimo de v a t y lo denotaremos por siguiente(v). Los comentarios hechos en párrafos anteriores permiten aplicar criterios de dualidad para mostrar la equivalencia entre el algoritmo que presentaremos y el de Bellman dado en la sección V.2. En efecto, el algoritmo que presentamos puede ser ligeramente modificado para calcular los caminos mínimos desde cada vértice de G hasta un vértice t. Si queremos utilizar el algoritmo para calcular caminos mínimos desde un vértice t hasta cada vértice de G, bastaría con invertir la dirección de los arcos de G y aplicar el algoritmo a este nuevo grafo $G^{-1}$. Obtendremos así un camino mínimo desde cada vértice en $G^{-1}$ hasta t. Ahora bien, $C^{-1}$ es un camino mínimo de v a t en $G^{-1}$ si y sólo si el camino C obtenido de $C^{-1}$ invirtiendo el sentido a los arcos es un camino mínimo de t a v en G. Por lo tanto, conociendo los caminos mínimos desde cada vértice v de $G^{-1}$ a t, los caminos mínimos de t a cada vértice v de G se obtienen de los primeros invirtiendo el sentido a los arcos.

Algoritmo de Programación Dinámica Regresiva para el cálculo de caminos mínimos:

{ Entrada: Un grafo de precedencia *G=(V,E)* sin arcos múltiples. Dos vértices *s* y *t* de *G*. Una función de costos $c : E \rightarrow \Re$.

Salida: Un camino de costo mínimo de *s* a *t*. Si cmin(*s*)=+∞ no hay camino de *s* a *t*. Si no, el camino se podrá construir siguiendo los apuntadores siguiente(s): <*s*,siguiente(*s* ), siguiente(siguiente(*s* )),...,*t* >.}

```
Comienzo
    Inicialmente ningún vértice ha sido visitado;
    Inicialmente cmin(v)=+∞, ∀v∈ V(G) ;
    Búsqueda en Profundidad(s )
Donde
    Búsqueda en Profundidad(x :vértice):
    Variables y :vértice;
    Comienzo
        Marcar x como visitado;
        { Si ya hemos visitado a t , no hace falta visitar sus
        descendientes }
        Si x =t entonces cmin(x )←0
            si no
            Comienzo
                Para cada y sucesor de x hacer:
                    Si y no ha sido visitado entonces
                        Búsqueda en Profundidad(y );
                { Se visitaron todos los descendientes de x. Así x es el próximo en el orden topológico inverso}
                Para cada y sucesor de x hacer:
                    Si (cmin(y)≠+∞) y (cmin(x) >cmin(y) +c(x,y ))
                        entonces
                        Comienzo
                            cmin(x ) ← cmin(y ) + c(x,y );
                            siguiente(x ) ← y
                        fin
            fin
    fin
fin.
```




Árbol de la búsqueda en profundidad

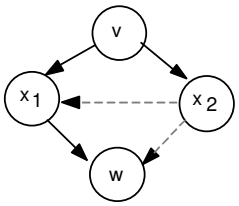



**figura V.9**

Es importante notar que este algoritmo funciona cualquiera sean s y t. Si al finalizar el algoritmo cmin(s) = +∞, entonces no existirá camino de s a t en G. El símbolo +∞ significa un número muy grande, basta con tomarlo mayor que el costo de cualquier camino en G. Note también que el algoritmo funciona cualquiera sea la función de costos c. Por lo tanto permite calcular el camino de costo máximo de s a t (basta con cambiar de signo a la función de costos c). Finalmente, es evidente que el algoritmo es O(max(|V|,|E|)).

En la figura V.9 vemos el árbol generado por el algoritmo de búsqueda en profundidad aplicado al grafo G de la figura. El primer vértice al cual visitan todos sus descendientes es $x_1$. Se calcula siguiente($x_1$) = t y cmin($x_1$) = 1. El siguiente vértice es $x_2$. El camino mínimo de $x_2$ a t se calcula entre los dos sucesores de $x_2$ que son $x_1$ y t. Como $c(x_2,x_1)+cmin(x_1) < c(x_2,t)+cmin(t)$ entonces siguiente($x_2$)=$x_1$ y cmin($x_2$)=2. Finalmente se alcanza a s y como $c(s,x_1)+cmin(x_1) > c(s,x_2)+cmin(x_2)$, se tiene que siguiente(s)=$x_2$ y cmin(s)=3. Por lo tanto el camino de costo mínimo de s a t es $<s,x_2,x_1,t>$.

### V.5.4 PLANIFICACIÓN DE PROYECTOS.

Los grafos de precedencia aparecen en muchas aplicaciones, particularmente en aquéllas donde se pretende determinar caminos de costo máximo entre dos vértices, como es el caso de los grafos que aparecen en Planificación de Proyectos.

Suponga que se quiere realizar un proyecto conformado por un gran número de actividades. Se puede representar cada actividad por un vértice de un grafo G y establecer un arco desde un vértice $v_i$ hasta un vértice $v_j$ para indicar que la actividad i precede a la actividad j. A cada arco asociamos un costo $c_{ij}$ que representa la mínima cantidad de tiempo necesario entre el comienzo de la actividad i y el comienzo de la actividad j; en efecto, es la duración mínima de la actividad i. El grafo resultante es obviamente dirigido sin circuitos, ya que una actividad no puede precederse a si misma.

El problema consiste en determinar el tiempo mínimo necesario para completar el proyecto. Este problema se reduce en términos del grafo a encontrar un camino de costo máximo entre un vértice s que representa el comienzo del proyecto, y un vértice t que representa su terminación. Esto se debe a que el tiempo mínimo para que comience una actividad es el máximo entre los tiempos mínimos para que concluyan las actividades predecesoras.

Un camino de costo máximo entre s y t se llama *camino crítico* debido a que las actividades que en él se encuentran determinan el tiempo global de terminación del proyecto pues cualquier retraso en el comienzo de una actividad en el camino crítico implica un retraso en el tiempo de culminación del proyecto.

Otro de los objetivos de la planificación de proyectos es determinar cuáles son las *actividades críticas* , es decir, aquéllas que no se puede retrasar sin que se retrase la duración de todo el proyecto y para las actividades no críticas determinar la holgura en tiempo que esta cada una posee para ser comenzada sin alterar el tiempo global de duración del proyecto.

Podemos utilizar cualquier algoritmo de los presentados en secciones anteriores para determinar un camino crítico de s a t. Sea cmax(v) el costo del camino máximo de s a v, con $v \in V(G)$. El valor cmax(t) es el costo de un camino crítico. Para conseguir el camino crítico basta con seguir los apuntadores que proporciona el algoritmo de caminos óptimos que utilicemos.

Sea TEC(i) el tiempo más temprano en que puede comenzar la actividad i. Se tiene que TEC(i)=cmax($v_i$), donde $v_i$ es el vértice correspondiente a la actividad i en el grafo. Sea TAC(i) el tiempo más tarde en que puede comenzar la actividad i sin afectar la duración cmax(t) del proyecto. Supongamos que las actividades están ordenadas de acuerdo a un orden topológico, es decir, si $(v_i,v_j) \in E(G)$ entonces i<j, con el vértice s numerado 1 y el vértice t numerado n (=$|V(G)|$). El cálculo de TAC(i) se puede efectuar recursivamente de acuerdo a las reglas siguientes:

- TAC(n) = cmax(t).

- $TAC(i) = \min \{TAC(j)-c_{ij} / (v_i,v_j) \in E(G)\}$.

La cantidad TAC(i)-TEC(i) representa la holgura en tiempo que posee la actividad i para ser comenzada sin afectar la duración global del proyecto. En otras palabras, si para cada actividad i del proyecto escogemos el tiempo de comienzo de la actividad en el intervalo [TEC(i),TAC(i)] entonces el proyecto no sufriría ningún retraso, su duración seguiría siendo cmax(t). Toda actividad i que esté en un camino crítico cumple con TAC(i)-TEC(i)=0 pues la actividad no puede ser retrasada sin afectar la duración del proyecto.

Es de hacer notar que para reducir los cálculos en la determinación de TAC, como los costos son positivos, podríamos eliminar las transitividades de G y trabajar sobre el grafo de precedencia inmediata.

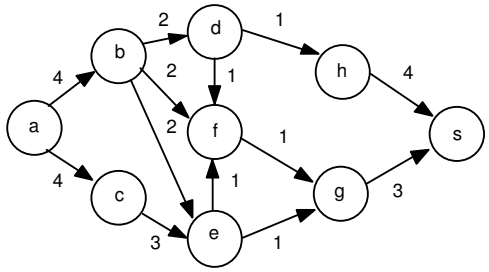



**figura V.10**

Daremos un ejemplo de cómo sería el cálculo. Supongamos que se tienen las actividades siguientes:

| Actividad | Duración | Precede a |
|---|---|---|
| a | 4 | b,c |
| b | 2 | d,e,f |
| c | 3 | e |
| d | 1 | f,h |
| e | 1 | f,g |
| f | 1 | g |
| g | 3 | |
| h | 4 | |

El grafo de la figura V.10 representa la situación. Note que se ha agregado un vértice s donde convergen los vértices g y h que no poseen actividades sucesoras. No es necesario agregar un vértice fuente pues ya hay una raíz.

Para hallar TEC, aplicamos el algoritmo de Bellman (sección V.2) partiendo de la actividad a, teniendo el cuidado de cambiar el signo a los costos antes de aplicar el algoritmo. Obtendremos el costo de un camino máximo de **a** a cada vértice del grafo. Note que no se puede aplicar el algoritmo presentado en la sección anterior, a menos que invirtamos el sentido a los arcos y tomamos como sumidero al vértice a. Obtenemos:

| **Actividad** | **Costo Máximo** | **Predecesor** |
|---|---|---|
| a | 0 | no tiene |
| b | 4 | a |
| c | 4 | a |
| d | 6 | b |
| e | 7 | c |
| f | 8 | e |
| g | 9 | f |
| h | 7 | d |
| s | 12 | g |

Para hallar TAC, aplicamos el algoritmo de Programación Dinámica Regresiva de la sección V.5.3, modificándolo ligeramente. Comenzamos colocando TAC(s)=TEC(s) y aplicamos recursivamente la fórmula:

$$TAC(i) = \min\{TAC(j)-c_{ij} / (v_i,v_j) \in E(G)\}$$

Seguimos un orden topológico inverso en el cálculo de TAC(i). Este orden resulta inmediato del algoritmo pues el siguiente vértice en el orden inverso es el siguiente vértice al cual se le han visitado todos sus descendientes en el algoritmo. Supongamos que la consideración de los sucesores de cada vértice en el algoritmo de Programación Dinámica Regresiva se hace en orden alfabético de la etiqueta del vértice. El algoritmo genera el árbol de búsqueda

en profundidad presentado en la figura V.11.

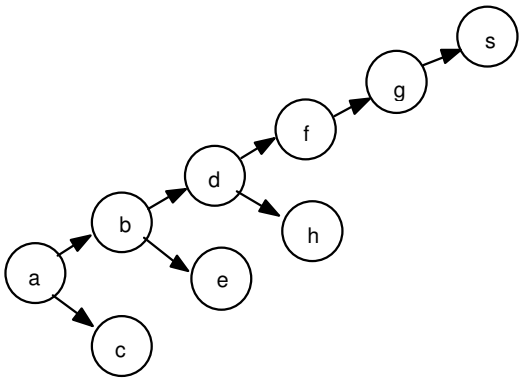



**figura V.11**

Por lo tanto:

**Actividad:** s g f h d e b c a

**TAC:** 12 9 8 8 7 7 5 4 0

Por lo tanto ya tenemos toda la información necesaria sobre el proyecto. Un camino crítico es <a,c,e,f,g,s>. La tabla y diagrama de tiempos siguientes resumen toda la información necesaria:

El diagrama de tiempo indica para cada actividad, lo más temprano que puede comenzar y lo más tarde que puede terminar (ver figura V.12)

| Actividad | TEC | TAC | ¿Es Crítica? | Holgura (TAC-TEC) | Duración |
|---|---|---|---|---|---|
| a | 0 | 0 | sí | 0 | 4 |
| b | 4 | 5 | no | 1 | 2 |
| c | 4 | 4 | sí | 0 | 3 |
| d | 6 | 7 | no | 1 | 1 |
| e | 7 | 7 | sí | 0 | 1 |
| f | 8 | 8 | sí | 0 | 1 |
| g | 9 | 9 | sí | 0 | 3 |
| h | 7 | 8 | no | 1 | 4 |

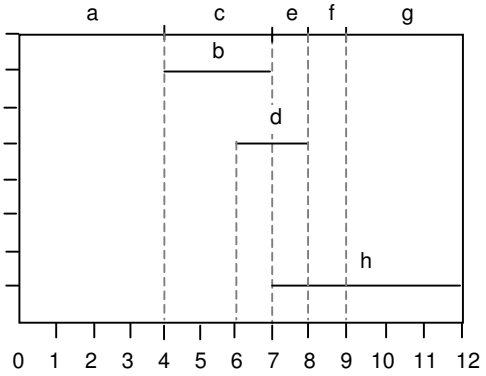



**figura V.12**

A continuación presentamos el algoritmo para el cálculo de TAC. Es una variante del algoritmo de Programación Dinámica Regresiva.

Cálculo de TAC:

{ Entrada: Un grafo de precedencia $G=(V,E)$ y una función de costos $c:E \rightarrow \Re$. Dos vértices $s$ y $t$, fuente y sumidero del grafo $G$, y TEC($t$ ).

Salida: Para cada vértice $v \in V(G)$, TAC(v).}

Comienzo

Inicialmente ningún vértice ha sido visitado;

Inicialmente TAC(v)=$+\infty$, $\forall v \in V(G)$;

Búsqueda en Profundidad($s$ );

Donde

Búsqueda en Profundidad($x$ : vértice):

Variables $y$ : vértice;

Comienzo

Marcar $x$ como visitado;

Si $x = t$ entonces TAC($x$ ) $\leftarrow$ TEC($x$ )

si no

Comienzo

Para cada sucesor $y$ de $x$ hacer:

Si $y$ no ha sido visitado entonces Búsqueda en Profundidad($y$ );

Para cada sucesor $y$ de $x$ hacer:

Si (TAC($y$)$\neq+\infty$) y (TAC($x$) > TAC($y$) - c($x,y$ )) entonces TAC($x$) $\leftarrow$ TAC($y$) - c($x,y$);

fin

fin

fin.

**Proposición V.5.4.1**

Cualquiera sea v vértice de G, se tiene que TAC (v) = cmax(s,t) – cmax(v,t)

**Demostración**: Inducción sobre la altura de un vértice. Utilizar la hipótesis inductiva: "todos los sucesores de un vértice dado v satisfacen la igualdad" y demuestre para v. La base de la inducción es comprobar la igualdad para el vértice t.

**Proposición V.5.4.2**

Sabemos que una actividad crítica v es una actividad con holgura cero (TAC(v)-TEC(v) = 0). Una actividad v es una actividad crítica si y sólo si v pertenece a un camino crítico de s a t.

**Demostración**:

($\Rightarrow$) Si TAC(v)- TEC(v)=0 entonces TAC(v) = TEC(v) = cmax(s,v). Como TAC(v) = cmax(s,t) – cmax(v,t) entonces cmax(s,t) = cmax(s,v) + cmax(v,t). Es decir, un camino máximo de s a v concatenado con un camino máximo de v a t es un camino de costo máximo, por lo que v está en un camino de costo máximo.

($\Leftarrow$) Sea P un camino de costo máximo de s a t que pasa por v. Entonces cmax(s,v)+cmax(v,t) = cmax(s,t) por principio de optimalidad en grafos de precedencia. Por la proposición V.5.4.1 tenemos que TAC(v) = TEC(v).

## V.6 EJERCICIOS

**1.** Use el algoritmo de Dijkstra (tanto la versión del modelo general de algoritmos de costo mínimo, como la versión de Dijkstra propiamente dicha) para encontrar la distancia y el camino más corto del vértice 6 a cualquier vértice del siguiente grafo:

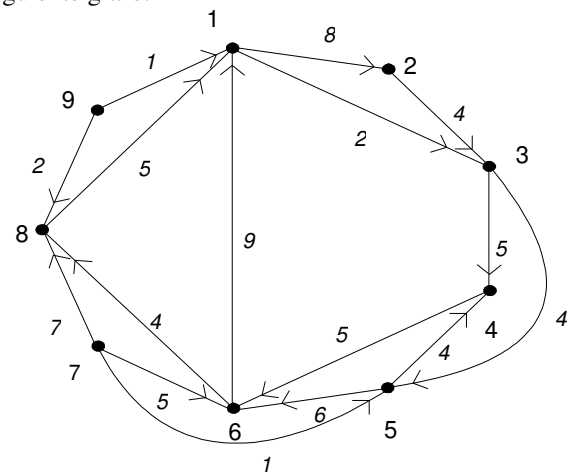



**figura V.13**

**2.** Dar tres razones por las cuales la siguiente lista de caminos no puede corresponder a un paso intermedio de un algoritmo de búsqueda de camino de costo mínimo, siguiendo el modelo general, con costos no negativos. *s*, *a*, *b* y *c* son vértices del grafo:

| | | |
|---|---|---|
| 1.- | (s, 0, -) | CERRADO |
| 2.- | (a, 3, 1) | ABIERTO |
| 3.- | (b, 2, 1) | CERRADO |
| 4.- | (b, 5, 2) | ABIERTO |
| 5.- | (c, 1, 3) | ABIERTO |

**3.** Modifique el algoritmo de DFS recursivo para hallar la altura de cada vértice en un grafo dirigido sin circuitos.

**4.** Muestre que el algoritmo de Dijkstra no funciona si los costos de los arcos son algunos negativos.

**5.** Seleccione las estructuras de datos apropiadas para lograr que el tiempo de algoritmo de Dijkstra sea $O(e.log(n))$ para un grafo con $e$ aristas y $n$ vértices. ¿En que caso sería conveniente?

**6.** Analizar cómo varía la eficiencia en BFS y Dijkstra si la lista de caminos abiertos se guarda en un heap en lugar de una lista lineal.

**7.** Modifique el algoritmo de Dijkstra para que obtenga todos los caminos mínimos de un vértice *v* dado a cualquier otro vértice *w*, en un grafo dirigido con costos no negativos en los arcos.

**8.** Se desea determinar el camino de costo máximo entre un par de vértices de un grafo de precedencia *G* con costos positivos. Se sugieren dos modificaciones al algoritmo de Dijkstra:

- Se multiplican los costos de *G* por –1 y se construye con ello un nuevo grafo *G'*. Se aplica Dijkstra sobre *G'*. El camino mínimo de *v* a *w* en *G'* constituye el camino máximo en *G*.

- Sea *k* el mayor costo de un arco de *G*. Se calcula la siguiente función de costo: $c'(e)=k-c(e)$, donde $c(e)$ es el costo del lado *e* en *G*, para crear el grafo *G'*. El camino mínimo en *G'* es el camino máximo en *G*.

**9.** Analice si estos algoritmos logran el objetivo deseado.

**10.** Modifique Dijkstra para detectar circuitos de costo negativo en un digrafo.

**11.** Muestre si el algoritmo que se da a continuación obtiene el costo de un camino mínimo de $v_0$ a cualquier vértice *v* en un grafo arbitrario con aristas que pueden tener costo negativo:

- Comienzo
- $S \leftarrow \{v_0\}$;
- $D[v_0] \leftarrow 0$;
- Para cada v en $V-\{v_0\}$ hacer $D[v] \leftarrow c(v_0, v)$;
- Mientras $S \neq V$ hacer
- Comienzo
- Escoger un vértice w en V-S tal que D[w] sea mínimo;
- $S \leftarrow S \cup \{w\}$;
- Para todo $v \in V$ tal que $D[v] > D[w] + c(w, v)$ hacer
- Comienzo
- $D[v] \leftarrow D[w] + c(w, v)$;
- $S \leftarrow S - \{v\}$;
- Fin
- Fin
- Fin

**12.** Samuel J, Pettacci A. es miembro fundador y activo de los bomberos de la USB, y es quien maneja el camión de bomberos. Siendo Samuel estudiante de computación, y por lo tanto versado en métodos eficientes para evitar esfuerzos innecesarios, no le hace gracia dejar el agradable ambiente de la casa de su novia en la Urbina por más tiempo del estrictamente necesario. Usualmente, la cantidad de tiempo que les toma llegar a la sede al resto de los bomberos es de 15 minutos.

La pregunta que Samuel debe responder es "¿Cuántos minutos después que suena el buscapersonas puede despedirse de su novia, si quiere llegar no más tarde que el último miembro del grupo?". Samuel construyó de memoria la red de la siguiente figura, fijando para cada tramo un estimado del tiempo necesario para atravesarlo. La casa de la novia de Samuel es el vértice 1 y la sede de los bomberos es el vértice 14. Diga como resolver el problema de Samuel y resuélvalo.

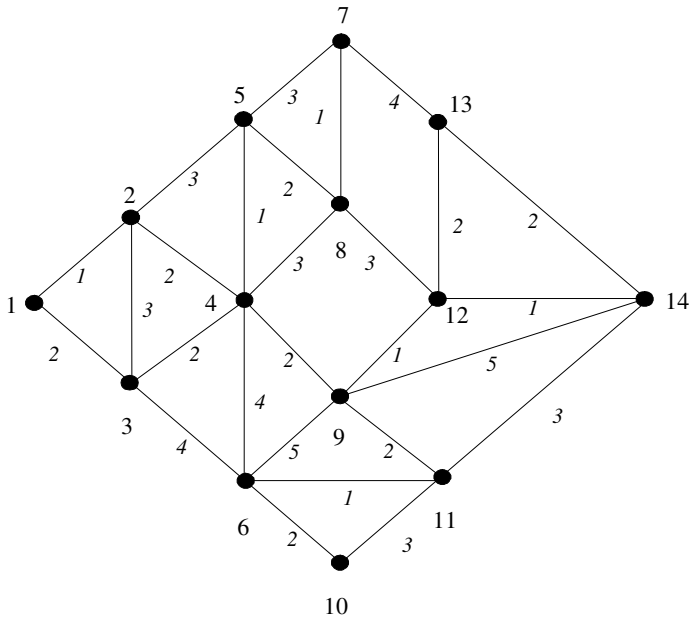



**figura V.14**

**13.** Dadas dos coordenadas $(x_1,y_1)$ $(x_2, y_2)$ en un tablero de ajedrez, hallar la ruta mínima que debería recorrer un caballo de ajedrez para ir de $(x_1,y_1)$ a $(x_2,y_2)$. Utilice el algoritmo de Dijkstra y el algoritmo BFS para resolver el problema y compárelos.

**14.** Escriba un algoritmo para hallar el camino mínimo entre dos vértices *v*, *w* de un grafo *G*, tomando en cuenta una de las siguientes restricciones:

a) El camino de *v* a *w* no debe contener vértices de *V'*, donde *V'* es un subconjunto propio de *V*.

b) El camino de *v* a *w* debe contener todos los vértices de un subconjunto propio *V'* de *V*.

c) El camino no debe contener aristas de un conjunto *E'* que es un subconjunto propio de *E*.

d) El camino debe contener todas las aristas de *E'*, subconjunto propio de *E*.

**15.** El grafo de la siguiente figura muestra los canales de comunicación (lados) y los tiempos de comunicación en minutos (asociados a los lados) entre 8 centros de comunicaciones (vértices). A las 3:00 pm desde el vértice *a* se envía un mensaje a cada uno de sus vecinos. El resto de los centros de comunicación retransmiten a cada uno de sus vecinos tan pronto lo reciben por primera vez. Se quiere saber a qué hora recibe el mensaje cada vértice de la red. Diga que algoritmo utilizaría y muestre la corrida en frío del mismo.

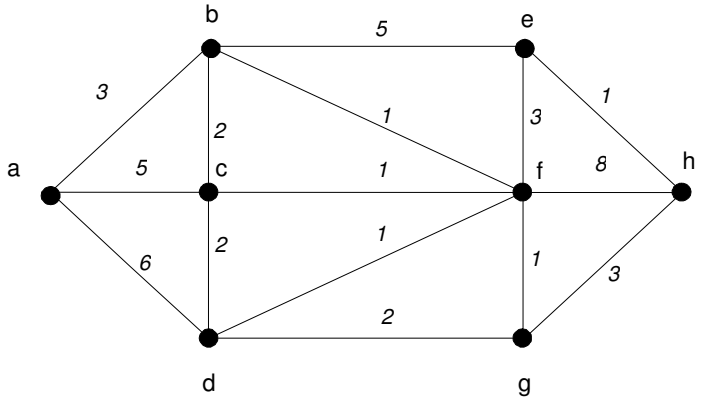



**figura V.15**

**16.** Una compañía manufacturera está planificando producir aires acondicionados de 3 componentes: gabinete, ventilador y motor. El precio de fabricación es de 50 dólares para los gabinetes, 75 dólares para los ventiladores y 100 dólares los motores. Sin embargo, una vez que los ventiladores están en producción, el costo de producción de gabinetes y motores se reduce en un 5%. Si los motores se producen primero, el costo de las otras unidades se reduce en un 10%. Además, después de estar 2 unidades en producción, se produce una reducción extra del 5% para la tercera unidad. Defina un grafo que represente el problema y resuélvalo como un problema de camino mínimo.

**17.** Sea *G* un digrafo sin arcos múltiples. ¿Cómo podrían ser utilizados los algoritmos de Floyd y Dijkstra para calcular la clausura transitiva del grafo *G*?

**18.** Para el siguiente grafo:

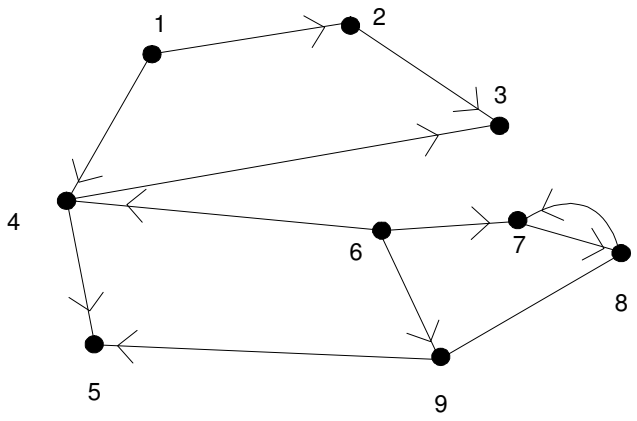



**figura V.16**

a) Encuentre la longitud del camino más corto entre cada par de vértices.

b) Determinar la altura y el nivel de cada vértice si no existiera el arco (7,8).

**19.** Mostrar que el siguiente algoritmo, debido a Dantzig, encuentra todas las distancias mínimas en un digrafo sin circuitos negativos, al igual que el algoritmo de Floyd. Sea $C^k(i, j)$ la distancia mínima de $i$ a $j$, con $1 \le i,j \le k$ sin utilizar ningún vértice mayor que $k$. Sea $c(i, j) = c(e)$ si $e=\{i, j\} \in E$ y $c(i, j) = \infty$ si $\{i, j\} \notin E$.

- $C^1(1,1) \leftarrow 0$
- $k \leftarrow 1$
- Repetir
- $k \leftarrow k + 1$
- Para $1 \le i \le k$ hacer
- Comienzo
- $C^k(i, k) \leftarrow Min_{1 \le j \le k-1}\{C^{k-1}(i, j)+c(j, k)\}$
- $C^k(k, i) \leftarrow Min_{1 \le j \le k-1}\{c(k, j)+C^{k-1}(j, i)\}$
- fin
- Para $1 \le i, j < k$ hacer
- $C^k(i, j) \leftarrow Min\{C^{k-1}(i, j), C^k(i, k) + C^k(k, j)\}$
- Hasta k=n

**20.** Dé un ejemplo de una estimativa no consistente para la cual el número de iteraciones del algoritmo A* sea mayor que el número de vértices alcanzables desde s.

**21.** Determinar la altura y el nivel de cada vértice del grafo siguiente:

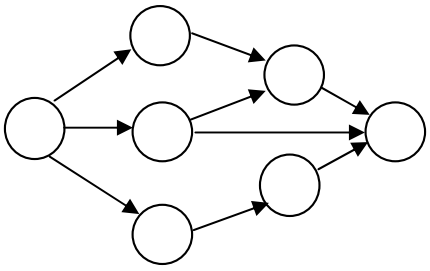


**figura V.17**

**22.** Sea *G=(V,E)* un digrafo sin arcos múltiples y *c* una función que asocia un número real a cada arco de *G*. Supongamos que se quiere hallar un camino de costo máximo desde un vértice *s* hasta cada vértice alcanzable desde *s*.

a) ¿Qué condiciones habría que imponer a *G* y *c* para que pueda existir caminos de costo máximo desde *s* a cada vértice alcanzable desde *s*?

b) Suponiendo que se satisfacen las condiciones dadas en a), ¿Para hallar un camino de costo máximo desde *s* hasta cada vértice alcanzable desde *s* bastaría con invertir el signo de los costos de los arcos y aplicar a *G* con estos nuevos costos un algoritmo de búsqueda de caminos mínimos basado en el modelo general de etiquetamiento?

**23.** Aplicar programación dinámica progresiva y regresiva para hallar un camino de costo máximo de **a** a **g** en el grafo siguiente:

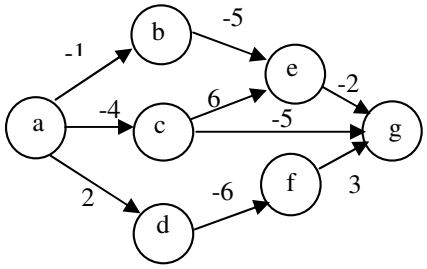



**figura V.18**

**24.** Se tiene un proyecto de construcción conformado por varias actividades: A, B, C, D, E, F y G, con duraciones respectivas 1, 2, 5, 3, 4, 3, 2. Algunas actividades preceden a otras en su ejecución, como se indica a continuación:

| Actividad | Precede a Actividad |
|---|---|
| A | B, C |
| B | D, E, F |
| C | E |
| D | F |

| Actividad | Precede a Actividad |
|---|---|
| E | F, G |
| F | G |
| G | --- |
| | |

Determine los tiempos más tempranos y tardíos en que debe comenzar una actividad para que el proyecto se ejecute en el menor tiempo posible.

**25.** La compañía EMPR S.A. tiene planeado para su inmediata ejecución un proyecto que involucra 7 actividades, distinguidas con las letras de la A hasta la G. Existen relaciones de precedencia entre muchas de estas actividades, la cuales se indican mediante el siguiente grafo:

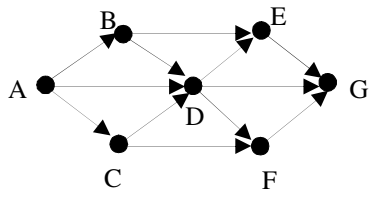



**figura V.19**

La duración de cada actividad es como sigue: A:2, B:3, C:4, D:1, E:8, F:4, G: 2

a) Encuentre un ordenamiento topológico del grafo.

b) Encuentre el tiempo mínimo requerido para culminar todo el proyecto, así como las actividades críticas.

**CAPÍTULO VI**

# *ÁRBOLES Y ARBORESCENCIAS*

## INTRODUCCIÓN

En este Capítulo se trata el concepto de árbol y el de arborescencia. Se determinan sus propiedades y se utilizan para definir otras estructuras como son los árboles y arborescencias planas, los árboles binarios y los árboles de juego. Se discute sobre estructuras de datos para representar árboles y arborescencias en el computador, así como los tipos de recorridos en arborescencias planas.

## VI.1 ÁRBOLES. PROPIEDADES

Como se definió en el capitulo III, un *árbol* es un grafo conexo sin ciclos. Definiremos también una *foresta o bosque* como un grafo donde cada componente conexa es un árbol.

A continuación se hará un desarrollo teórico que facilitará el camino hacia la obtención de una serie de resultados básicos sobre árboles.

Sea G = (V,E) un grafo con E = $\{e_1, e_2, \dots, e_m\}$, un ciclo C en G lo podemos representar como una m-tupla $(c_1, c_2, \dots, c_m)$ donde

$$c_i = \begin{cases} 0 & \text{si } e_i \notin C \\ 1 & \text{si } e_i \in C \end{cases}$$

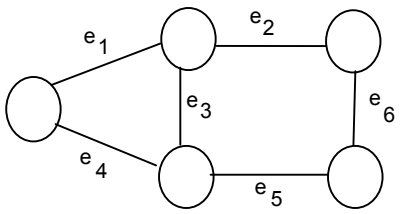
Si C y C' representan dos ciclos de G, la suma $C \oplus C'$ se obtiene de la siguiente manera:

$C \oplus C' = (c_1, c_2, \dots, c_m) \oplus (c'_1, c'_2, \dots, c'_m) = (d_1, d_2, \dots, d_m)$

$$d_i = \begin{cases} 0 & \text{si } c_i = c'_i \\ 1 & \text{si } c_i \neq c'_i \end{cases}$$

en otras palabras, $C \oplus C'$ no es más que la m-tupla que representa los lados no comunes de C y C'.

La figura VI.1 ilustra varios ciclos de un grafo.




$C_1 = < e_2, e_3, e_5, e_6 > \Rightarrow C^1 = (0, 1, 1, 0, 1, 1)$

$C_2 = < e_1, e_3, e_4 > \Rightarrow C^2 = (1, 0, 1, 1, 0, 0)$

$C_3 = < e_1, e_2, e_6, e_5, e_4 > \Rightarrow C^3 = (1, 1, 0, 1, 1, 1)$

**figura VI.1**

Es de notar que la operación $\oplus$ es conmutativa y asociativa.

**Definición VI.1.1**

Decimos que un conjunto $\boldsymbol{C}$ de ciclos de un grafo G=(V,E) es *independiente* si no existe ningún sub-conjunto $\boldsymbol{C'} = \{C_1, \dots, C_s\}$ de $\boldsymbol{C}$ tal que $C^1 \oplus C^2 \oplus \dots \oplus C^s = (0, 0, \dots, 0)$, de otro modo el conjunto se llama *dependiente.*

Intuitivamente, un ciclo es dependiente de otros si se puede formar a partir de ellos.

Por ejemplo, el ciclo $C_3$ de la figura VI.1 se puede formar a partir de los ciclos $C_1$ y $C_2$, es decir $C^3 = C^1 \oplus C^2$ , por lo que $\boldsymbol{C} = (C_1, C_2, C_3)$ es un conjunto dependiente.

**Observación:** Un conjunto de ciclos $\boldsymbol{C}$ que consta sólo de uno o dos ciclos será siempre independiente.

En lo que sigue y a menos que se especifique lo contrario, identificaremos un ciclo C con la m-tupla que lo representa.

**Definición VI.1.2**

Un conjunto independiente de ciclos $\boldsymbol{C}$, de un grafo G=(V,E), es *maximal* si no existe otro conjunto independiente $\boldsymbol{C'}$ de G tal que $\boldsymbol{C} \subset \boldsymbol{C'}$.

**Proposición VI.1.1**

Sea $\boldsymbol{C}$ un conjunto independiente de ciclos de G=(V,E), $\boldsymbol{C}$ es maximal si y sólo si para todo ciclo C de G tal que $C \notin \boldsymbol{C}$:

$\exists$ ciclos $C_1, \dots, C_k \in \boldsymbol{C}$ con $C = C_1 \oplus \dots \oplus C_k$ . Además esta combinación es única.

**Demostración:**

Sea $\boldsymbol{C}$ maximal y $C \notin \boldsymbol{C}$. Si C no puede expresarse como suma o combinación de un sub-conjunto de $\boldsymbol{C}$, entonces $\boldsymbol{C} \cup \{C\}$ es independiente, lo cual contradice la hipótesis de maximalidad de $\boldsymbol{C}$.

Por otro lado, sea $\boldsymbol{C}$ es un conjunto independiente tal que $\forall C \in \boldsymbol{C}$: existe un sub-conjunto de $\boldsymbol{C}$ cuya combinación es C. Si este conjunto $\boldsymbol{C}$ no fuera maximal, entonces existiría $C' \notin \boldsymbol{C}$ tal que $\boldsymbol{C} \cup \{C'\}$ es independiente, lo cual implica que C' no se puede expresar como combinación de un sub-conjunto de $\boldsymbol{C}$, contradiciendo la hipótesis.

Para ver que la combinación es única, basta con suponer que existe otra combinación de C, es decir

$C = D_1 \oplus \dots \oplus D_l$, con $\{C_1, \dots, C_k\} \neq \{D_1, \dots, D_p\}$ .

Entonces $C_1 \oplus \dots \oplus C_k \oplus D_1 \oplus \dots \oplus D_p = 0$. (1)

Sean $C_1,\dots,C_i,D_1,\dots,D_i$ los ciclos comunes a ambas colecciones, entonces también $C_1 \oplus \dots \oplus C_i \oplus D_1 \oplus \dots \oplus D_i = 0$, de donde (1) se convierte en:

$0 \oplus C_{i+1} \oplus \dots \oplus C_k \oplus D_{i+1} \oplus \dots \oplus D_p = 0$ ,

lo cual implica que

$C_{i+1} \oplus \dots \oplus C_k \oplus D_{i+1} \oplus \dots \oplus D_p = 0$,

lo cual es una contradicción pues $\{C_{i+1},\dots,C_k, D_{i+1},\dots,D_p\}$ es sub-conjunto de $\boldsymbol{C}$.

¤

Esta proposición sirve para caracterizar todos los ciclos de un grafo a partir de un conjunto independiente maximal del mismo.

**Proposición VI.1.2**

Todas las colecciones independientes maximales de ciclos de un grafo G tienen la misma cardinalidad.

**Demostración:**

Sean $\boldsymbol{C}$ y $\boldsymbol{D}$ dos colecciones independientes maximales de G:

$\boldsymbol{C} = \{C_1, \dots, C_k\}$ , $\boldsymbol{D} = \{D_1, \dots, D_s\}$ , con $k < s$.

Si $\boldsymbol{D}$ es independiente entonces $D_1$ no puede expresarse como combinación de ningún sub-conjunto de $\boldsymbol{D} - \{D_1\}$, pero sí en cambio existe un único sub-conjunto $\{C_i^1\}$ de $\boldsymbol{C}$ cuya suma es $D_1$. Por lo tanto, al menos uno de los ciclos en $\{C_i^1\}$ no es combinación de ciclos en $\boldsymbol{D} - \{D_1\}$.

Reordenando, podemos suponer que $C_1$ es tal ciclo y así, el conjunto $\boldsymbol{D_1'} = \{C_1, D_2, \ldots, D_s\}$ es independiente.

De igual manera, $D_2$ no puede expresarse como combinación de un sub-conjunto en $\boldsymbol{D_1'}$, pero sí de un sub-conjunto $\{C_i^2\}$ de $\boldsymbol{C}$. Existirá entonces en $\{C_i^2\}$ un ciclo $C_i^2 \neq C_1$ que no es combinación de ciclos en $\boldsymbol{D_1'}$ y podemos suponer, reordenando, que $C_i^2 = C_2$.

Así $\boldsymbol{D_2'} = \boldsymbol{D_1'} - \{D_2\} \cup \{C_2\}$ es independiente.

Continuando con el mismo razonamiento, llegamos a construir

$\boldsymbol{D_k'} = \{ C_1, \ldots, C_k, D_{k+1}, \ldots, D_s\}$ independiente, lo cual contradice la maximalidad de $\boldsymbol{C}$, de donde concluimos entonces que k=s.

¤

**Definición VI.1.3**

Llamamos *rango* de los ciclos de G, denotado r(G), al número máximo de ciclos independientes de G, es decir, la cardinalidad de cualquier conjunto independiente maximal.

**Proposición VI.1.3**

Sea G=(V,E) un grafo y $\boldsymbol{C}$ un conjunto independiente maximal de G. Si añadimos a G una nueva arista **e**={v,w}, entonces si **e** forma un ciclo C en G'= (V,E U {**e**}), el rango de los ciclos de G', r(G') es r(G) + 1. Si **e** no forma un ciclo en G' entonces r(G') = r(G).

**Demostración:**

Si se forma el ciclo C en G', este posee un lado {v,w} que no está presente en ninguno de los ciclos de $\boldsymbol{C}$, de donde $\boldsymbol{C'} = \boldsymbol{C} \cup \{C\}$ es un conjunto independiente en G' de cardinalidad r(G)+1.

Veamos que $\boldsymbol{C'}$ es maximal, es decir que si C' es un ciclo en G' tal que $C' \notin \boldsymbol{C'}$, entonces C' es una combinación de ciclos en $\boldsymbol{C'}$.

Si $e \notin C'$ entonces C' es un ciclo de G, por lo tanto se puede expresar como combinación de ciclos en $\boldsymbol{C} \subset \boldsymbol{C'}$.

Si $e \in C'$, obsérvese que :

$C' = (C \oplus C) \oplus C'$ pues $(C \oplus C) = (0, \ldots, 0)$, de donde $C' = (C \oplus C') \oplus C$.

Además, nótese que $e \notin C \oplus C'$ por ser común a ambos. Por lo tanto $C \oplus C'$ se puede expresar como combinación en $\boldsymbol{C}$ ($C \oplus C'$ representa a un conjunto de ciclos disjuntos en G', ninguno de los cuales contiene a e, ver proposición III.1.3.7).

Así $C \oplus C' = \sum_i C_i$ con $\{Ci\} \subseteq \boldsymbol{C} \subset \boldsymbol{C'}$, de

donde $C' = \sum_i C_i \oplus C$ es una combinación de ciclos en $\boldsymbol{C'}$.

Concluimos entonces que C' es maximal y por lo tanto

r(G') = r(G) + 1.

**Proposición VI.1.4**

Sea G=(V,E) un grafo. Sea n=|V|, m=|E| y p(G) el número de componentes conexas de G. El rango de los ciclos de G es

$$r(G) = m - n + p(G).$$

**Demostración:**

Usaremos inducción sobre el número de lados de G .

Si G = (V, {}) entonces r(G) = 0 = |E| – |V| + p(G).

Supongamos que la propiedad es cierta para todo grafo con a lo sumo m lados. Veamos que también es cierta para grafos con m+1 lados.

Sea G=(V,E) un grafo con m+1 lados (m≥0), sea e={v,w} un lado cualquiera en G, y sea G'=(V,E-{e})

Sabemos que existe una cadena entre v y w en G' si al agregar e ={v,w} a G', se forma un nuevo ciclo C .

En este caso r(G) = r(G') + 1, según la proposición VI.1.3.

- si existe cadena entre v y w en G', tenemos

$|E| = |E(G')| + 1$ y $p(G') = p(G)$ y así

$r(G) = r(G') + 1 = |E(G')| - |V| + p(G') + 1 = |E| - |V| + p(G)$

- si no existe cadena entre v y w en G', entonces e no genera un nuevo ciclo en G, de donde:

$r(G) = r(G')$, $|E| = |E(G')| + 1$, $p(G) = p(G') - 1$,

y así de nuevo

$r(G) = r(G') = |E(G')| - |V| + p(G') + 1 - 1$

$= |E| - |V| + p(G)$.

¤

**Definición VI.1.4**

Decimos que un grafo G es *arista maximal* (respectivamente *minimal* ) con respecto a una propiedad si G cumple dicha propiedad pero la pierde siempre que una arista es agregada al ( respectivamente eliminada del) grafo.

Podemos ahora enunciar el siguiente resultado sobre árboles:

**Proposición VI.1.5**

Los siguientes enunciados son equivalentes para un grafo G=(V,E) de orden n≥1.
(1) G es un árbol.
(2) G tiene (n–1) lados y no tiene ciclos.
(3) G es conexo y tiene (n–1) lados.
(4) G es arista maximal con respecto a la propiedad "no tiene ciclos".
(5) G es arista minimal con respecto a la propiedad "es conexo".
(6) Para cada par de vértices **x**, **y** en V(G) existe una única cadena simple de **x** a **y**.

**Demostración:**

Mostraremos (1) ⇒(2)⇒(3)⇒(4)⇒(5)⇒(6)⇒(1).

Sea |E|=m, n=|V| = orden de G, p(G)=Nº de componentes conexas de G.

(1) ⇒ (2): Si G es conexo entonces p(G) = 1. Por ser G un árbol, no tiene ciclos, luego
$r(G) = 0 = m - n + 1$, de donde $|E| = m = n-1$.

(2) ⇒ (3): Si G no tiene ciclos entonces $r(G) = 0 = m - n + p(G)$. Pero $m = n - 1$, de donde $p(G) = n - (n-1) = 1$, luego G es conexo.

(3) ⇒ (4): Si G conexo entonces p(G) = 1, además $m = n-1 \Rightarrow r(G) = (n-1) - n + 1 = 0$ luego G no tiene ciclos. Si agregamos una arista **e** a G, el grafo resultante

G' = (V, E') tendrá | E'| = n, p(G') = 1, luego r(G')=n – n + 1 = 1 y por lo tanto G' tiene un ciclo.

(4) ⇒ (5): Veamos que G es conexo. Supongamos que no lo es, entonces G tendrá por lo menos dos componentes conexas $C_1$ y $C_2$. Sea $v \in C_1$, $w \in C_2$. Si agregamos e={v,w} a E(G), este lado no formará un ciclo en G, lo cual contradice la hipótesis que G es arista maximal sin ciclos.

Tenemos entonces que r(G) = 0 y p(G) = 1 ya que G es conexo. Así $0 = r(G) = m - n + 1$ lo cual implica que $m = n - 1$.

Si quitamos una arista **e** a G, el grafo resultante G"=(V, E") no posee ciclos y

$|E"| = m" = m - 1$, luego

$0 = r(G'') = m'' - n + p(G'') = m - 1 - n + p(G'')$

$= n - 2 - n + p(G'') = p(G'') - 2$ , de donde $p(G'') = 2$ y por lo tanto G" no es conexo.

(5) $\Rightarrow$ (6): Supongamos que existen dos cadenas simples diferentes C y C' entre dos vértices $v,w \in V$, entonces por la proposición III.1.3.8 existe un ciclo en G. Sea $e=\{x,y\}$ una arista en ese ciclo,entonces en $G =(V,E-\{e\})$ existe aún una cadena entre cualquier par de vértices, es decir que es conexo, lo cual contradice la arista minimalidad de G con respecto a la propiedad de ser conexo.

(6) $\Rightarrow$ (1): Si existe cadena entre cualquier par de vértices, entonces G es conexo, de donde $p(G) = 1$. Si G tuviera un ciclo, entonces existiría al menos un par de vértices con al menos dos cadenas distintas entre ellos, lo cual contradice nuestra hipotesis de unicidad de cadenas entre cada par de vértices.

¤

**Proposición VI.1.6**

Un árbol $G = (V,E)$ con $|V| \geq 2$ tiene al menos dos vértices de grado 1.

**Demostración:**

Sea $n = |V|$, por la proposición VI.1.5 sabemos que $|E| = n-1$. por ser G conexo : $d(x) \geq 1$, $\forall x \in V$. Si para todos los vértices menos quizá uno $d(x) \geq 2$ entonces:

$$2|E| = \sum_{x \in V} d(x) \geq 2n - 1 > 2(n-1)$$

lo cual es una contradicción, por lo tanto deben existir al menos 2 vértices de grado 1.

¤

**Proposición VI.1.7**

Un grafo $G = (V,E)$ es conexo si y sólo si contiene un grafo parcial que sea un árbol.

**Demostración:**

Basta recordar cualquiera de los métodos para recorrer grafos, DFS o BFS, vistos en el capitulo IV, ya que con cualquiera de ellos obtenemos un grafo parcial que es un árbol si y sólo si el grafo es conexo.

¤

### VI.1.1. ÁRBOL COBERTOR DE COSTO MÍNIMO

Sea G=(V,E) un grafo conexo y sea c una función definida sobre los lados de G, $c: E(G) \rightarrow \Re$. Un grafo como el anterior nos permite representar diversas situaciones como por ejemplo las que se presentan en el estudio de redes de comunicación (vial, teléfonica, eléctrica, por ejemplo).

Supongamos que los vértices representan ciudades y los lados posibles interconexiones entre ellas. La función $c:E \rightarrow \Re$ podría representar los costos asociados a la instalación de cada interconexión. El problema en cuestión será buscar una red que conecte todas las ciudades al menor costo posible. Si además suponemos todos los costos positivos, será obvio que mientras menos lados tenga la red, menor será su costo.

**Definición VI.1.1.1**

Sea G un grafo conexo. Un grafo parcial de G conexo que es a su vez un árbol se llama un *Árbol Cobertor de G.*.

En vista de lo anterior y puesto que debemos conservar la conexidad del grafo, es evidente que necesitamos un árbol cobertor del grafo. Pero hay aún más, el árbol cobertor T que se busca debe ser el de menor costo, es decir, se busca un grafo parcial T de G que sea un árbol y tal que $\sum_{e \in T} c(e)$ sea mínima.

Un árbol que cumpla con la propiedad anterior se llama un *Árbol Mínimo Cobertor de G.*

**Proposición VI.1.1.1**

Sea G=(V,E) un grafo conexo y c una función c: E$\to\Re$. Sea U un subconjunto propio de V. Sea e={x,y} un lado en $\Omega$(U) (el cociclo definido por U) tal que , $\forall$ e'$\in \Omega$(U) : c(e) $\le$ c(e'). Entonces existe un árbol mínimo cobertor T de G que contiene a e.

**Demostración:**

Sea T=(V,E(T)) un árbol mínimo cobertor de G. Si e está en T, la propiedad es cierta. Si e no está en T, entonces agregando e a T se formará un ciclo en T'=(V,E(T)$\cup${e}) que lo incluye ( Prop. VI.1.5 (4) ). Esto implica que existe otro lado e' = {u,v} en T' tal que e'$\in \Omega$(U). Si eliminamos de T' el lado e', obtenemos entonces un nuevo árbol cobertor T" de G, tal que $\sum_{e\in T"} c(e) \le \sum_{e\in T} c(e)$ puesto que por hipótesis, c(e) $\le$ c(e'). Por lo tanto T" es un árbol mínimo cobertor y la propiedad queda demostrada.

¤

Los dos algoritmos que veremos a continuación nos permiten encontrar un árbol mínimo cobertor en un grafo G conexo, y ambos están basados en la propiedad anterior. La diferencia entre ellos radica en la forma cómo se escoge el conjunto U de la proposición VI. 1.1.1.

**Algoritmo de Prim:**

La idea básica de este algoritmo consiste en inicializar como vacío al conjunto que contendrá los lados del árbol T y con un vértice cualquiera **u** al conjunto U **.** Luego se selecciona el lado {**u,v**} en $\Omega$(U) , como se indica en la proposición VI.1.1.1. El vértice **v** se agrega al conjunto U. El árbol T buscado se irá formando al agregar, uno a uno, los lados que cumplan la propiedad, al mismo tiempo que crece U al incorporársele el otro extremo del lado recién agregado a T. Este proceso se repite hasta que los lados en T formen un árbol cobertor de G, o lo que es lo mismo hasta que U sea igual a V, es decir n -1 veces.

Supongamos que se numera el conjunto V de vértices de G, V={$v_1, v_2, \dots , v_n$}

El algoritmo es el siguiente:

Algoritmo de Prim ( Versión 1):
{Entrada: un grafo G=(V,E) no orientado conexo y una función de costos en las aristas.
Salida: Un conjunto T de lados tales que G(T) es un árbol mínimo cobertor de G.}
Variable U: conjunto de vértices .
Comienzo
(0) T $\leftarrow$ {}; U $\leftarrow$ {$v_1$};
(1) Mientras U $\neq$ V hacer
{ Invariante : |T|=|U|-1 y T está contenido en un árbol cobertor de costo mínimo}
Comienzo
1.1 Escoger e={u,v}, el lado de costo mínimo en $\Omega$(U),
es decir, con u$\in$U, v$\in$V – U;
1.2 T $\leftarrow$ T $\cup$ {e};
1.3 U $\leftarrow$ U $\cup$ {v};
Fin;
Fin;

**Proposición VI.1.1.2**

El algoritmo de Prim construye un árbol mínimo cobertor.

**Demostración:**

Para demostrar esta proposición basta con demostrar el siguiente invariante: al comienzo de una iteración cualquiera existe un árbol mínimo cobertor que contiene a T.

En la primera iteración es evidente que se cumple el invariante.

Supongamos que se cumple al inicio de una iteración k, veamos que también se cumplirá al comienzo de la iteración k+1:

Al comienzo de la iteración k: sea Tmin el árbol mínimo cobertor que contiene a T y e* el lado de costo mínimo en $\Omega$(U). Si e* está en Tmin, entonces al comienzo de la iteración k+1, T estará contenido en Tmin. Si e* no está en Tmin, el grafo formado al agregar e* a Tmin posee un ciclo C. Entonces existirá un lado e' en C $\cap \Omega$(U) (nótese que e' no puede

estar en T). El árbol que se obtiene al agregar e* y eliminar e' de Tmin es un árbol mínimo cobertor (ver Proposición VI.1.1.1) que contiene a T∪{e*}, el cual es el conjunto T al comienzo de la iteración k+1.

¤

A efectos de implementación, una forma sencilla de encontrar a cada paso el lado {u,v} de menor costo que conecte U con V–U es la siguiente:

- en un arreglo VECINO[i] mantenemos para cada vértice $v_i$ en V–U el número del vértice en U que esté conectado a $v_i$ por el lado de menor costo;

- en otro arreglo COSTOMIN[i] tenemos el costo del lado $\{v_i, VECINO[i]\}$.

El arreglo COSTOMIN se inicializa de la siguiente forma:

- para cada vértice vi adyacente a v1, $COSTOMIN[i] = c(\{v_i,v_1\})$.

- para los vértices $v_i$ no adyacentes a $v_1$ a COSTOMIN[i] se le asigna un valor suficientemente grande.

En cada iteración del algoritmo, recorriendo COSTOMIN encontramos el vértice $v_k$ en V–U conectado por un lado de costo mínimo en Ω(U) a un vértice en U y se lo anexamos al conjunto U. Al final, los lados {i, VECINO[i]} serán los lados del árbol. Será necesario en cada iteración actualizar ambos arreglos, teniendo en cuenta que el vértice $v_k$ entra en U y que alguno de los vértices que restan en V–U pudiera estar conectado a $v_k$ por un lado de costo mínimo en Ω(U). Será necesario tener un arreglo que indique si un vértice está o no en V-U. Cada vez que un vértice $v_k$ entra en U, a COSTOMIN[k] se le asigna un valor lo suficientemente grande para evitar que sea considerado de nuevo.

Con una implementación como ésta, la complejidad del algoritmo de Prim es $O(n^2)$, puesto que se debe recorrer n–1 veces un arreglo de tamaño n.

Podemos ver además, que este algoritmo de Prim es otro caso particular del algoritmo general de etiquetamiento. Para ello, asignaremos como atributo de cada cadena abierta del algoritmo de etiquetamiento el costo de su último lado. En cada iteración se escoge la cadena abierta que tenga menor atributo y se cierra esa cadena. Las cadenas cerradas no se reemplazan y serán las que forman el árbol mínimo cobertor. Como justificación de esta versión del algoritmo de Prim podemos considerar el conjunto U formado por los vértices finales de las cadenas cerradas, de modo que el último lado de cada cadena abierta es un lado en Ω(U). Al escoger la cadena abierta con menor atributo estamos escogiendo un lado como lo señala la proposición VI.1.1.1.

Algoritmo de Prim (Versión 2):
{Entrada: un grafo G=(V,E) no orientado.
Salida: una lista de cadenas en G tales que el sub-grafo G(T), inducido por el conjunto de lados de las cadenas es un árbol mínimo cobertor de G.}
Comienzo
(0) Escoja un vértice $v_1$ en V, abra el camino $P_o = < v_1 >$ y asígnele su atributo $A(P_o) = 0$;
(1) Mientras existan cadenas abiertas hacer:
Comienzo
1.1 Escoja el camino abierto $P_s = <v_1, \dots, v_s>$ con menor atributo.
1.2 Cierre $P_s$.
1.3 Obtenga los vecinos $v^1, \dots, v^q$ de $v_s$ (el vértice terminal de $P_s$).
1.4 Construya los caminos expandidos $P^i = Exp(P_s, v^i)$, $1 \le i \le q$, y calcule sus atributos $A(P^i) = c(\{v_s, v_i\})$.
1.5 Aplique la rutina de eliminación de cadenas.
Fin;
Donde
Rutina de Eliminación de Cadenas:
Comienzo
1. Para cada $P^i = Exp(P_s, v^i)$, $1 \le i \le q$, hacer:

```
        Comienzo
         1.1Si existe un camino listado con vértice terminal
              igual a v^i entonces
               Comienzo
               Si existe un camino abierto P_k con vértice terminal
              igual a v^i y A (P_k) >  A( P^i ) entonces
                    Comienzo
                      1.1.2     Eliminar P^k de la lista;
                      1.1.3     Abrir P^i;
                    Fin;
               Fin;
        1.2  Si no abra P^i
         Fin;
  Fin;  {Rutina de eliminación}
Fin.   { Prim }
```

A continuación se muestra un ejemplo de aplicación del algoritmo de Prim (Versión 2) al grafo de la figura VI.2.

| | | | |
|---|---|---|---|
| Iter 0: | $P_o$ = < a > | Abierto | A ( Po ) = 0 |
| Iter 1: | $P_o$ = < a > | Cerrado. | |
| | $P_1$ = < a, b > | Abierto | A ( $P_1$ ) = 1 |
| | $P_2$ = < a, d > | Abierto | A ( $P_2$ ) = 2 |
| Iter 2: | $P_o$ = < a > | Cerrado. | |
| | $P_1$ = < a, b > | Cerrado. | |
| | $P_3$ = < a, b, c > | Abierto | A ( $P_3$ ) =2 |
| | $P_4$ = < a, b, d > | Abierto | A ( $P_4$ ) = 1 |

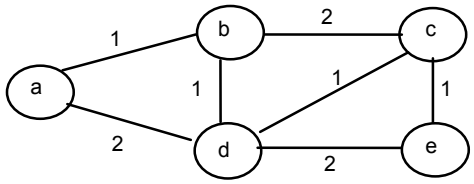



**figura VI.2**

Nota: El camino $P_2$ = < a, d > fue eliminado para abrir el camino $P_4$ = < a, b, d >.

| | | | |
|---|---|---|---|
| Iter 3: | $P_o$ = < a > | Cerrado. | |
| | $P_1$ = < a, b > | Cerrado. | |
| | $P_4$ = < a, b, d > | Cerrado. | |
| | $P_5$ = < a, b, d, c > | Abierto | A ( $P_5$ ) = 1 |
| | $P_6$ = < a, b, d, e > | Abierto | A ( $P_6$ ) = 2 |
| Iter 4: | $P_o$ = < a > | Cerrado. | |
| | $P_1$ = < a, b > | Cerrado. | |
| | $P_4$ = < a, b, d > | Cerrado. | |
| | $P_5$ = < a, b, d, c > | Cerrado. | |
| | $P_7$ = < a, b, d, c, e > | Abierto | A ( $P_7$ ) = 1 |
| Iter 5: | $P_o$ = < a > | Cerrado. | |
| | $P_1$ = < a, b > | Cerrado. | |
| | $P_4$ = < a, b, d > | Cerrado. | |
| | $P_5$ = < a, b, d, c > | Cerrado. | |

$P_7$ = < a, b, d, c, e >  Cerrado.

**Algoritmo de Kruskal;**

Otra forma de construir un árbol mínimo cobertor de un grafo G=(V,E) es comenzando con un grafo T= (V,{}) donde V es el conjunto de vértices de G. De esta forma cada vértice v en T forma una componente conexa. A medida que avanza el algoritmo iremos agregando lados a T y se formarán nuevas componentes a partir de las anteriores. Para decidir qué lados agregar, se examinan los lados de E en orden creciente de costo. Si el lado en consideración NO forma ciclo con los otros lados ya en T, es decir, si conecta dos vértices en componentes distintas, entonces se agrega ese lado a T, en caso contrario se descarta y se examina el siguiente lado. Cuando todos los vértices forman una sola componente en T, es decir cuando ya han sido seleccionados los |V|–1 lados requeridos, T es un árbol mínimo cobertor.

El algortimo en cuestión es el siguiente:

Algoritmo de Kruskal (Versión 1):
{Entrada: un grafo G = (V,E) conexo no orientado.
Salida: un conjunto T de lados tal que G(T) es un árbol mínimo cobertor de G.}
Variables : Numcomp: entero; e : lado;
{NumComp representa el Nº de componentes conexas del grafo en cada iteración }
Comienzo
(0) T ← {}; NumComp ← |V|; E' ← E;
(1) Mientras NumComp ≠ 1 hacer
{Invariante: existe un árbol mínimo cobertor que contiene a T y NumComp = al número de componentes conexas del grafo inducido por T = número de componentes conexas de G(E-E')}
Comienzo
1.1 e* ← lado de menor costo en E';
1.2 E' ← E' – {e*};
1.3 Si (G( T ∪ {e*} )) no contiene ciclos entonces
Comienzo
T ← T ∪ {e*} ;
NumComp ← NumComp – 1;
Fin;
Fin;
Fin.

**Proposición VI.1.1.3**

El algoritmo de Kruskal construye un árbol mínimo cobertor.

**Demostración:**

Para demostrar esta proposición basta con demostrar el siguiente invariante: al comienzo de una iteración cualquiera existe un árbol mínimo cobertor que contiene a T y NumComp es igual al número de componentes conexas de G(E-E').

En la primera iteración es evidente que se cumple el invariante.

Supongamos que se cumple al inicio de una iteración k, veamos que también se cumplirá al comienzo de la iteración k+1:

Al comienzo de la iteración k: sea Tmin el árbol mínimo cobertor que contiene a T. Si e* forma ciclo en G(T∪{e*}) entonces el conjunto T al comienzo de la iteración k+1 será igual al conjunto T al comienzo de la iteración k. En caso contrario, sea U el conjunto de vértices de una de las componentes conexas que contiene un extremo de e*. Así, e* está en Ω(U) y es el de menor costo entre todos los lados en Ω(U). Si e* está en Tmin, entonces al comienzo de la iteración k+1, T estará contenido en Tmin. Si e* no está en Tmin, el grafo formado al agregar e* a Tmin posee un ciclo C. Entonces existirá un lado e' en C ∩ Ω(U) (nótese que e' no puede estar en T). El árbol que se obtiene al agregar e* y eliminar e' de Tmin es un árbol mínimo cobertor (ver Proposición VI.1.1.1) que contiene a T∪{e*}, el cual es el conjunto T al comienzo de la iteración k+1.

Por otra parte es evidente que NumComp es igual al número de componentes conexas de G(E-E'). Por lo que podemos garantizar que el While termina ya que NumComp crece de iteración a iteración (aunque no estrictamente), y como la cardinalidad de E' decrece estrictamente en 1 en cada iteración, en alguna iteración el número de componentes conexas de G(E-E') será igual a 1 porque G es conexo.

¤

Este algoritmo resulta sumamente sencillo a simple vista, sin embargo, notaremos que es necesario revisar ciertos detalles en cuanto a la implementación del mismo. Lo primero es con respecto a la forma de seleccionar los lados en orden creciente de costo. Ordenar desde un principio todos los lados resulta bastante inconveniente pues normalmente se estará haciendo mucho más trabajo del necesario, ya que rara vez se llegan a examinar todos los lados antes de encontrar los |V|–1 requeridos. Lo que se necesita entonces es disponer de un operador que permita encontrar en cada iteración el lado de menor costo entre los que aún no han sido estudiados y que lo excluya de E(G). Para lograr lo anterior podemos colocar los lados en un árbol parcialmente ordenado o heap y utilizar un operador que llamaremos LADOMIN. Con este operador podemos extraer el lado de menor distancia y reordenar el heap en un tiempo *O* (log|E|). Igualmente podemos ordenar parcialmente los lados antes de comenzar en un tiempo *O* (|E|), esto lo realizará el operador ORDENAR, similar al procedimiento CREAR de la sección VII.4.

Lo segundo que se debe revisar detalladamente es como se almacenarán los vértices de cada componente, de forma que pueda revisarse de forma eficiente la posible formación de ciclos cada vez que se examina un nuevo lado. Una solución es formar conjuntos con los vértices que pertenecen a una misma componente y de esta manera, con una representación adecuada para conjuntos ( AHO et al, [1] ), se puede determinar en un tiempo constante a qué componente pertenecen los extremos de un lado, mediante un operador ENCONTRAR aplicado a cada uno de sus extremos para luego verificar si esas componentes son distintas. En caso de ser distintas se deben unir las componentes en cuestión y agregar el lado a T; en caso contrario se desecha el lado examinado. Para unir las dos componentes se requerirá de un operador UNION.

Las n-1 uniones que serán requeridas se pueden lograr en un tiempo *O* (nlogn) con la misma representación de conjuntos.

El operador INICIALIZAR crea las componentes conexas del grafo G=(V,{}), las cuales constan de un solo vértice cada una y serán identificadas con un número de 1 a |V|.

Utilizando los operadores descritos arriba, el algoritmo resulta:

```
Algoritmo de Kruskal  (Versión 2):
{Entrada:  Un grafo G = (V,E) no orientado conexo y una función de costos en las aristas.
 Salida :  Un conjunto de lados T tal que G(T) es un árbol mínimo cobertor del grafo G.}
Variables  Comp, NumComp: entero;
           e: registro x,y:vértice Fin;
           Xcomp,Ycomp : entero;
{ Comp es usado para etiquetar las componentes. NumComp
representa el Nº de componentes conexas del grafo en cada iteración};
Comienzo
(0)  T ← {};  NumComp ← |V|;
(1)  Para i ← 1 hasta |V| hacer
     Comienzo
       INICIALIZAR ( i, i );
     Fin;
(2)  ORDENAR (E);
(3)  Mientras NumComp≠1 hacer
     Comienzo
       LADOMIN (E, e);  { e={e.x, e.y} }
       ENCONTRAR (e.x, Xcomp);
       ENCONTRAR (e.y, Ycomp);
       Si Xcomp ≠ Ycomp  entonces
            Comienzo
              UNION (Xcomp, Ycomp);
            T ← T ∪ {e};
            NumComp ← NumComp -1;
            Fin;
     Fin;
Fin.
```

Con una implementación como ésta, el tiempo de ejecución del algoritmo será *O* (|E|log|E|), pues la inicialización es *O* (|V|), ORDENAR es *O* (|E|), las n-1 veces que se realiza la UNION se logra en un tiempo *O* (|V|log|V|) y LADOMIN es

*O* (log|E|) y será repetida p veces, donde p es el número de lados que serán examinados antes de formar el árbol (normalmente considerablemente menor que |E|), siendo esto lo que consume el mayor de tiempo del algoritmo.

### VI.2 ARBORESCENCIAS. PROPIEDADES

En el Capítulo III hemos definido una raíz de un árbol orientado A como un vértice desde el cual son alcanzables todos los demás vértices del árbol. Se definió así mismo una *arborescencia* como un árbol orientado con un vértice raíz r. Se vieron también algunas consecuencias directas de esta definición que dan lugar a la siguiente proposición.

**Proposición VI.2.1**

Sea G = (V,E) un grafo orientado de orden n. Las condiciones siguientes son equivalentes y caracterizan una *arborescencia* de raíz **r**.

(i) G es un árbol con una raíz **r**.

(ii) G tiene un vértice raíz **r** y $\forall x \in V$, $x \neq \mathbf{r}$ : existe un único camino de **r** a x.

(iii) G admite **r** como raíz y es arco minimal bajo esta propiedad, es decir, si se elimina un arco de G, **r** deja de ser raíz.

(iv) G tiene una raíz **r** y $d^-(\mathbf{r}) = 0$, $d^-(x) = 1$, $\forall x \in V$, $x \neq \mathbf{r}$.

(v) G es conexo con $d^-(\mathbf{r}) = 0$ y $d^-(x) = 1$, $\forall x \in V$, $x \neq \mathbf{r}$.

(vi) G no tiene ciclos y $d^-(\mathbf{r}) = 0$, $d^-(x) = 1$, $\forall x \in V$, $x \neq \mathbf{r}$.

(vii) G tiene una raíz **r** y no tiene ciclos.

(viii) G tiene una raíz **r** y $|E| = n - 1$.

**Demostración:**

(i) $\Rightarrow$ (ii) G tiene una raíz **r** y por lo tanto existen caminos de **r** a x, $\forall x \in V$. Por ser G un árbol no tiene ciclos y en consecuencia tampoco circuitos, por lo que todos los caminos son elementales.

Supongamos que existe $y \in V$ tal que existen 2 caminos distintos de **r** a y, entonces en el grafo subyacente de G existen dos cadenas elementales, y por tanto simples, distintas de **r** a x, lo cual contradice la hipótesis que G es un árbol (Prop. VI.1.5 (6)).

(ii) $\Rightarrow$ (iii) Sea $e \in E$ tal que **r** es raíz de $G' = (V, E - \{e\})$, entonces existirían dos caminos distintos de **r** al extremo final de e, contradiciendo (ii).

(iii) $\Rightarrow$ (iv) Por ser **r** una raíz sabemos que existe camino de **r** a x, por lo tanto $\forall x \in V, x \neq \mathbf{r}$, $d^-(x) \geq 1$. Supongamos que $\exists\, y \in V$ con $d^-(y) \geq 2$; entonces $\exists\, v_1 \neq v_2$ en V tales que $(v_1,y), (v_2,y) \in E$. Sea $G' = (V, E - \{(v_1,y)\})$, **r** es raíz de G' contradiciendo (iii).

Por otra parte, si $d^-(r) \geq 1$ entonces existiría $x \neq r$ tal que $(x,r) \in E$ y si eliminamos este arco de G **r** seguiría siendo raíz, contradiciendo de nuevo la arco minimalidad de G con respecto a esta propiedad.

(iv) $\Rightarrow$ (v) G es conexo puesto que admite una raíz.

(v) $\Rightarrow$ (vi) Si $d^-(r)=0$ y $d^-(x)=1$, $\forall x \in V$, $x \neq r$ entonces $|E| = \sum_{x \in V} d^-(x) = n-1$

por lo tanto, siendo G conexo, no tiene ciclos.

(vi) $\Rightarrow$ (vii) Si G no tiene ciclos tampoco tiene circuitos y por lo tanto es una grafo de precedencias cuya única fuente es el vértice **r**. Por la Prop. V.5.1.4 todos los caminos más largos partirán de **r** .

$d^-(r)=0$ $d^-(x) =1$ $\forall x \in V$, $x \neq \mathbf{r} \Rightarrow$ todo vértice menos **r** tiene un predecesor, y por lo tanto algún camino que le llega, el más largo de los cuales parte de r, luego **r** es raíz de G.

(vii) $\Rightarrow$ (viii) **r** raíz y G sin ciclos $\Rightarrow$ G conexo y sin ciclos, luego G es un árbol, por lo tanto $|E| = n - 1$.

(viii) $\Rightarrow$ (i) G con raíz $\Rightarrow$ G conexo. $|E| = n - 1$ y G conexo $\Rightarrow$ G es conexo y sin ciclos y por lo tanto es un árbol.

¤

**Observación:**

Resulta inmediato de la proposición anterior que una arborescencia no tiene circuitos y por lo tanto, todos los caminos son elementales. Además, toda arborescencia tiene un único vértice fuente, la raíz, y al menos un vértice sumidero, una hoja.

### VI.2.1 ARBORESCENCIAS COBERTORAS ÓPTIMAS

Un problema similar al de conseguir un árbol óptimo (mínimo o máximo) cobertor se presenta en grafos o redes orientados. Son muchos los problemas que se pueden plantear en los cuales se requiere determinar la forma de encontrar un punto desde el cual comunicarse con todos los demás de un grafo, respetando la orientación de los arcos (vías de comunicación, por ejemplo) y al menor costo global posible. Nuevamente se entiende por costo global de la red la suma de los costos de todos los arcos utilizados. En este caso lo que se desea obtener es una arborescencia mínima cobertora en el grafo asociado al problema.

Definiremos un *Bosque Orientado* como un bosque en el cual cada componente conexa es una arborescencia.

Sea G =(V,E) un digrafo y sea c: E $\rightarrow \Re$ una función que asocia a cada arco e$\in$E un número c(e)$\in\Re$. El problema que tratamos aquí consiste en encontrar un grafo parcial B de G tal que B sea un Bosque orientado y la suma $\sum_{e\in B} c(e)$ sea máxima.

Un subgrafo que cumpla estas condiciones se llama *Bosque Orientado de Peso Máximo de G*. El problema que planteamos al principio de esta sección y algunos otros que mencionaremos, se reducen a variaciones del problema de buscar un Bosque Orientado de Peso Máximo.

Es fácil observar que un bosque orientado B de un grafo G es una arborescencia cobertora si y sólo si el número de arcos de B es igual al número de vértices de G menos uno, y este es el máximo de arcos que puede tener un bosque.

Un bosque orientado de peso máximo de G no incluye ningún arco e con c(e)<0 , por lo tanto, si c(e)≤o, $\forall$ e$\in$E, el bosque orientado de peso máximo tendrá un conjunto vacío de arcos. Más aún, G puede tener un subgrafo que sea una arborescencia y c(e)>0, $\forall$ e$\in$E, y sin embargo, el bosque orientado de peso máximo de G no ser una arborescencia, como se muestra en el ejemplo de la figura VI.3.

Si un grafo G=(V,E) con c:E $\rightarrow\Re$ tiene un subgrafo parcial que es una arborescencia, entonces una arborescencia cobertora de peso máximo de G se puede conseguir basándonos en la siguiente propiedad y en el corolario que le sigue.

**Proposición VI.2.1.1**

Sea G=(V,E) un digrafo y c:E $\rightarrow\Re$.

Sea c':E $\rightarrow\Re$ con c'(e) = c(e) + k, $\forall$e$\in$E, y $k > \sum_{e\in E}|c(e)|$.

Si B es un bosque orientado de peso máximo para c', entonces B es el bosque con mayor peso con respecto a c, entre todos los bosques orientados cobertores de G que poseen el mayor número de arcos posible.

**Demostración:**

Veremos primero que un bosque orientado de peso máximo en G con c'j = cj + k, $k > \sum_{e\in E}|c(e)|$, tiene un número máximo de arcos.

Sean B1 y B2 dos bosques orientados cobertores de G con pesos P'(B1) y P'(B2), tales que:

|E(B1)| > |E(B2)|. Entonces

$$P'(B1) = \sum_{e\in E(B1)} (c(e)+k) = \sum_{e\in E(B1)} c(e) + |E(B1)|*k$$

$$\geq \sum_{e\in E(B1)} c(e) + (|E(B2)| + 1)*k = \sum_{e\in E(B1)} c(e) - \sum_{e\in E(B2)} c(e) + \sum_{e\in E(B2)} c(e) + (|E(B2)| + 1)*k >$$

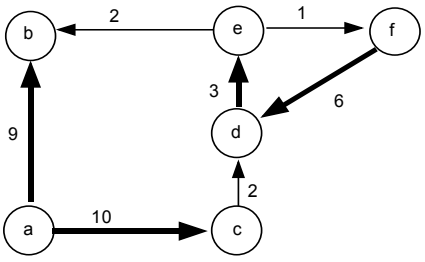
$$\sum_{e\in E(B1)} c(e) - \sum_{e\in E(B2)} c(e) + \sum_{e\in E}|c(e)| + \sum_{e\in E(B2)} c(e) + |E(B2)| *k =$$

$$\sum_{e\in E(B1)-E(B2)} c(e) - \sum_{e\in E(B2)-E(B1)} c(e) + \sum_{e\in E} |c(e)| + \sum_{e\in E(B2)} c(e) + |E(B2)| * k$$

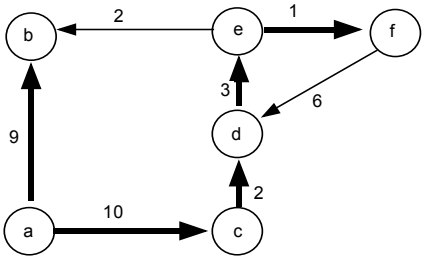
$$\geq \sum_{e\in E(B2)} c(e) + |E(B2)| * k = P'(B2)$$

De lo anterior se deduce que si B es un bosque orientado cobertor de G con respecto a c', entonces B tiene número máximo de arcos. Además, si $B_1$, $B_2$ son dos bosques orientados cobertores de G con $|E(B_1)| = |E(B_2)|$ y $P'(B_1) > P'(B_2)$ entonces $\sum_{e\in E(B1)} c(e) = P(B1) > P(B2) = \sum_{e\in E(B2)} c(e)$, lo cual completa la demostración.

¤



(a) Bosque de peso máximo



(b) Arborescencia cobertora de peso máximo

**figura VI.3**

**Corolario VI.2.1.1**

Si G admite una arborescencia, el bosque máximo B con respecto a c' es una arborescencia de peso máximo con respecto a c en G.

**Demostración**

La prueba se basa en el hecho que ningún bosque tiene más arcos que una arborescencia cobertora y que si A1, A2 son dos arborescencias cobertoras de G con pesos P(A1) > P(A2) con respecto a c entonces

$$P'(A1) = \sum_{e\in E(A1)} (c(e)+k) = P(A1) + (|V| - 1) * k$$

$$> P(A2) + (|V| - 1) * k = P'(A2)$$

donde P'(A1), P'(A2) son los pesos de las arborescencias con respecto a c'.

¤

Sin embargo, no será necesario modificar los pesos de los arcos para buscar una arborescencia máxima cobertora, ya que el algoritmo que estudiaremos se puede modificar fácilmente para tratar directamente este problema.

Una arborescencia A en G de peso mínimo, es decir, una arborescencia con $\sum_{e\in E(A)} c(e)$ mínima es una arborescencia de peso máximo en G' =(V,E) con c'(e) = –c(e), $\forall e\in E$.

Por otra parte, si lo que se busca es una arborescencia cobertora de peso máximo o mínimo cuya raíz sea un vértice dado **x**, basta con agregar al grafo original un vértice ficticio $x_o$ y un arco $(x_o, x)$ con cualquier peso positivo. Si existe una arborescencia en G con raíz **x**, entonces existe una arborescencia con raíz $x_o$ en el grafo modificado.

Veremos a continuación un algoritmo debido a Edmonds (1966) para encontrar un bosque de peso máximo en un grafo G = (V,E) con c: $E\rightarrow\Re$.

Definiremos conjuntos $D^i$ y $E^i$, inicialmente vacíos y a partir de ellos iremos generando una secuencia de grafos $G^i$, siendo $G^o$ = G, hasta conseguir en algún $G^f$ un bosque orientado de peso máximo, formado por los arcos en el conjunto $E^f$. El algoritmo es el siguiente:

Algoritmo de Búsqueda de Bosque Orientado de Peso Máximo de G = (V,E):

Comienzo

Paso 1: Escoja un vértice cualquiera x en $G^i$ tal que $x\notin D^i$. Agregue x a $D^i$. Si existen arcos en $G^i$ con peso positivo, dirigidos hacia x, entonces tome uno, digamos **e**, con peso máximo y luego inclúyalo en $E^i$. Repita este paso hasta que:

(a) Los arcos de $E^i$ no formen un bosque en $G^i$, es decir, hasta que se forme un circuito con la inclusión de e en $E^i$, o hasta que

(b) Todos los vértices de $G^i$ estén en $D^i$, y $E^i$ no posea circuitos, en cuyo caso los arcos de $E^i$ forman un bosque cobertor en $G^i$.

Note que cada arco **e** que se agrega a $E^i$ en el paso 1 es el único en $E^i$ que llega a algún vértice x en $D^i$, el cual hasta el momento de la inclusión de **e** en $E^i$ era una raíz de alguna componente del bosque $B' = (V(G^i), E^i)$. Si el otro extremo de **e** está en la misma componente que x, entonces al incluir **e** se formará un circuito $Q^i$, es decir, se presentará el caso (a).

Si ocurre (a) entonces:

Paso 2 : Almacene $Q^i$ junto con el arco $e_o{}^i$ de menor peso en $Q^i$. Contraiga los vértices en $Q^i$ en un nuevo vértice $v^{i+1}$ para formar el siguiente grafo $G^{i+1}$ cuyos vértices serán $V(G^{i+1}) = (V(G^i) \approx \{v^{i+1}\}) - V(Q^i)$. Los arcos de $G^{i+1}$son aquellos arcos de $G^i$ que tienen a lo sumo un extremo en $Q^i$. Por cada **a** arco en $G^i$ con un extremo ( y sólo uno) en $Q^i$ se crea un **a'** arco en $G^{i+1}$ que resulta de reemplazar en **a** el extremo en $Q^i$ por el vértice $v^{i+1}$. Denotemos por $F^{i+1}$ el conjunto de los arcos **a'** con extremo inicial $v^{i+1}$.

Haga $E^{i+1} = (E^i\cap E(G^{i+1}))\cup F^{i+1}$ y $D^{i+1} = D^i - V(Q^i)$

(note que $v^{i+1} \notin D^{i+1}$ y que en $E^{i+1}$ puede existir arcos paralelos).

Los pesos $c^{i+1}$ de los arcos en $G^{i+1}$ serán los mismos que en $G^i$ salvo para aquellos vértices cuyo extremo final sea $v^{i+1}$. Para cada uno de estos arcos $e_S$, sea $x_S$ el extremo de $e_S$ en $Q^i$ y $e_k$ el único otro arco en $Q^i$ que llega a $x_S$, entonces

$$c^{i+1}(e_S) = c^i(e_S) + c^i(e_o{}^i) - c^i(e_k).$$

Note que: $c^i(e_o) \geq 0$

$$c^i(e_k) \geq c^i(e_o)$$

$$c^i(e_k) \geq c^i(e_S)$$

(ya que siendo ambos, $e_S$ y $e_k$, incidentes a $x_k$, $e_k$ fue escogido para entrar en $E^i$ en lugar de $e_S$).

Repita nuevamente el paso 1 con i = i+1.
Luego de algunas repeticiones de Paso 1, Paso 2, ocurrirá el caso (b).
Al ocurrir (b) ejecute el Paso 3.

Paso 3 : Al llegar a la ejecución de este paso el grafo $(V(G^i), E^i)$, que llamaremos $B^i$, es un bosque orientado.

Construya $H^i$ a partir de $B^i$ sustituyendo el vértice $v_i$ por el circuito $Q^{i-1}$ (de cuya contracción resultó $v^i$). Es evidente que $H^i$ tiene un único circuito elemental que es $Q^{i-1}$.

Ahora construiremos el bosque $B^{i-1}$ de la siguiente manera:

Si $v^i$ es raíz de alguna componente en $B^i$, entonces elimine de $H^i$ el arco de menor peso en $Q^{i-1}$; el grafo así obtenido será $B^{i-1}$.

Si $v^i$ no es raíz en $B^i$ entonces, sea **e** el único arco que llega a $v^i$ en $B^i$. Este arco **e** corresponde a un arco e' en $G^{i-1}$ que llega a un vértice $x_k$ de $Q^{i-1}$. Sea $e_k$ el único otro arco en $Q^{i-1}$ que llega a $x_k$. $B^{i-1}$ se obtiene de eliminar el arco $e_k$ de $H^i$.

Si $i-1 \neq 0$, vuelva a repetir el paso 3 con $i=i-1$.

Fin;

Es evidente que al detenerse el algortimo, $B^o$ es un bosque orientado cobertor. Solo faltaría mostrar que $B^o$ es además un bosque de peso máximo.

Informalmente, puede verse que cada arco incluido en $E^i$ en el paso 1 es el mejor posible. Al pasar de $G^i$ a $G^{i+1}$ los nuevos costos de los arcos con su extremo final en $Q^i$ se determinan de manera tal que de ser elegido uno de ellos para entrar en $E^{i+1}$, entonces al construir el bosque $B^i$ en el paso 3, el arco en cuestión resultará en una mayor contribución al peso del bosque que los arcos del circuito que serían incluidos en caso contrario. Esto puede verse de la siguiente manera:

Si $e_S$ es un arco cuyo extremo final es $v^{i+1}$ y se le incluye en $E^{i+1}$ en el paso 1, entonces $c^{i+1}(e) = c^i(e) = c^i(e_S) + c^i(e_o{}^{i-1}) - c^i(e_k) > 0$, donde $e_k$ es el arco de $Q^i$ cuyo vértice final es el mismo de $e_S$. Es decir que:

$$c^i(e_S) + c^i(e_o{}^i) > c^i(e_k)$$

y justamente $e_k$ es el arco que será incluido en $H^i$ al construir $B^i$.

Si por el contrario, $v^{i+1}$ es raíz en $B^{i+1}$ es porque $c^{i+1}(e) \leq 0$, para cualquier arco e cuyo extremo final es $v^{i+1}$, por lo que resulta más conveniente excluir $e_o{}^i$ del circuito. El ejemplo de la figura VI.4 muestra esta situación. Para una demostración más formal de lo anterior el lector interesado puede referirse a Edmonds, J. 1968. "Optimum Branchings"[6].

Para obtener una arborescencia de peso máximo en un grafo que contenga una arborescencia cobertora, la única modificación necesaria es eliminar en el paso 1 la exigencia de tomar sólo arcos positivos. En efecto, en lo que a los cálculos del algoritmo se refiere, la única consecuencia de sumar a todos los arcos una constante positiva lo suficientemente grande es que en todo momento los pesos de los arcos serán positivos.

Esto último es fácil de observar, ya que si e es un arco cuyo vértice final es un vértice del circuito $Q^i$ encontrado en el paso 1, entonces:

$$c^{i+1}(e) = (c^i(e) + k) + (c^i(e_o) + k) - (c'(e_k) + k)$$

$$= c^i(e) + c^i(e_o) + k - c^i(e_k) > 0.$$

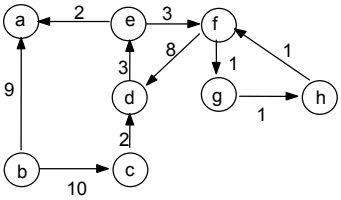
Como resultado, en el paso 1, siempre que para un vértice x, $d^-(x) > 0$, se incluirá un arco con vértice final x en $E^i$ asegurando así que se llegue a una arborescencia de pesos máximos (si el grafo tiene un subgrafo que sea una arborescencia cobertora).

Note que en el ejemplo de la figura VI.4 si sumamos la constante k=100, lo cual es mayor que $\Sigma_{e \in E} c(e)$, el costo del arco $(c,v^2)$ en $G^2$ sería igual a 97, con lo que sería incluido en en $E^2$ y así quedaría la arborescencia de la figura VI.5 como resultado final. El mismo resultado se obtendría si en el paso 1 permitimos escoger arcos negativos.

### VI.3 EQUIVALENCIA ENTRE ÁRBOLES Y ARBORESCENCIAS. ÁRBOLES ENRAIZADOS

Sea T un árbol no orientado y **r** un vértice cualquiera de T. Por ser T conexo, existe una cadena simple (única) desde **r** hasta todo otro vértice del árbol. Podemos entonces dotar de una orientación a cada uno de los lados de T de forma que

exista un camino (único) desde **r** hasta todos los demás vértices de T. El grafo orientado así obtenido es único y, además, es una arborescencia cuya raíz es el vértice **r**.




( a ) $G = G^0$

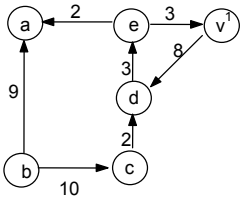
$D^o = \{ a, b, c, g, h, f \}$ $D^1 = \{ a, b, c, d, e, v^1 \}$

$E^o = \{(b,a),(b,c),(f,g),(g,h),(h,f)\}$

$Q^o = \langle f,(f,g),g,(g,h),h,(h,f),f \rangle$

$E_o^0 = (h,f)$ , $c^1(e, v^1) = 3 + 1 - 1 = 3$

$e^1 = (e,v^1)$, $c^2 = (c,v^2) = 2+3-8 = -3$

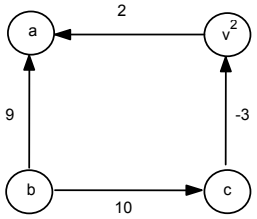



( b ) $G^1$

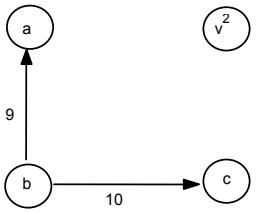
$D^1 = \{ a, b, c, d, e, v^1 \}$

$E^1 = \{(b,a),(b,c),(v^1,d),(d,e),(e,v^1)\}$

$Q^1 = \langle v^1,(v^1,d),d,(d,e),e,(e,v^1),v^1 \rangle$

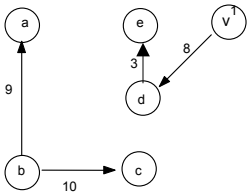



( c ) $G^2$

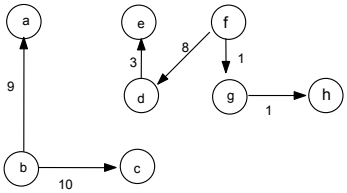



$B^2$

$D^2 = \{ a, b, c, v^2 \}$ $E^2 = \{ (b,a), (b,c) \}$

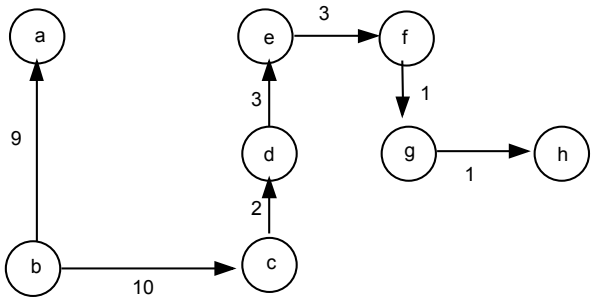



(d) $B^1$




$B^0$

**figura VI.4**




Arborescencia cobertora de peso máximo para el grafo G de la figura VI.4

**figura VI.5**

Un árbol no orientado en el cual distinguimos un vértice **r** se llama a veces un *árbol enraizado en **r*** y, como vimos anteriormente, existe una correspondencia entre los árboles enraizados y las arborescencias. Esta última afirmación permite a veces utilizar indistintamente los términos arborescencia y árbol, como suele ocurrir cuando se trabaja con árboles en Computación, siempre y cuando se especifique cuál es el vértice distinguido o raíz **r**.

En cuanto a la representación gráfica de los árboles enraizados, estos se suelen dibujar con el vértice raíz **r** en el tope, y el resto de los vértices pendiendo de **r**, con las hojas en la parte inferior de la figura. Este sentido de arriba hacia abajo corresponde a la orientación implícita de los lados.

En base a lo dicho en el punto anterior, a partir de este momento y mientras no se especifique lo contrario, no haremos distinción entre árboles enraizados y arborescencias, siempre que se distinga el vértice raíz.

### VI.4 TERMINOLOGÍA: NIVEL, ALTURA, LONGITUD

Sea A un árbol y **r** un vértice distinguido o raíz. Los vértices adyacentes a **r** se llaman sus *hijos* o *sucesores* y recíprocamente, **r** es su *padre* o *predecesor*. Sea B el bosque generado por los vértices de A menos la raíz. Se llama *sub-árbol* de **r** a cada uno de los árboles de B, cuyas raíces son los hijos de **r**. Se dice que dos vértices son hermanos si tienen el mismo padre. Un vértice sin sucesores se llama *vértice terminal* u *hoja*. Todo vértice que no sea una hoja se llama *vértice interior*. En consecuencia, todos los vértices interiores son raíces de algún sub-árbol no vacío de A.

Vemos entonces que es posible dar una definición equivalente y recursiva para árboles enraizados (extensible a arborescencias de manera natural) de la siguiente manera:

**Definición VI.4.1**

Un *árbol enraizado* es un grafo conexo A = (V,E) en el cual:

a) Hay un único vértice distinguido **r**, llamado raíz del árbol

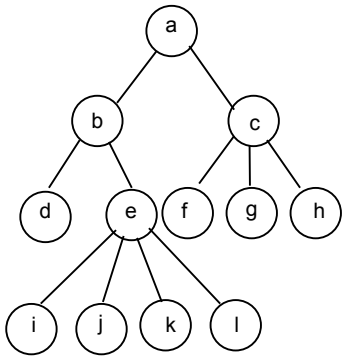
b) El resto de los vértices (excluyendo la raíz) están particionados en $m \geq 0$ subconjuntos (disjuntos) $V_1, V_2, \ldots, V_m$ tales que los subgrafos inducidos por cada $V_i$, $i = 1, \ldots, m$ son a su vez árboles enraizados. Estos subgrafos inducidos se llaman *sub-árboles* de **r** , y se denotan $A_1, A_2, \ldots, A_m$. La raíz de cada sub-árbol $A_i$ es el único vértice en $A_i$ que es adyacente a **r** en A, es decir, el único hijo de **r** en $A_i$. El conjunto de lados E es la unión disjunta de los lados en los sub-árboles $A_1, \ldots, A_m$ más los lados incidentes en **r**.

Esta nueva definición excluye la existencia de ciclos desde el momento que vemos que no hay lados entre sub-árboles de un mismo vértice.

Trabajaremos ahora con los niveles de los vértices en un árbol. Como se definió en la Sec. V.5, el nivel $\eta(x)$ de un vértice x en un grafo es el largo máximo de un camino elemental terminado en x. Si A es un árbol enraizado, los caminos más largos parten de la raíz, por lo que la raíz tiene nivel 0, y todos los demás vértices tienen nivel igual al de su padre más 1, o lo que es lo mismo, para todo vértice no terminal con nivel i, el nivel de sus hijos es i+1.

Una forma de obtener los niveles de los vértices en un árbol A es utilizando el algoritmo de búsqueda en amplitud BFS, comenzando por la raíz **r**, a la cual se le asigna nivel(r) = 0, y asignando nivel(x) + 1 a todos los vértices visitados desde un vértice x. De esta misma manera obtenemos la longitud o largo del camino (o cadena) desde la raíz hasta cada vértice, que será igual a su nivel. Definimos la *Altura de un vértice x* como la longitud del camino más largo desde x hasta

una hoja. Se define la *Altura de un Árbol* como la longitud del camino más largo en el árbol, es decir, la altura de la raíz, o lo que es lo mismo, el máximo de los niveles de los vértices del árbol.




Árbol enraizado con raíz a

Hojas: d, f, g, h, i, j, k, l.
Nodos interiores: a, b, c, e.
Raíces de los sub-árboles de a: b, c.
Nivel del nodo f: 2
Altura del nodo b: 2
Altura del árbol: 3

**figura VI.6**

Consideremos a continuación un árbol A de altura **a** en el cual cada vértice puede tener a lo sumo **g** hijos. Se denota $N_a(g)$ al número máximo de vértices que puede tener un árbol con las características anteriores. Este número $N_a(g)$ se puede determinar a partir del número máximo de vértices que puede haber en cada nivel del árbol, así:

Nivel 0: 1 vértice
Nivel 1: máximo g vértices
Nivel 2: máximo $g.g = g^2$ vértices
…
Nivel a: máximo $g^{a-1}.g = g^a$ vértices

de donde $N_a(g) = \sum_{i=0}^{a} g^i = (g^{a+1}-1)/(g-1)$

Un árbol en el cual todas las hojas están en el máximo nivel y todos los vértices interiores tienen el máximo número de hijos se llama un *Árbol completo*.

Es de especial interés el caso en el que g=2, donde $N_a(g) = \sum_{i=0}^{a} 2^i = (2^{a+1}-1)$

Podemos observar que en un árbol A con número máximo de hijos g=2 y k hojas debe tener un camino de longitud al menos $\log_2(k)$. Aplicando un razonamiento similar al anterior también se puede determinar que la altura mínima que puede tener un árbol de n vértices, donde cada uno puede tener a lo sumo 2 hijos es $[\log_2 n]$.

Estos árboles con g=2 han sido estudiados y utilizados ampliamente en computación, en especial los llamados "árboles binarios" que veremos más adelante, debido a las enormes facilidades que ofrecen para el manejo de información. Así mismo, los resultados anteriores han facilitado la obtención de muchos otros en el estudio de la complejidad de algoritmos que trabajan con árboles binarios.

### VI.5 ÁRBOLES Y ARBORESCENCIAS PLANAS

Existe una clase de grafos que posee una importante propiedad: se puede dibujarlos sobre un plano ( papel, por ejemplo) de forma tal que los vértices sean puntos en el plano y dos lados cualesquiera no tengan más intersecciones que los

vértices que puedan tener como extremos comunes. A estos grafos se les llama *Grafos Planares* . Es evidente que no todos los grafos son planares, como por ejemplo no lo es $K_5$, ni el grafo de la figura VI.7

Un mismo grafo puede tener más de una representación planar, por ejemplo las dos representaciones planares de $K_4$ que se muestran en la figura VI.8.

Los árboles son grafos planares debido a la existencia de una única cadena entre cualquier par de vértices.

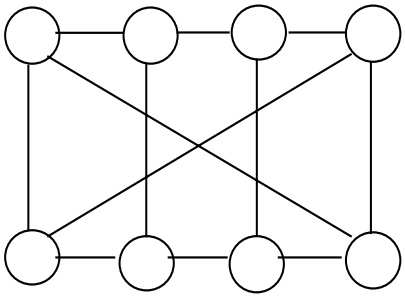
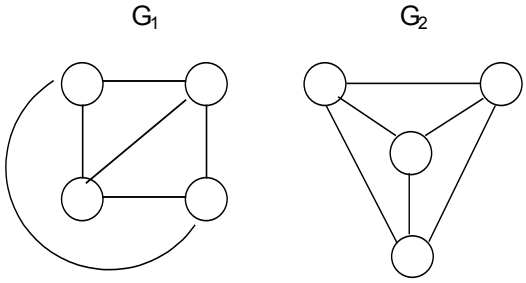


**figura VI.7**

Dado un árbol A, se puede distinguir diferentes representaciones planares de A: $A_1$, $A_2$,… , como se muestra en la figura VI.9 para el árbol A con V={a,b,c,d,e} y E={{a,b}, {a,c}, {c,d}, {c,e}}.

Lo dicho anteriormente para árboles en general, obviamente aplica para árboles enraizados y arborescencias.




Dos representaciones planares del mismo grafo

**figura VI.8**

### VI.5.1 ÁRBOLES ORDENADOS

Si observamos ahora los árboles $A_3$ y $A_4$ de la figura VI.9, y distinguimos en ellos el vértice c como raíz, podemos diferenciar uno del otro según las posiciones relativas de sus sub-árboles en cada una de las representaciones planares y podemos entonces numerarlas de izquierda a derecha, de forma que el 1º sub-árbol de c en $A_3$ es distinto del 1º sub-árbol de c en $A_4$.

**Proposición VI.5.1.1**

Toda representación planar de un árbol enraizado ( o arborescencia ) puede ser caracterizada asignando a cada vértice no terminal una lista ordenada o permutación de sus hijos (raíces de sus sub-árboles).

Para los árboles $A_3$ y $A_4$ de la figura VI.9 tenemos:

$A_3$: P(c) = <a, d, e>, P(a) = <b>

$A_4$: P(c) = <d, e, a>, P(a) = <b>

**Definición VI.5.1.1**

Llamamos *Árbol Ordenado* un árbol enraizado donde cada vértice no terminal tiene asociada una permutación de sus hijos.

Si nos remitimos a la definición VI.4.1 de árboles enraizados, vemos que un árbol ordenado es uno en el que es relevante el orden en que aparecen listados los sub-árboles en tal definición.

En este sentido, los árboles A3 y A4 con las permutaciones dadas arriba son dos árboles ordenados diferentes y sin embargo, si los consideramos simplemente como árboles enraizados, son el mismo.

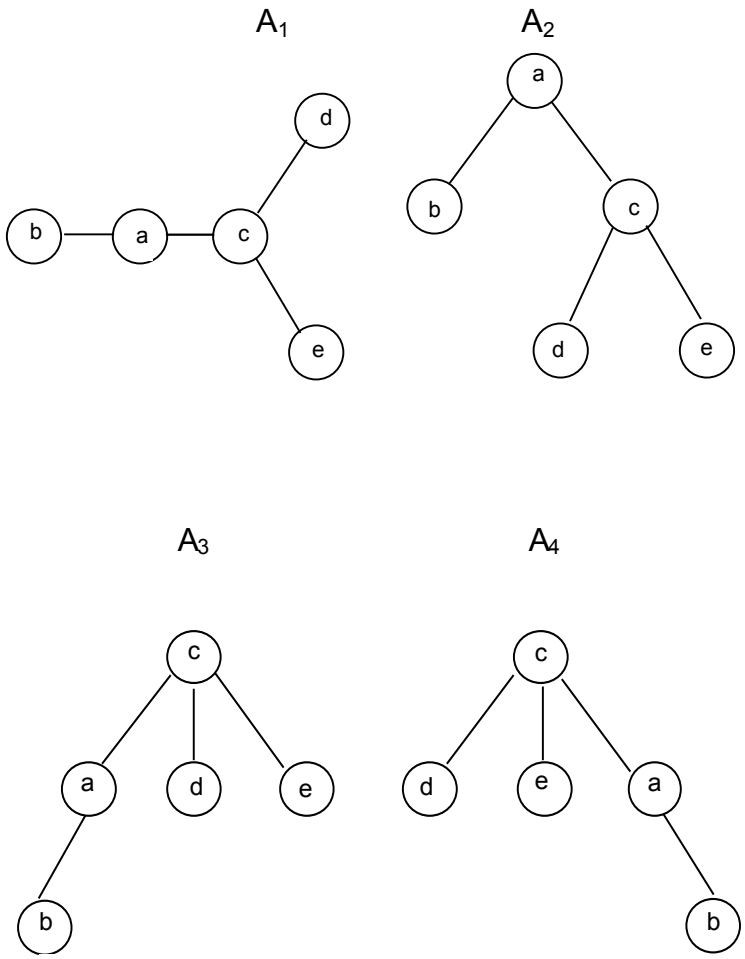



**figura VI.9**

Vemos entonces que la representación planar de árboles nos lleva de forma natural a los árboles ordenados. Pero no es sólo esto, la utilización de árboles en computación y en consecuencia, la necesidad de representarlos en el computador impone igualmemte la necesidad de ordenar los hijos, debido a la organización secuencial de la memoria y a la ejecución secuencial de las instrucciones.

### VI.5.1.1 REPRESENTACIÓN DE EXPRESIONES ALGEBRAICAS

Como ejemplo de utilización de árboles ordenados, consideremos la siguiente expresión algebraica:

(1) $((a + b) * 5) / (9 - (f * \log h)$

En esta expresión aparecen operadores de uno y de dos operandos, y en el caso de los segundos, en particular de los operadores (/) y (–), es importante el orden de los operandos.

Una expresión como ésta puede ser fácilmente representada haciendo uso de árboles ordenados, donde las hojas son los operandos y los operadores están en los vértices internos y en la raíz. El orden de los sub-árboles corresponde al orden de los operandos. Tal representación es la que se muestra en la figura VI.10 para la expresión (1).

### VI.5.2 ÁRBOLES BINARIOS

Como ya hemos mencionado antes, existe un tipo de árbol al que se ha dedicado mucho estudio debido a su enorme utilización en computación, son estos los árboles binarios.

**Definición VI.5.2.1**

Un *Árbol Binario* es un árbol enraizado en el cual cada vértice tiene a lo sumo dos hijos, los cuales se denominan *hijo izquierdo* e *hijo derecho* . De igual manera, los sub-árboles de cada vértice se denominan *sub-árbol izquierdo* y *sub-árbol derecho* respectivamente.

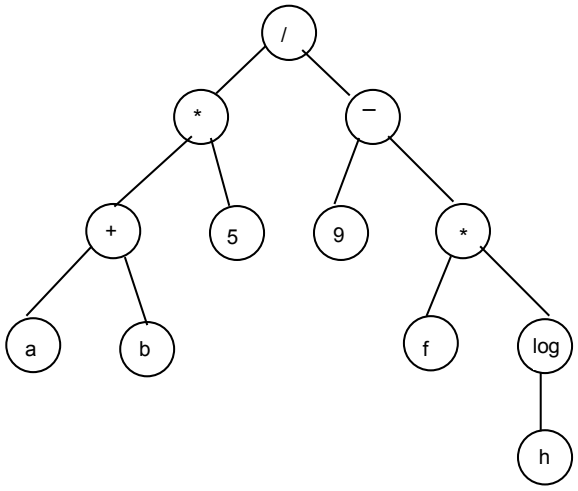



**figura VI.10**

Es importante notar que un árbol binario no es lo mismo que un árbol ordenado con a lo sumo dos hijos por vértice, pues en este último, si un vértice tiene un solo hijo, éste es simplemente el primero (y único); en el caso de un árbol binario se debe especificar si este hijo es el derecho o el izquierdo, lo cual diferencia un árbol de otro. Los árboles $A_1$, $A_2$ y $A_3$ de la figura VI.11 son iguales si los vemos como simples árboles enraizados, sin embargo, como árboles ordenados $A_1$ es distinto de $A_2$ y $A_3$ que son iguales, y como árboles binarios los tres son distintos.

Los árboles binarios han sido utilizados como estructuras para almacenamiento, ordenamiento y recuperación de información, así como para la representación y solución de problemas de toma de decisiones, entre otros.

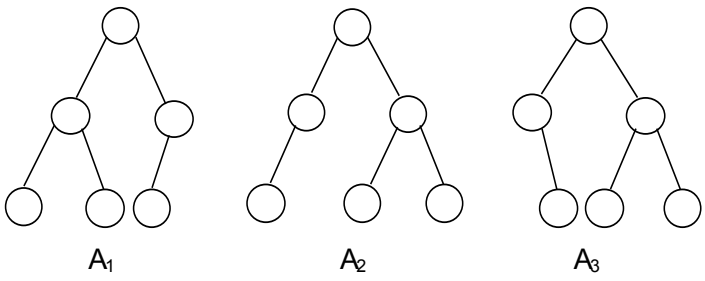


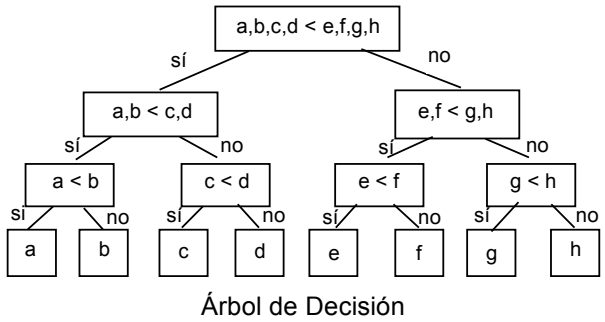

**figura VI.11**

### VI.5.2.1 ÁRBOLES DE DECISIÓN

Son muchos los casos en los cuales la resolución de un determinado problema se facilita si se logra subdividirlo adecuadamente en problemas menores y más sencillos (estrategia conocida como "Divide-and-Conquer").

Consideremos un pequeño ejemplo. Se tiene 8 monedas {a, b, c, d, e, f, g, h} de las cuales una es falsa y de menor peso que las demás. Si disponemos de una balanza podemos determinar cuál es la moneda falsa de la siguiente manera. Se coloca la mitad de las monedas de un lado de la balanza y la otra mitad del otro lado. Del lado que resulte menos pesado sabremos que está la moneda falsa. Tomamos las monedas de este lado y nuevamente las repartimos equitativamente a ambos lados de la balanza. Una de las dos monedas del lado más liviano es la falsa, luego, volviéndolas a pesar, tendremos la respuesta a nuestro problema.

El árbol de la figura VI.12 es una representación del método empleado para resolver el problema. En este árbol cada vértice interno representa una decisión o comparación y dependiendo del resultado de ésta se puede pasar al hijo izquierdo o al derecho sucesivamente hasta llegar a las hojas que representan cada una de las respuestas posibles al problema original.



Árbol de Decisión

**figura VI.12**

Este tipo de árboles se conoce con el nombre de *Árbol de Decisión*.

Otro ejemplo frecuente de uso de este tipo de árboles es en el ordenamiento de n números distintos. Puesto que cada una de las n! permutaciones de estos números podría ser el ordenamiento correcto, el árbol de decisión asociado al problema, en el cual cada vértice representa los ordenamientos posibles luego de las comparaciones realizadas para llegar a él y los lados el resultado de la comparación entre dos claves, deberá tener n! hojas, si se excluyen comparaciones innecesarias o extemporáneas.

A partir del resultado visto en VI.4 podemos concluir que para llegar a un ordenamiento son necesarias como mínimo log(n!) comparaciones, lo cual es de orden nlogn [ AHO et al., " Data Structutres and Algorithms", pp 284-286]. Este resultado es básico en el análisis de los diferentes métodos de ordenamiento por comparación, pues establece una cota inferior para el mínimo de comparaciones que tendrá que realizar cualquiera de estos algoritmos.

### VI.5.2.2 ÁRBOLES BINARIOS PERFECTAMENTE BALANCEADOS

Definimos un *Árbol binario Perfectamente Balanceado* o *Equibrado* como un árbol binario en el cual para cada vértice, el número de vértices en su sub-árbol izquierdo y el número de vértices en su sub-árbol derecho difieren a lo sumo en una unidad.

Un árbol perfectamente balanceado con n vértices será un árbol de áltura mínima en el sentido que no puede existir otro árbol binario con el mísmo número de vértices y de altura menor. En un árbol de este tipo todas las hojas estarán en los dos últimos niveles, es decir, si a es la altura del árbol, entonces las hojas estarán todas en los niveles a y a–1.

La construcción de un árbol perfectamente balanceado con n vértices se puede expresar recursivamente de la siguiente manera:

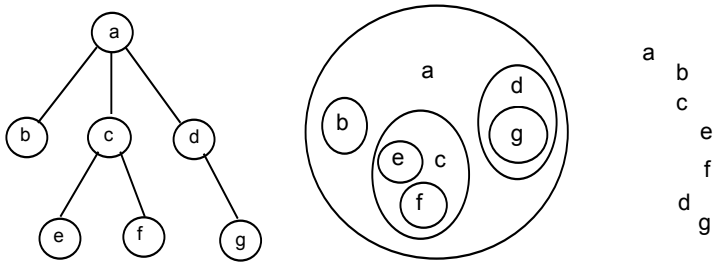
Algoritmo de construcción de un árbol Perfectamente Balanceado:
{Entrada: n vértices.
Salida: el árbol perfectamente balanceado con los n vértices }.
Comienzo
Si n>o entonces:

1) Tomar un vértice como raíz.
2) Construir el sub-árbol izquierdo con $n_i = n$ **div** 2 vértices, siguiendo estos mismos pasos.
3) Construir el sub-árbol derecho con $n_d = n - n_i - 1$ vértices, siguiendo estos mismos pasos.

Fin.

### VI.6 REPRESENTACIÓN DE ÁRBOLES Y ARBORESCENCIAS

Como grafos que son, tanto los árboles como las arborescencias admiten cualquiera de las representaciones para grafos presentadas en el capítulo II. Sin embargo, por sus características especiales, es interesante estudiar nuevas formas de representación que se adaptan mejor a estas estructuras, sobre todo para los árboles ordenados en los cuales se conoce el número máximo de hijos por vértice y para árboles binarios.




Distintas representaciones gráficas para un mismo árbol

**figura VI.13**

En cuanto a la representación gráfica, ya hemos venido utilizando la convención de colocar la raíz en el tope para los árboles enraizados y arborescencias. Sin embargo, los árboles enraizados por su estructura particular son suceptibles de ser representados gráficamente de varias otras formas que nada tienen que ver con la usual representación de grafos, como se muestra en la figura VI.13. Más aún, el método usual utilizado para numerar capítulos y secciones de capítulos en los libros, el mismo que usamos en este libro, es otra manera de representar árboles como se muestra en la figura VI.14.

1: a
- 1.1: b
- 1.2: c
  - 1.2.1: e
  - 1.2.2: f
- 1.3: d
  - 1.3.1: g

Numeración como índice de los vértices del árbol de VI.13

**figura VI.14**

Dedicaremos ahora nuestra atención a nuevas formas de representación en el computador.

#### VI.6.1 REPRESENTACIÓN UTILIZANDO LISTAS

##### VI.6.1.1 REPRESENTACIÓN HOMOGÉNEA ASCENDENTE

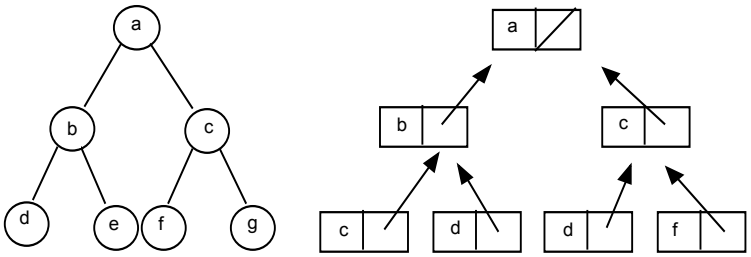
Este tipo de representación para arborescencias, extensible a árboles enraizados, se basa en el hecho de que para todo nodo x que no sea la raíz: $d^-(x) = 1$. De esta manera podemos representar cada nodo como un registro con dos campos: uno para el nombre o etiqueta del nodo y el otro para almacenar un apuntador a su nodo antecesor o padre. Evidentemente, el apuntador al padre de la raíz es NULO.

La declaración para esta estructura sería:

```
tipo  apun_ha = apuntador a  nodo_ha;
tipo  nodo_ha = Registro
                    valor:T;
                    padre: apun_ha
                Fin
```

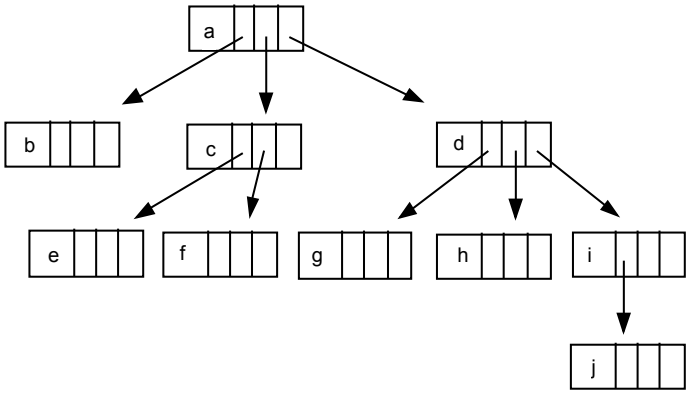
Se le llama homogénea a este tipo de representación por utilizar un único tipo de registro para representar toda la estructura. Una ventaja de esta representación es que es independiente del número de hijos que puedan tener los nodos.

Por otro lado, una desventaja es que esta representación no puede reflejar ningún tipo de orden entre los hijos o sub-árboles de un nodo, por lo que es necesario descartarla cuando el orden es propiedad relevante del árbol que se quiere representar.




Representación homogénea ascendente

**figura VI.15**




Representación homogénea descendente de un árbol
con máximo 3 hijos por nodo

**figura VI.16**

### VI.6.1.2 REPRESENTACIÓN HOMOGÉNEA DESCENDENTE

Podemos mencionar dos tipos de representaciones homogéneas descendentes, es decir, que representan la relación de un elemento o nodo hacia sus descendientes.

La primera de estas representaciones requiere de un conocimiento previo del número máximo de hijos por nodo. Se puede entonces representar todos los nodos del árbol utilizando una misma estructura base que consiste en un registro con un campo de información para almacenar el valor del nodo, y tantos campos de tipo apuntador como el máximo de hijos que puedan tener los nodos en el árbol a representar.

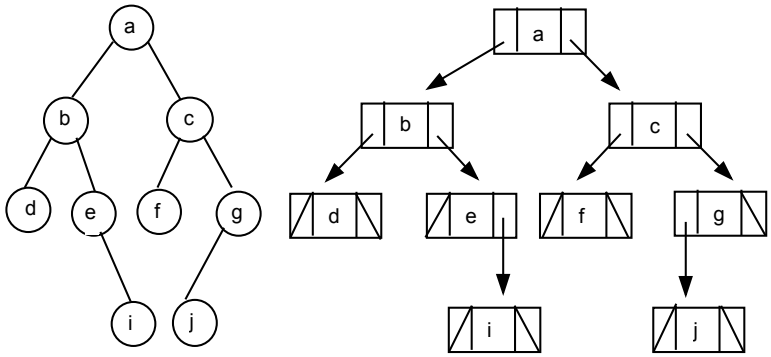
Esta representación permite tomar en consideración el orden relativo entre los hijos si ello fuese necesario. La mayor desventaja que presenta es la cantidad de memoria desperdiciada en apuntadores en todos aquellos nodos con menos hijos que el máximo permitido, en particular en las hojas. Vale la pena sin embargo mencionar que en el caso de árboles con a lo sumo 2 hijos por nodo esta desventaja es mínima, pues pensar en menos campos apuntadores, correspondería ya a las listas lineales. Por esta última razón es esta representación la usualmente seleccionada para este tipo de árboles.

En el caso particular de los árboles binarios, esta es la representación estándar y a los dos apuntadores a los hijos se les distingue como apuntador al hijo izquierdo y apuntador al hijo derecho (ver figura VI.17).

La declaración para estas estructuras sería:

```
tipo  apun_n = apuntador a  árbol_n;
tipo  árbol_n = Registro
                  valor:T;
                  apuntadores: Arreglo [1.. n] de  apun_n
                fin;

tipo  apun2 = apuntador a  binario;
tipo binario = Registro
                  valor:T;
                  izquierdo, derecho: apun2
                fin;
```




Representación homogénea descendente
de árboles binarios

**figura VI.17**

La segunda de las representaciones homogéneas descendentes que presentaremos, no requiere del conocimiento previo del número máximo de hijos por nodo. Este tipo de representación utiliza un único tipo de registro de tres campos: uno para el valor del nodo, y dos campos de tipo apuntador, uno con la dirección del primero de una lista de los hijos del nodo, y el segundo con la dirección del siguiente hermano del nodo dentro de la lista de hijos de su padre.

La declaración para esta estructura sería:

```
tipo  apun = apuntador a  nodo;
tipo  nodo = Registro
                valor:T;
                hijos, sig_hermano: apun
              fin;
```

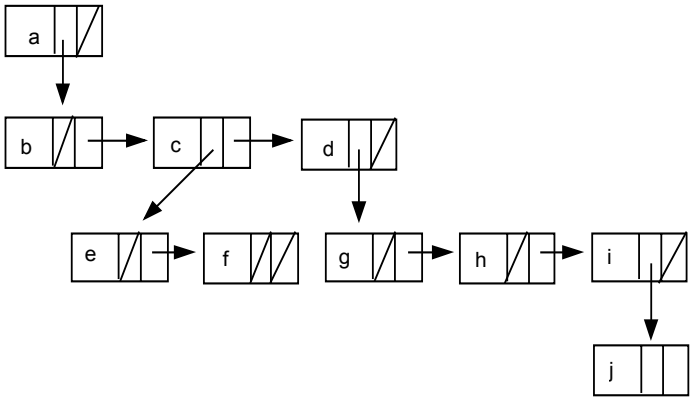
La figura VI.18 muestra la representación del mismo árbol de la figura VI.16 usando esta nueva estructura.

Este último tipo de representación se conoce como *representación de árboles usando árboles binarios.* Es fácil comprender el por qué, basta con cambiar los nombres de los campos "hijos" y "sig_hermano" por "izquierdo" y "derecho" respectivamente (ver figura VI.19).

### VI.6.2 REPRESENTACIÓN USANDO ARREGLOS

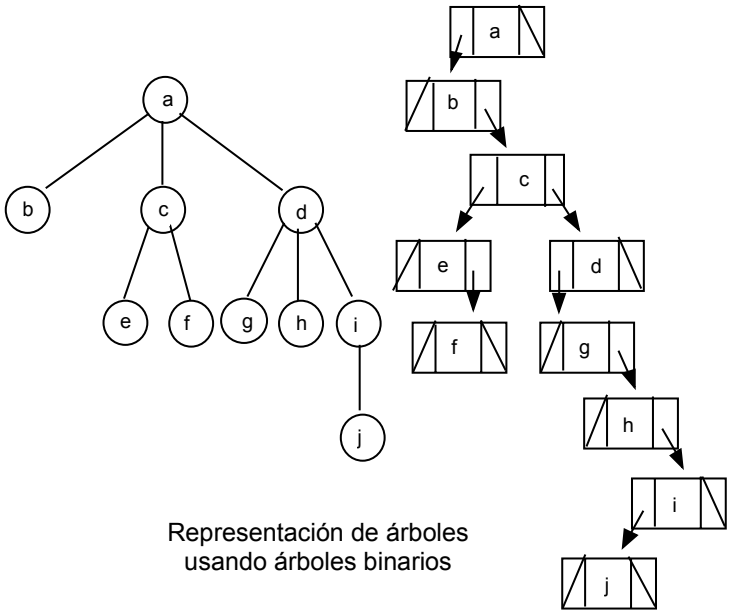
La primera de las representaciones usando arreglos que mencionaremos es equivalente a la homogénea ascendente usando listas. Utiliza dos arreglos, uno para el valor del nodo y otro para guardar las posición de su nodo padre. De esta manera podemos almacenar árboles con cualquier número de hijos. Como desventajas presenta las mismas de la representación equivalente usando listas, además de las ya conocidas asociadas a la declaración estática del tipo arreglo.

Limitaremos las siguientes representaciones con arreglos a árboles con no más de 2 hijos por nodo, en consideración a la cantidad de memoria, que, en otros casos, quedaría frecuentemente sin uso y desperdiciada.




Representación homogénea descendente del árbol de la figura VI.16

**figura VI.18**



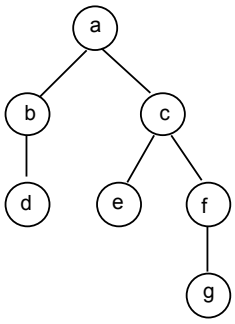

Representación de árboles usando árboles binarios

**figura VI.19**

La primera de estas representaciones es muy similar a la primera de las representaciones homogéneas descendentes usando listas, en el sentido que utiliza tres arreglos, uno para el valor de cada nodo y los otros dos para almacenar las posiciones de cada uno de los hijos (ver figura VI.20).

La otra representación a la que haremos mención usa un único arreglo unidimensional A[1.. N] en el cual , para cada nodo en posición A[i] sus hijos estarán en las posiciones A[2∗i] y A[2∗i +1], o lo que es lo mismo, para cada nodo en una

posición i, con i≥N/2, su padre será el nodo en la posición i **div** 2. Este tipo de representación, aún pareciendo económica, sólo es recomendable en el caso de árboles completos o de altura mínima, en los cuales las hojas estarán todas en los dos últimos niveles. La última posición no vacía en el arreglo corresponderá a la hoja más a la derecha en el último nivel del árbol.




| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| a | b | c | d | e | f | g |
| 2 | 4 | 5 | 0 | 0 | 7 | 0 |
| 3 | 0 | 6 | 0 | 0 | 0 | 0 |

Representación usando Arreglos

**figura VI.20**

### VI.7 RECORRIDOS DE ÁRBOLES

Al utilizar árboles como modelos para representar situaciones o simplemente como una forma de organizar información, a menudo se hace necesario disponer de un método sistemático para recorrer o visitar todos los nodos del árbol con el fin de realizar alguna operación.

Para poder organizar y explicar diferentes recorridos se hace necesario suponer un orden entre los elementos del árbol. En otras palabras, es necesario dotar a cada nodo de un ordenamiento de sus sub-árboles, aún cuando la estructura original no posea dicha organización. Igualmante, visto que todo árbol puede ser considerado como un bosque formado por un único árbol, podemos explicar los tres principales tipos de recorridos, conocidos como *Pre-orden*, *In-orden* u *orden central* y *Post-orden*, referidos a bosques, con lo cual disponemos de métodos más generales y de mayor alcance.

Estos recorridos se pueden expresar recursivamente como sigue:

Pre-Orden:
{ Entrada: Un Bosque B formado por los árboles $A_1$, …, $A_m$. Las raíces $r_1$, …, $r_m$ de cada uno de los árboles.}
Comienzo
(1) Visitar la raíz del primer árbol del bosque B.
(2) Recorrer en Pre-Orden el bosque formado por los sub- árboles de la raíz del primer árbol, si los hay.
(3) Recorrer en Pre-Orden el bosque formado por los árboles restantes, respetando el orden entre ellos.
Fin.

In-Orden u Orden Central:
{ Entrada: Un Bosque B formado por los árboles $A_1$, …, $A_m$. Las raíces $r_1$, …, $r_m$ de cada uno de los árboles.}
Comienzo
(1) Recorrer en In-Orden el bosque formado por los sub-árboles de la raíz del primer árbol, si los hay.
(2) Visitar la raíz del primer árbol del bosque B.
(3) Recorrer en In-Orden el bosque formado por los árboles restantes, respetando el orden entre ellos.
Fin.

Post-orden:
{ Entrada: Un Bosque B formado por los árboles $A_1$, …, $A_m$. Las raíces $r_1$, …, $r_m$ de cada uno de los árboles.}
Comienzo
(1) Recorrer en Post-orden el bosque formado por los sub-árboles de la raíz del primer árbol, si los hay.
(2) Recorrer en Post-orden el bosque formado por el resto de los árboles, respetando el orden entre ellos.
(3) Visitar la raíz del primer árbol del bosque B.
Fin.

Debido a la importancia de los árboles binarios, y a la singular organización de sus sub-árboles, distinguidos como izquierdo y/o derecho, y no primero y segundo, lo cual los distingue de los árboles ordenados, se hace necesario especificar cómo se adaptan los recorridos anteriores a los árboles binarios.

Cuando en un árbol binario no existen sub-árboles, es decir, si un nodo no tiene hijo izquierdo ni derecho entonces "recorrer el sub-árbol" significa "no hacer nada". En caso contrario los recorridos se harán siguiendo los pasos indicados a continuación, según el caso:

Pre-Orden:
{Entrada: Un árbol binario A, el nodo r, raíz de A.}
Comienzo
(1) Visitar la raíz.
(2) Recorrer el sub-árbol izquierdo en Pre-orden.
(3) Recorer el sub-árbol derecho en Pre-orden.
Fin.

In-Orden u Orden Central:
{Entrada: Un árbol binario A, el nodo r, raíz de A.}
Comienzo
(1) Recorrer el sub-árbol izquierdo en In-orden.
(2) Visitar la raíz.
(3) Recorrer el sub-árbol derecho en In-orden.
Fin.

Post-orden:
{Entrada: Un árbol binario A, el nodo r, raíz de A.}
Comienzo
(1) Recorrer el sub-árbol izquierdo en Post-orden.
(2) Recorrer el sub-árbol derecho en Post-orden.
(3) Visitar la raíz.
Fin.

Estos recorridos son de especial interés pues pueden ser aplicados a árboles en general, ya que todos pueden ser representados como árboles binarios.

Es interesante notar que los recorridos del bosque o árbol original y del árbol binario asociado se corresponden uno a uno.

Recorridos del bosque de VI.21(a):

| | |
|---|---|
| Pre-Orden: | a b c d e f g h i j k l. |
| In-Orden: | b d e c f a h j k l i g. |
| Post-orden: | e d f c b l k j i h g a. |

Recorridos del árbol binario de VI.21(b)

| | |
|---|---|
| Pre-Orden: | a b c d e f g h i j k l. |
| In-Orden: | b d e c f a h j k l i g. |
| Post-orden: | e d f c b l k j i h g a. |

### VI.8 ÁRBOLES DE JUEGO

Consideremos algún juego como por ejemplo damas, ajedrez o la "vieja", en el cual intervienen dos jugadores. En juegos como estos, tiene cierta ventaja el jugador con capacidad de prever más jugadas. Suponiendo que el juego tiene un número finito de posibles jugadas y que existe alguna regla que asegura que el juego eventualmente termina, las secuencias de jugadas posibles a partir de una situación o estado inicial, pueden ser representadas utilizando un árbol enraizado o arborescencia llamado *Árbol de Juego* .

La raíz de este árbol representa el estado inicial del juego; los lados incidentes en la raíz representan las posibles jugadas o movimientos del Primer jugador, y los nodos del primer nivel corresponden a los posibles estados del juego luego de cada uno de estos movimientos; los lados entre los nodos del primer nivel y los del segundo representan las jugadas que puede realizar el Segundo jugador según el estado o situación en que encuentre el juego, y así sucesivamente. Los lados que se encuentran entre nodos de un nivel par y los del siguiente corresponderán a jugadas del Primer jugador, y los lados entre nodos de un nivel impar y el siguiente, corresponderán a jugadas del Segundo jugador. Las hojas del árbol representan estados o situaciones a partir de los cuales el juego no puede continuar, bien porque alguno de los jugadores gana, o porque termina el juego con un empate.

El árbol de la figura VI.22 corresponde al juego de NIM en el cual, a partir de un número inicial k de palitos, cada jugador va tomando x palitos, con $1 \leq x \leq 3$, siempre que el número kr de palitos restantes sea mayor que 3, y $1 \leq x \leq kr$ en caso contrario. El jugador que toma el último palito pierde. A cada nodo se asocia un número, igual al número de palitos restantes en ese estado del juego.

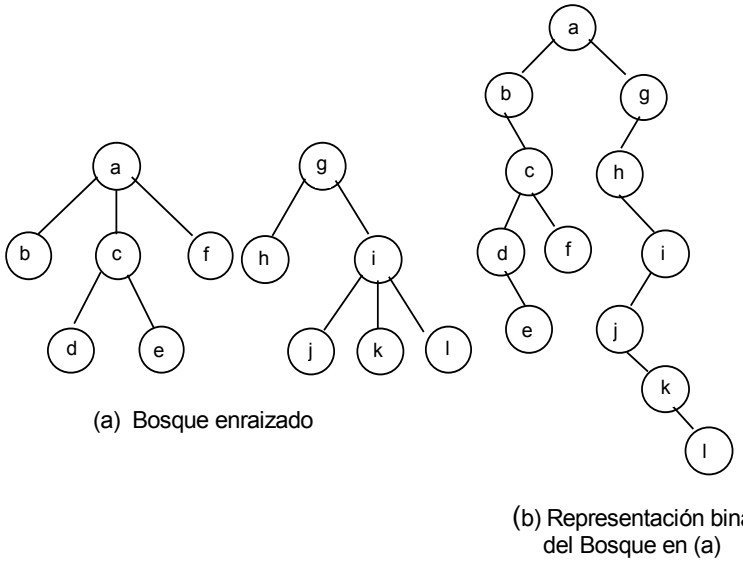



**figura VI.21**

A cada nodo del árbol se le asocia además un valor, comenzando por las hojas. Estos valores serán: 1, -1 y 0, según que la hoja represente un juego ganado, perdido o empatado para el Primer jugador. Los valores se propagarán luego hacia el resto de los nodos de acuerdo a una estrategia conocida como *min-max*. Note que mientras que el Primer jugador busca ganar, el Segundo busca que el Primero pierda. Esto último en términos de los valores asociados a los nodos significa que el primer jugador hará movimientos que le conduzcan a estados con el máximo valor posible, mientras que el Segundo hará todo lo contrario.

El método *min-max* consiste en asignar a cada nodo no terminal el máximo de los valores de sus hijos, cuando los lados entre ellos corresponden a jugadas del Primer jugador (nodos en nivel par), y el mínimo cuando corresponden a jugadas del Segundo. Esto es fácil de lograr hacieno un recorrido del árbol en Post-orden.

Un jugador tiene una estrategia de juego ganadora, si a partir de su primer turno puede siempre hacer una jugada que le garantice el triunfo, independiente de las jugadas de su adversario. En términos de los valores asociados a los nodos, esto se expresa como sigue: si la raíz tiene valor 1, el Primer jugador tiene una estrategia ganadora; si tiene valor -1, el Segundo jugador tiene una estrategia ganadora; si tiene valor 0, lo mejor que puede hacer el Primer jugador es lograr un empate.

En el árbol de juego de la figura VI.22, el valor 1 corresponde a las hojas en niveles pares y -1 a las hojas en niveles impares (en este juego siempre hay un ganador, por lo que no se asigna el valor 0 a ninguna hoja). Propagando los valores hasta la raíz de acuerdo al método min-max, podemos ver que el juego tiene una estrategia ganadora para el segundo jugador. ¿Qué ocurriría si el juego se iniciara con 6 palitos? ¿Y con 4?

Note sin embargo, que el árbol correspondiente a este juego puede llegar a ser inmensamente grande a medida que aumenta el número de palitos con que se da inicio al juego, es decir, el número en la raíz. En la práctica, incluso juegos

sencillos como la "vieja", suelen dar lugar a árboles sumamente grandes. Esta situación obviamente se repite para la mayoría de los juegos de verdadero interés, en los cuales a veces incluso utilizando un computador es imposible examinar en un tiempo "prudencial" todas las ramas posibles.

La idea de árboles de juego que acabamos de explicar, donde los nodos toman valores 1, -1 y 0, se puede generalizar a árboles en los cuales se deben asignar otros valores a los nodos. Estos valores, que llamaremos *beneficio*, representan un estimado de la probabilidad de que una determinada jugada resulte en un triunfo para el primer jugador. Esta generalización es útil para árboles grandes, como los que mencionamos antes, los cuales son imposibles de examinar en su totalidad.

La mayoría de los juegos que se juegan con el computador generan árboles muy grandes, por lo que en base a lo dicho arriba, lo que se hace es simular el juego hasta un número predefinido, llamado *paso*, de movimientos sub-siguientes.

En general, ocurrirá que las hojas de un árbol de juego, dado un paso pre-fijado, son ambiguas, ya que al no representar un juego terminado, no corresponden a un triunfo, ni a un empate, ni a un juego perdido. Para poder asignar valores a los nodos se deberá disponer entonces de una función, que dependerá del juego, que evaluará el beneficio que puede aportar una determinada jugada al primer jugador. De esta manera, nuevamente aplicando el método min-max, se podrán propagar los valores hasta la raíz. Estas funciones por ser estimaciones, no necesariamente garantizan un triunfo o empate. La figura VI.23 muestra un árbol para el juego de la "vieja" con un paso de 2, con la primera jugada de las X ya realizada (por lo tanto a efectos del árbol, el Primer jugador son las O). La función que se utiliza es la siguiente: número de filas, columnas y diagonales que permanecen abiertas para el primer jugador (las O), menos las que permanecen abiertas para el contrario.

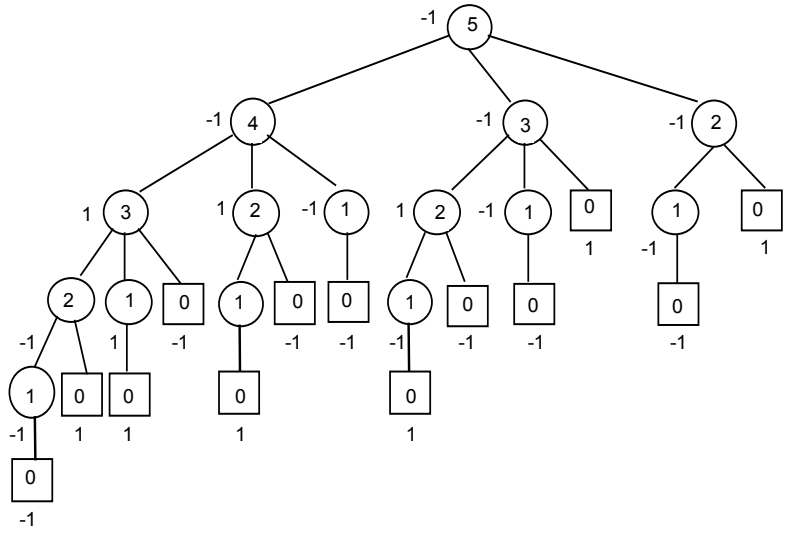


**figura VI.22**

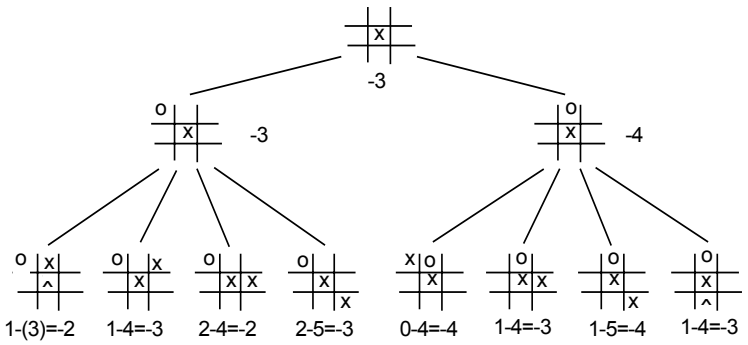



**figura VI.23**

Observe que en este ejemplo, si el jugador X se encuentra ante la situación de la derecha en el árbol, la función le indica que escoja una de las jugadas con valor -4 y sin embargo, la segunda jugada posible que se le presenta, con un valor de -3, es una jugada con una estrategia ganadora para X. Mientras mayor sea el paso, mejor será la información que proporcionan los valores de los nodos, pero para la escogencia del paso se debe tomar en cuenta la capacidad y velocidad de la computadora, lo que constituye un fuerte factor limitante.

Afortunadamente existe un método que se utiliza en combinación con el min-max y que permite escoger un mayor paso, al permitir descartar de antemano algunas ramas. Este método es conocido como *Poda Alfa-Beta* y se puede utilizar siempre que el paso sea mayor o igual a 2. Para ver mejor cómo funciona lo ilustraremos con el ejemplo de la figura VI.24

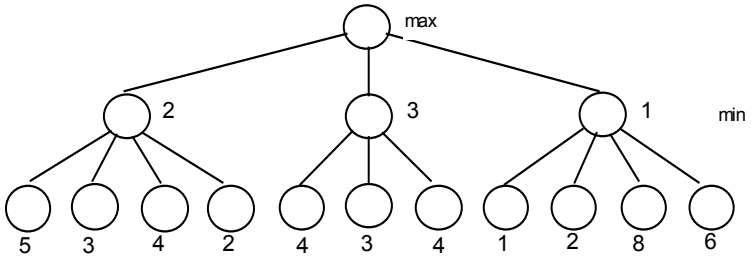



**figura VI.24**

Supongamos que el Primer jugador analiza las posibles respuestas de su contrincante de izquierda a derecha. Luego de haber examinado las respuestas a sus dos posibles movimientos, al analizar el tercero encuentra que su contrincante puede lograr que solo obtenga un beneficio de 1, con esto sabe que esa jugada, la tercera, no le conviene y no necesita continuar examinando las otras posibles respuestas de su adversario a esa jugada. Se dice que las ramas que no van a ser analizadas son *podadas* del árbol.

Este método se puede acelerar más aún,es decir, recortar más ramas, si se puede disponer de algún criterio que permita de alguna manera distinguir mejores de peores jugadas. Si se analizan las posibles jugadas en base a un tal criterio, es posible que resulten podadas un número mayor de ramas. Por esta razón, conviene tratar los árboles de juego como árboles ordenados.

La mayoría de los juegos por computadora utilizan este método de poda y además van evaluando el árbol a medida que se va generando. De esta forma, las ramas podadas en realidad nunca llegan a existir.

### VI.9 EJERCICIOS

**1.** Hallar todas las colecciones maximales de ciclos independientes del grafo siguiente:

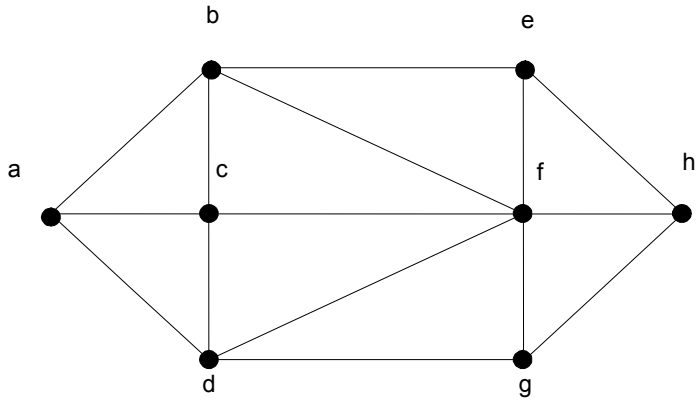



**figura VI.25**

**2.** Sea $G=(V,E)$ un grafo no orientado. Demostrar:

a) Si $G$ es conexo entonces $G$ tiene al menos $|E| - |V| + 1$ ciclos distintos.

b) Escriba un algoritmo que encuentre $|E| - |V| + 1$ ciclos distintos de $G$, si G es conexo.

c) Si $G$ tiene $p$ componentes conexas, entonces $G$ tiene al menos $|E| - |V| + p$ ciclos distintos.

**3.** Utilizar el algoritmo de Kruskal para construir un árbol mínimo cobertor del grafo $G=(V,E)$ respecto a la función de costos $c$.

| **Lado** | **Costo** | **Lado** | **Costo** |
|---|---|---|---|
| {a, b} | 2 | {e, f} | 8 |
| {a, c} | 1 | {e, g} | 7 |
| {a, d} | 8 | {e, h} | 1 |
| {b, d} | 6 | {f, h} | 8 |
| {c, d} | 4 | {g, h} | 5 |
| {c, e} | 7 | {e, i} | 7 |
| {c, f} | 3 | {g, j} | 8 |
| {d, f} | 2 | {h, j} | 4 |
| {d, f} | 8 | {i, j} | 8 |

**4.** ¿Cómo modificaría el algoritmo de Kruskal para hallar el árbol cobertor de costo máximo?

**5.** Pruebe que la siguiente afirmación es falsa: Si $G$ es un grafo con una función de costos $c:E\rightarrow\aleph$ tal que $G$ tiene un único árbol mínimo cobertor, entonces $G$ no tiene ciclos.

**6.** Diseñe un algoritmo que halle las componentes conexas de un grafo basándose en un esquema parecido al algoritmo de PRIM.

**7.** Sea $T=(V_T,E_T)$ un árbol y $v$ un vértice de $T$. Podemos inducir una orientación sobre $T$ de forma que el resultado sea una arborescencia $T'$ con raíz $v$. Decimos que dos vértices $x, y \in V_T$ forman un par descendiente lineal con respecto a $T'$ si existe un camino de $x$ a $y$ o de $y$ a $x$ en $T'$. Sea $G=(V,E)$ un grafo no orientado, $v \in V$ y T un árbol cobertor de $G$. Sea $T'$ la arborescencia obtenida al orientar $T$ de forma que $v$ sea raíz de $T'$. Se dice que $T'$ es un árbol cobertor lineal de $G$ si y sólo si para cada lado $\{x, y\} \in E$, los vértices $x$, $y$ forman un par descendiente lineal con respecto a $T'$. Demuestre que dado $G$ y $v$, existe un árbol cobertor lineal de $G$ con raíz $v$.

**8.** En el siguiente grafo:

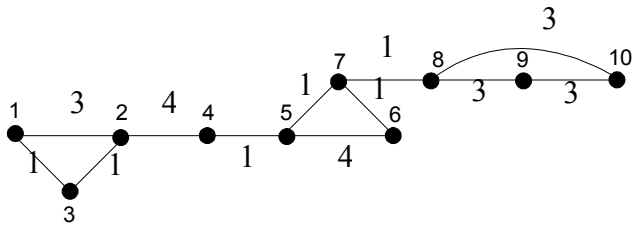



**figura VI.26**

a) Halle un árbol mínimo cobertor usando Kruskal.

b) Halle un árbol mínimo cobertor usando Prim.

**9.** La siguiente tabla representa los arcos de un digrafo cuyos vértices son $V$={a, b, c, d, e, f, g}. Este grafo corresponde a una red de comunicación y los costos de utilización de cada arco aparecen también en la tabla.

| N. Inicial | N. Terminal | Costo | N. Inicial | N. Terminal | Costo |
|---|---|---|---|---|---|
| a | b | 2 | e | b | 2 |
| b | d | 2 | f | c | 3 |
| c | a | 1 | f | e | 1 |
| d | c | 3 | g | e | 2 |
| d | e | 1 | g | f | 4 |

Se desea determinar la red que comunique el vértice g con cada uno de los otros vértices de forma que la red resultante tenga el mínimo costo global.

**10.** Muestre que si $G=(V,E)$ es un grafo no dirigido cuyas aristas tienen costos positivos todos distintos, entonces $G$ tiene un único árbol mínimo cobertor.

**11.** Sea $G=(V,E)$ un grafo conexo y no orientado y con al menos un istmo $e$. Sea $G_1$ y $G_2$ las dos componentes conexas de $G'=(V, E-\{e\})$, sean $T_1$ y $T_2$ los árboles cobertores mínimos de las dos componentes conexas de $G'$. Sea $T=T_1 \cup T_2 \cup \{e\}$. ¿ $T$ es un árbol mínimo cobertor de $G$ ?.

**12.** En los campos de golf de LAMUNITA Golf Club se desea instalar un sistema de riego automatizado. Para tal fin se debe instalar una red de tuberías para distribuir agua con fertilizantes para la grama. Los tubos a instalar tienen diferentes diámetros y costos según el terreno por donde deben pasar. Los puntos de riego están marcados con las letras de la **a** a la **f**. Las tuberías posibles de instalar con sus respectivos diámetros y costos son las siguientes:

| Tubería | Costo | Diámetro | Tubería | Costo | Diámetro |
|---|---|---|---|---|---|
| {a, b} | 3 | 3 | {b, d} | 5 | 5 |
| {a, c} | 1 | 5 | {b, f} | 1 | 2 |
| {a, d} | 1 | 2 | {c, e} | 3 | 5 |
| {a, e} | 2 | 4 | {d, e} | 2 | 4 |
| {b, e} | 2 | 2 | {d, f} | 1 | 1 |
| {b, e} | 2 | 3 | {e, f} | 4 | 3 |

Sin embargo, para facilitar el paso del agua con fertilizantes, se ha determinado que las tuberías a utilizar deben tener un diámetro mayor o igual a 3. Bajo estas condiciones, determine la red de riego de menor costo. Indique el algoritmo a utilizar y cualquier modificación al mismo si lo considera necesario.

**13.** El departamento de vialidad del municipio Apié posee un presupuesto limitado para la construcción o reparación de las vías de comunicación del municipio. El municipio Apié consta de diez urbanizaciones y es la intención del departamento proveer a los residentes de las comunidades de un sistema de carreteras que permita interconectar todos los poblados. El cuadro siguiente muestra los costos de conexión directa entre las comunidades, para todas las conexiones posibles:

| Conexión | Costo de comunicación |
|---|---|
| 1-2 | 7 |
| 2-3 | 4 |
| 3-7 | 2 |
| 3-8 | 6 |
| 4-5 | 6 |
| 5-6 | 4 |
| 6-7 | 5 |
| 6-10 | 4 |
| 6-9 | 3 |
| 10-9 | 5 |
| 1-9 | 7 |

Determine, utilizando sus conocimientos de grafos, cómo deben ser conectados las ciudades de manera que la red de vialidad sea de costo mínimo.

**14.** Demuestre que el número de hojas de una arborescencia con grado de salida igual a k para todo vértice del grafo (k > 1) es siempre mayor que el número de vértices internos.

**15.** Encontrar un bosque orientado de peso máximo en el siguiente grafo:

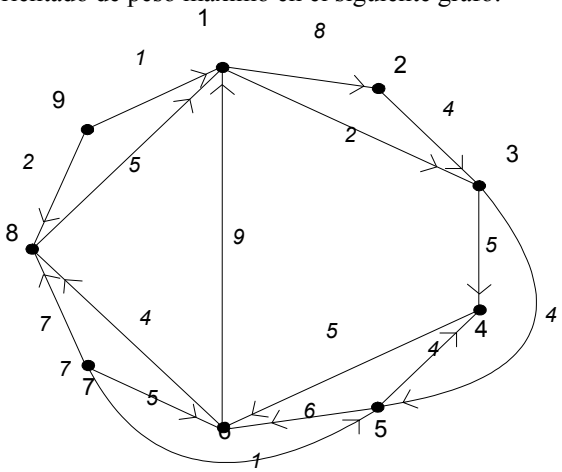



**figura VI.27**

**16.** Dadas las siguientes expresiones aritméticas, construya los respectivos árboles planos que las representan:

a) a + b – (c * d)
b) g * j – y / w
c) a – b / c + d
d) w + (x – y) * h
e) x + y / w * h
f) w + (a * d) – (b * c – j)

**17.** Dada una expresión aritmética con operadores binarios (+, -, *, /):

a) Diseñe el procedimiento que construya el árbol binario correspondiente.
b) Escriba un procedimiento que acepte valores para cada una de las variables en la expresión y luego recorra el árbol para evaluar la expresión al tiempo que imprime la misma notación postfija.

**18.** Diseñe y codifique una función que reciba un árbol binario y un vértice de ese árbol, y devuelva el nivel en cual se encuentra ese vértice.

**19.** Se dice que un árbol $T_1$ es similar a un árbol $T_2$ si se tiene una de las dos condiciones siguientes:

a) $T_1$ tiene el mismo dibujo que $T_2$
b) $T_1$ es igual a $T_2$ visto en un espejo.

Diseñe una función que reciba dos árboles $T_1$ y $T_2$ y devuelva 1 si los árboles son similares y 0 en caso contrario.

**20.** Un árbol binario de $n$ hojas es estrictamente binario si tiene $2n$-1 vértices. Diseñe un procedimiento que verifique si un árbol es estrictamente binario.

**21.** Sea $h$ la altura de un árbol binario $T$. Determine las dimensiones mínimas de un arreglo en el cual podría almacenarse todo el árbol. ¿Cómo sería el acceso?

**22.** Suponga que dispone de los siguientes operadores:

- Nivel($T$,$x$): devuelve un entero correspondiente al nivel del vértice cuya clave es $x$ en el árbol $T$.
- Padre($T$,$x$): devuelve la clave del vértice padre del vértice cuya clave es $x$.

Usando estos operadores, escriba un procedimiento PAC($T$, $x$, $y$) que devuelva la clave del vértice correspondiente al primer antecesor común de los vértices cuyas claves son $x$ e $y$ respectivamente.

**23.** Para una expresión aritmética en preorden en la cual sólo aparecen los operadores binarios (+, -, /, *), diseñe un algoritmo que construya el árbol binario asociado a la expresión.

**24.** Considere los siguientes recorridos de árboles binarios:

a) (1) Recorrer el subárbol derecho.
(2) Visitar la raíz.
(3) Recorrer el subárbol izquierdo.

b) (1) Visitar la raíz.
(2) Recorrer el subárbol derecho.
(3) Recorrer el subárbol izquierdo.

¿Qué relación existe entre estos dos recorridos y los recorridos pre, pos e inorden?.

**CAPÍTULO VII**

# *FLUJO EN REDES*

## INTRODUCCIÓN

En este capítulo estudiaremos varios problemas de Flujo en Redes. Estos problemas pueden todos ser formulados como problemas de Programación Lineal y resueltos utilizando los métodos clásicos, sin embargo, los algoritmos que presentaremos aquí son algoritmos que nos interesan por adaptarse directamente a cada problema tratado y por fundamentarse en propiedades de la estructura subyacente en toda red, a saber, el digrafo subyacente formado por los vértices y arcos de la misma.

## VII.1 BÚSQUEDA DE CAMINOS MÁS CORTOS EN REDES

En el capítulo V se estudiaron varios algoritmos de búsqueda de caminos de costo mínimo que parten de un vértice s, considerando siempre $cmin(s,v) < -\infty$, $v \in V$. Volvemos aquí sobre el mismo problema levantando esta exigencia, para lo cual incluiremos en los algoritmos a estudiar mecanismos para detectar la presencia de circuitos y poder detener la ejecución del algoritmo.

Definiremos primero lo que llamamos una RED.

**Definición VII.1.1**

Una red $R = (V,E,f_1, \dots , f_m)$ es una tupla donde V y E son los vértices y arcos de un digrafo G = (V,E) sin bucles, llamado el *digrafo subyacente* y donde $f_1, \dots , f_m$ son funciones $f_i: E \rightarrow \Re$ que pueden representar magnitudes tales como distancia, costo, capacidad, etc.

Consideremos una red R = (V,E,d) donde la función d la llamaremos distancia. Mientras no se especifique lo contrario, supondremos que el digrafo subyacente G = (V,E) es sin arcos múltiples.

Dados dos vértices **x, y** en V, denotaremos por $\mathbf{C}_x^y$ un camino del vértice **x** al vértice **y**. Recordemos que el costo de un camino C en una red es la suma de los costos de los arcos que conforman el camino, y se denota por d(C). Si a la función asociada a los arcos la llamamos distancia estaremos hablando de la distancia del camino.

**Definición VII.1.2**

Un circuito en R = (V,E,d) se dice *Absorbente* si su costo o distancia es negativo.

Definamos a continuación dos funciones **I, T** : $E \rightarrow V$, las cuales asocian a cada arco sus vértices Inicial y Terminal respectivamente.

**Definición VII.1.3**

Dados dos vértices x, y en una red R=(V,E,d), si existe un camino más corto de x a y entonces la distancia de ese camino se llama *Distancia mínima* de x a y. Así, la distancia mínima de un vértice a si mismo, si existe, es nula.

Sea s una raíz de R = (V,E,d) sin circuitos absorbentes, podemos definir otra función $\boldsymbol{\pi}$: $V \rightarrow \Re$ que asocia a cada vértice **x** la distancia mínima de **s** a **x**.

**Proposición VII.1.1**

Sea R = (V,E,d) sin circuitos absorbentes y con una raíz **s**, entonces:

$\forall e \in E$: $\boldsymbol{\pi}(T(e)) - \boldsymbol{\pi}(I(e)) \leq d(e)$. (I)

**Demostración:**

Sea **e**=(x,y), entonces C = $\mathbf{C}_S^x$ || <x, e, y> donde $\mathbf{C}_S^x$ es un camino de costo mínimo, es un camino de **s** a **y**, de donde d(C) = $\boldsymbol{\pi}$(x) + d(**e**) ≥ $\boldsymbol{\pi}$(y).

¤

**Proposición VII.1.2**

Sea R = (V,E,d) como antes, **s** una raíz de R y ∀x∈V: $\boldsymbol{\pi}$(x) = distancia mínima de **s** a x en R, entonces la subred R(E'), donde E' = {e ∈ E / $\boldsymbol{\pi}$(T(e)) - $\boldsymbol{\pi}$(I(e)) = d(e)} también tiene raíz **s** y todos los caminos de costo mínimo en R están en R(E'), la red inducida por E'.

Recíprocamente, todo camino de **s** a **x** en R(E') es un camino de costo mínimo.

**Demostración:**

Sea $\mathbf{C}_S^z$ el camino de costo mínimo de **s** a **z** en R, e = (x,y) un arco de este camino mínimo. Entonces las subsecuencias $\mathbf{C}_S^x$, $\mathbf{C}_S^y$ son caminos de costo mínimo a **x** e **y** respectivamente. Así: d($\mathbf{C}_S^y$) = $\boldsymbol{\pi}$(y) = d($\mathbf{C}_S^x$) + d(e) = $\boldsymbol{\pi}$(x) + d(e), de donde vemos que todo arco perteneciente a un camino mínimo de **s** a cualquier otro vértice **x** está en E'.

Sea ahora C un camino de **s** a cualquier vértice **x** en R(E'), veamos que este es un camino de costo mínimo.

$$d(C) = \sum_{e\in C} d(e) = \sum_{e\in C} (\boldsymbol{\pi}(T(e)) - \boldsymbol{\pi}(I(e))) = \boldsymbol{\pi}(x) - \boldsymbol{\pi}(s) = \boldsymbol{\pi}(x)$$

luego siendo la distancia de C igual a la distancia mínima de **s** a **x,** C es un camino de costo mínimo.

¤

Cabe observar que luego de esta demostración podemos caracterizar todos los arcos sobre un camino de costo mínimo como los arcos que verifican la desigualdad (I) de la Proposición VII.1.1 como una igualdad.

**CorolarioVII.1.2.1**

Todo circuito en R(E') tiene distancia nula**.**

**Algoritmo General**

A continuación damos la modificación que habría que hacer a la rutina de eliminación del modelo general de caminos de costo mínimo presentado en el capítulo V, para detectar circuitos absorbentes. En cuanto a los atributos, es fácil ver que a cada camino $P^j$ = Ext ($P_i$, $x^j$) le debe corresponder como atributo A($P^j$) = A($P_i$) + d(e), donde e = ($x_i$,$x^j$). Este atributo corresponde al costo del camino, es decir, la distancia del vértice inicial s a $x^j$ . La selección del próximo camino abierto a cerrar se hará tomando el que tenga menor atributo. La Rutina de eliminación de caminos quedaría como sigue:

```
Rutina de Eliminación de Caminos para el Alg. General de Camino Mínima :
Comienzo
    Para  todo camino P^j obtenido por expansión de P_i a x^j hacer:
        Si  no existe camino P_k en la lista  con igual vértice final x^j que P^j  entonces   Abrir P^j
        sino { ya existe otro camino listado hasta  x^j }
            Si  A (P_k) > A (P^j) entonces
                    Si  x^j está en el camino P^j entonces
                            Comienzo
                            Escribir (' Hay Circuito Absorbente ');
                            PARAR
                            Fin
                    sino   Comienzo
                             Eliminar P_k;
                             Abrir P^j;
                             Fin;
Fin.
```

El algoritmo anterior permite detectar circuitos absorbentes si los hay, y en caso contrario encuentra la arborescencia de distancias mínimas de raíz **s**.

Una segunda versión del algoritmo, la cual presentamos a continuación utiliza el algoritmo de Dijkstra para encontrar una arborescencia inicial A de raíz **s**, no necesariamente de distancias mínimas, y luego trata de mejorar esa arborescencia hasta que sea óptima o hasta detectar un circuito absorbente. Para mejorar la arborescencia inicial, el algoritmo se basa en las Proposiciones VII.1.1 y VII.1.2.

**CaminoMinGeneral** (V, E, I, T, d, **s;** VAR **π**, A, EsRaiz, CircuitAbs, C)
{Entrada:V, E, I, T, d, **s**.
Salida:**π**, A, EsRaiz, CircuitAbs, C.
EsRaiz y CircuitAbs son variables booleanas que indican respectivamente si **s** es o no raíz y si existe o no un circuito absorbente. En caso que ocurra esto último, C sería tal circuito . A es un arreglo con las referencias a los antecesores de cada vértice en la arborescencia de caminos de costo mínimo. El algoritmo de Dijkstra devuelve en **π** un vector de costos tal que **π**(x), x en V, que, como puede haber circuitos de costo negativo, no necesariamente es el costo del camino mínimo de s a x. La variable booleana EsRaiz devuelve verdadero si y sólo si s es raíz de G=(V,E).}

```
Comienzo
    Dijkstra (V,E,I,T,d,s; π,A,EsRaiz);
    CircuitAbs ← FALSO;
    Si (EsRaiz) entonces
    Comienzo
        E' ← {e∈E / e = (A(x),x), x∈V};
        Mientras (∃ê∈E: d(ê) < π(T(ê))-π(I(ê)) ) y (no CircuitAbs) hacer
            Si (G=(V,E' ∪ {ê}) contiene un circuito C ) entonces CircuitAbs ← VERDADERO
            sino
            Comienzo
                x ← T(ê); E' ← E' ∪{ê} - {(A(x),x)};
                A(x) ← I(ê);
                ∂ ← π(T(ê)) - π(I(ê)) - d(ê);
                π(x) ← π(x) - ∂;
                Para todo y descendiente de x en el árbol A hacer: π(y) ← π(y) - ∂;
            Fin
    Fin
Fin.
```

Se observa que en cada iteración del algoritmo $\sum_{v\in V}$ **π**(v) decrece estrictamente, así que no se vuelve nunca a una arborescencia hallada anteriormente. Siendo finito el número de arborescencias de R = (V,E,d), queda asegurada la terminación del algoritmo.

Para determinar si existe circuito absorbente basta mirar si T(**ê**) es antecesor de I(**ê**) en **A**, donde **ê** es el arco que se está agregando en ese momento. Por ser A una arborescencia, esto último es muy sencillo pues existe un único camino de s a x en A y basta recorrerlo en sentido inverso.

## VII. 2 FLUJO MÁXIMO

Prácticamente hablando, un problema de flujo máximo consiste, por ejemplo, en buscar la cantidad máxima de agua que puede fluir, en un determinado momento, desde una estación de bombeo hasta un reservorio, a través de un sistema de acueductos. Podría representar también la cantidad máxima de vehículos por unidad de tiempo que soporta una red vial entre dos ciudades.

Para definir formalmente el problema del Flujo Máximo recordaremos que en una red R = (V,E,c), un *VÉRTICE FUENTE* es un vértice **s** tal que no existe ningún arco e∈E, tal que T(e) = **s**. Se llama *VÉRTICE SUMIDERO* o *POZO* a un vértice **p** tal que no existe ningún arco e∈E con I(e) = **p**.

**Definición VII.2.1**

Sea R = (V,E,c) una red con un vértice fuente **s** y un vértice sumidero **p**. Agreguemos a R un arco $\mathbf{e_r}$ = (p,s), con $c(\mathbf{e_r}) = +\infty$. El arco $\mathbf{e_r}$ se llama *Arco de Retorno.*

Este arco se agrega a la red para efectos de facilitar la resolución del problema, lo cual veremos a continuación.

**Definición VII.2.2**

Dada una red R = (V,E,c) con un vértice fuente **s**, un vértice sumidero **p** y la función c: E $\rightarrow$ $\mathfrak{R}$. Sea $\tilde{E} = E \cup \{\mathbf{e_r}\}$. Una función **f** de $\tilde{E}$ en $\mathfrak{R}$ se llama un *FLUJO* sobre el digrafo subyacente G = (V, $\tilde{E}$) si cumple la siguiente igualdad para todo vértice x en V se tiene que $\sum_{e\in\tilde{E}/T(e)=x} f(e) - \sum_{e\in\tilde{E}/I(e)=x} f(e) = 0$, es decir, todo el flujo que llega a un vértice debe ser igual al flujo que sale del vértice.

Dada una red R = (V,E,c) con un vértice fuente **s** y un vértice sumidero **p**, donde la función **c**:E $\rightarrow\mathfrak{R}$ representa una "capacidad" definida sobre los arcos, buscar el Flujo Máximo consiste en buscar un vector $\mathbf{(f(e_1), f(e_2),...,f(e_m))}$, $\tilde{E} = \{e_1, e_2, ..., e_m\}$ y $m=|\tilde{E}|$, tal que:

i) **f** sea un **flujo** sobre el grafo G(V, $\tilde{E}$), donde $\tilde{E} = E \cup \{(\mathbf{p},\mathbf{s})\} = E \cup \{\mathbf{e_r}\}$

ii) $0 \le f(e) \le c(e), \quad \forall e \in \tilde{E}$.

iii) $f(\mathbf{e_r}) = f((\mathbf{p},\mathbf{s}))$ sea máximo bajo las condiciones (i) e (ii).

Entre otras cosas, es por ello que es necesario agregar el arco de retorno, para que la ley de conservación pueda cumplirse incluso para los vértices **s** y **p**. Debido a esto mismo, el flujo por el arco de retorno, representa la cantidad total de flujo que "pasa" por la red en cualquier momento y es esta cantidad $f(\mathbf{e_r})$ la que se busca maximizar al resolver el problema.

Un flujo **f** en R = (V, $\tilde{E}$,c) que satisface las condiciones (i) e (ii) se llama un *flujo factible*.

Otros problemas, tales como la presencia de varios vértices fuente ($s_1,..s_k$) y/o varios vértices sumidero ($p_1,..,p_q$), pueden ser resueltos fácilmente llevándolos al caso expuesto anteriormante. Para ello basta con agregar un vértice fuente ficticio $\mathbf{s_O}$, o "super-fuente" y/o un vértice pozo o sumidero ficticio $\mathbf{p_O}$, o "super-pozo", con arcos $(\mathbf{s_O},s_i)$, i=1,..,k, $(p_j,\mathbf{p_O})$, j=1,..,k con capacidad infinita, además del arco de retorno $(\mathbf{p_O},\mathbf{s_O})$.

Igualmente, problemas tales como la presencia de restricciones de capacidad sobre los vértices, lo cual podría representar limitaciones de capacidad de intersecciones, pueden también ser fácilmente resueltos bajo el esquema anterior, representando a cada vértice **x** como dos nuevos vértices **x'** e **x"** unidos por un arco (**x'**, **x"**) con capacidad igual a la capacidad del vértice **x** original.

Este problema de Flujo Máximo puede ser formulado como un Problema de Programación Lineal (P.P.L.). Para ello usaremos la Matriz de Incidencias del digrafo subyacente.

Si denotamos $\tilde{M}$ la Matriz de Incidencias del digrafo G = (V, $\tilde{E}$), el problema del flujo máximo se convierte en el siguiente P.P.L.

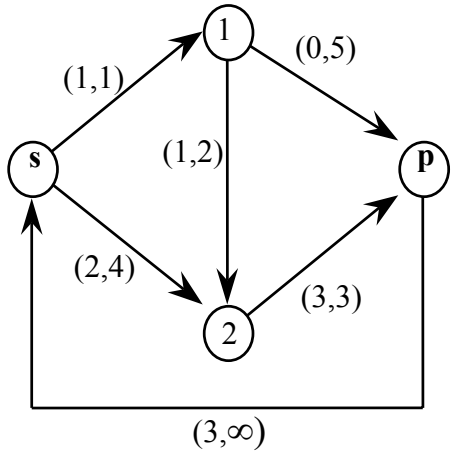
**max** $f(\mathbf{e_r})$

**s.a.** $\begin{cases} \tilde{M}.f = 0 \quad \text{donde f es el vector de flujo} \\ f \le c \quad \text{c es el vector de capacidad} \\ f \ge 0 \end{cases}$

### VII.2.1 ALGORITMO DE FORD & FULKERSON

El algoritmo de Ford & Fulkerson es un algoritmo para la búsqueda del flujo máximo en una red R=(V,E,c). Una primera idea para este algoritmo es: dado un flujo **f** incial factible, buscar un camino (elemental) **C** de **s** a **p** tal que ningún arco del camino esté saturado, es decir que $\forall e\in\mathbf{C}$: f(e) < c(e). De conseguirse este camino, el flujo puede ser aumentado en cada uno de los arcos en $\boldsymbol{\varepsilon} = \mathbf{Min}_{\ e\in\mathbf{C}} [c(\mathbf{u}) - f(\mathbf{u})] > 0$.

Sin embargo este procedimiento podría bloquearse, como se ve en la Figura VII.1, sin que se haya encontrado el flujo máximo.




- (flujo,capacidad)
- Valor Flujo Máximo = 4

**figura VII.1**

Una segunda idea consiste en buscar una cadena (elemental) **C** tal que, si agregamos el arco de retorno $\mathbf{e}_r$ a esta cadena **C**, podemos dividir los arcos de **C** en:

$\mathbf{C}^+ = \{e \in \mathbf{C}$ / e tiene la misma orientación que $\mathbf{e}_r$ en $\mathbf{C}\}$

$\mathbf{C}^- = \{e \in \mathbf{C}$ / e tiene orientación contraria a $\mathbf{e}_r$ en $\mathbf{C}\}$

Cada uno de los arcos en **C** debe ser tal que:

Si $e \in \mathbf{C}^+$ : $f(e) < c(e)$

Si $e \in \mathbf{C}^-$ : $f(e) > 0$.

Una cadena como la descrita anteriormante se llama una *cadena de aumento.* Una vez hallada esta cadena, el flujo a través de la red puede aumentarse en:

$$\varepsilon = \mathbf{Min}\ \{\ \mathbf{Min}_{\ e \in C^+}\ [c(e) - f(e)],\ \mathbf{Min}_{\ e \in C^-}\ [f(e)]\ \} > 0$$

haciendo:

$$f(e) = \begin{cases} f(e)+\varepsilon & \text{si } e \in C^+ \\ f(e)-\varepsilon & \text{si } e \in C^- \end{cases}$$

Este será un nuevo flujo factible ya que:

i) Las restricciones de conservación de flujo se mantienen sobre todos los vértices, lo cual es facil ver ya que:

- Si **C** no pasa por $x \in V$, es evidente que el flujo a través de **x** se mantiene constante.

- Si **x** es un vértice tal que **C** pasa por él, habrá exactamente dos arcos de **C** incidentes en **x**. En esta situación se pueden presentar cuatro casos. Llamemos e1 y e2 los dos arcos incidentes en **x**, entonces:

$T(e1) = T(e2) = \mathbf{x}$ ó $I(e1) = I(e2) = \mathbf{x}$ ó $T(e1) = I(e2) = \mathbf{x}$ ó $I(e2) = T(e1) = \mathbf{x}$.

En todos estos casos las modificaciones sobre el arco e1 compensan las modificaciones en el arco e2 de forma que las leyes de conservación de flujo se mantienen.

ii) El nuevo valor del flujo **f** verifica de nuevo: $0 \le f(e) \le c(e)$, $\forall e \in E$, por la forma como se escoge $\varepsilon$.

Sea Y un subconjunto de vértices de R = (V,E,c). Sea

$\Omega(Y) = \{ e \in E$ / $I(e) \in Y, T(e) \in Y$ ó $T(e) \in Y, I(e) \in Y \}$ el *cociclo inducido* por el conjunto Y.

Los arcos de $\Omega(Y)$ los dividimos en dos subconjuntos $\Omega^+$, $\Omega^-$, donde:

$$\Omega^+(Y) = \{ e \in E / I(e) \in Y, T(e) \in Y \}$$

$$\Omega^- (Y) = \{ e \in E / T(e) \in Y, I(e) \in Y \}$$

**Definición VII.2.1.1**

Se llama un *Corte que separa **s** de **p*** a un subconjunto **K** de arcos de R = (V,E,c) tal que, existe un subconjunto de vértices **Y** con $\mathbf{s} \in Y$, $\mathbf{p} \in Y$ con $\mathbf{K} = \Omega^+(Y)$.

Es interesante notar que si **K** es un corte que separa **s** de **p**, entonces todo camino de **s** a **p** en R = (V,E,c) tiene un arco en **K**.

**Definición VII.2.1.2**

Se define la *Capacidad de un Corte **K*** en R = (V,E,c) como la suma de las capacidades de los arcos en **K**, es decir: $c(\mathbf{K}) = \sum_{e \in \mathbf{K}} c(e)$.

**Proposición VII.2.1.1**

Para todo flujo factible **f** en R = (V, $\tilde{E}$ ,c) y para todo corte **K** que separa **s** de **p** se tiene que $f(\mathbf{e_r}) \le c(\mathbf{K})$.

**Demostración:**

Sea Y un subconjunto de vértices tal que $\mathbf{K} = \Omega^+(Y)$. Si sumamos las restricciones de conservación de flujo sobre todos los vértices en Y obtenemos:

$$\sum_{e \in \Omega^+(Y)} f(e) - \sum_{e \in \Omega^-(Y)} f(e) = 0$$

Puesto que $\mathbf{s} \in Y$, $\mathbf{p} \in Y$, entonces $\mathbf{e_r} = (\mathbf{p},\mathbf{s}) \in \Omega^-(Y)$, de donde despejando se obtiene:

$$f(e_r) = \sum_{e \in \Omega^+(Y)} f(e) - \sum_{e \in \Omega^-(Y) - \{(p,s)\}} f(e)$$

pero $0 \le \mathbf{f}(\mathbf{e}) \le c(\mathbf{e})$, $\forall e \in E \quad \Rightarrow \quad \sum_{e \in \Omega^-(Y) - \{(p,s)\}} f(e) \ge 0$,

y $\sum_{e \in \Omega^+(Y)} f(e) = \sum_{e \in K} f(e) \le \sum_{e \in K} c(e)$ luego $f(\mathbf{e_r}) \le c(\mathbf{K})$.

¤

En la próxima sección presentaremos en detalle un algoritmo para resolver un problema más general que el problema de flujo máximo dado en esta sección y que podemos utilizar sin cambio alguno para resolver el problema de flujo máximo.

### VII.2.2 Flujo Máximo en Redes Canalizadas

Llamamos *Red Canalizada* a una red R = (V,E∪{$\mathbf{e_r}$},b,c) con fuente s y pozo p, $\mathbf{e_r}$ = (s,p), en la cual además de la función "capacidad" que acota superiormente el flujo, tenemos otra función b: E —> $\mathfrak{R}$ que corresponde a una cota inferior para el flujo por los arcos (por esto decimos que el flujo está canalizado). El problema consiste en hallar un vector f tal que:

(i) f sea un flujo en el grafo (V,E∪{$\mathbf{e_r}$})

(ii) $\forall e \in E$: $b(\mathbf{e}) \le f(\mathbf{e}) \le c(\mathbf{e})$.

(iii) $f(\mathbf{e_r})$ sea máximo.

Note que si b(e)=0, $\forall e \in E$, entonces el problema anterior no es mas que el problema de flujo máximo visto en la sección anterior. Note también que $c(\mathbf{e_r})$ puede ahora no ser infinito.

La solución inicial trivial f(e) = 0, $\forall e \in E$, puede no ser factible en este nuevo problema por lo que habrá que disponer de un mecanismo para encontrar al menos una solución inicial factible al problema. Esto se verá en la próxima sección. Sin embargo, para el problema de flujo máximo de la sección anterior se tiene que la solución trivial es factible.

A continuación presentamos un algoritmo para determinar un flujo máximo en una red canalizada. Este algoritmo requiere que el flujo **f** inicial sea un flujo factible en R = (X,V,b,c). Cuando b(e)=0, ∀e∈E, podemos tomar como flujo inicial a f(**e**)=0, ∀e∈E. MODIF es una variable booleana que toma el valor VERDADERO si el flujo **f** es modificado por el algoritmo. ExisteFM, también booleana, indica si existe un Corte de capacidad finita, en cuyo caso ExisteFM = VERDADERO.

```
FLUJOMAX(V, E, I, T, e_r, s, p, b, c, fi; VAR f, Y, MODIF, ExisteFM): { Ford & Fulkerson }
{Entradas: V, E, I, T, e_r, s, p, b, c, fi
 Salidas: f, Y, MODIF, ExisteFM}
 Comienzo
   MODIF ← FALSO;
   f ← fi;
   ExisteFM ← VERDADERO;
   Repetir
      MARCAR (V,E,I,T,e_r,s,p,b,c,f; Y,A,epsilon);
      Si (p∈Y y epsilon ≠ ∞) entonces
      Comienzo
            x ← p;
            C+ ←{e_r};
            C- ← ∅;
            M ODIF ← VERDADERO;
            Mientras (x≠s) hacer:
            Comienzo
                  e ← A(x);
                  Si (x = T(e)) entonces
                  Comienzo
                        C+ ← C+ ∪ {e};
                        x ← I(e);
                  Fin
                  sino Comienzo
                        C- ←C- ∪ {e};
                        x ←T(e);
                           Fin;
            Fin ;
            Para todo e∈C+ hacer : f(e) ←f(e) + epsilon ;
            Para todo e∈C- hacer: f(e) ←f(e) - epsilon ;
      Fin
   hasta que (p∉Y ó epsilon = ∞);
Si (epsilon = ∞) entonces ExisteFM ←FALSO ;

Donde
MARCAR(V,E,I,T,e_r,s,p,b,c,f; VAR Y,A,epsilon):
{ Entradas: V,E,I,T,e_r,s,p,b,c,f.
  Salidas: Y,A,epsilon
Y es un conjunto de vértices y A es un vector que asocia a cada vértice un arco del cual él es extremo. Si al
final del algoritmo p∈Y entonces se puede aumentar el flujo a través del camino de aumento <p, x1, x2, ... , s>
que se reconstruye a partir de A de la siguiente forma: x1 es el otro vértice extremo del arco A(p), x2 es el
otro vértice extremo de A(x1), etc. Si
∂(s) es la cantidad máxima en que se desea incrementar el flujo del arco e_r}
Variable ExisteC : booleana;
Comienzo
      Y ← {s};
      ExisteC ← VERDADERO;
      ∂(s) ← c(e_r) - f(e_r);
```

```
        epsilon ← 0;
        Mientras (p∉Y y ExisteC ) hacer
            Si (∃e=(x,y) / x∈Y, y∉Y, f(e) < c(e)) entonces {Marcado Directo }
            Comienzo
                    Y ← Y ∪ {y};  A(y) ←  e;
                    ∂(y) ← Min [ ∂(x), c(e) - f(e)]
            Fin
            sino {Marcado Inverso }
               Si (∃e=(x,y) / y∈Y, x∉Y, f(e) > b(e)) entonces
               Comienzo
                        Y ← Y ∪ {x};  A(x) ←  e;
                        ∂(x) ← Min [ ∂(y), f(e) - b(e)]
               Fin
               sino ExisteC  ← FALSO;
        Si (ExisteC) entonces epsilon ← ∂(p);
Fin ;  { MARCAR }
Fin.  { Flujo Max }
```

**Justificación del Algoritmo:**

i) Sea $\hat{E}$ = { e∈E/ ∃y∈ Y con A[y] = e }  (A e Y son salidas de Marcar).

El grafo G(V,$\hat{E}$) es claramente un árbol. La cadena que une **s** a **p** en G = (V,$\hat{E}$) unido con $\mathbf{e_r}$ es un ciclo **C** donde $\mathbf{C}^+$ son los arcos orientados como $\mathbf{e_r}$ y $\mathbf{C}^-$ son los arcos orientados en sentido contrario a $\mathbf{e_r}$.

ii) Por la forma como se calcula **ε,** al modificar los valores de f(**e**) se mantiene la factibilidad.

i) + ii) ⇒ el flujo que se obtiene al final de cada iteración es siempre factible y estrictamente mejor que el anterior, pues al f($\mathbf{e_r}$) le asigna f($\mathbf{e_r}$) + **ε**, (**ε** > 0).

ε puede ser infinito cuando exista un camino de aumento de costo no acotado en cuyo caso el flujo es no acotado.

Una vez que termina el algoritmo con **p**∉Y, se puede ver que hemos encontrado un corte **K** de **s** a **p** con c(**K**) = f($\mathbf{e_r}$), lo cual según la Proposición VII.2.1.1 implica que **f** no puede aumentar más (y además no podremos conseguir otro corte de capacidad menor).

Veamos a continuación que efectivamente se ha detectado tal corte:

Sea Y el último conjunto de vértices marcados. Por construcción **p**∉Y.

Sea **e** = (x,y) ∈ $\mathbf{\Omega}^+$(Y) ⇒ f(**e**) = c(**e**) (sino y = T(**e**) sería marcado en la marcación directa).

Sea **e** = (y,x) ∈ $\mathbf{\Omega}^-$ (Y) ⇒ f(**e**) = 0  (sino y = I(**e**) sería marcado en la marcación inversa).

Tomando **K** = $\mathbf{\Omega}^+$(Y) tenemos f($\mathbf{e_r}$) = $\sum_{e\in \mathbf{K}}$ c(e) = c(**K**).

¤

**Proposición VII.2.1.2 (Flujo Max - Corte Min)**

El valor máximo de f($\mathbf{e_r}$) para un flujo factible **f** en R = (V,E,c) es igual a la capacidad de un corte de **s** a **p** de capacidad mínima. En particular f($\mathbf{e_r}$) será no acotado si y sólo si no existe en R = (V,E,c) un corte de capacidad finita que separe **s** de **p**.

**Demostración:**

Podemos remitirnos al hecho de que el problema de flujo máximo es un P.P.L. que tiene al menos una solución: f(**e**) = 0, ∀e∈ $\tilde{E}$ . Además si existe algún corte de capacidad finita, el problema es acotado y por lo tanto tiene solución óptima finita.

¤

En cuanto a la terminación del algoritmo de Ford & Fulkerson (F&F), podemos ver que siempre que las capacidades de los arcos sean números racionales, podemos considerar que todas son múltiplos enteros de algún número $\partial$ en **R**. Así, si existe corte finito, al comenzar con f(**e**) = 0, $\forall e \in E$, e será siempre múltiplo de $\partial$, luego los valores del vector **f** para cada solución serán también múltiplos de $\partial$, y f($\mathbf{e_r}$) aumenta en cada iteración en un múltiplo de $\partial$ estrictamente mayor que cero. Este crecimiento al ser el problema acotado se detiene en un número finito de pasos.

Por otro lado, con una implementación como la propuesta por Edmonds & Karp (1972), en la cual las cadenas de aumento se buscan tratando el conjunto de vértices marcados como una COLA y examinando primero todos los arcos incidentes al vértice en el Principio de la Cola antes de pasar a examinar los vecinos del siguiente vértice en la Cola, el procedimiento MARCAR resulta similar al algoritmo de Dijkstra, y aún más eficiente pues para marcar un nuevo vértice basta con que este cumpla las condiciones para una marcación bien sea directa o inversa, sin necesidad de hacer más comparaciones. Esto nos da un orden $O(n^2)$ para MARCAR. En cuanto al número de veces que este procedimiento es llamado por FlujoMax, Edmonds & Karp (1972) encuentran una cota $O(n^3)$ basandose en la implementación ya mencionada. Con estos resultados, obtenemos una cota $O(n^5)$ para el algoritmo de Flujo Máximo.

Si las capacidades de los arcos son números irracionales, es posible que el algoritmo no pare, o peor aún, que converja a soluciones lejanas al óptimo (Ford & Fulkerson, 1962). Esto sin embargo, en los casos reales no ocurre puesto que siempre se pueden tomar aproximaciones racionales lo suficientemente buenas ya que los irracionales no tienen una representación exacta en la computadora.

En el caso que no exista corte finito, esto implica que existe al menos un camino $\mathbf{C}_S^p$ de **s** a **p** en el cual todos los arcos tienen capacidad infinita. Siendo finito el número de caminos elementales y estando siempre este camino $\mathbf{C}_S^p$ no saturado, es seguro que en la primera etapa de la marcación (marcación directa) este camino será encontrado y se detectará al ser $\boldsymbol{\varepsilon}$ = **infinito** para tal camino.

### VII.3 FLUJOS FACTIBLES

Consideremos una red R = (V,E,b,c) con b: $E \to \mathfrak{R}$, c: $E \to \mathfrak{R}$ y $b(e) \le c(e)$, $\forall e \in E$.

Se desea encontrar un vector **f** en $\mathbf{R}^m$ (m=|E|), que llamaremos un *flujo factible* en R = (V,E,b,c), el cual debe satisfacer las siguientes condiciones:

i) **f** es un flujo en G = (V,E).

ii) $b(\mathbf{e}) \le f(\mathbf{e}) \le c(\mathbf{e})$, $\forall e \in E$.

Veremos en lo que sigue un método que nos permitirá encontrar un flujo factible, siempre que este exista, y una condición necesaria y suficiente para su existencia.

**Proposición VII.3.1 (Teorema de Hoffman)**

Una condición necesaria y suficiente para que exista un flujo factible en R = (V,E,b,c) es que:

$$\forall \text{ cociclo } \Omega(Y) \text{ en } (V,E): \sum_{e \in \Omega^-(Y)} b(e) \le \sum_{e \in \Omega^+(Y)} c(e) \qquad \text{(II)}$$

**Demostración:**

Sea f un flujo factible en R = (V,E,b,c) y $\Omega(Y)$ un cociclo, entonces, sumando todas las ecuaciones de conservación de flujo sobre los vértices en Y obtenemos:

$$\sum_{e \in \Omega^-(Y)} f(e) = \sum_{e \in \Omega^+(Y)} f(e), \quad \text{pero } \sum_{e \in \Omega^-(Y)} b(e) \le \sum_{e \in \Omega^-(Y)} f(e) \text{ y } \sum_{e \in \Omega^+(Y)} f(e) \le \sum_{e \in \Omega^+(Y)} c(e)$$

de donde se obtiene la condición (II) buscada.

¤

El siguiente algoritmo es un algoritmo constructivo, el cual cuando no puede encontrar un flujo factible pone en evidencia la existencia de un cociclo que no cumple la condición de Hoffman, lo cual demuestra la suficiencia de la misma.

**Algoritmo para encontrar un Flujo Factible.**

**FlujoFact**(V, E, I, T, b, c; VAR **f**, Y, ExistFF)
{Entradas: V, E, I, T, b, c
Salidas: **f**, Y, ExistFF.
La variable booleana ExisteFF será verdadera sy y sólo si existe un flujo factible en (V,E,b,c).}
Comienzo
ExistFF ← VERDADERO;
f ← 0 ; {Flujo Inicial = 0 , ∀e∈E }
Repetir
INDIC ← $\sum_{e / f(e)>c(e)} [f(e)-c(e)] + \sum_{e / f(e)<b(e)} [b(e)-f(e)]$;
Si (INDIC ≠ 0 y ExistFF) entonces
Comienzo
Si (∃**ê** = (x,y) ∈ E / f(**ê**) > c(**ê**) ) entonces
Comienzo
$I(e_r)$ ← **p** ← y;
$T(e_r)$ ← **s** ← x;
$\tilde{c}(e_r)$ ← f(**ê**) - c(**ê**);
α ←**1**;
Fin
Sino { ∃ê = (x,y) ∈ E / f(**ê**) < b(**ê**) }
Comienzo
$I(e_r)$ ← **p** ← x;
$T(e_r)$ ← **s** ← y;
$\tilde{c}(e_r)$ ← b(**ê**) - f(**ê**);
α ←**-1**;
Fin **;**
Para todo e∈E hacer:
Comienzo
$\tilde{b}(e)$ ←**Min** [ b(e), f(e) ]
$\tilde{c}(e)$ ←**Max** [ f(e), c(e) ]
$\tilde{f}(e)$ ←f(e);
Fin ;
$\tilde{f}(e_r)$ ←0;
$\tilde{b}(e_r)$ ←0;
FlujoMax (V,E,I,T,$e_r$,s,p, $\tilde{b}$ , $\tilde{c}$ , $\tilde{f}$ ; $\tilde{f}$ , Y, MODIF, ExistFM);
Si ( $\tilde{f}(e_r)$ = $\tilde{c}(e_r)$ ) entonces { El flujo se pudo cambiar }
Comienzo
f(**ê**) ← $\tilde{f}$ (**ê**) - α* $\tilde{f}(e_r)$;
Para todo e ∈E / e ≠ **ê** hacer
f(e) ← $\tilde{f}$ (e);
Fin
sino ExistFF←FALSO; { El flujo no cumple la condición de Hoffman }
Fin
hasta que (INDIC = 0 ó no ExistFF).
Fin. {FlujoMax }

**Observaciones:**

Si INDIC = 0 entonces **f** es un flujo factible. Las nuevas cotas $\tilde{b}$, $\tilde{c}$ se definen de forma tal que f = $\tilde{f}$ sea factible en R = (V,E $\cup$ {$\mathbf{e_r}$}, $\tilde{b}$, $\tilde{c}$) para poder llamar a FLUJOMAX que requiere un flujo inicial factible.

El algoritmo anterior se detiene cuando INDIC = 0, es decir cuando se tiene un flujo **f** factible, o cuando ExistFF es falso.

En este segundo caso, sea Y el último conjunto de vértices "marcados" que devuelve FLUJOMAX :

a) Si $\alpha = 1$ entonces:

$\forall e \in \Omega^+(Y) - \{\hat{\mathbf{e}}\}$: $f(e) = \tilde{f}(e) = \tilde{c}(e) \geq c(e)$

$\forall e \in \Omega^-(Y)$: $f(e) = \tilde{f}(e) = \tilde{b}(e) \leq b(e)$

$f(\hat{\mathbf{e}}) = \tilde{f}(\hat{\mathbf{e}}) - \alpha * \tilde{f}(\mathbf{e_r}) > c(\hat{\mathbf{e}})$

b) Si $\alpha = -1$ entonces:

c) $\forall e \in \Omega^+(Y)$: $f(e) = \tilde{f}(e) = \tilde{c}(e) \geq c(e)$

$\forall e \in \Omega^-(Y) - \{\hat{\mathbf{e}}\}$: $f(e) = \tilde{f}(e) = \tilde{b}(e) \leq b(e)$

$f(\hat{\mathbf{e}}) = \tilde{f}(\hat{\mathbf{e}}) - \alpha * \tilde{f}(\mathbf{e_r}) < b(\hat{\mathbf{e}})$.

En ambos casos:

$$0 = \sum_{e \in \Omega^+(Y)} f(e) - \sum_{e \in \Omega^-(Y)} f(e) > \sum_{e \in \Omega^+(Y)} c(e) - \sum_{e \in \Omega^-(Y)} b(e)$$

de donde podemos concluir que f no factible $\Rightarrow \exists$ un cociclo $\Omega(Y)$ que no cumple la condición de Hoffman.

### VII.4 FLUJO DE COSTO MÍNIMO

Se plantea el problema de Flujo de Costo Mínimo de la siguiente manera:

Sea R = (V,E,a,b,c) con $a: E \to \mathfrak{R}$, $b: E \to \mathfrak{R}$, $c: E \to \mathfrak{R}$, $b(e) \leq c(e)$, $\forall e \in E$.

Se desea encontrar un vector f tal que:

i) f sea un flujo en G=(V,E)

ii) $b(e) \leq f(e) \leq c(e)$, $\forall e \in E$.

iii) $\sum_{e \in E} [a(e) * f(e)]$ sea mínimo.

Así como el problema de Flujo Máximo, el problema de Flujo de Costo Mínimo, también puede formularse como un P.P.L. Si M es la matriz de incidencia de G entonces:

$$\textbf{min} \quad \mathbf{a}^t * \mathbf{f}$$

$$\textbf{s.a.} \begin{cases} M.f = 0 \\ f \geq b \\ f \geq -c \end{cases}$$

Antes de ver como resolver este problema, veremos como otros problemas conocidos, pueden plantearse como problemas de flujo de costo mínimo.

a) Camino Mínimo de s a p en R = (V,E,d).

En este caso el planteamiento lo hacemos construyendo una nueva red $R' = (V, E \cup \{e_r\}, a,b,c)$, donde

$e_r = (p,s),\ a(e_r) = 0,\ \ b(e_r) = 1,\ c(e_r) = \infty$

$\forall e \in E : a(e) = d(e),\ \ b(e) = 0, c(e) = \infty$

b) El problema de Flujo Máximo de s a p en $R = (V, E \cup \{e_r\}, b, c)$, con $e_r=(p,s)$, se puede reducir al problema de flujo de costo mínimo como sigue:

Construiremos la nueva red $R'=(V, E \cup \{e_r\}, a,b,c)$, donde

$e_r = (p,s)\ \ a(e_r) = -1, b(e_r) = 0, c(e_r) = \infty$

$\forall e \in E : a(e) = 0, b(e) = b(e), c(e) = c(e).$

c) Problema de Transporte.

Supongamos **N** fuentes con capacidades $\mathbf{a_i}$, **M** destinos con demandas $\mathbf{b_j}$, y costos unitarios de transporte $\mathbf{c_{ij}}$ de la fuente **i** al destino **j**.

La formulación como P.P.L. de este problema es:

$$\textbf{Min} \quad \sum_i \sum_j c_{ij} X_{ij}$$

$$\textbf{s.a.} \sum_j x_{ij} \le a_i, \quad \forall i$$

$$\sum_j x_{ij} \ge b_j, \quad \forall j$$

$$x_{ij} \ge 0, \quad \forall i,j.$$

Construimos entonces la siguiente red $R=(F \cup D \cup \{(p,s)\}, E1 \cup E2 \cup E3, a,b,c)$ donde:

F = { cjto. de vértices correspondientes a cada fuente}

D = { cjto. de vértices correspondientes a cada destino}

**s** = super-fuente, **p** = super-destino o sumidero.

E1 = {(**s**,Fi) / Fi es una fuente}

E2 = {(Dj,**p**) / Dj es un destino}

E3 = {(Fi,Dj) / Fi es una fuente, Dj es un destino}

$$c(e)=\begin{cases} c_{ij} & \text{si } e=(F_i,D_j) \in E_3 \\ 0 & \text{en otro caso} \end{cases}$$

$$b(e)=\begin{cases} b_j & \text{si } e=(D_j,p) \in E_2 \\ 0 & \text{en otro caso} \end{cases}$$

$$a(e)=\begin{cases} a_i & \text{si } e=(s,F_i) \in E_1 \\ 0 & \text{en otro caso} \end{cases}$$

$e_r = (p,s)$: $a(e_r) = 0$, $b(e_r) = \sum \mathbf{b_j}$, $c(e_r) = \sum \mathbf{a_i}$

**Definición VII.4.1**

Dado un flujo **f** factible en una red R = (V,E,a,b,c), podemos construir una red $\hat{R}$ =(V, $\hat{E}$, d) donde $\hat{E} = \hat{E}' \cup \hat{E}''$, donde $\hat{E}'$ y $\hat{E}''$ son construidos como sigue:

Si f(**e**) < c(**e**) entonces crear $e' \in \hat{E}'$ con :

I(e') = I(e), T(e') = T(e), d(e') = a(e).

Si f(**e**) > b(**e**) entonces crear $e'' \in \hat{E}''$ con :

I(e") = T(e), T(e") = I(e), d(e") = -a(e)

**Proposición VII.4.1**

Un flujo factible **f** en R = (V,E,a,b,c) es de costo mínimo si y sólo si la red $\hat{R}$ (f) no tiene circuitos absorbentes.

**Demostración.**

Supongamos que $\hat{R}$ (F) si tiene un circuito absorbente C, llamemos $C^+ = C \cap \hat{E}'$, $C^- = C \cap \hat{E}''$.

Llamemos $\Gamma^+$ = arcos en E asociados a arcos en $C^+$, $\Gamma^-$ = arcos en E asociados a arcos en $C^-$. Por ser C un circuito absorbente se tiene que $\sum_{e \in C} d(e) < 0$.

Pero $\sum_{e \in C} d(e) = \sum_{e' \in C^+} d(e') + \sum_{e'' \in C^-} d(e'') = \sum_{e \in \Gamma^+} a(e) - \sum_{e \in \Gamma^-} a(e) < 0$

Sea $\varepsilon$ = **Min** { $\mathbf{Min}_{e \in \Gamma^+}$ [c(e) - f(e)], $\mathbf{Min}_{e \in \Gamma^-}$ [ f(e) - b(e)] } > 0

Podemos entonces construir un nuevo flujo **g** como:

g(**e**) = f(**e**) + $\varepsilon$, si $\mathbf{e} \in \Gamma^+$

g(**e**) = f(**e**) - $\varepsilon$, si $\mathbf{e} \in \Gamma^-$

g(**e**) = f(**e**), si $\mathbf{e} \notin (\Gamma^+ \cup \Gamma^-) = \Gamma$.

El costo de este flujo es:

$$\mathbf{a} * \mathbf{g} = \sum_{e \in E} a(e)*g(e) = \sum_{e \in \Gamma^+} a(e)*(f(e)+\varepsilon) + \sum_{e \in \Gamma^-} a(e)*(f(e)-\varepsilon) + \sum_{e \notin \Gamma} a(e)*f(e) =$$

$$= \sum_{e \in E} a(e)*f(e) + \varepsilon * \left[ \sum_{e \in \Gamma^+} a(e) - \sum_{e \in \Gamma^-} a(e) \right] < \mathbf{a} * \mathbf{f}$$

de donde se observa que **g** tiene menor costo que **f**.

Por otro lado, si $\hat{R}$ (f) no tiene circuitos absorbentes, entonces, agregando los arcos necesarios para que un cierto vértice **s**, escogido arbitrariamente, sea una raíz de $\hat{R}$ (f) y dando a estos arcos una distancia sufientemente grande M>>0, podemos asegurar que existen caminos mínimos desde **s** hasta cada vértice **x** de $\hat{R}$ (f).

De esta manera, podemos asociar los valores:

$\pi(x)$ = distancia mínima de **s** a **x**, $x \in V$, tales que:

$\forall e \in \hat{E}$: $\pi(T(\mathbf{e})) - \pi(I(\mathbf{e})) \leq d(\mathbf{e})$.

Plantearemos a continuación el dual del problema de flujo de costo mínimo:

**max** 0.X + **b**.Y - **c**Z

**s.a.** X.E + Y - Z = **a**, Y, $Z \geq \mathbf{0}$

Definamos:

$t(e) = \pi(T(e)) - \pi(I(e))$ , $\alpha(e) = \text{Max}\ [0, a(e) - t(e)]$ y $\beta(e) = \text{Max}\ [0, t(e) - a(e)]$

Así,

si $e' \in \hat{E}'$ : $\pi(T(e')) - \pi(I(e')) = \pi(T(e)) - \pi(I(e)) \leq d(e) = a(e) \Rightarrow t(e) \leq a(e)$

si e''$\in\hat{E}$'': $\pi(T(e'')) - \pi(I(e'')) = \pi(I(e)) - \pi(T(e)) \le d(e) = -a(e) \Rightarrow \quad t(e) \ge a(e)$

Veamos que si las variables duales (X, Y, Z) toman los valores ($\pi$, $\alpha$, $\beta$), esto es una solución factible.

$\alpha(e) \ge 0$, $\forall e \in E$, pues si $a(e) - t(e) < 0$ entonces $\alpha(e) = 0$.

$\beta(e) \ge 0$, $\forall e \in E$, pues si $t(e) - a(e) < 0$ entonces $\beta(e) = 0$.

La restricción dual se transforma en:

$$\pi(T(e)) - \pi(I(e)) + \alpha(e) - \beta(e) = a(e)$$

y es fácil verificar que esta igualdad siempre se cumple.

Ahora sólo falta ver que las soluciones ($\pi$, $\alpha$, $\beta$), para el dual, y **f** para el primal, cumplen las condiciones de holgura complementaria y que por lo tanto ambas son óptimas a sus respectivos problemas:

$f(e) > b(e) \Rightarrow t(e) \ge a(e) \Rightarrow \alpha(e) = 0$

$f(e) < c(e) \Rightarrow t(e) \le a(e) \Rightarrow \beta(e) = 0$

$\alpha(e) > 0 \Rightarrow a(e) > t(e) \Rightarrow f(e) = b(e)$

$\beta(e) > 0 \Rightarrow t(e) > a(e) \Rightarrow f(e) = c(e)$.

¤

El algoritmo que presentamos a continuación, conocido como algoritmo Primal, encuentra su justificación en la Proposición VII.4.1.

```
FlujoCostoMin;
{Entradas:  V, E, I, T, a, b, c, fi;
Salidas:    f, Y, ExisteFCM }.
Variable Circuitabs : Booleana;
Comienzo
    FlujoFact (V,E,I,T,b,c,fi;  f,Y,ExisteFCM);
    Si (ExisteFCM) entonces
        Repetir
            Construir  R̂ =(V,Ê,d) como en la definición VIII.4.1, con Ê= Ê'∪Ê";
            { aplicar CaminoMinGeneral a  R̂ }
            CaminoMinGeneral (V,Ê, Î,  T̂ , d, s; π, A, EsRaiz, CircuitAbs, C);
            Si (CircuitAbs) entonces
                    Comienzo
                    C+ ← C ∩ Ê' ;
                    C- ← C ∩ Ê" ;
                    Γ+ ← arcos de E asociados a C+;
                    Γ- ← arcos de E asociados a C-;
                    ε ← Min {Min_{e∈Γ+} [c(e)-f(e)],
                                Min_{e∈Γ-} [f(e)-b(e)]}; { ε > 0 }
                    Si  (ε ≠ ∞) entonces
                            Para todo  e ∈ E  hacer
                            Según e hacer
                            Comienzo
                                    e∈Γ+   :   f(e) ← f(e) + ε;
                                    e∈Γ-   :   f(e) ← f(e) - ε;
                                    e∉Γ    :   f(e) ← f(e);
                            Fin ;
                     sino   ExisteFCM ← FALSO;
                    Fin ;
        hasta que (no CircuitAbs)  ó  (no ExisteFCM);
```

sino ExisteFCM ← FALSO
Fin.

### VII.5 FLUJO MÁXIMO DE COSTO MÍNIMO

Sea R = (V, E ∪ {$\mathbf{e_r}$}, a, c) con $\mathbf{e_r}$ = (p, s) y a: E → $\Re^+$, c: E → $\Re^+$, donde $\Re^+$ representa a los reales no negativos.

El problema de hallar el "flujo máximo de costo mínimo" en R = (V, E, a, c) consiste en buscar un vector **f** tal que:

i) **f** sea un flujo en G(V,E ∪ {$\mathbf{e_r}$})

ii) $0 \le f(e) \le c(e), \forall e \in E$.

iii) f($\mathbf{e_r}$) sea máximo.

iv) $\sum_{e\in E}$ a(e)*f(e) sea mínimo bajo i), ii) y iii).

Es decir, se busca entre todos los flujos factibles en R = (V,E ∪ {$\mathbf{e_r}$}, a, c) aquellos con f($\mathbf{e_r}$) máximo, y entre ellos aquel que tenga el costo mínimo. Tenemos entonces una mezcla de dos problemas ya vistos: Flujo Máximo y Flujo de Costo Mínimo.

Sean **F** = **max** f($\mathbf{e_r}$) sobre R = (V,E ∪ {$\mathbf{e_r}$}, a, c) y **A** = $\sum_{e\in E}$ a(e) + 1.

Existen dos formas de llevar el problema de Flujo Máximo de Costo Mínimo a un problema de Flujo de Costo Mínimo :

a) Podemos considerar R'=(V, E , a', b', c') donde:

E' = E ∪ {$\mathbf{e_r}$}; a'($\mathbf{e_r}$) = -A; c($\mathbf{e_r}$) = ∞; ∀e∈E: (a'(e) = a(e), c'(e) = c(e)); ∀e∈E': b'(e) = 0

Así, dando el valor negativo - A << 0 al arco $\mathbf{e_r}$, al minimizar $\sum_{e\in E'}$ a(e)*f(e) se tratará de dar el valor más grande a f($\mathbf{e_r}$).

b) Otra forma es con R"=(V, E", a", b",c") con E" = E ∪ {$e_r$}, donde:

a"($e_r$) = 0, a"(e) = a(e), ∀e∈E

b"($e_r$) = **F**, b"(e) = 0, ∀e∈E

c"($e_r$) = ∞, c"(e) = c(e), ∀e∈E

**Definición VII.5.1**

Dado un flujo factible **f** en R = (V,E ∪ {$\mathbf{e_r}$}, a, c), llamaremos $\hat{R}$ (**f**) la red $\hat{R}$ =(V,$\hat{E}$, $\hat{d}$ ) donde:

$\hat{E} = \hat{E}' \cup \hat{E}''$ construyendo $\hat{E}'$ y $\hat{E}''$ como sigue

si e∈E con f(e) < c(e), entonces e'∈$\hat{E}'$:

I(e') = I(e), T(e') = T(e), d(e') = a(e).

si e∈E con f(e) > 0, entonces e"∈$\hat{E}''$:

I(e") = T(e), T(e") = I(e"), d(e") = -a(e).

**Observación:**

En este caso $\hat{R}$ (**f**) no tendrá arcos asociados a $\mathbf{e_r}$.

**Definición VII.5.2**

Llamamos una "Solución Parcial" al problema de Flujo Máximo de Costo Mínimo, a una solución óptima al problema P(v), v∈$\Re^+$ definido como:

**min** a * **f**

**s.a.** $M * \mathbf{f} = 0$

$f(e) \leq c(e), \forall e \in E$

$f(e_r) \geq v$

$f(e) \geq 0, \quad \forall e \in E$

Donde M es la matriz de incidencias de $(V, E \cup \{e_r\})$

El dual D(v) de este problema sería entonces:

**max** $w = v.\gamma(e_r) - \sum_e c(e).\gamma(e)$

**s.a.** $\pi(T(e)) - \pi(I(e)) - \gamma(e) \leq a(e), \forall e \in E$

$\pi(s) - \pi(p) + \gamma(e_r) = 0$

$\gamma(e) \geq 0, \forall e \in E$

**Proposición VII.5.1**

Si $(\pi, \gamma)$ es solución óptima de D(v) entonces:

$\gamma(e) = \mathbf{Max}\ e \in E\ [0; \pi(T(e)) - \pi(I(e)) - a(e)]$

**Demostración:**

$\gamma(e)$ debe ser mayor o igual a 0, $\forall e \in E$, además, en D(v) cada $\gamma(e)$ aparece en una única restricción: la restricción asociada al arco **e**. Luego se puede despejar:

$$\gamma(e) \geq \pi(T(e)) - \pi(I(e)) - a(e)$$

como $-c(e) \leq 0$ es el coeficiente de $\gamma(e)$ en la función objetivo, concluimos que el mínimo valor que puede tomar $\gamma(e)$ es $\mathbf{Max}_{e \in E}\ [0;\ \pi(T(e)) - \pi(I(e)) - a(e)]$ que corresponde a su cota inferior.

¤

**Proposición VII.5.2**

Sea **f** una solución factible a P(v), entonces:

**f** es solución óptima a P(v) **si y sólo si** $\hat{R}$ (f) no tiene circuitos absorbentes **si y sólo si** $\forall x \in V$ puede hallarse valores $\pi(x)$ tal que:

(**1**) $0 < f(e) < c(e) \Rightarrow \pi(T(e)) - \pi(I(e)) = a(e)$

(**2**) $\pi(T(e)) - \pi(I(e)) > a(e) \Rightarrow f(e) = c(e)$

(**3**) $\pi(T(e)) - \pi(I(e)) < a(e) \Rightarrow f(e) = 0.$

**Demostración:**

Supongamos que $\hat{R}$ (f) tiene un circuito absorbente C, veremos que en ese caso **f** no puede ser solución óptima a P(v).

Por ser C un circuito absorbente: $\sum_{e \in C} [d(e)] < 0.$

Sean $C^+ = C \cap \hat{E}'$, $C^- = C \cap \hat{E}''$

$\Gamma^+ = \{e \in E / e' \in C^+\}$, $\Gamma^- = \{e \in E / e'' \in C^-\}$

$\sum_{e \in \Gamma^+} a(e) - \sum_{e \in \Gamma^-} a(e) < 0$, además $e \in \Gamma^- \Rightarrow f(e) > 0$, y $e \in \Gamma^+ \Rightarrow f(e) < c(e)$

luego $\varepsilon = \mathbf{Min}\ \{\ \mathbf{Min}_{e \in \Gamma^+} [c(e) - f(e)],\ \mathbf{Min}_{e \in \Gamma^-} [f(e)]\} > 0$, y entonces el nuevo flujo **g**:

$g(e) = f(e) + \varepsilon, \quad \text{si } e \in \Gamma^+$

$g(e) = f(e) - \varepsilon, \quad \text{si } e \in \Gamma^-$

g(e) = f(e), si $e \in \Gamma^+ \cup \Gamma^-$ es factible a P(**v**) y su costo :

$$\mathbf{a}*\mathbf{g} = \sum_{e\in E} a(e)*g(e) = \sum_{e\in\Gamma^+} a(e)*(f(e)+\varepsilon) + \sum_{e\in\Gamma^-} a(e)*(f(e)-\varepsilon) + \sum_{e\in\Gamma} a(e)*f(e) = \sum_{e\in E} a(e)*f(e) + \varepsilon * \left[ \sum_{e\in\Gamma^+} a(e) - \sum_{e\in\Gamma^-} a(e) \right]$$

$< \mathbf{a} * \mathbf{f}$

Por otra parte si $\hat{R}(f)$ no tiene circuitos absorbentes, entonces podemos agregar a $\hat{R}(f)$ los arcos necesarios, con distancia M>>0, de forma que el vértice **s** sea una raíz y entonces podemos asegurar que existe distancia mínima de **s** a x, $\forall x \in V$.

En otras palabras, podemos encontrar valores $\pi(x)$, $\forall x \in V$ tales que:

$\pi(T(e)) - \pi(I(e)) \leq d(e)$, $\forall e \in E$

Así, si $e' \in \hat{E}'$ entonces $\pi(T(e')) - \pi(I(e')) \leq d(e') = a(e)$. (*)

si $e'' \in \hat{E}''$ entonces $\pi(T(e'')) - \pi(I(e'')) \leq d(e'') = -a(e)$

$\Rightarrow \pi(I(e)) - \pi(T(e)) \leq -a(e)$

$\Rightarrow \pi(T(e)) - \pi(I(e)) \geq a(e)$ (**).

Luego,

si $0 < f(e) < c(e)$, por (*) y (**): $\pi(T(e)) - \pi(I(e)) = a(e)$ q.e.d. (**1**)

si $\pi(T(e)) - \pi(I(e)) > a(e)$, negando (*) $\Rightarrow e \notin \hat{E}' \Rightarrow f(e) = c(e)$ q.e.d (**2**)

si $\pi(T(e)) - \pi(I(e)) < a(e)$, negando (**) $\Rightarrow e \notin \hat{E}'' \Rightarrow f(e) = 0$ q.e.d. (**3**)

Supongamos por último que disponemos de valores $\pi(x)$, $\forall x \in V$, que satisfacen las condiciones (**1**), (**2**), (**3**).

Definamos $g(e) = \mathbf{Max}\ [0,\ \pi(T(e)) - \pi(I(e)) - a(e)]$, $\forall e \in E$ y $g(\mathbf{e_r}) = \pi(\mathbf{s}) - \pi(\mathbf{p})$

Entonces $(\pi, g)$ es solución factible de D(v), el dual de P(v).

Las condiciones de holgura complementaria entre P(v) y D(v) serían:

i) $f(e) < c(e) \Rightarrow g(e) = 0$.

ii) $f(e) > 0 \Rightarrow \pi(T(e)) - \pi(I(e)) - g(e) = a(e)$

i') $g(e) > 0 \Rightarrow f(e) = c(e)$

ii') $\pi(T(e)) - \pi(I(e)) - g(e) < a(e) \Rightarrow f(e) = 0$

Al verificar que efectivamente **f** y $(\pi, g)$ satisfacen estas condiciones, se habrá demostrado que cada una de ellas es solución óptima a su respectivo problema.

i') Si $g(e) > 0 \Rightarrow$ ( por definición) $\pi(T(e)) - \pi(I(e)) > a(e) \Rightarrow$ ( por (**2**) ) $f(e) = c(e)$

ii') Si $\pi(T(e)) - \pi(I(e)) - g(e) < a(e) \Rightarrow$ ( por definición) $g(e) = 0 \Rightarrow \pi(T(e)) - \pi(I(e)) < a(e) \Rightarrow$ ( por (**3**) ) $\Rightarrow f(e) = 0$.

i) Negando i' ): Si $g(e) = 0 \Rightarrow f(e) < c(e)$.

ii) Negando ii' ): Si $f(e) > 0 \Rightarrow \pi(T(e)) - \pi(I(e)) - g(e) = a(e)$

¤

### VII.5.1 ALGORITMOS PARA EL PROBLEMA DE FLUJO MÁXIMO DE COSTO MÍNIMO

Formularemos a continuación dos versiones del algoritmo de busqueda de Flujo Máximo de Costo Mínimo, conocido también como Primal_Dual y propuesto por Ford & Fulkerson, basadas en cada una de las condiciones de la Proposición VII.5.2.

**Primera Versión.**

Se busca partiendo de $f^o$: $f(e) = 0, \forall e \in E$, construir una secuencia $f^1, f^2, \ldots, f^k$, factibles en $R = (V, E \cup \{e_r\}, a, c)$, donde cada $f^i(e_r)$ crezca monótonamente hasta alcanzar el valor máximo $f^k(e_r) = \mathbf{F}$. Cada solución parcial $f^i$ será óptima al problema $P(f^i(u_r))$. Al construir $\hat{R}(f^i)$ entonces ésta será sin circuitos absorbentes.

Para pasar de $f^i$ a $f^{i+1}$, se busca el camino más corto de **s** a **p** en $\hat{R}(f^i)$. Este camino corresponderá a una cadena de aumento de **s** a **p** de costo mínimo en $R = (V, E \cup \{e_r\}, a, c)$, y su distancia corresponderá al aumento mínimo en costo por unidad de flujo. Si no existe camino de **s** a **p** en $\hat{R}(f^i)$, entonces no existe cadena de aumento en $R = (V, E \cup \{e_r\}, a, c)$, luego $f^i$ es óptimo.

El algoritmo, presentado de manera informal, sería el siguiente:

**PRIMAL-DUAL (VERSIÓN 1)**: {Flujo Máximo de Costo Mínimo I};
{ Entradas: V, E, I, T, s, p, a, c, $e_r$;
Salidas: f, ExisteFMCM }.
Comienzo
f(e) ← 0, ∀e∈E ;
Camino ← VERDADERO;
ExisteFMCM ← VERDADERO;
Mientras ( Camino **y** ExisteFMCM) hacer:
Comienzo
CONSTRUIR $\hat{R}$ (f);
Si ( C tal que C es un Camino Mínimo de **s** a **p** ($C_s^p$) en $\hat{R}$ (f) ) entonces
Comienzo
- Sea Γ el conjunto de arcos que corresponden a C ∪ {$e_r$} en R=(V,E∪{$e_r$},a,c), con Γ=Γ⁺∪Γ⁻ donde Γ⁺ es el conjunto de arcos de Γ con la misma dirección que $e_r$;
ε ← **Min** [ **Min** $_{e\in\Gamma^+}$ [c(e)-f(e)], **Min** $_{e\in\Gamma^-}$ [f(e)]]; { ε>0 }
Si (ε = ∞) entonces ExisteFMCM ← FALSO
sino
Para todo e∈E hacer
Según e hacer
Comienzo
e∈Γ⁺ : f(e) ← f(e) + ε;
e∈Γ⁻ : f(e) ← f(e) - ε;
Fin ;
Fin
sino Camino ← FALSO;
Fin ;
Fin.

Ejemplo de construcción de $\hat{R}$ (f ):

Dada la red de la figura VII.2 donde los pares sobre los arcos son (a(e), c(e)), s=1, p=5.

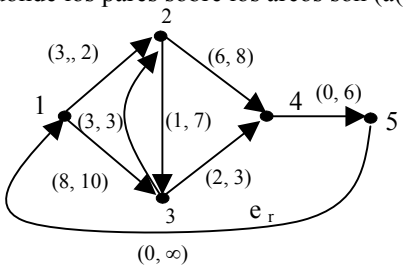



**figura VII.2**

Para el flujo f(1,2)=f(2,3)=f(3,4)=f(4,5)=f(5,1)=2; f(1,3)=f(3,2)=f(2,4)=0, $\hat{R}(f)$ sería:

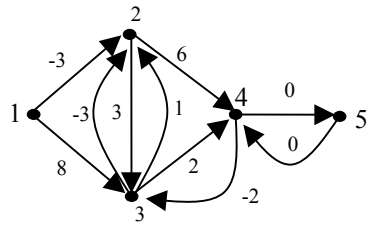



**figura VII.3**

**Segunda Versión (para el problema de Flujo Máximo de Costo Mínimo).**

Veremos a continuación una presentación algo distinta del mismo algoritmo ( además la más utilizada ). Esta otra presentación está basada en la segunda parte de la Proposición VII.5.2. La diferencia esencial es que la búsqueda del camino más corto de **s** a **p** en $\hat{R}$ (f) se simplifica gracias al conocimiento de los valores de $\pi(x)$ de la iteración anterior.

Dispondremos en cada iteración de una solución **f** óptima de $P(f(e_r))$, con valores de $\pi(x)$ que satisfacen:

(1) $0 < f(e) < c(e) \Rightarrow \pi(T(e)) - \pi(I(e)) = a(e)$.

(2) $\pi(T(e)) - \pi(I(e)) > a(e) \Rightarrow f(e) = c(e)$.

(3) $\pi(T(e)) - \pi(I(e)) < a(e) \Rightarrow f(e) = 0$.

Cada iteración consta de dos etapas:

**i) Cambio de flujo.**

Se construye R'=(V,E',c) donde $E' = \{ e \in E / \pi(T(e)) - \pi(I(e)) = a(e) \}$ y en esta red se busca el flujo máximo. Note que los arcos sobre esta red son aquellos que estarían en la arborescencia de distancias mínimas en $\hat{R}$ (f). Evidentemente f no es un flujo en R', pero sin embargo, el procedimiento de búsqueda de cadenas de aumento y de modificación del flujo permanece válido al ser trasladado a R.

**ii) Cambio de las variables duales ($\pi$).**

Cuando no se puede aumentar ya el flujo ( con f solución óptima de $P(f(u_r))$, sea Y el último conjunto de vértices marcados por FlujoMax, tenemos que:

$\mathbf{s} \in Y$, $\mathbf{p} \notin Y$.

$e \in \Omega^+(Y) \cap E' \Rightarrow f(e) = c(e)$

$e \in \Omega^-(Y) \cap E' \Rightarrow f(e) = 0$

Sean

$A^+ = \{ e \in \Omega^+(Y) / \pi(T(e)) - \pi(I(e)) < a(e) \}$

$A^- = \{ e \in \Omega^-(Y) / \pi(T(e)) - \pi(I(e)) > a(e) \}$

Si $A^+ = \varnothing \Rightarrow \forall e \in \Omega^+(Y): \pi(T(e)) - \pi(I(e)) \geq a(e) \Rightarrow f(e) = c(e)$

$A^- = \varnothing \Rightarrow \forall e \in \Omega^-(Y): \pi(T(e)) - \pi(I(e)) \leq a(e) \Rightarrow f(e) = 0$

Por lo tanto, si $A^+ \cup A^- = \varnothing$ entonces $\Omega^+(Y)$ es un corte saturado y el flujo actual es óptimo a P(**F**).

Si $A^+ \neq \varnothing$ ó $A^- \neq \varnothing$ entonces, sea

$\partial$ = **Min** { **Min** $_{e \in A^+}$ [a(e) - $\pi$(T(e)) + $\pi$(I(e))] , **Min** $_{e \in A^-}$ [$\pi$(T(e)) - $\pi$(I(e)) - a(e)]}

Es claro que $\partial > 0$ y así los valores

$\pi(x) = \pi(x)$, si $x \in Y$

$\pi(x) = \pi(x) + \partial$, si $x \in Y$

satisfacen (**1**), (**2**) y (**3**) y además representan las distancias mínimas de **s** a **x** en la nueva red $\hat{R}(f)$.

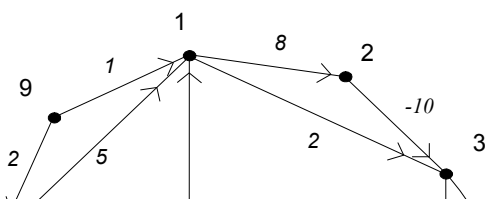
El segundo algoritmo para resolver el problema del Flujo Máximo de Costo Mínimo es entonces:

**FlujoMaxCostoMin (Versión 2)**:
{Flujo Máximo de Costo Mínimo II};
{ Entradas: V, E, I, T, s, p, $e_\Gamma$, a, c;
Salidas: f, ExisteFMCM }.
Comienzo
f(e) ← 0, ∀e∈E;
Dijkstra ( V,E,I,T,a,s; π,A,EsRaiz); { Valores iniciales de π, con a(e) > 0, ∀e }
Si (EsRaiz) entonces
Repetir
ExisteFMCM ← FALSO;
E' ← { e∈E/ π(T(e)) - π(I(e)) = a(e) };
FlujoMax (V, E', I', T', $e_\Gamma$, s, p, b, c, f; f, Y, MODIF, ExisteFM);
Si (ExisteFM) entonces { Existe un flujo máximo en la red }
Comienzo
$A^+ \leftarrow$ { $e \in \Omega^+(Y)$/ $\pi(T(e)) - \pi(I(e)) < a(e)$ };
$A^- \leftarrow$ { $e \in \Omega^-(Y)$/ $\pi(T(e)) - \pi(I(e)) > a(e)$ };
ExisteFMCM ← VERDADERO;
Fin;
Si ( $A^+ \cup A^- \neq \emptyset$) y (no ExisteFMCM) entonces
Comienzo
$\partial^+ \leftarrow$ **Min** $_{e \in A^+}[a(e) - \pi(T(e)) + \pi(I(e))]$; { Si $A^+ = \emptyset$ entonces $\partial^+ = \infty$ }
$\partial^- \leftarrow$ **Min** $_{e \in A^-}[\pi(T(e)) - \pi(I(e)) - a(e)]$; {Si $A^- = \emptyset$ entonces $\partial^- = \infty$ }
$\partial \leftarrow$ **Min** $\{\partial^+, \partial^-\}$; { $\partial > 0$ }
Para todo $x \in V$ tal que $x \notin Y$ hacer:
$\pi(x) \leftarrow \pi(x) + \partial$;
Fin ;
hasta que (no ExisteFMCM) ó ($A^+ \cup A^- = \emptyset$);
sino ExisteFMCM ← FALSO;
Fin.

La terminación de ambos algoritmos la justifican los mismos argumentos que el algoritmo de Flujo Máximo en el cual se basan.

### VII.6 EJERCICIOS

**1.** Use el algoritmo CaminoMinGeneral para encontrar la distancia y el camino más corto del vértice 6 a cualquier vértice del siguiente grafo:



**2.** Use el algoritmo CaminoMinGeneral para encontrar el camino más corto del vértice 6 a cualquier vértice del grafo del ejercicio 1 modificando el costo del arco (6,8) por –14. ¿Existe un circuito absorvente?.

**3.** ¿El siguiente grafo contiene un circuito absorvente?. Aplique el algotirmo CaminoMinGeneral.

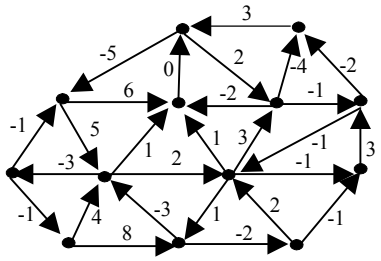



**figura VII.5**

**4.** Determinar una solución óptima del problema de flujo máximo en la siguiente red. Encontrar un corte de capacidad mínima.

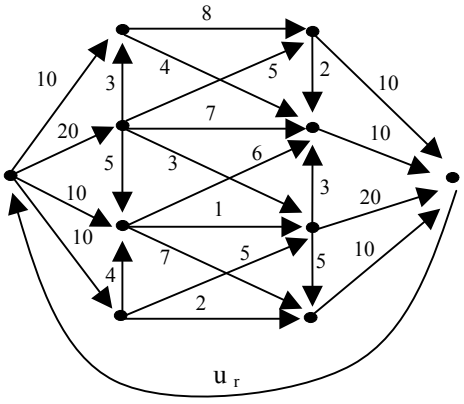
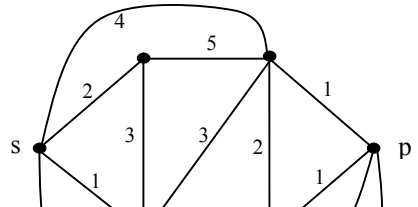



**figura VII.6**

**5.** Considere la siguiente red donde las capacidades son los números en las aristas:



a) Determine un flujo máximo de s a p.

b) Proponga una variante simple del algoritmo de Ford y Fulkerson que se aplique de manera directa en el caso de redes no orientadas.

**6.** En la red siguiente los dos números asociados a cada arco corresponden el primero y el segundo respectivamente a una cota inferior y una cota superior del valor del flujo en ese arco.

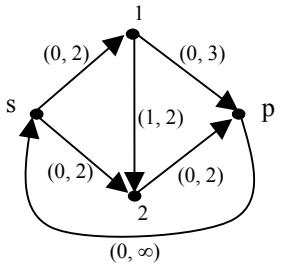



**figura VII.8**

a) Determine un flujo factible partiendo del flujo cero.

b) A partir del flujo f(s,1)=f(1,2)=f(2,p)=f($u_r$)=2, f(1,p)=f(s,2)=0, aplique el algoritmo de Ford y Fulkerson para hallar un flujo máximo.

**7.** Halle un flujo de costo mínimo en la siguiente red donde al lado de cada arco se representa (a(e), b(e), c(e)):

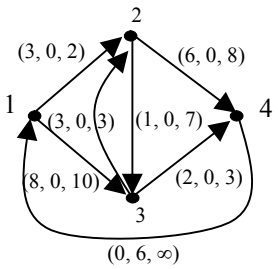



**figura VII.9**

**8.** Resuelva el siguiente problema de transporte:

| $A_{ij}$ | B1 = 15 | B2 = 20 | B3 = 30 | B4 = 35 |
|---|---|---|---|---|
| C1 = 25 | 8 | 5 | 6 | 7 |
| C2 = 25 | 8 | 2 | 7 | 6 |
| C3 = 50 | 9 | 3 | 4 | 8 |

**9.** Determine un flujo máximo de costo mínimo en la red del ejercicio 7, tomando a s=1, p=4 y reemplazando la terna de (p,s) por (0,0,6) (todo los b(e) deben ser cero como se muestra). Utilice los dos algoritmos dados.

**APÉNDICE 1**

## *GLOSARIO DE TÉRMINOS BÁSICOS*

**OBSERVACIÓN:** todas las definiciones para grafos son válidas tanto para grafos orientados como para no-orientados, a menos que se especifique lo contrario.

1. Grafo: Un *Grafo* G es una terna (V, E, $\phi$), donde V es un conjunto finito de elementos llamados *Vértices o Nodos* del grafo, E es un conjunto finito de elementos llamados *Lados* y $\phi$ es una función que asigna a cada elemento de E un par de elementos de V. Si todos estos pares no son ordenados se dice que el *grafo es no orientado o no dirigido* y sus lados se llaman *Aristas*. Cuando todos los pares {a,b} son ordenados, se denotan (a,b) y se les llama *Arcos*, en este caso se dice que el *grafo es orientado o dirigido* y se le llama también *Digrafo*. Cuando no exista confusión denotaremos un Grafo por el par (V,E) sin incluir la función $\phi$. En este caso, un lado del grafo representará en sí mismo el par que le asociaría la función $\phi$. Escribiremos por ejemplo: e={a,b}, con e en E.
2. Digrafo: Grafo dirigido. Ver grafo.
3. Grafo, representacion de: Un grafo generalmente se representa mediante un dibujo donde los NODOS aparecen como puntos y un lado {a,b} como una linea que une el punto a con el b. Si el grafo es orientado para cada arco (a,b) se coloca una flecha apuntando de a hacia b sobre la linea que une ambos puntos.
4. Extremos: Son los nodos a, b de un lado {a,b}. Si el lado es un arco (a,b), se dice que el nodo a es el nodo o extremo inicial y el nodo b el nodo o extremo terminal.
5. Digrafo, representación de: Ver representación de grafo.
6. Grafo no orientado: ver Grafo.
7. Grafo orientado o dirigido: Digrafo. Ver grafo
8. Arcos: lados de un digrafo.
9. Aristas: lados de un grafo no orientado.
10. Orden de un grafo: Es la cardinalidad del conjunto de nodos.
11. Lazo: Arco o arista donde ambos extremos coinciden, ej. (a,a) o {a,a}.
12. Bucle: Ver Lazo.
13. Grafo planar: Grafo G = (V,E) que puede ser representado sobre una superficie plana de forma tal que ningún par de lados se intersecten fuera de sus extremos.
14. Caras, de un grafo planar: Dada una representación de un grafo planar, se llaman caras del mismo a cada una de las regiones que quedan limitadas por las aristas considerando estas como fronteras.
15. Frontera de una cara, de un grafo planar G = (V,E): Dada una representación planar de G = (V,E), la frontera de una cara es el conjunto de aristas que en esa representación la delimitan.
16. Dual de un grafo planar G = (V,E): Es el grafo G* = (V*,E*), asociado a una determinada representación planar del grafo, en el cual existe una biyección entre el conjunto de nodos de G* y las caras o regiones de la representación planar del grafo original. Dos nodos de G* serán adyacentes si las regiones correspondientes son vecinas en la representación planar de G.

17. Multiplicidad de un par a,b: número de veces que el par (a,b) o {a,b}, según que el grafo sea o no dirigido, aparece en la familia E de lados del mismo.

18. Grafo simple: Un grafo se dice que es simple si todos los pares **a,b** de nodos tienen multiplicidad 1 y si el grafo no posee lazos.

19. Multigrafo: Grafo o digrafo en el cual se permiten pares de multiplicidad $\geq 1$. Ver grafo.

20. Adyacencia de nodos: Dos nodos a,b se dicen adyacentes si son extremos de un mismo lado (arco o arista). Ej. a es adyacente a b $\Leftrightarrow$ (a,b) está en E.

21. Adyacencia de lados: Dos lados (arcos o aristas) $e_1$, $e_2$ son adyacentes si tienen un extremo en común. Ej. (a,b) es adyacente a (b,c).

22. Incidencia nodos-lados: Un nodo a es incidente a un lado **e** (arco o arista) si es uno de sus extremos, i.e., **e**=(a,b) $\Rightarrow$ a y b son incidentes a **e**. En forma similar, un lado **e** es incidente a un nodo a si uno de sus extremos coincide con a, i.e., **e**=(a,b) es incidente al nodo a y al b.

23. Grado interior de un nodo: En un digrafo, es el número $d^-(x)$ de arcos cuyo extremo final coincide con el nodo dado x.

24. Grado exterior de un nodo: En un digrafo, es el número $d^+(x)$ de arcos cuyo extremo inicial coincide con el nodo dado x.

25. Grado de un nodo: Es el número d(x) de lados incidentes a ese nodo, donde un lazo se considera doblemente incidente. En un digrafo $d(x) = d^+(x) + d^-(x)$ (ver grado interior, exterior). Se denota $\partial(G) = \min_{(x \text{ en } V)} d(x)$ y $\Delta(G) = \max_{(x \text{ en } V)} d(x)$.

26. Sucesor: Un nodo b es sucesor del nodo a en un digrafo G = (V,E), si el arco (a,b) está en E. Suc(x) = conjunto de sucesores de x.

27. Predecesor: Un nodo a es predecesor del nodo b en un digrafo G = (V,E), si el arco (a,b) está en E. Pred(x) = conjunto de predecesores de x.

28. Fuente: Es un nodo que no tiene predecesores, es decir, un nodo x con $d^-(x)=0$.

29. Pozo o Sumidero: Es un nodo que no tiene sucesores, es decir, un nodo x con $d^+(x)=0$.

30. Vecindad de un nodo: Conjunto de nodos adyacentes al nodo dado. Se denota usualmente Ady(x)

31. Vecindad de un conjunto de nodos: Dado un subconjunto A de nodos de un grafo G = (V,E), la vecindad del conjunto A se define como la unión de las vecindades de cada uno de los nodos en A menos el conjunto A, es decir :

$$Ady(A) = U_{(x \text{ en } A)} Ady(x) - A.$$

32. Conjunto de adyacencias: ver Vecindad.

33. Nodo aislado: Nodo cuya vecindad es el conjunto vacío o el mismo nodo.

34. Digrafo simétrico: Un digrafo simple G = (V,E) es simétrico si para todo arco (a,b) en E $\Rightarrow$ (b,a) está en E.

35. Digrafo anti-simétrico: Un digrafo G = (V,E) es anti-simétrico si (a,b) en E y (b,a) en E $\Rightarrow$ a=b.

36. Digrafo transitivo: Un digrafo G = (V,E) es transitivo si (a,b) en E, (a,c) en E $\Rightarrow$ (a,c) en E.

37. Grafo completo: Un grafo simple no orientado G = (V,E) es completo si todo par de nodos a,b son adyacentes. Se denota $K_n$ al grafo completo de n nodos.

38. Torneo (Tournois): Es un digrafo G = (V,E) anti-simétrico sin lazos ni arcos repetidos , tal que todo par a,b de nodos son adyacentes.

39. Grafo irreflexivo: Grafo sin lazos.

40. Grafo bipartito: Un grafo G = (V,E) es bipartito si el conjunto de nodos V puede particionarse en dos subconjuntos V1, V2 de forma que los nodos pertenecientes a un mismo subconjunto Vi sean dos a dos no-adyacentes. Se denota a veces G = ( V1, V2, E ).

41. Grafo bipartito completo: Grafo simple bipartito G = (V,E) con V = V1 + V2 tal que entre todo par de nodos a,b, con a en V1 y b en V2 existe un lado en E que los une. Se denota Kp,q al grafo no-orientado bipartito completo con |V1|=p, |V2|=q.

42. Grafo cúbico: Es un grafo donde todos los nodos tienen grado 3.

43. p-grafo ( idem. k-grafo, etc): Es un digrafo tal que todos sus nodos tienen grado exterior menor o igual a p.

44. Grafo regular: Es un grafo donde todos los nodos tienen igual grado.

45. Subgrafo o Subgrafo parcial: de un grafo G = (V,E), es un grafo G' = (A,F) donde A es un subconjunto de V, y F es un subconjunto de lados en E cuyos extremos están ambos en A.

46. Grafo parcial: Es un subgrafo G' = (A,F) de un grafo G = (V,E) donde el conjunto A de nodos es igual a V.

47. Subgrafo engendrado por un subconjunto de nodos A: Es el subgrafo $G_A$ = (A,F) de G = (V,E), donde F es el subconjunto de E de lados cuyos extremos están ambos en A. Se denota a veces simplemente $G_A$.

48. Subgrafo engendrado por un subconjunto de lados F: Es el subgrafo G = (A,F), donde los nodos son los extremos de los lados en F. A veces se denota como G(F).

49. Cadena entre dos nodos $x_o$, $x_n$: es una secuencia alternada de nodos y lados :
$< x_o, e_1, x_1, ...,x_{n-1}, e_n, x_n >$ tal que para todo i=1,...,n las extremidades de $e_i$ son $x_{i-1}$ y $x_i$.

50. Camino entre dos nodos $X_o$, $X_n$: es una secuencia alternada de nodos y arcos:
$< x_o, e_1, x_1, ...,x_{n-1}, e_n, x_n >$ tal que para todo i=1,...,n el extremo inicial de $e_i$ es $x_{i-1}$ y el extremo terminal es $x_i$.

51. Cadena (Camino) simple: Es una cadena (camino) donde ninguno de los lados (arcos) que la conforman aparece más de una vez en la secuencia.

52. Cadena (Camino) elemental: Es una cadena (camino) en la cual cada vértice aparece de la secuencia que lo define a lo sumo una vez.

53. Ciclo: es un cadena simple donde los nodos inicial y final coinciden, i.e., $x_o=x_n$. Un ciclo es una cadena simple circular o cerrada.

54. Pseudo-ciclo: es una cadena donde los nodos inicial y final coinciden, en la cual se admiten arcos repetidos en la secuencia, incluso consecutivamente.

55. Circuito: es un camino simple donde los nodos inicial y final coinciden, i.e., $x_0=x_n$. Un circuito es un camino simple circular o cerrado.

56. Ciclo (Circuito) elemental: Es un ciclo (cicuito) donde todos los vértices distintos de $x_o$ y $x_n$ aparecen a lo sumo una vez en la secuencia $<x_0, e_0, \dots, x_n>$ que lo define.

57. Cuerda, de un ciclo: Es una arista que une dos vértices no adyacentes del ciclo.

58. Longitud de un camino o cadena: Es el número de lados de la secuencia que define el camino o la cadena.

59. Longitud de un circuito o ciclo: Es el número de lados de la secuencia que define el circuito o el ciclo.

60. Clausura transitiva de un grafo: Es el digrafo $G^+$ = (V,F) tal que para todo par de nodos a,b de V existirá un arco (a,b) en F de $G^+$ si y sólo si existe un camino de longitud $\geq 1$ en G desde a hasta b.

61. Grafo conexo: Es un grafo tal que entre todo par de nodos existe una cadena que los une.

62. Grafo fuertemente conexo: Es un digrafo tal que entre todo par de nodos a,b existe un camino de a hacia b.

63. Componentes conexas: Son los subgrafos $G_c$ = (C,F) conexos de G = (V,E), tales que C es maximal bajo esta propiedad.

64. Componentes fuertemente conexas: Son los subgrafos $G_c$ = (C,F) fuertemente conexos de G = (V,E), tales que C es maximal bajo esta propiedad.

65. Grafo de Precedencia: Es un digrafo sin circuito.

66. Nudo: Es un nodo de un grafo fuertemente conexo de grado estrictamente mayor que dos.

67. Anti-nudo: Es un nodo de un grafo fuertemente conexo que no es un nudo, es decir que su grado es igual a dos.

68. Conjunto de articulación: Es un subconjunto A de nodos de un grafo G = (V,E) tal que la supresión de A y de los lados adyacentes a los nodos de A, aumenta el número de componentes conexas del grafo. Si G = (V,E) es conexo, se llama PIEZAS (relativas a A) a los subgrafos conexos $G_{AUCi}$, donde $G_{Ci}$ son las componentes conexas de $G_{V-A}$.

69. Punto de articulación: Es un nodo o vértice cuya supresión aumenta el número de componentes conexas del grafo.

70. Bloque: Es un conjunto de A de nodos de un grafo G = (V,E) que genera un subgrafo $G_A$ conexo, sin puntos de articulación y maximal bajo esta propiedad.

71. Istmo: Es un lado cuya supresión aumenta el número de componentes conexas del grafo.

72. Clique: Un subgrafo completo. Se denota w(G) la cardinalidad de la clique máxima de un grafo G = (V,E).

73. Grafo k-conexo (lado-conexo) : Grafo conexo que permanece conexo tras la supresión de cualquier número menor que k de nodos (lados).

74. Conexidad ó Conectividad (lado-conexidad ó lado-conectividad): de un grafo conexo, es el número mínimo de nodos (lados) necesarias para desconectar el grafo.

75. Grafo k-fuertemente conexo: es un digrafo que permanece fuertemente conexo tras la supresión de cualquier número menor que k de nodos.

76. Conjunto estable: Subconjunto de nodos S de un grafo G = (V,E) tal que sus elementos son dos a dos no-adyacentes.

77. Conjunto independiente: Ver conjunto estable.

78. Número de estabilidad de un grafo: Es la cardinalidad del máximo conjunto estable. Se denota usualmente como $\alpha(G)$.

79. Grafo $\alpha$-crítico: Es un grafo G = (V,E) tal que la supresión de cualquiera de sus lados aumenta en uno el numero de estabilidad, es decir que, para todo lado e en E: $\alpha(G' = (V,E-e)) = \alpha(G) + 1$

80. Número de independencia: Ver número de estabilidad.

81. Conjunto Dominante: Es un subconjunto D del conjunto de nodos de un grafo G = (V,E) tal que: Para todo nodo x en V-D, existe un lado incidente a x (por su extremo inicial en digrafos) cuyo otro extremo (extremo final en digrafos) esta en D.

82. Conjunto absorbente: Ver conjunto dominante.

83. Núcleo (Kernel): Es un subconjunto del conjunto de nodos de un grafo que es a la vez estable y dominante.

84. Partición: de un conjunto D, es un conjunto de subconjuntos no vacíos de D disjuntos dos a dos tales que su unión es D.

85. Recubrimiento: de nodos por aristas o arcos en un grafo G = (V,E), es un subconjunto F de E tal que, para todo nodo x en V existe una arista o arco en F adyacente a x.

86. Cubrimiento: de un conjunto R, es un conjunto de subconjuntos de R tales que su unión es R.

87. Apareamiento: Es un subconjunto F del conjunto E de lados de un grafo,tal que los elementos de F son dos a dos no adyacentes.

88. Apareamiento perfecto, en un grafo G = (V,E): Es un apareamiento tal que todo nodo x en V es adyacente a un lado en F, es decir, un cubrimiento F de E, tal que todo par de elementos de F son no-adyacentes.

89. Transversal: Es un subconjunto T de nodos de un grafo G = (V,E) tal que cada lado del grafo es incidente a por lo menos uno de los nodos en T. Se dice que un trasversal en un recubrimiento de los lados por los nodos.

90. Cadena Alternada, con respecto a un apareamiento: Dado un apareamiento F, en un grafo G = (V,E), una cadena alternada con respecto al apareamiento es una cadena elemental tal que sus lados están alternadamente en F y en E-F.

91. Árbol: Grafo conexo sin ciclos.

92. Árbol Cobertor, de un grafo G = (V,E): Es un grafo parcial A = (V,F) de G, es decir, cuyos nodos son los mismos de G, que es un árbol.

93. Raíz: Es un nodo x de un digrafo G = (V,E) tal que para todo nodo y≠x en V, existe un camino de x a y.

94. Anti-raíz: Es un nodo x de un digrafo G = (V,E) tal que, para todo nodo y≠x en V, existe un camino de y hacia x en G = (V,E).

95. Arborescencia: Es un árbol orientado con una raiz.

96. Bosque: Es un grafo donde cada componente conexa es un árbol.

97. Floresta: Ver Bosque.

98. Coloración de los nodos de un grafo: es una partición del conjunto de nodos del grafo en subconjuntos estables.

99. Coloración de los lados de un grafo: Es una partición del conjunto de lados en subconjuntos tales que cada uno de ellos es un apareamiento.

100. Número cromático: es la cardinalidad de una coloración mínima de los nodos, es decir la mínima partición del conjunto de nodos en conjuntos estables. Se denota usualmente como $\delta(G)$.

101. Índice cromático, de un grafo G = (V,E): Es la cardinalidad de una coloración mínima de los lados, es decir, es la cardinalidad de una partición mínima:
$E_1, \ldots, E_q$ de E, tal que cada uno de los $E_i$ es un apareamiento. Se denota usualmente q(G).

102. Grafo complementario, de un grafo simple G = (V,E): es el grafo simple $G^-(V,F)$ donde F={{a,b}/a,b están en V y {a,b} no está en E}.

103. Digrafo simétrico, de un digrafo G = (V,E): es el digrafo $G^{-1}= (V,E^{-1})$ donde
$E^{-1}$={(b,a)/a,b,están en V y (a,b) está en E}.

104. Grafo adjunto, de un grafo no orientado G = (V,E): Es el grafo no orientado G'(W,F), tal que existe una correspondencia uno a uno entre el conjunto de aristas E de G y el conjunto de nodos W de G', y dos nodos de W seran adyacentes en G' si y sólo si las aristas correspondientes son adyacentes en G.

105. Distancia entre dos nodos: en un grafo orientado (no-orientado), es la longitud del camino (cadena) más corto que los une. Se denota $\partial_G(x,y)$.

106. Diámetro, de un grafo: es el máximo de las distancias (mínimas) entre cualquier par de nodos, i.e., Max $\{\partial_G(x,y)/$ x,y están en V$\}$. Se denota $\partial$ (G).

107. Distanciamiento, de un nodo x: Dado un grafo G = (V,E), es el máximo de las distancias (mínimas) desde x hasta el resto de los nodos del grafo. Se denota como $e(x) = \text{Max}_{(y \text{ en } X)} \partial_G(x,y)$.

108. Excentricidad, de un nodo x: Ver distanciamiento.

109. Centro, de un grafo: Es un nodo con distanciamiento mínimo.

110. Radio, de un grafo: Es el distanciamiento mínimo del mismo, es decir, el distanciamiento de un centro del grafo.

111. Hamiltoniano, camino o cadena: Camino o cadena elemental que pasa por todos los nodos del grafo.

112. Hamiltoniano, circuito o ciclo: Circuito o ciclo elemental que pasa por todos los nodos del grafo.

113. Grafo hamiltoniano: Un grafo (digrafo) es hamiltoniano si posee un ciclo (circuito) hamiltoniano.

114. Euleriano, camino, cadena, ciclo o circuito: Un camino, cadena, ciclo o circuito de un grafo se dice que es euleriano si cada uno de los elementos del conjunto de lados E aparece una y sólo una vez en la secuencia que lo define.

115. Grafo Euleriano: Un grafo (digrafo) es euleriano si posee un ciclo (circuito) euleriano.

116. Factor: Es una familia de ciclos elementales disjuntos dos a dos, tales que todo nodo del grafo pertenece a uno y solo uno de estos ciclos.

117. Cociclo: Dado un subconjunto A de nodos de un grafo G = (V,E), un cociclo es el conjunto de lados del grafo que tienen uno de sus extremos en A y el otro en V-A. Se denota $\Omega(A)$. Si el grafo es dirigido, entonces se define $\Omega^+(A)$ como el conjunto de arcos tales que su extremo inicial esta en A y su extremo terminal esta en V-A. Igualmente se define $\Omega^-(A)$ como el conjunto de arcos tales que su extremo inicial está en V-A y su extremo terminal en A.

118. Cociclo elemental: Es un cociclo minimal, es decir, un cociclo que no contiene ningún subconjunto propio de arcos que a su vez sea un cociclo.

119. Cocircuito: Es un cociclo $\Omega(A)=\Omega^+(A) \cup \Omega^-(A)$ donde uno de los conjuntos $\Omega^+(A)$, $\Omega^-(A)$ es vacío.

120. Conjunto desconectante: Es un subconjunto F de arcos de un grafo G = (V,E) tal que la supresión de F aumenta el número de componentes conexas del grafo.

121. Corte que separa dos nodos a y b, en un digrafo G = (V,E): Es un subconjunto F de arcos del grafo tal que existe un subconjunto A de nodos con $F = \Omega^+(A)$, a en A y b en V-A.

122. Clase de grafos: Es un conjunto de grafos que satisfacen ciertas propiedades.

123. Isomorfismo de grafos: Dos grafos G = (V,E) y H = (W,F) se dicen isomorfos si existe una biyección f de V en W y otra de E en F, tal que $g(\{a,b\}) = \{f(a), f(b)\}$.

124. Parámetro de una clase de grafos: Es una función tal que su dominio es la clase, su rango generalmente es un cuerpo y para todo par de grafos isomorfos en la clase, el valor de la función es el mismo.

125. Invariante de una clase de grafos: Ver parámetro de una clase de grafos.

**APÉNDICE 2**

## *DICCIONARIO DE TÉRMINOS*

| ESPAÑOL | INGLÉS | FRANCÉS | Nº término de Apéndice 1 |
|---|---|---|---|
| adyacencia, de lados | edge-adjancency | arête-adjacence | 21 |
| adyacencia, de nodos | vertex-adjacency | sommet-adjacence | 20 |
| anti-nudo | anti-node | anti-nœud | 67 |
| anti-raíz | anti-root | anti-racine | 94 |
| apareamiento | matching | couplage | 87 |
| apareamiento perfecto | perfect matching | couplage parfait | 88 |
| árbol | tree | arbre | 91 |
| árbol cobertor | spanning tree | arbre maximal ou arbre couvrant | 92 |
| arborescencia | arborescence | arborescence | 95 |
| arco | arc | arc | 8 |
| arista | edge | arête | 9 |
| bloque | block | bloc | 70 |
| bosque | forest | forêt | 96 |
| bucle | loop | boucle | 12 |
| cadena alternada | alternating chain | chaîne alternée | 90 |
| " elemental | elementary " | " élémentaire | 52 |
| " entre dos nodos | chain | chaîne | 49 |
| " simple | simple " | " simple | 51 |
| camino elemental | elemantary path | chemin élémentaire | 52 |
| " entre dos nodos | path | chemin | 50 |
| " simple | simple " | " simple | 5 |
| cara, de grafo planar | face | face | 14 |
| centro | centre | centre | 109 |
| ciclo | cycle | cycle | 53 |
| " elemental | elementary cicle | cycle élémentaire | 56 |
| circuito | circuit | circuit | 55 |
| " elemental | elementary " | " élémentaire | 56 |
| clase de grafos | class of graghs | classe de graphes | 122 |
| clausura transitiva | transitive closure | fermeture transitive | 60 |
| clique | clique | clique | 72 |
| cociclo | cocycle | cocycle | 117 |
| " elemental | elementary cocycle | " élémentaire | 118 |
| cocircuito | cocircuit | cocircuit | 119 |
| coloración, de lados | edge-coloring | coloration des arêtes | 99 |
| " , de nodos | vertex-coloring | " des sommets | 98 |
| componentes conexas | connected component | composant connexe | 63 |
| " fuertemente conexas | strongly connected component | " fortement connexe | 64 |
| conexidad | vertex-connectivity | connectivité | 74 |
| conjunto absorbente | dominating set | ensemble absorbant | 82 |
| " de Adyacencias | neighbourhood or adjacency set | voisinage | 32 |
| " de articulación | cut-set | ensemble d'articulation | 68 |
| " desconectante | edge-cut set | coupe | 120 |
| " dominante | ensemble absorbant | dominating set | 81 |
| " estable | independent or stable set | stable | 76 |
| " independiente | independent set or stable set | stable | 77 |
| corte | edge-cut set | coupe | 121 |
| cubrimiento | cover | recouvrement | 86 |

| | | |
|---|---|---|
| cuerda | chord | corde, diagonal 57 |
| diámetro | diameter | diamètre 106 |
| digrafo | digraph | graph orienté 2 |
| " anti-simétrico | anti-symmetric digraph | graph orienté anti-symétrique 35 |
| " simétrico | symmetric digraph | " " symétrique 34 |
| " transitivo | transitve " | " " transitif 36 |
| ",representación de | digraph representation | graph orienté, representation de 5 |
| distancia, entre nodos | distance | écart 105 |
| distanciamiento | associated number | écartement 107 |
| dual, de grafo planar | dual, of a planar graph | dual d'un graphe planaire 16 |
| euleriano, cadena | eulerian chain | chaîne eulérienne 114 |
| " , camino | " path | chemin eulérien 114 |
| " , ciclo | " cycle | cycle " 114 |
| " , circuito | " circuit | circuit " 114 |
| excentricidad | eccentricity | écartement 108 |
| extremos | endpoints | extremités 4 |
| factor | factor | facteur 116 |
| floresta | forest | forêt 97 |
| frontera, de una cara | boundary | frontière 15 |
| fuente | source | souce 28 |
| grado | degree | degrée 25 |
| grado exterior | outer-degree | demi-degré extérieur 24 |
| grado interior | inner-degree | demi-degré interieur 23 |
| grafo | graph | graphe 1 |
| " α–critico | α–critical graph | " α–critique 79 |
| " adjunto | adjoint, line " | " adjoint 104 |
| " bipartio | bipartite " | " biparti 40 |
| " bipartito completo | complete bipartite | " biparti complet 41 |
| " complementario | complementary graph | " complémentaire 102 |
| " completo | complete " | " complet 37 |
| " conexo | connected " | " connexe 61 |
| " cúbico | cubic " | " cubique 42 |
| " de precedencia | precedence or acircuitic graph (DAG) | " des precedences ou sans circuit 65 |
| " dirigido | directed graph | " orienté 7 |
| " euleriano | eulerian " | " eulerien 115 |
| " fuertemente conexo | strongly connected " | " fortement connexe 62 |
| " hamiltoniano | hamiltonian graph | " hamiltonien 113 |
| " irreflexivo | irreflexive " | " irreflexif 39 |
| " k-conexo | k-connected " | " k-connexe 73 |
| " k-fuertemente conexo | k-strongly connected graph | " k-fortement connexe 75 |
| " no orientado | non-oriented graph | " non-orienté 6 |
| " orientado | oriented " | " orienté 7 |
| " parcial | partial " | " partiel 46 |
| " planar | planar " | " planaire 13 |
| " regular | regular " | " regulier 44 |
| " simple | simple " | " simple 18 |
| " , representación de | representation of a graph | " , representation de 3 |
| hamiltoniano, cadena | hamiltonian chain | chaîne hamiltonienne 111 |
| " , camino | " path | chemin hamiltonien 111 |
| " , ciclo | " cycle | cycle " 112 |
| " , circuito | " circuit | circuit " 112 |
| incidencia nodos-lados | vertex-edge incidence | incidence sommet-arête 22 |
| índice cromático | chromatic index | indice chromatique 101 |
| invariante | | invariant 125 |
| isomorfismo | isomorphisme | isomorphisme 123 |
| istmo | isthmus, cut-edge | isthme 71 |
| k-grafo | k-graph | k-graph 43 |
| kernel | kernel | noyau 83 |
| lado-conexidad | edge-connectivity | connexite 74 |
| lazo | loop | boucle 11 |
| longitud de un camino o cadena | length of a path or chain | longueur d'un chemin ou une chaîne 58 |
| longitud de un ciclo o circuito | length of a cycle o circuit | " d'un cycle, circuit 59 |
| multigrafo | multigraph | multigraphe 19 |
| multiplicidad de un par | multiplicity of a pair | multiplicité d'une paire 17 |
| nodo aislado | isolated vertex | sommet isolé 33 |

| | | | |
|---|---|---|---|
| núcleo | kernel | noyau | 83 |
| nudo | node, junction point | nœud, carrefour | 66 |
| número cromático | chromatic number | nombre chromatique | 100 |
| numero de estabilidad | stability number | nombre de stabilité | 78 |
| " de independencia | independence number | " " " | 80 |
| orden de un grafo | order of a graph | ordre d'un graphe | 10 |
| p-grafo | p-graph | p-graphe | 43 |
| parámetro | parameter | paramétre | 124 |
| partición | partition | partition | 84 |
| pozo | sink | puits | 29 |
| predecesor | predecessor | prédécesseur | 27 |
| pseudo-ciclo | pseudo-cycle | pseudo-cycle | 53 |
| punto de articulación | cut-vertex or articulation point | point d'articulation | 69 |
| radio | radius | rayon | 110 |
| raíz | root | racine | 93 |
| recubrimiento de nodos por lados | covering of nodes by edges | recouvrement des sommets par arêtes | 85 |
| subgrafo | sub-graph | sous-graphe | 45 |
| " engendrado por lados | (partial)subgraph induced by edges | sous-graphe induit par des arêtes | 48 |
| " engendrado por nodos | subgraph induced by a set of vertices | sous-graphe induit par des sommets | 47 |
| subgrafo parcial | partial sub-graph | sous-graphe partiel | 45 |
| sucesor | successor | successeur | 26 |
| sumidero | sink | puits | 29 |
| torneo | tournament | tournois | 38 |
| transversal | transversal | transversal | 89 |
| vecindad, de un conjunto de nodos | neighbourhood of a set of nodes | voisinage d'un ensemble de sommets | 31 |
| vecindad, de un nodo | neighbourhood of a vertex | voisinage d'un sommet | 30 |

## *BIBLIOGRAFÍA*


**[1]** Aho, Hopcroft, Ullman. *Data Structures and Algorithms*. Addison-Wesley. 1983.

**[2]** Aho, Hopcroft, Ullman. *The Design and Analysis of Computer Algorithm.* Addison-Wesley. 1974.

**[3]** Berge Claude. *Graphes et Hypergraphes*. Dunod. 1970.

**[4]** Brassard G., Bratley P. *Fundamentos de Algoritmia*. Prentice Hall. 1997.

**[5]** Crema A., Sanchez A. *Acerca del Rango-Circuito de un Grafo*. Publicación Nº 87-05. Fac. de Ciencias. Escuela de Computación. U.C.V. Agosto 1987.

**[6]** Edmonds Jack. *Optimum Branchings*. Lectures in Applied Mathematics. Vol. 2 AMS. pp 346-361. 1968.

**[7]** Fernández J.A.: *Notas sobre un lenguaje algorítmico*. notas de curso. 1987.

**[8]** Gonzaga Clovis. *Busca de Caminhos em Grafos e Aplicaçoes*. I. Reuniao de Matemática Aplicada. IBM. Rio de Janeiro 1978.

**[9]** Jensen P., Barnes W. *Network Flow Programming*. John Wiley & Sons. 1980.

**[10]** Knuth, D.E. *The Art of Computer Programming Volume 1: Fundamental Algorithms*. Addison-Wesley. 1968.

**[11]** Meza O. *Notas del curso de Matemáticas Discretas*. U.C.V. 1985.

**[12]** Ortega M. *Grafos y Algoritmos*. Publicación Interna de la Univ. Metropolitana. 1986.

**[13]** Sakarovitch M. *Optimisation dans les Réseaux*. Publicación I.M.A.G., Grenoble, Francia. 1979.

**[14]** Sincovec, Wiener. *Data Structures using Modula 2*. John Wiley & sons. 1986.

**[15]** Wirth N. *Algoritmos y Estructuras de Datos* Prentice Hall. 1987.

**[16]** Xuong N.H. *Graphes: Théorie, Algorithmes et Applications*. Publicación del I.M.A.G., Grenoble,. Francia. 1984.