TEORIA DE GRAFICAS

INTRODUCCIÓN

La teoría de gráficas o teoría de grafos es aplicada en una gran cantidad de áreas tales como ciencias sociales, lingüística, ciencias físicas, ingeniería de comunicación, y otras. La teoría de grafos también juega un papel importante en varias áreas de la ciencia de la computación, tales como teoría de cambio y lógica de diseño, inteligencia artificial, lenguajes formales, gráficos por computadora, sistemas operativos, compiladores, y organización y recuperación de información.

CONCEPTOS BÁSICOS DE LA TEORIA DE GRAFOS

La terminología usada en la teoría de grafos no es estándar. No es poco usual encontrar que varios términos diferentes son usados como sinónimos. Esta situación, de cualquier manera, llega a ser más complicada cuando descubrimos que un término en particular es usado por diferentes autores para referirse a conceptos desiguales. Esta situación es natural debido a la gran diversidad de campos en los que la teoría de gráficas se aplica.

En esta sección debemos definir un grafo como un sistema matemático abstracto. De cualquier manera, para dar algo de sentido a la terminología usada y también para desarrollar algunas ideas intuitivas, se representará un grafo por medio de un diagrama. Ese diagrama se llamará igualmente grafo. Las definiciones y términos aquí presentados no están restringidos a aquellos grafos que pueden ser representados por medio de diagramas, aunque parezca ser el caso ya que estos términos tengan una fuerte asociación con dicha representación. Debemos resaltar que una representación diagramática es posible únicamente en casos muy simples. Los métodos alternativos para representar grafos serán igualmente discutidos.

Definiciones básicas

Primeramente se consideraran varios grafos que son presentados por medio de diagramas. Algunos de estos grafos pueden ser considerados como grafos de cierta relación, pero hay algunos otros que no pueden ser interpretados de esta manera.

Considérese el diagrama de la figura 1.1. Para el propósito de nuestro estudio, estos diagramas representarán grafos. Nótese que cada diagrama consiste de un conjunto de elementos que son representados por puntos o círculos y están en ocasiones etiquetados como $v_1, v_2, \ldots,$ o $1,2,\ldots.$ También, en cada diagrama ciertas parejas de dichos puntos están conectados por líneas o arcos.

Definición. Un grafo $G = \langle V, E, \varphi \rangle$ consiste de un conjunto *V* no vació llamado el conjunto de *nodos (puntos, vértices)* del grafo, donde *E* es el conjunto de los ejes del grafo, y φ es un mapeo del conjunto de ejes *E* a un conjunto ordenado o desordenado de elementos pares de *V*.

Debemos asumir de lo anterior que ambos, los conjuntos *V* y *E* de un grafo son finitos. Sería conveniente denotar un grafo *G* como $\langle V, E \rangle$, o simplemente como G. Nótese que la definición de grafo implica que para cada eje del grafo *G* podemos asociar una pareja ordenada o no ordenada de nodos pertenecientes al grafo. Si un eje $x \in E$ está asociado con un par ordenado *(u, v)* o un par no ordenado *(u, v)*, donde $u, v \in V$, entonces se dice que el eje *x* conecta o une los nodos *u* y *v*. Cualquier par de nodos que estén conectados por un eje en un grafo son llamados **nodos adyacentes.**

Definición. En un grafo *G = (V,E)*, un eje que esté asociado a un par ordenado de *V x V* es llamado **eje dirigido de G,** mientras un eje que esté asociado a un par no ordenado de nodos es llamado **eje no dirigido**. Un grafo del cual cada nodo es dirigido es llamado **dígrafo** o **grafo dirigido.** Un grafo en el cual cada eje es no dirigido es llamado **grafo no dirigido.** Si algunos ejes son dirigidos y otros no dirigidos en un grafo, se trata de un grafo mixto.

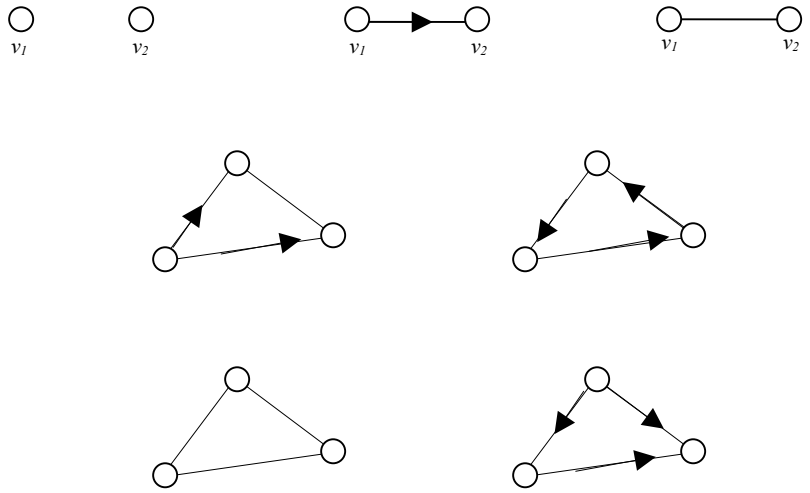


Fig.1 Los grafos (b), (e) y (g) son grafos dirigidos. Los grafos mostrados en (c) y (f) son no dirigidos. El grafo (e) es mixto. El grafo de (a) puede considerarse como dirigido y no dirigido.

El mapa de una ciudad que muestra solamente un sentido de las calles es un ejemplo de grafo dirigido en el que losa nodos son las intersecciones y los ejes son las calles.

Un mapa que muestra solamente los dos sentidos de las calles es un ejemplo de grafo no dirigido

Mientras que un mapa que muestra todo (un sentido y dos sentidos en las calles) es un ejemplo de grafo mixto.

Sea (V,E) un grafo y sea $x \in E$ un eje dirigido asociado con el par ordenado de nodos (u,v). Entonces se dice que el eje x se inicializa u origina en el nodo u y finaliza o termina en el nodo v. Los nodos u y v son también llamados el nodo inicial y final de x. El eje x € que uno los nodos u y v, puede ser dirigido o no dirigido, se dice que es incidente a los nodos u y v.

El eje de un grafo que une a un nodo consigo mismo es conocido como (LOOP OR SLING).

Los grafos mostrados en la figura 1 no tienen más de un eje entre cada par de nodos. En el caso de ejes dirigidos, los dos posibles ejes entre un par de nodos que son opuestos en dirección son considerados distintos. En algunos grafos dirigidos, así como en algunos no dirigidos, podemos tener pares de nodos unidos por más de un eje, como se muestra en la figura 2a y b. Estos ejes son llamados paralelos. Nótese que no hay ejes paralelos en la figura 2c.

En la figura 2a hay dos ejes hay dos ejes paralelos que unen a los nodos 1 y 2, dos ejes paralelos que unen a los nodos 2 y 3, mientras hay dos circuitos que paralelos en 2. en 2c, hay dos ejes paralelos asociados con la pareja ordenada $\langle v_1, v_2 \rangle$.

Cualquier grafo que contenga ejes paralelos es llamado multigrado. Por el otro lado, si no hay más que un eje uniendo cada par de nodos (no más de un eje dirigido en el caso de un grafo dirigido), entonces estos grafos son llamados grafos simples. (Ver fig. 1)

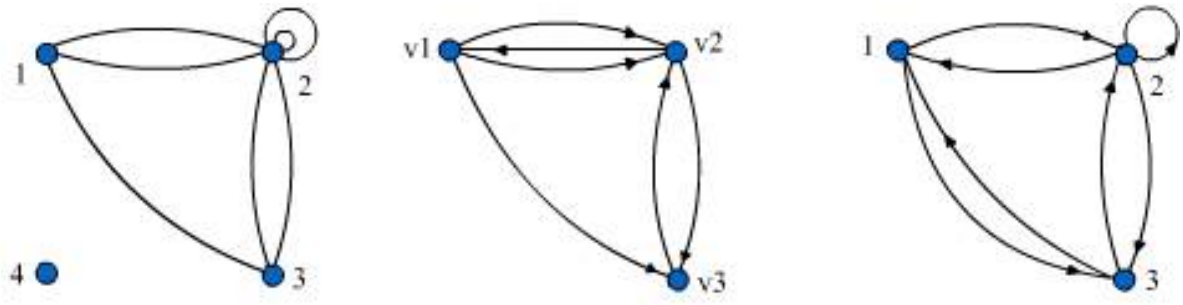


Fig. 2

Los grafos en la figura 2a y b pueden ser representados por los diagramas mostrados en las figuras 3a y b en donde el numero de cada eje muestra la multiplicidad de estos. También podemos considerar la multiplicidad como un peso asignado a un eje. Esta interpretación nos permite generalizar el concepto del peso de los números que no son necesariamente enteros. También podemos tener grafos como los mostrados en la figura 3c y d en los que el número en los ejes muestran el peso de cada uno de ellos. Un grafo en donde los pesos son asignados a cada eje es llamado (WEIGHTED GRAPH).

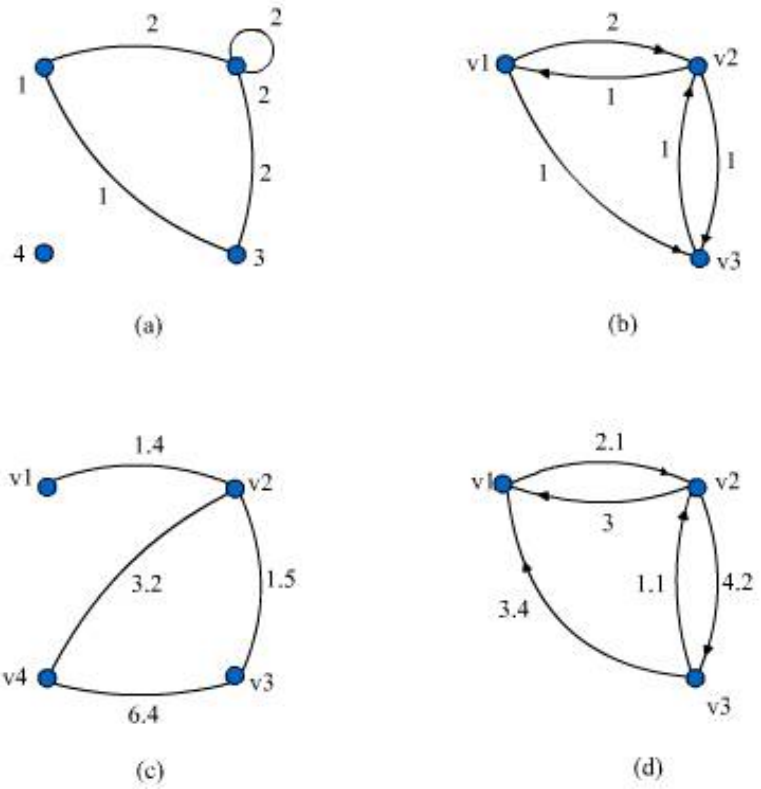


Fig. 3

Un grafo representando las calles de una ciudad se le debe de asignar pesos a cada calle (que representan los ejes) de acuerdo con la densidad de tráfico en cada calle.

En un grafo un nodo que no es adyacente a ningún otro nodo es llamado *nodo aislado*. Un grafo que contiene únicamente nodos aislados se conoce como un *grafo nulo*. En otras palabras, un conjunto de ejes en un grafo nulo esta vacío.

Para un grafo dado podemos obtener una variedad de diagramas colocando los nodos en posiciones arbitrarias y también representando los ejes con arcos o líneas de diferente formas. Por esto mismo podemos tener grafos que aparentemente son totalmente distintos pero en realidad representan lo mismo, como se muestra en las figuras 4 (a)(a´), (b)(b´) y (c)(c´).

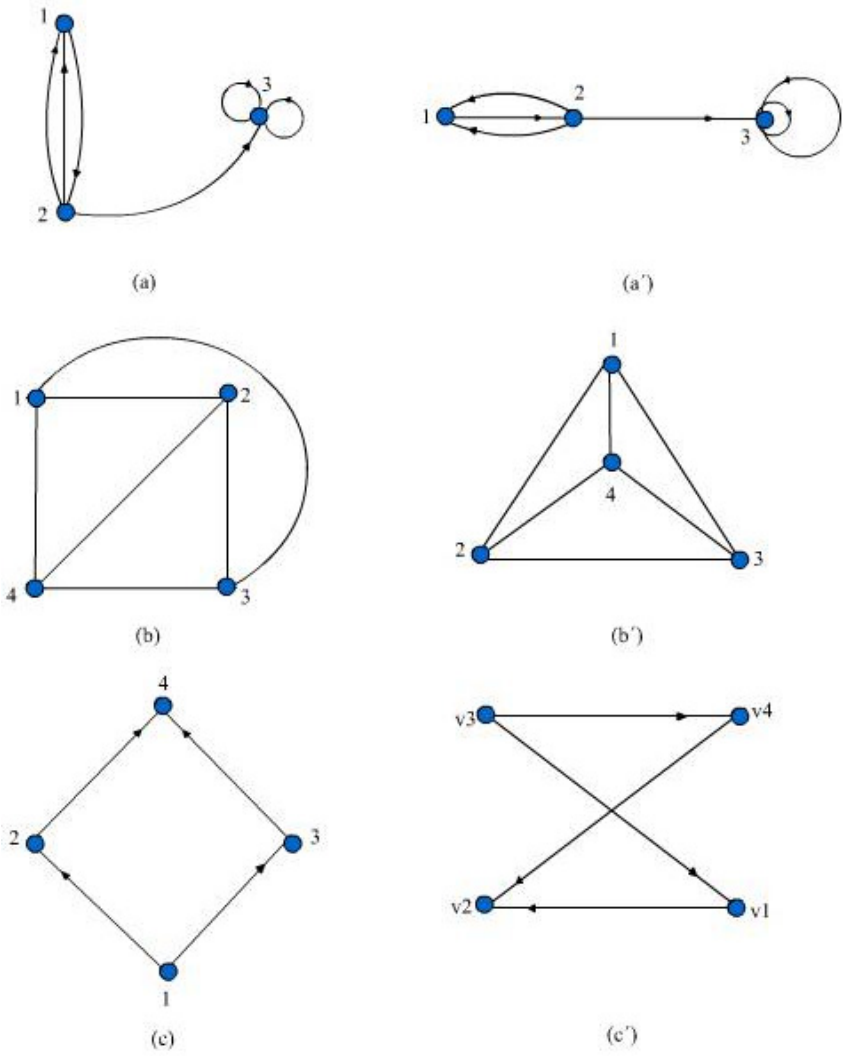


Fig. 4

Definición: Dos grafos son isomorfos si existe correspondencia uno a uno entre los nodos de los dos grafos, y además conservan la adyacencia entre los nodos así como la dirección de los ejes, si existen.

De acuerdo con la definición de isomorfismo nos damos cuenta que un par cualquiera de nodos que esta unido por un eje debe de tener los nodos correspondientes en el otro grafo también unidos por un eje, así mismo debe existir una correspondencia uno a uno entre los ejes. Los grafos mostrados en la figura 4 (c) y (c´) son isomorfos.

Definición: En un grafo dirigido, para cualquier nodo *v* al número de ejes que tienen a *v* como su nodo inicial se le llama *grado de salida* del nodo *v*. Al número de ejes que tiene a *v* como su nodo terminal es llamado *grado de entrada* de *v*, y a la suma del grado de entrada y de salida del nodo *v* es *llamado grado total*. En el caso de un grafo dirigido, el grado total del nodo *v* es igual al número de ejes que inciden en *v*.

El grado total de un nodo aislado es cero.

El concepto del grado de un nodo puede generalizarse a un conjunto de nodos. Sea *G = (V : E)* un grafo dirigido y sea $X \subseteq V$ un subconjuntos de nodos. Al número de ejes de G que tienen su nodo inicial en X pero su nodo terminal no pertenece a X es llamado *grado de salida* de X. De manera similar, al número de ejes de G que tienen su nodo terminal en X pero su nodo inicial no pertenece a X es llamado *grado de entrada* de X.

Un resultado simple que envuelve la definición de grados de un nodo es que la suma de los grados de todos los nodos de un grafo debe ser un número par el cuál es igual a dos veces el número de ejes del grafo.

Sea V(H) un conjunto de nodos de un grafo H y V(G) sea un conjunto de nodos de un grafo G tal que $V(H) \subseteq V(G)$. Si, en adición, cada eje de H es también un eje de G, entonces el grafo H es llamado subgrafo de el grafo G, el cual es expresado por $H \subseteq G$. Naturalmente, el grafo G por si mismo, así como el grafo nulo que se obtiene de G al eliminar todos sus ejes, son también subgrafos de G. Otro subgrafo de G puede obtenerse eliminando algunos nodos y ejes de G.

Sea G = (V , E) un grafo dirigido simple. Entonces cada eje de E puede ser expresado por medio de un par ordenado de elementos de V. Únicamente dicha pareja define al eje.

Trayectorias, Alcances y Conectividad

Sea G = (V , E) un grafo dirigido simple. Considerando una secuencia de ejes de G tal que el nodo terminal de cada eje en la secuencia es el nodo inicial del siguiente eje, si existe alguno en la secuencia. Un ejemplo de secuencia de este tipo es:

$$((v_{i1}, v_{i2}), (v_{i2}, v_{i3}), \ldots, (v_{ik-2}, v_{ik-1}), (v_{ik-1}, v_{ik}))$$

Definición: Cualquier secuencia de ejes de un grafo dirigido tal que el nodo terminal de cada eje en la secuencia es el nodo inicial de otro eje, si existe, siendo el siguiente en la secuencia define la trayectoria de un grafo.

Una trayectoria se define como un viaje a través de los nodos que aparecen en la secuencia, y que se origina en el nodo inicial del primer eje y finaliza en el nodo terminal del último eje de la secuencia.

Definición: El número de ejes que aparecen en la secuencia de una trayectoria es llamado longitud de la trayectoria.

Considérese el grafo dirigido simple mostrado en la figura 5. Algunas de las trayectorias que se originan en el nodo 1 y terminan en el nodo 3 son:

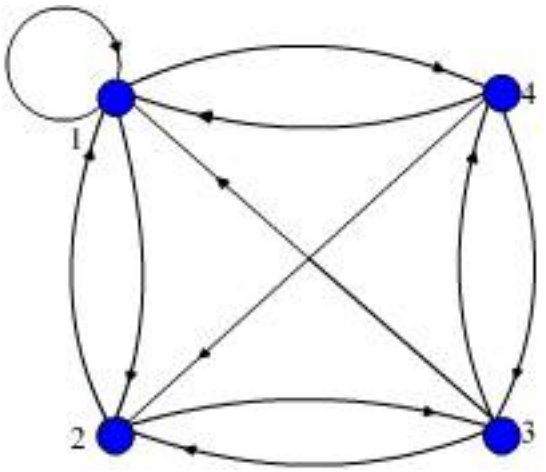


Fig. 5

P1: ((1,2),(2,3))
P2: ((1,4),(4,3))
P3: ((1,2),(2,4),(4,3))
P4: ((1,2),(2,4),(4,1),(1,2),(2,3))
P5: ((1,2),(2,4),(4,1),(1,4),(4,3))
P6: ((1,1),(1,1),...,(1,2),(2,3))

Definición: La trayectoria en un diagrama en el que todos los ejes son distintos es llamada *trayectoria simple*. La trayectoria en la que todos los nodos son distintos en cada camino es llamada *trayectoria principal.*

Naturalmente cada trayectoria principal de un diagrama es simple. Las trayectorias P1, P2 y P3 del grafo dirigido de la figura 5 son elementales.

Definición: Una trayectoria que se origina y termina en el mismo nodo se conoce como *circuito*. Un circuito es llamado *simple* si su trayectoria es simple, en un circuito no aparecen más de un eje. Un circuito es llamado principal si no pasa por un nodo más de una vez.

Nótese que en un circuito el nodo inicial aparece al menos dos veces incluso si este es un circuito principal. Los siguientes son algunos de los circuitos del grafo de la fig. 5.

C1= ((1,1))
C2= ((1,2),(2,1))
C3= ((1,2),(2,3),(3,1))
C4= ((1,4),(4,3),(3,1))

C5= ((1,4),(4,3),(3,2),(2,1))

Obsérvese que cualquier trayectoria que es no principal contiene circuitos que pasan por algunos nodos más de una vez en la trayectoria.

Un grafo dirigido simple que no contiene circuitos se conoce como acíclico. En la figura 4 se muestran algunos grafos dirigidos que son acíclicos.

Definición: Un nodo *v* de un grafo dirigido simple, se dice que es *accesible* para el nodo *u* del mismo grafo dirigido, si existe una trayectoria de *u* a *v*.

Nótese que el concepto de accesibilidad es independiente del número de trayectorias alternativas entre *u* y *v* ni tampoco de su longitud.

Si un nodo *v* es accesible para el nodo *u*, entonces la trayectoria de mínima longitud se conoce como *geodesic.* La longitud del g*eodesic* del nodo *u* al nodo *v* se conoce como distancia y se denota por *d(u,v).* Se asume que d(u,v) = 0 para cualquier nodo u.

La distancia de un nodo *u* a un nodo *v*, si *v* es accesible para *u*, se denota *d u,v* y satisface las siguientes propiedades:

$$d(u, v) >= 0$$
$$d(u, u) = 0$$
$$d(u, v) + d(v, w) >= d(u, w)$$

La última desigualdad se conoce como el *triángulo de la desigualdad*. Si *v* no es accesible para *u* se representa de la siguiente manera *d(u,v) =oo.* Además si *v* es accesible para *u* y viceversa, entonces *d (u,v)* es no necesariamente igual que *d (v, u).*

Teorema: En un grafo dirigido simple, la longitud de cualquier elemento de una trayectoria es menor o igual a *n – 1*, donde *n* es el número de nodos en el grafo. De manera similar la longitud de cualquier circuito principal no excede a *n*.

DEMOSTRACIÓN: La demostración se basa en el hecho de que en cada trayectoria principal los nodos que aparecen en la secuencia son distintos. El número de nodos en cualquier trayectoria principal de longitud *k es k+1*. Si tenemos únicamente *n* nodos distintos en el grafo, no podemos tener una trayectoria principal de longitud mayor a *n – 1*. Para un circuito principal de longitud *k,* la secuencia contiene *k* nodos distintos.

Un grafo no dirigido se dice que es *conectado* si para cada par de nodos del grafo los dos nodos son accesibles para algún otro. Esta definición no puede aplicarse a un grafo dirigido sin algunas modificaciones, porque en un grafo dirigido si el nodo *u* es accesible para un nodo *v*, el nodo *v* no puede ser accesible para el nodo *u*, por lo tanto a un grafo dirigido conectado se le llama (weakly connected).

Definición: Un grafo dirigido simple se dice que es *unilateralmente conectado* si para cada par de nodos del grafo al menos uno de los nodos del par es accesible para el otro nodo. Si

para cualquier par de nodos del grafo ambos nodos del par son accesibles entre si, entonces se dice que el grafo es *fuertemente conectado.*

Definición: Para un grafo dirigido simple, el subgrafo más fuertemente conectado se conoce como *componente fuerte*. Similarmente, el subgrafo más unilateralmente conectado se conoce como componente unilateral.

Teorema: En un grafo simple dirigido, G = (V, E), cada nodo del grafo dirigido (lies) exactamente en un componente fuerte.

REPRESENTACION MATRICIAL

Proposición.- La suma de los grados de los vértices es igual al doble del número de aristas

Demostración
Al realizar la suma de los grados de todos los vértices, como cada arista tiene 2 extremos se cuenta exactamente 2 veces. Por tanto la suma de los grados de los vértices es igual al doble del número de aristas

Definición (matriz de adyacencia de un grafo)
Sea G un grafo de orden n. Llamaremos matriz de incidencia de G a la matriz nxn que llamaremos A = (aij) donde aij = 1 si {i,j}ÎA y aij = 0 en otro caso.

La matriz de adyacencia siempre es simétrica porque aij = aji

Cuando se trata de grafos ponderados en lugar de 1 el valor que tomará será el peso de la arista. Si el grafo es no dirigido hay que asegurarse de que se marca con un 1 (o con el peso) tanto la entrada a[i][j] como la entrada a[j][i], puesto que se puede recorrer en ambos sentidos.

La implementación en C estaría dada por la siguiente forma:

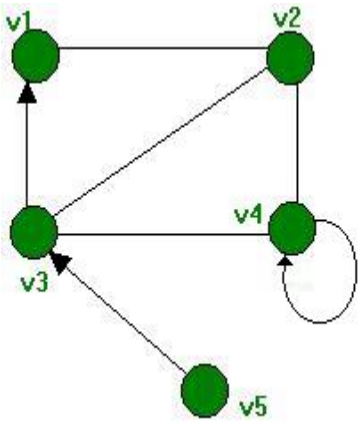
```
int V,A;
int a[maxV][maxV];
void inicializar()
{
int i,x,y,p;
char v1,v2;
// Leer V y A
memset(a,0,sizeof(a));
for (i=1; i<=A; i++)
{
scanf("%c %c %d\n",&v1,&v2,&p);
x=v1-'A'; y=v2-'A';
a[x][y]=p; a[y][x]=p;
}
}
```

En esta implementación se ha supuesto que los vértices se nombran con una letra mayúscula y no hay errores en la entrada. Evidentemente, cada problema tendrá una forma de entrada distinta y la inicialización será conveniente adaptarla a cada situación. En todo caso, esta operación es sencilla si el número de nodos es pequeño. Si, por el contrario, la entrada fuese muy grande se pueden almacenar los nombres de nodos en un árbol binario de búsqueda o utilizar una tabla de dispersión, asignando un entero a cada nodo, que será el utilizado en la matriz de adyacencia.

Como se puede apreciar, la matriz de adyacencia siempre ocupa un espacio de V*V, es decir, depende solamente del número de nodos y no del de aristas, por lo que será útil para representar grafos densos.

A contunuación podemos observar un grafo y su matriz de adyacencia:

	v1	v2	v3	v4	v5
v1	0	1	0	0	0
v2	1	0	1	1	0
v3	1	1	0	1	0
v4	0	1	1	1	0
v5	0	0	1	0	0



Esta matriz esta formada por las conexiones entre los vertices del grafo, de modo tal que si el un vertice "x" esta conectado con un arco con el vértice "y", aparecera el número uno en la casilla (vx, vy) en caso contrario lo que veremos en la matriz será un cero.

En el ejemplo anterior la casilla (v1, v1) tiene el valor de cero puesto que no existe ningún arco que conecte al vértice v1 consigo mismo.

Al contrario de v1 el vértice cuatro tiene un bucle por lo que (v4, v4) adquiere el valor de uno.

La casilla (v1, v2) tiene el valor de uno puesto que existe un arco que une v1, con v2. Como este arco no posee dirección, lo podemos recorrer en ambos sentidos y por tanto (v2, v1) también posee el valor de uno.

No es el caso con v3 y v5 donde el arco que los uno posee dirección y sólo puede recorrerse de v5 a v3 por lo que (v5, v3) posee el valor de uno, mientras que (v3, v5) tiene valor cero.

Teorema.- Sea G un grafo de n vértices con n > 1 y sea A su matriz de adyacencia. Se cumple que el valor del coeficiente de la matriz es igual al número de caminos de longitud k con extremos vi y vj

Demostración
Por inducción en k

Para k = 1 es la definición de A

Supongamos que es cierto para k y vamos a verlo para k+1.
La casilla (i,j) de Ak+1 es el producto de la fila i de Ak por la columna j de A, es decir, si llamamos (a, b, c,...) a la fila i de Ak y (a', b', c',...) a la columna j de A, entonces la casilla (i,j) de Ak+1 es aa' + bb' +.....

El número de caminos de longitud k+1 de i a j que pasan en último lugar por el vértice 1 será 0 si no hay arista de 1 a j o bien coincidirá con el número de caminos de longitud k de i a 1 si existe arista de 1 a j. En resumen, como el número de caminos de i a 1 es el primer elemento de la fila i de Ak, y el primer elemento de la columna j de A vale 1 ó 0 dependiendo de si hay o no arista de 1 a j, tendremos que el número de caminos de longitud k+1 de i a j que pasan en último lugar por el vértice 1 será siempre aa'.

De manera análoga el número de caminos de longitud k+1 de i a j que pasan en último lugar por el vértice 2 será siempre bb', y así sucesivamente.

El número total de caminos de longitud k+1 de i a j será la suma de todos los anteriores, es decir, aa'+bb'+cc'+...., es decir, el elemento (i,j) de la matriz Ak+1.

Nota.- Si existe un camino de longitud m (m ³ n) entre 2 vértices cualquiera, entonces existe un camino de longitud £ n-1 entre esos dos vértices.

Definición.- Un grafo G se dice conexo si cada par de vértices está unido al menos por un camino.

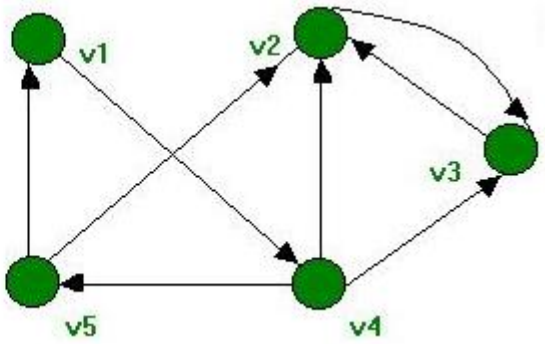
Definición.- La relación entre vértices dada por v está relacionado con w si hay un camino que los une es de equivalencia. Las clases de equivalencia de esta relación se llaman las componentes conexas del grafo.

Un método para comprobar si un grafo es conexo es por medio del teoreme visto anteriormente:
- Se halla la matriz de adyacencia y se eleva a la n-ésima potencia
- Se calcula la suma de las potencias de A hasta An.
- Si todos sus elementos son distintos de cero, el grafo es conexo.

Ejemplo:

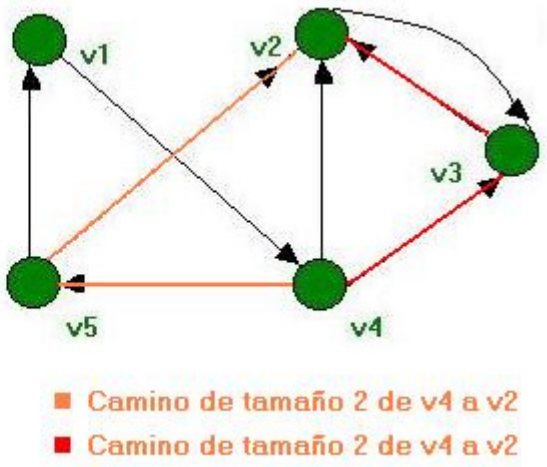
Sea el grafo A y su matriz de adyacencia.

$$A = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$



Si elevamos esta matriz al cuadrado podrémos observar la cantidad de caminos de tamaño dos que existen entre los vértices, tal como se muestra a continuación.

$$A^2 = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$



Por ejemplo a42=2 por lo que existen dos caminos de tamaño dos que van de v4 a v2.

Siguiendo con el ejemplo obtenemos la matriz quinta de A.

$$A^3 = \begin{pmatrix} 1 & 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 \\ 0 & 2 & 1 & 0 & 1 \end{pmatrix}$$

$$A^4 = \begin{pmatrix} 0 & 1 & 2 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 3 & 2 & 0 & 1 \\ 1 & 2 & 2 & 0 & 0 \end{pmatrix} \qquad A^5 = \begin{pmatrix} 0 & 3 & 2 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 3 & 3 & 0 & 0 \\ 0 & 2 & 2 & 1 & 0 \end{pmatrix}$$

Ahora bien si sumamos las matrices que obtuvimos anteriormente, generaremos una matriz en donde estarán representados la suma de todos los posibles caminos para llegar a cierto vértice del grafo a otro.

$$B = \begin{pmatrix} 1 & 7 & 6 & 2 & 2 \\ 0 & 2 & 3 & 0 & 0 \\ 0 & 3 & 2 & 0 & 0 \\ 2 & 10 & 9 & 1 & 2 \\ 2 & 7 & 6 & 2 & 1 \end{pmatrix}$$

Y si en esta matriz sustituimos todos los números diferentes de ceros por unos, obtendremos la siguiente matriz.

$$P = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Esta matriz se conoce como la matriz de caminos y nos indica si es posible llegar a cierto nodo desde otro, una matriz de caminos unitaria proviene de un grafo conexo.

Otro método para obtener una matriz de caminos es a través de conjunciones y disyunciones.

P=A V A(2) V A(3).....V A(k)

Donde

K = número de vértices en el grafo.

aij(2)=V(aik or akj) de k=1 hasta n

REPRESENTACIÓN MATRICIAL

Proposición.- La suma de los grados de los vértices es igual al doble del número de aristas

Demostración

Al realizar la suma de los grados de todos los vértices, como cada arista tiene 2 extremos se cuenta exactamente 2 veces. Por tanto la suma de los grados de los vértices es igual al doble del número de aristas

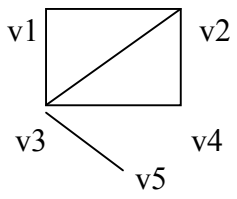
Definición (matriz de adyacencia de un grafo)

Sea G un grafo de orden n. Llamaremos matriz de incidencia de G a la matriz $n \times n$ que llamaremos $A = (a_{ij})$ donde $a_{ij} = 1$ si $\{i, j\} \in A$ y $a_{ij} = 0$ en otro caso.

La matriz de adyacencia siempre es simétrica porque $a_{ij} = a_{ji}$

Ejemplo:

	v1	*v2*	*v3*	*v4*	*v5*
v1	0	1	1	0	0
v2	1	0	1	1	0
v3	1	1	0	1	1
v4	0	1	1	0	0
v5	0	0	1	0	0



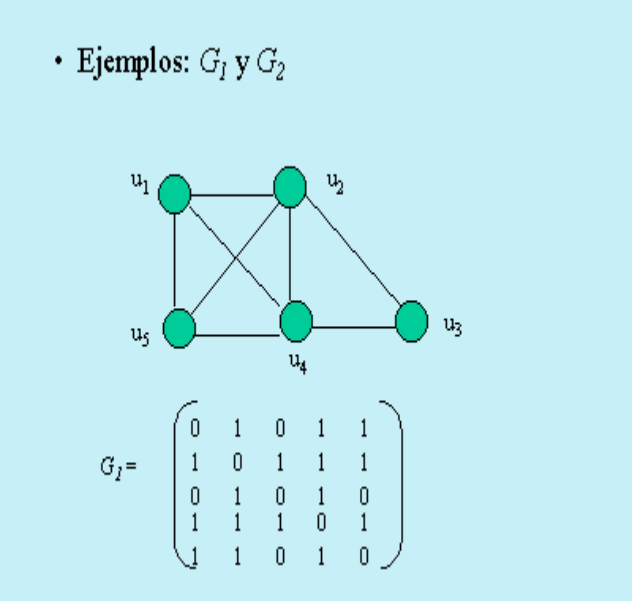
Cuando se trata de grafos ponderados en lugar de 1 el valor que tomará será el peso de la arista. Si el grafo es no dirigido hay que asegurarse de que se marca con un 1 (o con el peso) tanto la entrada $a[i][j]$ como la entrada $a[j][i]$, puesto que se puede recorrer en ambos sentidos.

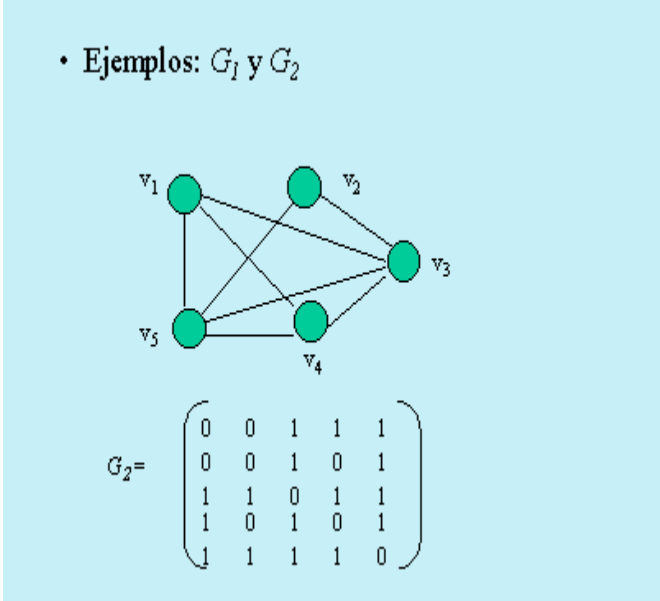
```
int V,A;
int a[maxV][maxV];

void inicializar()
{
  int i,x,y,p;
  char v1,v2;
  // Leer V y A
  memset(a,0,sizeof(a));
  for (i=1; i<=A; i++)
  {
    scanf("%c %c %d\n",&v1,&v2,&p);
    x=v1-'A'; y=v2-'A';
    a[x][y]=p; a[y][x]=p;
```

```
    }
}
```

En esta implementación se ha supuesto que los vértices se nombran con una letra mayúscula y no hay errores en la entrada. Evidentemente, cada problema tendrá una forma de entrada distinta y la inicialización será conveniente adaptarla a cada situación. En todo caso, esta operación es sencilla si el número de nodos es pequeño. Si, por el contrario, la entrada fuese muy grande se pueden almacenar los nombres de nodos en un árbol binario de búsqueda o utilizar una tabla de dispersión, asignando un entero a cada nodo, que será el utilizado en la matriz de adyacencia.

Como se puede apreciar, la matriz de adyacencia siempre ocupa un espacio de V*V, es decir, depende solamente del número de nodos y no del de aristas, por lo que será útil para representar grafos densos.





Una de las cosas que se pueden observar a simple vista en una matriz de adyacencia es que si el diagrama es reflexivo, entonces los elementos de la diagonal principal esta formado por unos.

Teorema.- Sea G un grafo de n vértices con $n>1$ y sea A su matriz de adyacencia. Se cumple que el valor del coeficiente a_{ij}^k de la matriz A^k es igual al número de caminos de longitud k con extremos v_i y v_j

Demostración

Por inducción en k

Para k = 1 es la definición de A

Supongamos que es cierto para k y vamos a verlo para k+1.

La casilla (i,j) de A^{k+1} es el producto de la fila i de A^k por la columna j de A, es decir, si llamamos (a, b, c,...) a la fila i de A^k y (a', b', c',...) a la columna j de A, entonces la casilla (i,j) de A^{k+1} es aa' + bb' +.....

El número de caminos de longitud k+1 de i a j que pasan en último lugar por el vértice 1 será 0 si no hay arista de 1 a j o bien coincidirá con el número de caminos de longitud k de i a 1 si existe arista de 1 a j. En

resumen, como el número de caminos de i a 1 es el primer elemento de la fila i de A^k, y el primer elemento de la columna j de A vale 1 ó 0 dependiendo de si hay o no arista de 1 a j, tendremos que el número de caminos de longitud k+1 de i a j que pasan en último lugar por el vértice 1 será siempre aa'.

De manera análoga el número de caminos de longitud k+1 de i a j que pasan en último lugar por el vértice 2 será siempre bb', y así sucesivamente.

El número total de caminos de longitud k+1 de i a j será la suma de todos los anteriores, es decir, aa'+bb'+cc'+...., es decir, el elemento (i,j) de la matriz A^{k+1}.

Nota.- Si existe un camino de longitud m ($m \geq n$) entre 2 vértices cualquiera, entonces existe un camino de longitud $\leq n-1$ entre esos dos vértices.

Definición.- Un grafo G se dice conexo si cada par de vértices está unido al menos por un camino.

Definición.- La relación entre vértices dada por v está relacionado con w si hay un camino que los une es de equivalencia. Las clases de equivalencia de esta relación se llaman las componentes conexas del grafo.

Nota.- Un método para comprobar si un grafo es conexo es el siguiente:

- Se halla la matriz de adyacencia y se eleva a la n-1-ésima potencia
- Se calcula la suma de las potencias de A hasta A^{n-1}
- Si todos sus elementos son $\neq 0$, el grafo es conexo.

Ejemplo:

Sea la matriz de adyacencia

$$A = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

$$A^2 = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix} \qquad A^3 = \begin{pmatrix} 1 & 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 \\ 0 & 2 & 1 & 0 & 1 \end{pmatrix}$$

$$A^4 = \begin{pmatrix} 0 & 1 & 2 & 1 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 3 & 2 & 0 & 1 \\ 1 & 2 & 2 & 0 & 0 \end{pmatrix} \qquad A^5 = \begin{pmatrix} 0 & 3 & 2 & 0 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 3 & 3 & 0 & 0 \\ 0 & 2 & 2 & 1 & 0 \end{pmatrix}$$

Por lo que existe un camino de tamaño dos que va de v1 a v2 y 3 caminos de tamaño cinco de v4 a v3.

Ahora bien si sumamos las matrices que obtuvimos anteriormente, generaremos una matriz en donde estarán representados la suma de todos los posibles caminos para llegar a cierto vértice del grafo a otro.

$$B = \begin{pmatrix} 1 & 7 & 6 & 2 & 2 \\ 0 & 2 & 3 & 0 & 0 \\ 0 & 3 & 2 & 0 & 0 \\ 2 & 10 & 9 & 1 & 2 \\ 2 & 7 & 6 & 2 & 1 \end{pmatrix}$$

Y si en esta matriz sustituimos todos los números diferentes de ceros por unos, obtendremos la siguiente.

$$P = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Esta matriz se conoce como la matriz de caminos y nos indica si es posible llegar a cierto nodo desde otro, una matriz de caminos unitaria proviene de un grafo conexo.

Otro método para obtener una matriz de caminos es a través de conjunciones y disyunciones.

$P=A \vee A^{(2)} \vee A^{(3)} \ldots\ldots \vee A^{(k)}$

Donde

K = número de vértices en el grafo.

$a_{ij}^{(2)} = \bigvee (a_{ik} \wedge a_{kj})$ de k=1 hasta n

Por ejemplo para la matriz de adyacencia anterior

$$A^{(2)} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix} \qquad A^{(3)} = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

$$A^{(4)} = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

$$A^{(5)} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Y P estaría dada por:

$$P = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} = A \vee A^{(2)} \vee A^{(3} \vee A^{(4)} \vee A^{(5)}$$

Definición.- Una arista de un grafo G se dice de separación si G es conexo pero al suprimir la arista se divide en dos componentes conexos

Definición.- Dados dos grafos G=(V,E) y G´=(V´,E´), se denomina isomorfismo entre G y G´ a cualquier aplicación biyectiva f:$G \rightarrow G'$ tal que si a, b $\in$ V, entonces {a,b}$\in$E $\Leftrightarrow$ {f(a),f(b)}$\in$E´.

Es decir, es una aplicación biyectiva entre los vértices de V y los de V´ de modo que los vértices conectados siguen estándolo. En este caso, diremos que G y G´ son isomorfos.

Si G y G' son isomorfos son matemáticamente iguales y solo varía la apariencia, o sea, que se mantienen las adyacencias, estructura, caminos, ciclos, nº de vértices, nº de aristas, etc.

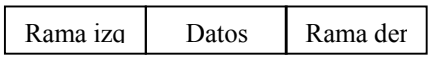
MANIPULACIÓN Y REPRESENTACIÓN DEL ALMACENAJE DE GRAFOS

Árboles: Representación y operaciones

Aunque hay formas de representar árboles basándose en técnicas de almacenaje secuencial (arreglos de vectores y matrices), este tema no será discutido aquí. La representación en computadora de árboles basados en almacenaje ligado o dinámico parece ser más popular debido a la facilidad con que los nodos pueden ser insertados y eliminados de un árbol, y porque las estructuras del árbol pueden crecer a un tamaño arbitrario, un tamaño que es comúnmente impredecible.

El análisis de esta sección se limitara a los árboles binarios ya que son fácilmente representables y manipulables. Un árbol general puede ser fácilmente convertido en un árbol binario equivalente como se verá más adelante. Por tal motivo las técnicas de almacenamiento dinámico son las que usaremos para representar árboles binarios.

Nos enfocaremos ahora a la tarea de usar las técnicas de almacenaje dinámico para representar árboles binarios. Recalcando que los árboles binarios tienen un nodo raíz sin descendientes o bien un subárbol izquierdo y derecho. Cada subárbol descendiente es en sí, un árbol binario, haciendo la distinción entre su rama izquierda y su rama derecha. Una manera conveniente de representar árboles binarios es usar es usar técnicas que involucren nodos con una estructura como la siguiente:



donde *Rama izq.* o *Rama der.* contienen un apuntador al subárbol izquierdo o al subárbol derecho del nodo en cuestión respectivamente. *Datos* contiene la información que está asociada con este nodo en particular. Cada apuntador puede tener un valor nulo.

Un ejemplo de árbol binario como grafo y su correspondiente representación dinámica en memoria son los mostrados en las figuras 3.1a y b respectivamente. Obsérvese la gran similitud entre las figuras. Dicha similitud muestra que la representación dinámica de un árbol es más cercana a la estructuración lógica de los datos involucrados. Esta propiedad puede ser útil en el diseño de algoritmos que procesen estructuras de árboles.

Examinaremos ahora un número de operaciones que son aplicables a los árboles. Una de las operaciones más usadas a las estructuras de árboles es el recorrido. Este es un procedimiento en el cual cada nodo es procesado exactamente una vez de alguna manera sistemática. Usando la terminología popularizada por Knuth, podemos recorrer un árbol

binario de tres formas, específicamente, en preorden, en orden, y en postorden. Las siguientes son definiciones recursivas de estos recorridos.

Recorrido en preorden
Procesar el nodo raíz.
Recorrer el subárbol izquierdo en preorden.
Recorrer el subárbol derecho en preorden.

Recorrido en orden
Recorrer el subárbol izquierdo en orden.
Procesar el nodo raíz.
Recorrer el subárbol derecho en orden.

Recorrido en postorden
Recorrer el subárbol izquierdo en postorden.
Recorrer el subárbol derecho en postorden.
Procesar la raíz.

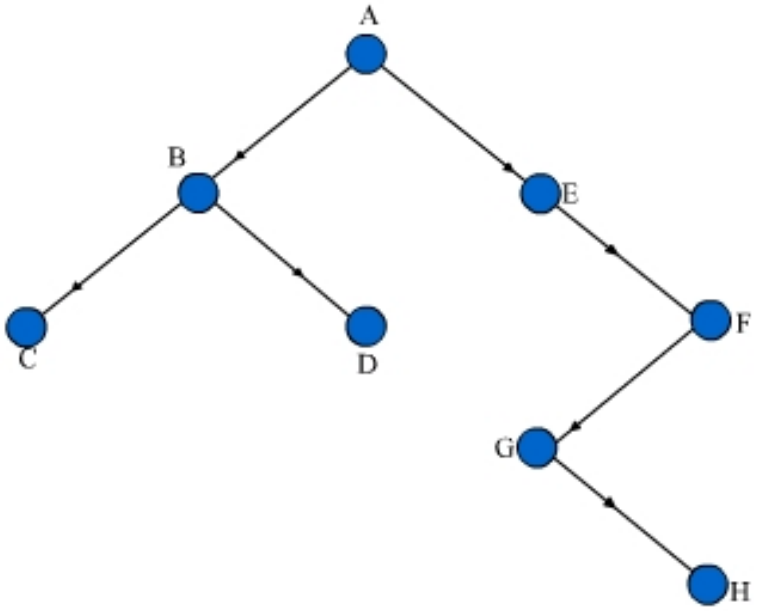
Si un subárbol en particular está vacío (por ejemplo cuando un nodo no tiene descendiente izquierdo o derecho), el recorrido se efectúa sin realizar acción alguna. En otras palabras, un subárbol vacío se considera completamente recorrido cuando es encontrado.

Si las palabras izquierdo y derecho son intercambiadas en las definiciones anteriores, tenemos entonces tres nuevos métodos de recorrido que son llamados *converso en preorden, converso en orden, y converso en postorden,* respectivamente.

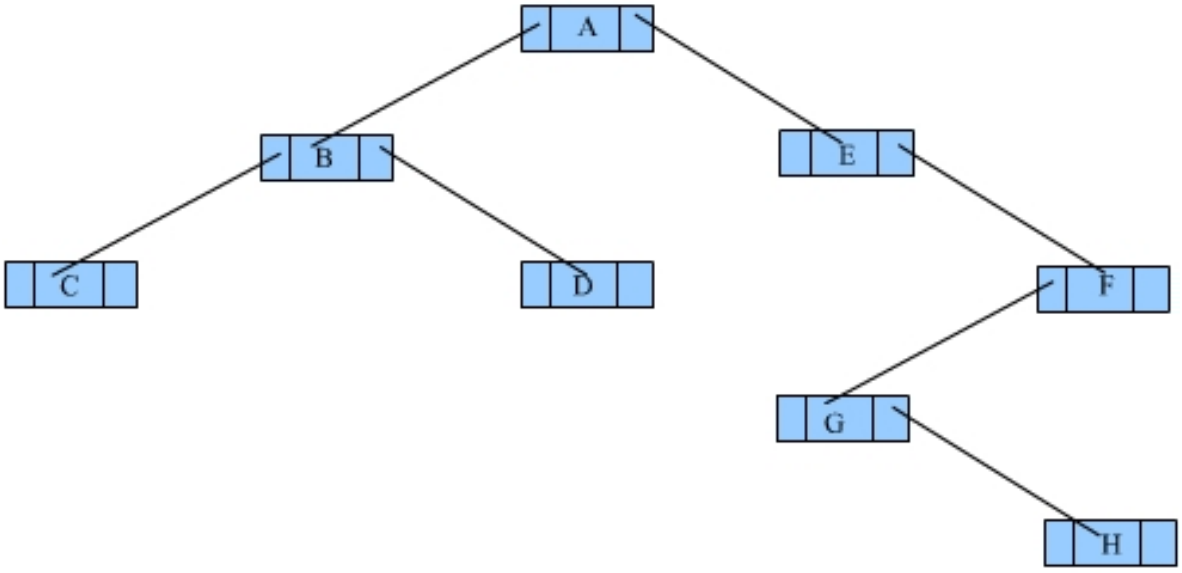
Los recorridos en preorden, en orden, y en postorden del árbol mostrado en la figura 3.1 procesara los nodos de la siguiente manera:

ABCDEFGH (preorden)
CBDAEGHF (orden)
CDBHGFEA (postorden)

Aunque los algoritmos recursivos sean probablemente los más sencillos de escribir para el recorrido de árboles binarios, formularemos algoritmos que son no recursivos. Debido a que el recorrer un árbol exige descender y subsecuentemente ascender partes del árbol, la información de los apuntadores que permita el movimiento en el árbol debe ser temporalmente almacenada. Obsérvese que la información estructural que se presenta ya en el árbol, permite el movimiento descendente desde la raíz del árbol. Debido a que el ascenso del árbol debe ser hecho de manera opuesta al descenso del mismo, es requerido un espacio para guardar los valores del apuntador mientras el árbol es recorrido.



(a)



(b)

Fig. 3.1

Estructuras de Lista y Grafos.

Este subtema será tocado primeramente con la representación de una estructura que es más general que un árbol. Dicha estructura es llamada estructura de lista, y varios lenguajes de programación han sido desarrollados para permitir un procesamiento sencillo similar a los que serán descritos. La necesidad del procesamiento de listas proviene de de un alto costo en el rápido almacenaje computacional y de la impredecible naturaleza del requerimiento de almacenaje de los programas y datos en computadora. Hay varias aplicaciones en la manipulación de símbolos en las cuales esta incertidumbre es particularmente aguda. Se mostrara que una estructura de lista puede ser usada para representar un grafo dirigido. Segundamente, daremos una introducción a la representación de la estructura general de un grafo. Dichas representaciones están basadas no solo en la naturaleza de los datos, sino también en las operaciones que serán realizadas sobre los datos.

En el contexto del procesamiento de listas, definimos una *lista* como una secuencia finita de cero o más *átomos* o *listas,* donde un átomo es concebido como un objeto (por ejemplo, una cadena de símbolos) el cual se distingue de una lista por ser estructuralmente indivisible. Si agrupamos a las listas entre paréntesis y separamos los elementos por comas, entonces lo siguiente puede ser considerado una lista:

(a, (b, c), d, (e,f,g))
( )
(((a)))

La primera lista contiene cuatro elementos, propiamente, el átomo *a,* la lista *(b, c)* la cual contiene a los átomos *b* y *c*, el átomo *d,* y la lista *(e, f, g)* cuyos elementos son los átomos *e, f,* y *g.* La segunda lista no contiene elemento alguno, pero una lista nula continua siendo una lista valida de acuerdo con nuestra definición. La tercera lista contiene un elemento, la lista *((a))* la cual contiene el elemento simple *(a)*, el cual contiene al átomo *a*.

Existe una relación distinta entre grafos y listas. Una lista es un grafo dirigido con un nodo fuente (un nodo cuyo grado de entrada es 0) correspondiente a la lista completa, y con cada nodo inmediatamente conectado al nodo de la fuente que corresponde a un elemento de la lista incluso siendo un nodo con grado de salida 0 (para átomos) o siendo un nodo que tenga ramas (para elementos que son listas) emanando de él. Cada nodo excepto el nodo fuente. Los ejes que salen de un nodo son considerados listas ordenadas. Esto significa que podemos distinguir entre el primer eje, el segundo eje, etc., que corresponden al orden de los elementos desde el primer elemento, el segundo elemento, etc. Además, no hay ciclos en el grafo.

La teoría anterior podría bien aplicarse a los árboles. De cualquier manera, las listas son, de hecho, extensiones de los árboles excepto porque una lista se puede contener a sí misma como elemento y un árbol no. Existen algunas listas que no pueden ser representadas por árboles, pero cada árbol puede ser representado como lista. Las lista pueden tener una estructura anidada esencialmente recursiva que ningún árbol puede tener.

Así hay algunas listas que tienen una representación finita en nuestra notación de paréntesis y comas, pero que corresponde a grafos infinitos.

Los grafos de algunos ejemplos de listas aparecen en la figura 3.2. Si, de cualquier manera, *M* es la lista *(a, b, M),* entonces tenemos un grafo "infinito" para *M*, como se muestra en la figura 3.3.

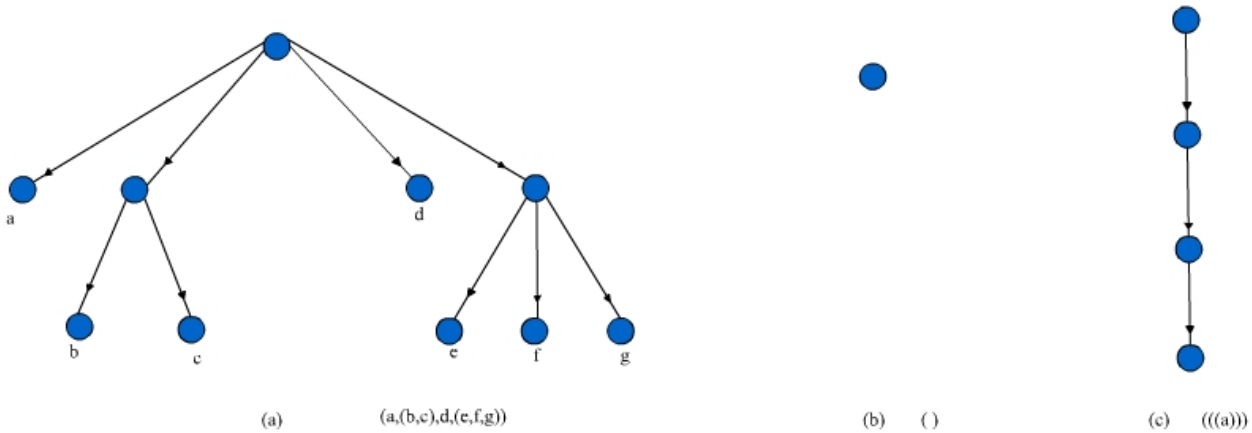


Fig. 3.2

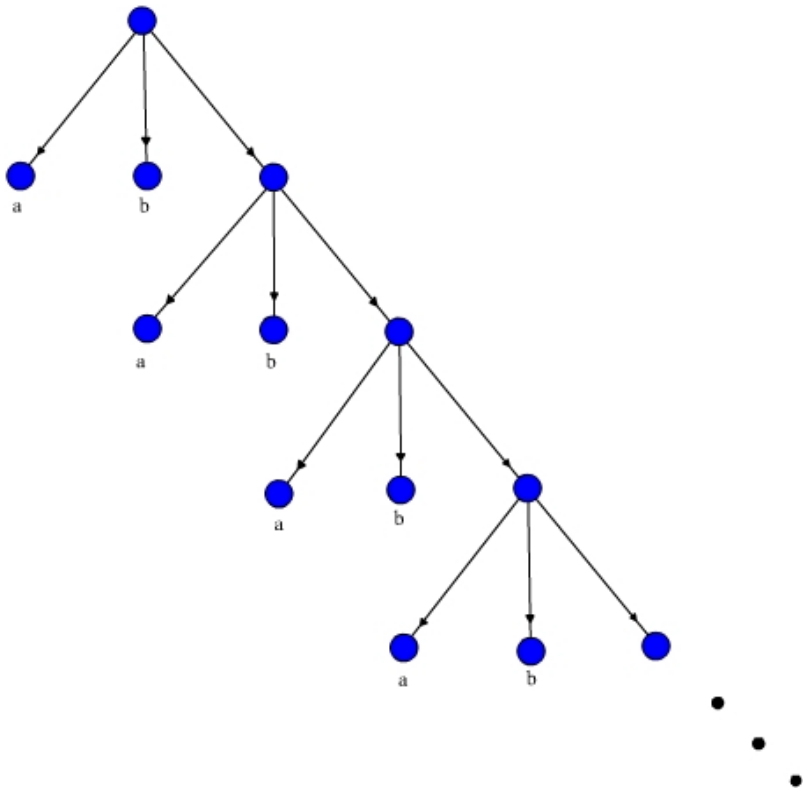


Fig. 3.3

Las técnicas de asignación dinámica pueden ser usadas para representar listas en la memoria de una computadora. En dicha representación existen dos tipos de nodos, uno para los átomos y otro para los elementos de lista. Un nodo atómico contiene dos campos: el primero almacena el valor del átomo (por ejemplo, una cadena de símbolos), y el segundo contiene un apuntador al siguiente elemento de la lista. Un nodo de lista contiene también dos campos: el primer campo apunta la representación del almacenamiento de la lista, y el segundo apunta al elemento que sigue a este elemento de lista en particular. Se asume que un nodo atómico y un nodo de lista son distinguibles.

Obsérvese que además de orden, una lista tiene también profundidad. La profundidad de una lista es el número de niveles que contiene. Así, en la figura 3.3 los elementos *a* y *d* están en el nivel 1, y *b* y *c* están en el nivel 2. El número de pares de paréntesis que rodean un elemento indican su nivel. El elemento *d* en la lista *(a,(b, (c, (d))))* tiene un nivel de 4.

El orden y la profundidad son más fácilmente en términos de nuestra representación de almacenamiento, donde el orden está indicado por las flechas horizontales y la profundidad por flechas verticales que apuntan hacia abajo. Así, la lista *(a, (b, c), d, (e, f, g))* sería representada por la estructura de almacenamiento de la figura 3.4. En el almacenamiento, varias listas pueden compartir sublistas comunes. Por ejemplo, las listas *(a (b, c), d)* y *(1, 5.2, (b, c))* podrían ser representadas como en la figura 3.5.

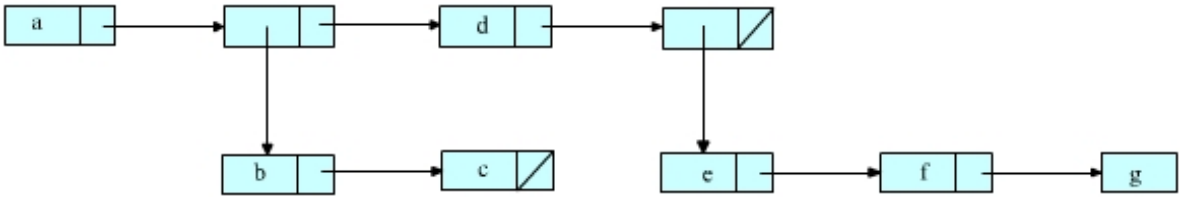


Fig. 3.4

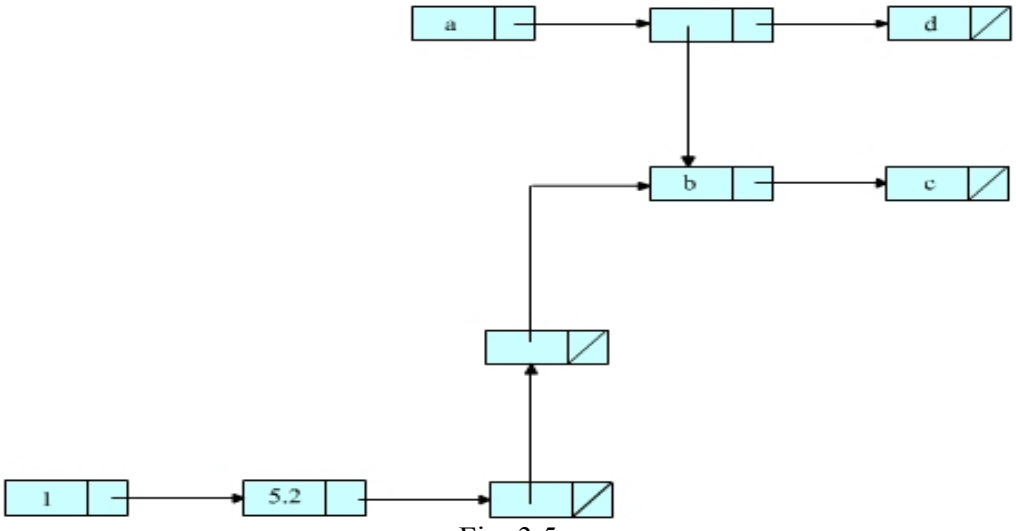


Fig. 3.5

La lista recursiva *M*, donde *M* es *(a, b, M),* puede ser representada como se muestra en la figura 3.6. Naturalmente usaremos esta representación de almacenaje en los casos en los que las estructuras estén compartidas en vez de generar un número infinito de nodos que correspondan a un grafo infinito, pero se debe tener mucho cuidado al manejar dicha estructura recursiva para evitar caer en un ciclo infinito.

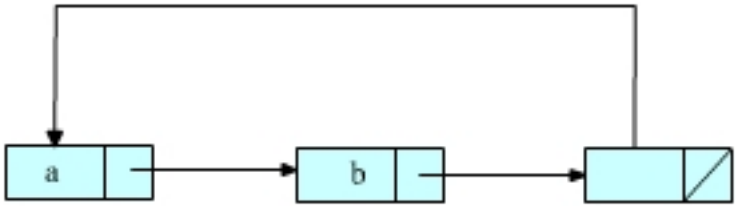


Fig. 3.6

Una estructura de lista se presenta frecuentemente en el proceso de información, aunque no sea siempre evidente. Considérese un enunciado simple en inglés que consta de sujeto, verbo y objeto. Dicho enunciado puede ser interpretado como una lista de tres elementos, cuyos elementos pueden ser atómicos (palabras simples) o listas (frases). Las siguientes oraciones y sus correspondientes representaciones en lista so ejemplos:

Man bites dog. = (Man, bites, dog)

The man bites the dog = ((The, man), bites, (the, dog))

The big man is biting the small dog = ((The, big, man), (is, biting), (the, small, dog))

El sujeto y el objeto del último ejemplo pueden ser incluso separados en sustantivos y adjetivos como se muestra a continuación:

(((The, big), (man)), (is, biting), (the, small), (dog)))

La representación del almacenamiento de este ejemplo se ve en la figura 3.7.

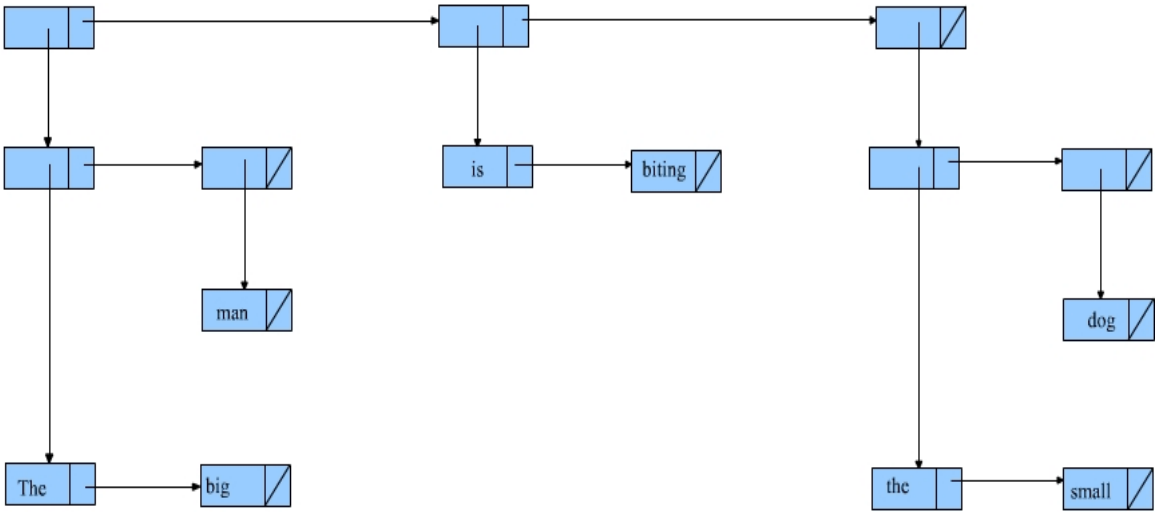


Figura 3.7

Por todo lo anterior, las estructuras de lista son usadas para representar dígrafos, y una propiedad de dicha representación es que las sublistas pueden estar compartidas. Como un ejemplo, un dígrafo y su representación en lista se muestran en la figura 3.8.

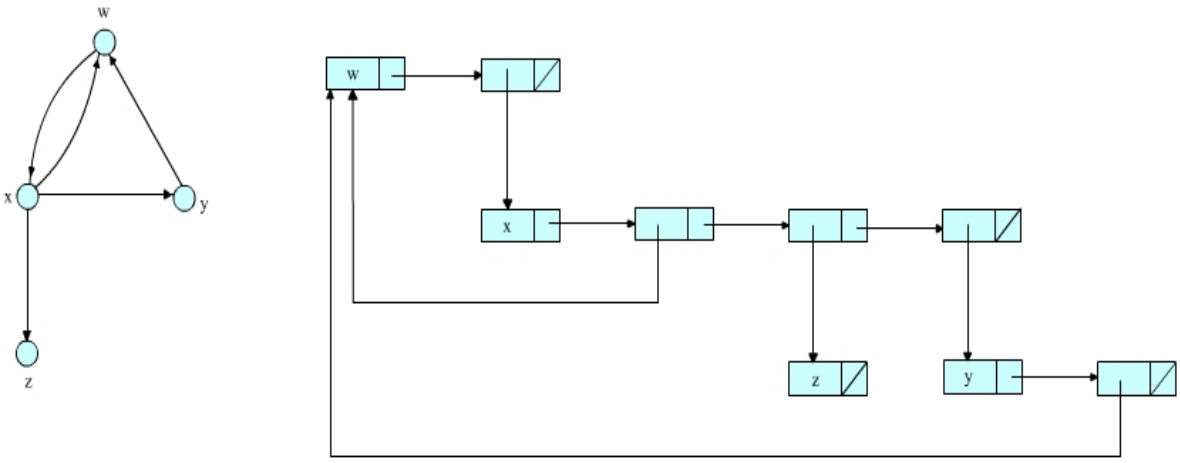


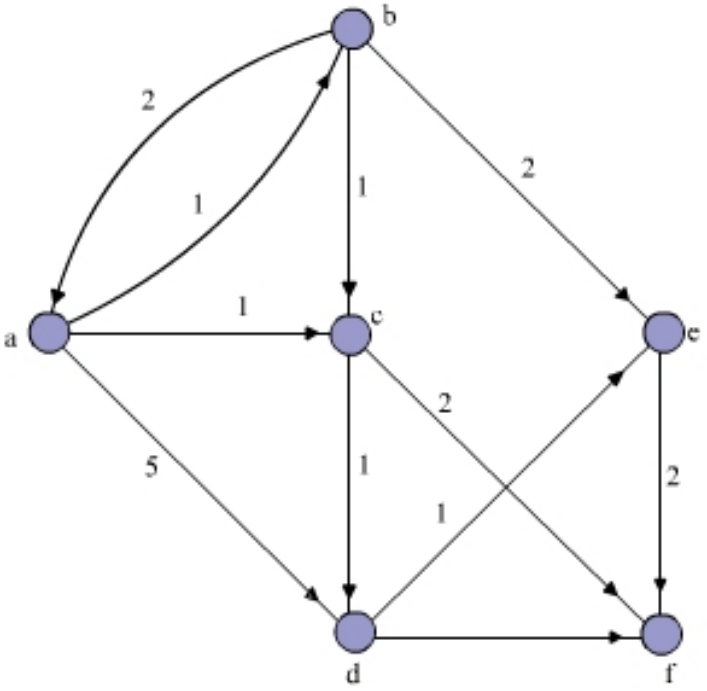
Figura 3.8

Discutiremos a continuación otro método de almacenamiento para los grafos. La mejor representación de almacenaje para algunos grafos generales depende de la naturaleza de los datos y de las operaciones que se vayan a realizar sobre estos datos. Además, la elección de una representación adecuada se ve afectada por otros factores como el número de nodos, el número promedio de ejes que salen del nodo, si un grafo es dirigido, la frecuencia de agregado/eliminados que serán hechos, etc.

En ocasiones se pueden usar arreglos (cuando existe como máximo un eje entre un par cualquiera de nodos) para representar grafos. En este caso los nodos son numerados de 1 a *n,* y se usa un arreglo bidimensional con *n* filas y *n* columnas para representar al grafo.

Así mismo, se podrían requerir vectores para almacenar datos de los nodos en dicha representación. Este enfoque no es muy adecuado para un grafo que tiene un gran número de nodos o muchos nodos que estén conectados a solo algunos ejes, ni cuando el grafo deba ser continuamente alterado.

Si hay un número de ramas entre un par de nodos y un número considerable número de nodos que están conectados a únicamente a otros pocos nodos, entonces la representación de la estructura de almacenaje para dicho grafo podría ser una de las mostradas en la figura 3.9. Obsérvese que el grafo tiene un cierto peso y que la representación de almacenaje está formada por una tabla directorio, y asociada a cada entrada en este directorio podemos tener una lista de ejes. Un directorio de entrada de nodos típico esta formado por el número de nodo, el dato asociado a él, el numero de ejes que emanan a de él, y un campo apuntador que da la lista de ejes asociadas a este nodo. Cada lista de ejes, en este caso, es almacenada como una tabla secuencial cuya entrada típica consiste en el peso de un eje y el número del nodo en que este eje en particular termina. Para un grafo que es continuamente alterado, una representación que almacene las listas de ejes como una lista ligada es más deseable. En este caso no es necesario tener el campo que denota el número de ejes en la tabla directorio de los nodos.



Node #	Node data	Number of edges	Pointer to edge list
1	a	3	
2	b	3	
3	c	2	
4	d	2	
5	e	1	
6	f	0	NULL

Weight	Destination node
1	2
3	3
5	4

1	3
2	5
6	1

1	4
2	6

1	5
2	6

2	6

Figura 3.9

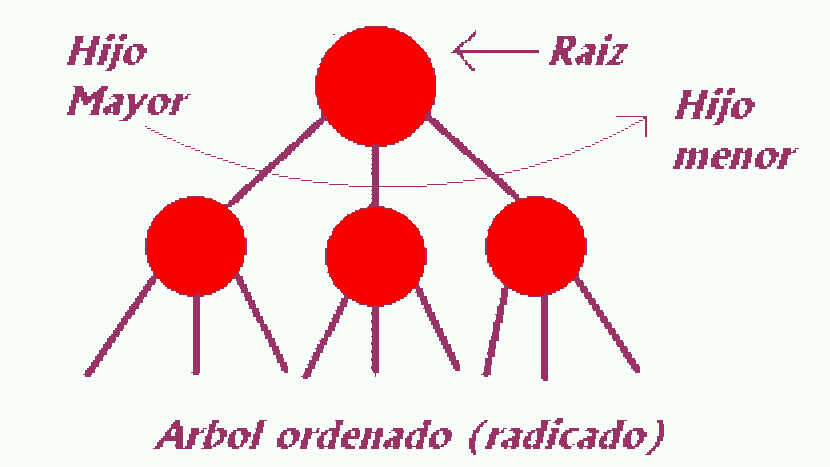
Arboles

Introducción

Los árboles (diagramas arborescentes o arborigramas) constituyen una de las subclases más útiles de los grafos (gráficas). En las ciencias de la computación se utilizan con frecuencia.

Intuitivamente, un árbol es un grafo que en su trazo se asemeja a un árbol con su ramificación hacia abajo.

Un árbol es un grafo simple en el cual existe un único camino entre cada par de vértices. Un árbol con raíz (radicado) es un árbol que tiene un vértice particular designado como raíz.



Propiedades

Sea T un grafo simple con n vértices. Entonces son equivalentes los siguientes enunciados:

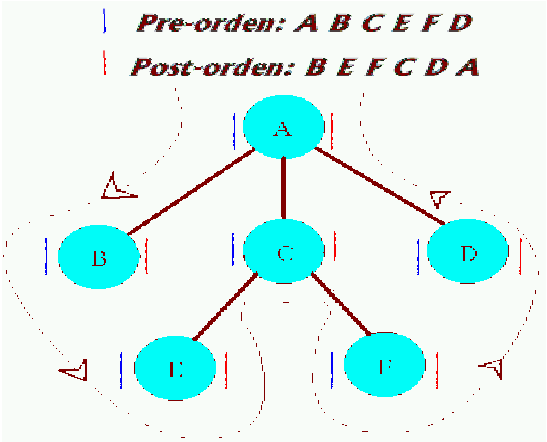
- T es un árbol.
- T es conexo y no contiene circuitos.
- T es conexo y tiene n-1 lados.
- T no contiene circuitos y tiene n-1 lados.

Sea T un árbol con raíz v0, Supóngase que x, y, z son vértices en T y que (v0, v1, ..., vn) es un camino en T. Entonces:

- Vn-1 es el padre de vn.
- V0, v1, ..., vn-1 son los antepasados de vn.
- Vn es el hijo de vn-1.

- Si x es un antepasado de y, entonces y es un descendiente de x.
- Si x y y son hijos de z, entonces x y y son hermanos.
- Si x no tiene hijos, entonces x es un vértice terminal (o una hoja).
- Si x no es un vértice terminal, entonces x es un vértice interno (o una rama).

Recorrido de un árbol



El recorrido de un árbol se refiere a "caminar" en forma sistemática para visitar cada vértice una vez. Existen tres métodos para recorrer un árbol, y son:

- Recorrido con orden inicial (pre-orden)
- Recorrido con orden intermedio (Inter-orden)
- Recorrido con orden final (post-orden)

Todos estos métodos son recorridos, y se explicaran a continuación.

El recorrido con orden inicial: Se recorrerá el árbol partiendo de la raíz procesando primero el nodo en el que nos encontremos, después la rama izquierda, y al final la rama derecha.

El recorrido con orden intermedio: Se recorrerá el árbol partiendo de la raíz procesando primero la rama izquierda, después el nodo en el que nos encontremos y al final la rama derecha.

El recorrido con orden final: Se recorrerá el árbol partiendo de la raíz procesando primero la rama izquierda, después la rama derecha y al final el nodo en el que nos encontremos.

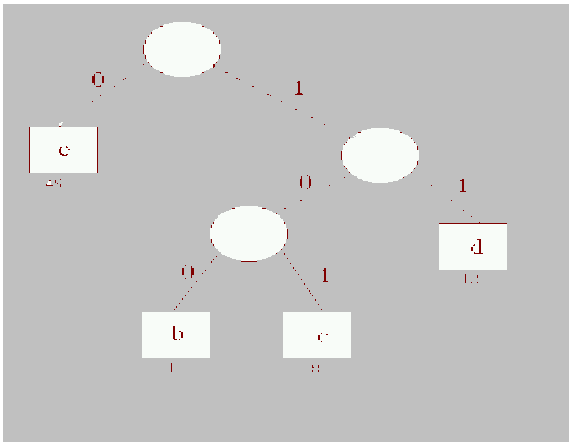
El procesar una rama se refiere a pasar al primer nodo de esa rama e ir siguiendo los algoritmos para sus ramas y así sucesivamente.

Compresión

Existen algunos algoritmos que se basan el la representación de los datos por medio de árboles para comprimir el volumen que estos ocupan.

Algoritmo de Huffman para la compresión de datos.

Este algoritmo se basa en una primera lectura del archivo para calcular la frecuencia de apariciones de cada caracter en el archivo. Teniendo una lista con las frecuencias de cada caracter se ordenan en orden creciente por su frecuencia, y se procede recursivamente hasta terminar con todos los caracteres uniendo los dos de menor frecuencia como hijos de un nodo que tendrá por valor la suma de los valores de sus hijos. Estos nodos después comenzaran a entrar en la suma como si fueran un caracter mas. Al terminar con el árbol completo se definirá que los hijos izquierdos tengan un valor de 0 y los derecho de 1. De manera que para llegar a un caracter la ruta que se siga en base a su posición relativa con respecto a la raíz nos dará el código de este caracter para su compresión de manera que los caracteres más frecuentes tendrán códigos pequeños y los de menor frecuencia tendrán códigos mas grandes.



Teoría de juegos

Una de las más importantes aplicaciones de los árboles se puede ver en la teoría de juegos, aplicando relaciones entre el juego y las decisiones que se toman en ellos, creando un árbol que le puede permitir a una computadora tomar las decisiones correctas y optimas para jugar un juego cualquiera.

Se presenta a continuación una muestra de como se soluciona el problema de las 8 reinas por medio de árboles y sus combinaciones posibles. El problema consiste en colocar 8 reinas en un tablero de 8 x 8, colocando sólo una en cada renglón y columna de manera que no se ataquen entre sí.

Sistemas Operativos

Abrazo Mortal (Deadlock)

En ambientes de multiprogramación, varios procesos pueden competir por un número finito de recursos.

Un proceso solicita recursos, y si los recursos no están disponibles en ese momento, el proceso entra en un estado de espera. Puede suceder que los procesos en espera nunca cambien de estado, debido a que los recursos que han solicitado están siendo detenidos por otros procesos en espera.

Por ejemplo, esta situación ocurre en un sistema con cuatro unidades de disco y dos procesos. Si cada proceso tiene asignadas dos unidades de disco pero necesita tres, entonces cada proceso entrara a un estado de espera, en el cual estará hasta que el otro proceso libere las unidades de cinta que tiene asignadas. Esta situación es llamada **Abrazo Mortal** (Deadlock).

Para prevenir un abrazo mortal o recuperarse de alguno que ocurra, el sistema puede tomar alguna acción relativamente extrema, tal como el derecho de tomar los recursos (preemption of resources) de uno o mas de los procesos que se encuentren en el abrazo mortal. A continuación describiremos algunos de los métodos que pueden utilizarse en un sistema operativo para manejar el problema de abrazo mortal.

EL PROBLEMA DE ABRAZO MORTAL (DEADLOCK).

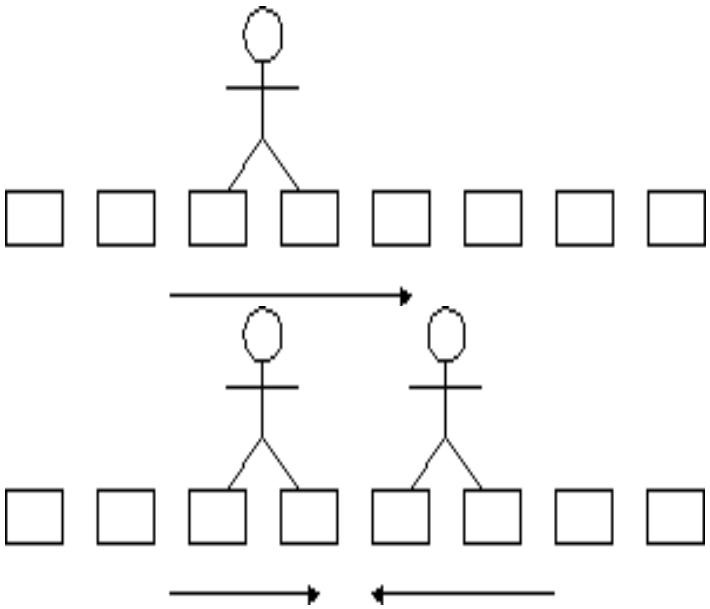
El problema de los abrazos mortales no es único al ambiente de los sistemas operativos.

Generalizando nuestra interpretación de recursos y procesos, podemos ver que un problema de abrazo mortal puede ser parte de nuestro ambiente de vida diaria.

Por ejemplo, considere el problema de cruzar un río que tiene un número de piedras de apoyo. A lo mas, un pie puede estar pisando

una piedra a la vez. Para cruzar el río, cada persona debe de utilizar cada una de las piedras de apoyo.

Podemos suponer que cada una de las personas que cruzan el río es un proceso y que cada piedra de apoyo es un recurso. Un abrazo mortal ocurre cuando dos gentes intentan cruzar el río desde orillas opuestas y se encuentran en el medio.



El pisar una piedra puede ser vista como el adquirir un recurso, mientras que el quitar el pie de la piedra corresponde a liberar el recurso. Un abrazo mortal ocurre cuando dos personas tratan de poner un pie en la misma piedra.

El abrazo mortal puede ser resuelto si cada una de las personas se retira hacia el lado del río desde el cual inició a cruzar. Esta retirada es llamada un "rollback".

Observe que si varias personas están cruzando el río desde el mismo lado, es necesario que más de una persona se retire con el fin de resolver el abrazo mortal. Si una persona empieza a cruzar el río sin averiguar si alguien mas esta tratando de cruzar el río desde el otro lado, entonces siempre puede ocurrir un abrazo mortal.

> La única manera de asegurar que un abrazo mortal no va a ocurrir, es el de pedir a cada persona que va a cruzar el río que siga un protocolo de aceptación previa. Tal protocolo debe de

pedir a cada una de las personas que deseen cruzar el río, el averiguar si alguien mas esta cruzando desde el otro lado. Si la respuesta es no, entonces puede proseguir. De otra manera, debe esperar hasta que la otra persona haya terminado de cruzar.

EL MODELO DEL SISTEMA

Un sistema consiste de un número finito de recursos que son distribuidos entre un número de procesos que compiten por ellos.

Los recursos son clasificados en diferentes tipos, cada uno de los cuales consiste de algún número de instancias iguales

a. Ciclos de CPU
b. Espacio en memoria
c. Archivos y dispositivos de E/S (tales como impresoras, unidades de cinta y de disco, etc.) son ejemplos de tipos de recursos. Si un sistema tiene dos CPU's, entonces el tipo CPU tiene dos instancias.

Un proceso debe solicitar un recurso antes de usarlo y liberarlo después de usarlo.

Un proceso puede solicitar tantos recursos como sean necesarios para llevar a cabo la tarea para la cual ha sido diseñado. Obviamente el número de recursos solicitados no debe exceder el número de recursos disponibles en el sistema.

En otras palabras, un proceso no debe pedir tres impresoras si en el sistema solo existen dos.

Bajo un modo normal de operación un proceso puede utilizar un recurso sólo en la siguiente secuencia:

1.- **PETICIÓN**. Si la petición no puede ser satisfecha inmediatamente (por ejemplo, el recurso esta siendo utilizado por otro proceso), entonces el proceso solicitante debe esperar hasta que pueda adquirir el recurso.

2.- **USO**. El proceso puede utilizar un recurso (por ejemplo, si el recurso es una impresora en línea, el proceso puede imprimir en la impresora).

3.- **LIBERACIÓN**. El proceso libera el recurso.

La petición y liberación del recurso son peticiones al sistema.

El uso de recursos puede también hacerse solo a través de llamadas al sistema.

Por lo tanto, para cada uso, el sistema operativo checa para asegurarse de que el proceso usuario ha pedido y se le han asignado los recursos. Una tabla del sistema registra cuando un recurso está libre o asignado, y si está asignado, a que proceso. Si un proceso pide un recurso que está asignado en ese momento a otro proceso, éste puede ser agregado a una cola de procesos en espera de ese recurso.

DEFINICIÓN DE ABRAZO MORTAL

Un conjunto de procesos está en un abrazo mortal cuando todos los procesos en ese conjunto están esperando un evento que sólo puede ser causado por otro proceso en el conjunto.

Los eventos a que nos referimos son concernientes con la asignación y liberación de recursos principalmente. Sin embargo, otro tipo de eventos pueden llevar a la existencia de abrazos mortales.

Los abrazos mortales pueden también involucrar diferentes tipos de recursos. Por ejemplo, considere un sistema con una impresora y una unidad de disco. Suponga que el proceso P tiene asignada la unidad de disco y que el proceso Q tiene asignada la impresora. Ahora, si P pide la impresora y Q pide la unidad de disco, ocurre un abrazo mortal.

CARACTERIZACIÓN DE UN ABRAZO MORTAL

Un abrazo mortal es una condición indeseable. En un abrazo mortal, los procesos nunca terminan de ejecutarse y los recursos del sistema esta amarrados, evitando que otros procesos puedan siquiera empezar.

CONDICIONES NECESARIAS

Una situación de abrazo mortal puede surgir sí y solo sí las siguientes cuatro condiciones ocurren simultáneamente en un sistema.

- **Exclusión Mutua**. Al menos un recurso es mantenido en un modo no-compartible; esto es, sólo un proceso a la vez puede usar el recurso. Si otro proceso solicita ese recurso, tiene que ser retardado hasta que el recurso haya sido liberado.

- **Retener y esperar**. Debe existir un proceso que retenga al menos un recurso y esté esperando para adquirir recursos adicionales que están siendo retenidos por otros procesos.

- **No existe el derecho de desasignar** (No preemption). Los recursos no pueden ser desasignados (preempted); esto es, un recurso sólo puede ser liberado voluntariamente por el proceso que lo retiene, después de que el proceso ha terminado su tarea.

- **Espera Circular**. Debe existir un conjunto {p0, p1, ...,pn} de procesos en espera tal que p0 esté esperando por un recurso que está siendo retenido por p1, p1 está esperando por un recurso que está siendo retenido por p2, ..., pn-1 está esperando por un recurso que está siendo retenido por pn y pn está esperando por un recurso que está siendo retenido por p0.

MÉTODOS PARA MANEJAR LOS ABRAZOS MORTALES

Existen dos métodos para manejar el problema de los abrazos mortales.

Podemos usar algún protocolo para asegurar que el sistema nunca entrará en un estado de abrazo mortal ó podemos permitir que el sistema entre en un estado de abrazo mortal y después recuperarnos.

El recuperarse de un abrazo mortal puede ser muy difícil y muy caro. Por ello, primeramente consideraremos los métodos para asegurar que no ocurran los abrazos mortales.

Comúnmente, existen dos métodos:

- **PREVENCIÓN** de abrazos mortales (Deadlock Prevention)
- **EVASIÓN** de abrazos mortales (Deadlock Avoidance).

PREVENCIÓN DE ABRAZOS MORTALES.

Para que ocurra un abrazo mortal, deben presentarse cada una de las cuatro condiciones necesarias. Al asegurarnos de que por lo menos una de estas condiciones no se presente, podemos prevenir la ocurrencia de un abrazo mortal. Examinando cada una de las condiciones necesarias por separado.

A. EXCLUSIÓN MUTUA.

La condición de exclusión mutua debe mantenerse para los tipos de recursos que no son compartibles.

Por ejemplo, una impresora no puede ser compartida simultáneamente por varios procesos. En general, no es posible prevenir los abrazos mortales negando la condición de exclusión mutua, debido a que algunos recursos son intrínsecamente no compartibles.

B.RETENER Y ESPERAR.

Para asegurar que la condición de retener y esperar nunca ocurra en el sistema, se debe garantizar que siempre que un proceso solicite un recurso, este no pueda retener otros recursos.

Un protocolo que puede ser usado requiere que cada proceso solicite y le sean asignados todos los recursos antes de que empiece su ejecución.

NO EXISTE EL DERECHO DE DESASIGNAR

La tercera condición necesaria es que no debe de existir el derecho de desasignar recursos que han sido previamente asignados. Con el fin de asegurar que esta condición no se cumple, el siguiente protocolo puede ser usado. Si un proceso que esta reteniendo algunos recursos solicita otro recurso que no puede ser asignado inmediatamente (esto es, que el proceso tenga que esperar), entonces todos los recursos que tiene este proceso en espera son desasginados. Esto es, los recursos son liberados implícitamente. Los recursos liberados son agregados a la lista de los recursos por los cuales esta esperando el proceso. El proceso solo puede volver a ser reinicializado cuando haya obtenido otra vez todos los recursos que tenia asignados, así como el nuevo recurso que estaba solicitando.

Alternativamente si un proceso solicita algunas recursos, primero verificamos si estos están disponibles. Si es así, entonces se le asignan. De otro modo, se verifica si están asignados a alguno de los procesos que están en espera de recursos adicionales. Si es así, los recursos deseados son desasginados del proceso en espera y asignados al proceso solicitante. De otra manera (no están disponibles, ni asignados a procesos en espera) el proceso que hace la solicitud debe esperar. Pero, mientras esta esperando, algunos de sus recursos pueden ser desasignados solamente si estos son solicitados. Un proceso puede ser reinicializado cuando se le asigna el recurso que había solicitado y recupera cualquier recurso que haya sido desasignado mientras esperaba.

ESPERA CIRCULAR

Con el fin de asegurarse de que la condición de espera circular nunca se presente, se puede imponer un ordenamiento total sobre todos los tipos de recursos. Esto es, asignamos a cada tipo de recurso un número entero único, el cual permita comparar dos recursos y determinar cuando uno precede al otro en el ordenamiento.

Mas formalmente, sea R={r1,r2,...,rn} el conjunto de tipos de recursos. Puede definirse una función uno a uno F:R->N, donde N es el conjunto de números naturales.

Por ejemplo, si el conjunto R de tipos de recursos incluye unidades de disco (UDD), unidades de cinta (UDC), lectoras ópticos (LO) e impresoras (I), entonces la función F puede ser definida como sigue:

F(LO) =1

F(UDD) = 5

F(UDC) = 7

F(I) = 12

Se puede considerar el siguiente protocolo para prevenir abrazos mortales:

Cada proceso puede solicitar recursos solamente en un orden creciente de numeración. Esto es un proceso puede solicitar inicialmente cualquier numero de instancias de un tipo de recurso, digamos ri. Después de esto, el proceso puede solicitar instancias de recursos del tipo rj si y solo si F(rj) > F(ri). Si varias instancias del mismo tipo de recurso son necesitadas, debe hacerse una sola petición para todas las instancias debe ser usada. Por ejemplo, usando la función definida anteriormente, un proceso que quiere usar el lector óptico (LO) y la impresora (I) debe solicitar primero el (LO) y después la (I).

Alternativamente, se puede requerir simplemente que siempre que un proceso solicite una instancia del recurso tipo rj, este tenga que liberar cualquier recurso del tipo ri tal que F(ri) >= F(rj).

Si este protocolo es usado, la condición de cola circular no puede presentarse. Se puede demostrar este hecho asumiendo que existe una cola circular (prueba por contradicción). Sea {p0, p2, p2, ..., pn} el conjunto de procesos que están en la espera circular, donde pi esta esperando por un recurso ri, el cual es retenido por pi+1. (Aplicando modulo aritmético sobre los índices, pn esta esperando por un recurso rn que esta siendo retenido por p0). Entonces, debido a que el proceso pi+1 esta reteniendo el recurso ri mientras esta esperando el recurso ri+1, se debe de tener F(ri) < F(ri+1), para toda i. Pero esto significa que F(r0) < F(r1) < ... < F(rn) < F(r0). Por transitividad, F(r0) < F(r0), lo cual es imposible. Por lo tanto, no puede existir una espera circular.

Debe notarse que la función F debe definirse de acuerdo al ordenamiento de uso normal de los recursos en el sistema. Por ejemplo, usualmente se utiliza primero una (UDD) antes que la (I), por lo cual es razonable definir F(UDD) < F(i).

EVASIÓN DE ABRAZOS MORTALES

Los algoritmos de prevención de abrazos mortales, como se discutió anteriormente, los previenen al restringir la manera en que se hacen las solicitudes. Las restricciones aseguran que al menos una de las condiciones necesarias no se presentará. Sin embargo, un efecto colateral de prevenir los abrazos mortales por este método es la posible baja utilización de los recursos y el reducido desempeño del sistema.

Un método alternativo para evitar los abrazos mortales requiere de información adicional acerca de cómo van a ser solicitados los recursos. Por ejemplo, en un sistema con una lectora óptica y una impresora, pudiera ser que se dijera que el proceso P solicitará la lectora óptica y después la impresora, antes de liberar ambos recursos. El proceso Q, por otro lado, solicitará primero la impresora y después la lectora óptica. Con este conocimiento de la secuencia completa de peticiones y liberaciones de cada proceso, se puede decidir para cada petición si el proceso debe esperar o no.

Cada solicitud requiere que el sistema considere los recursos actualmente disponibles, los recursos actualmente asignados a cada proceso y las solicitudes y liberaciones futuras de cada proceso, para decidir si la solicitud actual puede satisfacerse o debe esperar para satisfacer un posible abrazo mortal futuro.

Existen varios algoritmos que difieren en la cantidad y tipo de información que requieren. El modelo mas sencillo y mas útil requiere que cada proceso declare el máximo número de recursos de cada tipo que pudiera necesitar. Dada esta información a priori, para cada proceso, acerca del máximo número de recursos de cada tipo que puedan llegar a necesitar, es posible construir un algoritmo que asegure que el sistema nunca entrará en un estado de abrazo mortal. Este algoritmo define la aproximación de evasión de abrazos mortales. Un algoritmo de evasión de abrazos mortales dinámicamente examina el estado de asignación de recursos para asegurarse que nunca pueda existir una condición de espera circular.

El estado de asignación de los recursos es definido por el número de recursos disponibles, asignados y la demanda máxima de los procesos. Un estado es seguro si el sistema puede asignar recursos a cada proceso (hasta su máximo) en algún orden y aún puede evitar un abrazo mortal.

Mas formalmente, un sistema está en un estado seguro solo si existe una secuencia segura. Una secuencia de procesos <p1,p2,...,pn> es una secuencia segura para el estado actual de asignación si para cada pi, las solicitudes por recursos que pi puede aun solicitar pueden ser satisfechas con los recursos actualmente disponibles mas los recursos retenidos por todos los pj, con j<i. En esta situación, si los recursos requeridos por el proceso pi no están inmediatamente disponibles, entonces pi puede esperar hasta que todos los pj hallan terminado. Cuando ya han terminado, pi puede obtener todos los recursos que necesitaba, completando su tarea asignada, regresando los recursos asignados y terminando. Cuando pi termina, pi+1 puede obtener los recursos necesarios y así en adelante. Si no existe esa secuencia, entonces se dice que el sistema esta en un estado inseguro.

Un estado seguro no es un estado de abrazo mortal y un estado de abrazo mortal es un estado inseguro. Sin embargo, no todos los

estados inseguros son estados de abrazo mortal. Un estado inseguro puede llevar a un estado de abrazo mortal. Mientras el estado sea seguro, el sistema operativo puede evitar los estados inseguros (y de abrazo mortal). En un estado inseguro, el sistema operativo no puede prevenir que los procesos soliciten los recursos de tal forma que ocurra un abrazo mortal: el comportamiento de los procesos controla los estados inseguros.

Como ejemplo, considere un sistema con 12 unidades de disco (UDD) y tres procesos p0, p1 y p2. El proceso p0 requiere 10 UDD, p1 puede llegar a utilizar hasta 4 UDD y p2 puede llegar a utilizar hasta 9 UDD. Supóngase que al tiempo T0, el proceso p0 retiene 5 UDD, p1 retiene 2 y p2 retiene 2 (por ello, existen 3 UDD disponibles).

Necesidades Máximas Necesidades Actuales

p0 10 5

p1 4 2

p2 9 7

Al tiempo 0, el sistema está en un estado seguro. La secuencia <p1,p0,p2> satisface la condición de seguridad, debido a que a p1 pueden asignársele inmediatamente todas las UDD que requiere y después regresarlas cuando termine (el sistema tendrá entonces 5 UDD disponibles), p0 ahora podrá obtener todas las UDD que requiere y terminar (con lo que el sistema tendrá 10 UDD disponibles) y finalmente, p2 podrá obtener todas las UDD que requiera y regresarlas al terminar (en ese momento el sistema tendrá las 12 UDD disponibles).

Observe que es posible ir de un estado seguro a un estado inseguro. Suponga que el tiempo T1, p2 solicita que se le asigne 1 UDD mas. El sistema ya no estará mas en un estado seguro. En este punto, solamente p1 puede obtener todas las UDD que necesita. Cuando este termine y las regrese el sistema tendrá únicamente 4 UDD disponibles. Debido a que p0 tiene asignadas 5 UDD, pero requiere un máximo de 10, puede solicitar 5 mas. Como estas no están disponibles, p0 tendrá que esperar. Similarmente, p2 puede solicitar 5

UDD mas y tendrá que esperar, con lo cual se llega a un estado de abrazo mortal.

El error cometido fue permitir la solicitud del proceso p2 de una UDD mas. Si se hiciera que p2 esperará hasta que alguno de los otros procesos terminara y liberara sus recursos , entonces podría haberse evitado la situación de abrazo mortal.

Dado el concepto de estado seguro, se pueden definir algoritmos de evasión que aseguren que el sistema nunca entrará en un estado de abrazo mortal. La idea es simplemente asegurarse que el sistema siempre se permanece en un estado seguro. Inicialmente el sistema se encuentra en un estado seguro. Siempre que un proceso solicite un recurso que este actualmente disponible, el sistema debe decidir si el recurso puede ser asignado inmediatamente o si el proceso debe esperar. La solicitud se satisface solo si el sistema permanece en un estado seguro después de la asignación.

Observe que en este esquema si un proceso solicita un recurso que se encuentre actualmente disponible, puede que tenga que esperar. Por ello, la utilización de los recursos puede ser menor que sin un algoritmo de evasión de abrazos mortales.

DETECCIÓN DE ABRAZOS MORTALES

Si un sistema no utiliza algún protocolo para asegurar que no ocurrirá un abrazo mortal. Entonces se requiere implantar un esquema de detección y recuperación. Un algoritmo que examina el estado del sistema es invocado periodicamente para determinar cuando ha ocurrido un abrazo mortal. Si éste ha ocurrido, el sistema debe intentar recuperarse del abrazo mortal. Con la finalidad de hacerlo, el sistema debe:

a. Mantener información acerca del estado actual de asignación de recursos para cada uno de los procesos, así como también cualquier petición de asignación de recursos pendiente.

b. Proporcionar un algoritmo que utilice esa información para determinar cuando el sistema ha entrado en un estado de abrazo mortal.

Aún cuando no hemos trabajado sobre los puntos a y b mencionados, cabe hacer notar que el esquema de detección y recuperación requiere de "overhead" que incluye no solo el costo de ejecución de mantener la información necesaria y ejecutar el algoritmo de detección, sino que también las pérdidas potenciales inherentes al recuperarse de un abrazo mortal.

Varias instancias de un tipo de recurso

El algoritmo de detección emplea varias estructuras de datos variables que son muy similares a las utilizadas en el algoritmo del banquero.

Disponible. Un vector de longitud m indicando el número de recursos disponibles de cada tipo.

Asignación. Una matriz n x m definiendo el número de recursos de cada tipo que estan actualmente asignados a cada proceso.

Petición. Una matriz n x m indicando las peticiones actuales de cada proceso. Si

Petición[i,j] = k, entonces el proceso pi esta solicitando k instancias mas del tipo de recurso rj.

La relación menor-que (<) entre dos vectores esta definida como para el algoritmo del banquero. Para simplificar la notación, nuevamente trataremos las matrices Asignación y Petición como vectores y nos referiremos a ellos como Asignación i y Petición i, respectivamente. El algoritmo de detección descrito posteriormente se debe a Shoshani y Coffman [1970]; el algoritmo simplemente investiga cualquier secuencia de asignación posible para los procesos que aún no han sido completados.

1. Sea Trabajo y Terminar los vectores de longitud n y m respectivamente. Inicialice Trabajo := Disponible. Para i=1, 2, .., n, si Asignación i 0 entonces Terminar[i] := Falso; de otra manera, Terminar[i] := Verdadero.

2. Encuentre un índice tal que:

a. Terminar[i] = Falso y

b. Petición i Trabajo.

Si no existe tal I vaya al paso 4.

3. Trabajo := Trabajo + Asignacióni

Terminar[i] := Verdadero

vaya al paso 2.

4. Si terminar[i] = Falso para alguna i, 1 i n, entonces el sistema está en un estado de abrazo mortal. Además, si Terminar[i] = Falso entonces el proceso pi esta en el abrazo mortal.

Nos podemos preguntar porque se reclaman los recurso del proceso pi tan pronto como se determina que Petición i Trabajo. Sabemos que pi no esta actualmente en un abrazo mortal. Así, si suponemos una actitud optimista y suponemos que pi no requerira mas recursos para completar su tarea; este devolverá todas sus instancias de recursos al sistema. Si este no es es caso, puede ocurrir un abrazo mortal mas tarde, el cual será detectado la siguiente vez que el algoritmo de detección sea invocado.

Un ejemplo

Considere un sistema con cinco procesos {p0, p1,p2, p3, p4, } y tres tipos de recursos {A, B,C}. El tipo de recurso A tiene & instancias, el recurso B tiene 2 instancias y el recurso C tiene 6 instancias. Suponga que al tiempo T0 tenemos el siguiente estado de asignación de recursos:

Asignación Petición Disponible

A B C A B C A B C

p0 0 1 0 0 0 0 0 0 0

p1 2 0 0 2 0 2

p2 3 0 3 0 0 0

p3 2 1 1 1 0 0

p4 0 0 2 0 0 2

Aseguraremos que el sistema no está en un estado de abrazo mortal.De hecho si ejecutamos el algoritmo encontraremos que la secuencia <p0,p2,p3,p1,p4> resulta en Terminar := Verdadero para todo i.

Suponga ahora qye el proceso p2 hace una solicitud adicional por una instancia del tipo C. La matriz Petición se modifica como sigue.

Petición

A B C

p0 0 0 0

p1 2 0 2

p2 0 0 1

p3 1 0 0

p4 0 0 2

Decimos que ahora el sistema está en un estado de abrazo mortal. Aunque podemos reclamar los recursos mantenidos por el proceso p0, el número de recursos disponibles es insuficiente para satisfacer las peticiones de los otros procesos. Por ello, existe un abrazo mortal, que involucra a los procesos p1, p2, p3 y p4.

DETECCION DE PUNTOS MUERTOS

La detección de un deadlock se puede hacer por una gráfica de recursos. Dicha detección tendrá que llevarse a cabo por los manejadores distribuidos de transacciones, lo cual da lugar a algoritmos bastante complejos.

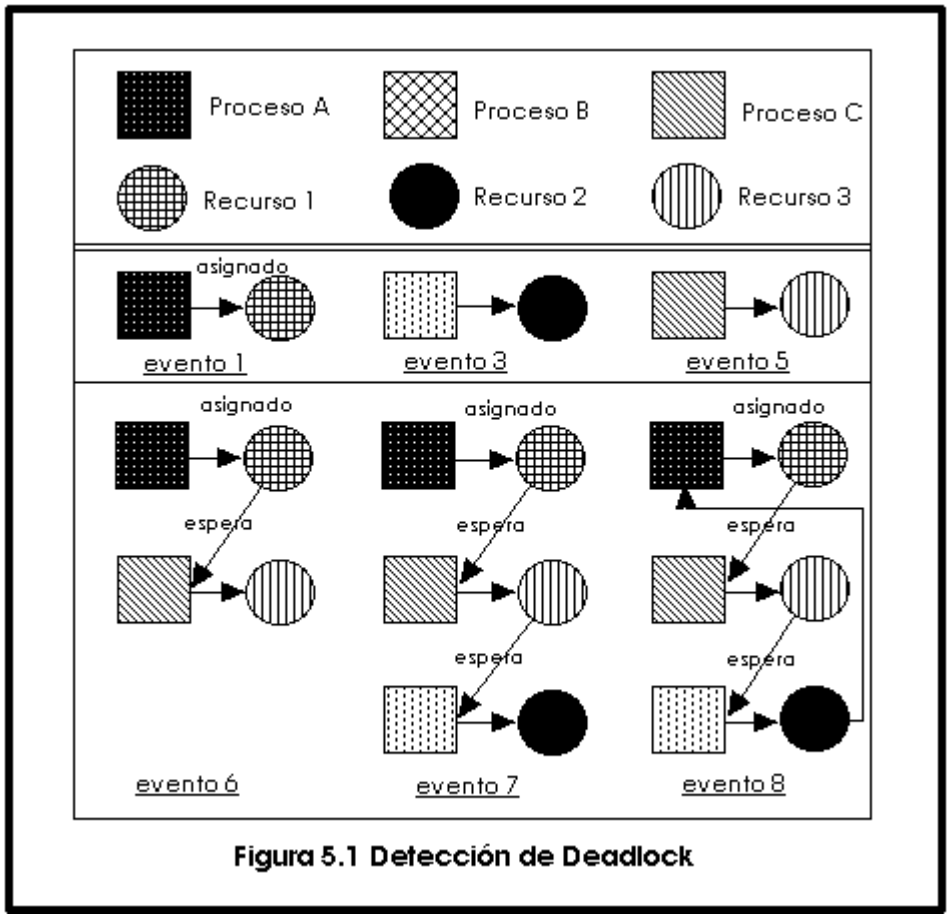


Figura 5.1 Detección de Deadlock

Una solución alternativa sería corregir el deadlock, y una opción para esto es que al solicitar un recurso (por ejemplo, un registro de una base de datos) se inicialice un contador que indique un tiempo de espera para recibir el bloqueo y si el tiempo de espera expira entonces se decide abortar. Esta opción tendrá un impacto muy grande dependiendo si el tiempo de espera es muy grande o pequeño, ya que no se sabe si el plazo expiró debido a un deadlock real o no. De esta manera, si el tiempo de espera es muy pequeño, crece la probabilidad de abortar una gran cantidad de transacciones que no estaban en deadlock, si el tiempo es muy grande se corre el peligro de desperdicialo cuando el deadlock existe e incluso de provocar otros.

Utilización del Algoritmo de Detección.

Ahora la pregunta es ¿Cuándo debemos invocar el algoritmo de detección?, La respuesta depende de dos factores:

1)¿Qué tan seguido creemos que ocurren los abrazos mortales?

2)¿Cuántos procesos se verán afectados por el abrazo mortal cuando este suceda?

Si los abrazos mortales ocurren frecuentemente, entonces el algoritmo de detección debe ser invocado mas frecuentemente. Los recursos asignados a los procesos en el abrazo mortal estarán ociosos hasta que el abrazo mortal sea eliminado.

Además, el número de procesos en el ciclo del estado de abrazo mortal puede crecer.

Los abrazos mortales pueden ocurrir cuando algún proceso hace una petición que no puede satisfacerse inmediatamente. Es posible que esta petición sea la que completa la cadena de procesos en espera. Lléndose a un extremo, puede invocarse el algoritmo de detección cada que una petición no pueda satisfacerse inmediatamente. En este caso, no solamente podremos identificar los procesos que se encuentran en el estado de abrazo mortal, sino el proceso específico que lo "provoco" (en realidad, cada uno de los procesos en abrazo mortal estan encadenados en un ciclo, de tal forma que todos han causado el abrazo mortal).

Es claro que el invocar el algoritmo de detección de abrazos mortales para cada petición introduce un considerable "overhead" en el tiempo de computación. Una alternativa menos costosa sería invocarlo en intervalos menos frecuentes, por ejemplo una vez cada hora o siempre que la utilización del CPU caiga por debajo del 40% (Un abrazo mortal eventualmente afecta el desempeño del sistema y causará que la utilización del CPU caiga). Si el algoritmo de detección es invocado en puntos arbitrarios de tiempo, puede ser que existan demasiados abrazos mortales, lo cual sería demasiado costoso de trabajar.

En general, no puede especificarse cual es la frecuencia ideal para invocar el algoritmo de detección en todos

Detección de Fallas en Circuitos Combinacionales

La detección de fallas es concerniente con la determinación donde existe una falla en un circuito. En el diagnostico de fallas, uno trata de localizar una falla específica en un sistema.

El diagnostico de fallas en los modernos sistemas de cómputo está haciéndose mas importante como la complejidad de dichos sistemas se incrementa. Un rápido y efectivo diagnostico de fallas en sistemas de computo es deseable desde que una vez que estos diagnósticos están en servicio, un tiempo menor puede ser mantenido al mínimo.

La detección y diagnostico de fallas en las primeras computadoras fueron realizados por técnicos. Aunque éstas computadoras eran físicamente muy extensas, el número de componentes en ellas era relativamente pequeño comparado con las computadoras actuales, y un ingeniero hábil puede localizar una falla en un periodo razonable de tiempo. Con la llegada de los transistores y los circuitos integrados, el número de componentes en una computadora moderna se ha incrementado enormemente y la detección de fallas y diagnósticos se han convertido en tareas cada vez mas difíciles.

La conclusión lógica es que si una computadora es totalmente operacional, debería diagnosticarse por si misma, o peor, otra computadora debería ser usada para realizar el diagnostico.

En esta sección consideraremos que un diagrama de circuito combinacional sea una grafica dirigida, cuyos nodos son las compuertas y sus aristas la conexiones entre las compuertas. A partir de esta representación un algoritmo será dado para generar una talla de fallas. El propósito de dicha tabla es que sea posible detectar y mas generalmente diagnosticar fallas. El algoritmo usará ciertas notaciones de relaciones semejantes a las de composición, simetría, transitividad, cerradura, etc. La primera parte de la sección da una descripción de fallas en circuitos combinacionales conectados. Esta discusión está seguida por una descripción de ciertas representaciones circuitales posibles de circuitos combinacionales. Un algoritmo que genera una matriz de falla es formulado.

Fallas en circuitos Combinacionales.

Como primera observación, pudimos ver el porque la gran velocidad de las computadoras digitales, todo esto es requerido en el orden de detectar y diagnosticar fallas es escribir un programa que asegure que una computadora pueda añadir todas las combinaciones de números apropiadamente, cambiando cualquier patrón de bits izquierdo o derecho, etc. si consideramos cuánto hora tomaría para realizar una prueba de la adición, llega a estar claro que tal acercamiento es vano. Asumamos que la computadora que se diagnosticará tiene una palabra de 32 bit y un tiempo de adición de 1 microsegundo el número de

combinaciones posibles de los operandos para una instrucción de la adición es aproximadamente 10^{19}. El número de instrucción adición requeridas para realizar una prueba tan exhaustiva tomaría cerca de 3×10^{5} años, una situación obviamente ridícula. Incluso, si este acercamiento no durara tanto indicaría si la computadora suma correctamente. Esta prueba sería una prueba funcional de la verificación más bien que una prueba de diagnóstico.

El ejemplo antedicho ilustra el punto que hay dos requisitos importantes de un procedimiento de diagnóstico:

1. El procedimiento debe ser tan rápidamente como sea posible, y por lo tanto debe implicar pocas pruebas como sea posible.

2. El resultado del procedimiento debe ser tan específico como sea posible. Por ejemplo, más bien que indicando eso "la computadora no puede sumar," debe indicar que es éste porque el llevar del bit 7 al bit 8 es incorrecto.

Para probar los circuitos combinacionales, debemos primero estar enterados de los tipos de fallas que puedan ocurrir y cuáles son sus efectos totales. Hay dos tipos principales de fallas que pueden ocurrir. El primer tipo consiste en fallas transitorias o intermitentes, y este tipo no se incluye en la discusión siguiente. El segundo tipo de falla es una falla permanente que puede originar de una falla de un componente, de un circuito abierto, de un corto circuito, etc. Tales averías serán denotadas como "fallas de pegado." Serán consideradas que existen en las terminales de la entrada y de salida de los componentes. Las fallas de pegado tienen el efecto de hacer que la compuerta de entrada parezca estar pegada en el valor 0 o 1. Se denotará la falla de pegado en uno y la falla de pegado en cero por los términos p-e-1 y p-e-0 respectivamente. Además el análisis sólo traerá circuitos que consisten de compuertas NAND.

Por ejemplo, considérese la compuerta NAND mostrada en la figura 1 Si la aplicación de los tres símbolos de entrada produce los resultados mostrados en la tabla 1, existe una falla p-e-1 en la terminal x, de acuerdo a la definición. El efecto de esta falla es que la salida se cambia de $x \uparrow y \uparrow z$ a $1 \uparrow y \uparrow z = y \uparrow z$; de forma similar, una falla p-e-0 en la terminal x produce una salida $0 \uparrow y \uparrow z = 1$. Esta última salida sería la misma en una falla p-e-1 en la salida. La tabla 1 muestra las salidas debidas a fallas p-e-1 y p-e-0 en la terminal x.

Tabla 1 ILUSTRACIÓN DE FALLAS PEGADAS EN

x	y	z	a p-e-1	a p-e-0
0	0	0	1	1
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	0	1

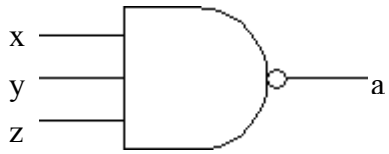


Figura 1 Una compuerta NAND

En el resto de esta sección también se supondrá que sólo existe una falla en el circuito, mientras se evalúa la detección de la falla.

Nociones de Detección de fallas.

Antes de intentar una formulación general del problema de detección de fallas, hay que examinar un ejemplo específico. En la figura 2, se da un circuito combinable que consiste de cuatro entradas y tres compuertas NAND.

Considérense todos los casos posibles en los que una terminal está pegada en 0 o en 1. hay catorce casos posibles que se muestran en la segunda columna de la Tabla 2. En este caso, (x,0) o (x,1)significa que la terminal x está pegada en 0 o en 1 respectivamente. Las fallas correspondientes se designan por $f_1, f_2, \ldots, f_{14}$ en la primera columna de la tabla. Sean las entradas en las terminales *a*, *b*, *c* y *d*, respectivamente. Para estas entradas la función de salida correspondiente a cualquiera de las fallas se muestra en la última columna de la tabla. Debido a la elección de entradas de inmediato se nota que la función de salida es $x_1 x_3$ para cada una de las fallas f_1, f_3 y f_{10}. Existe una situación similar para otros conjuntos de fallas. Por lo tanto, no sería posible distinguir entre las fallas f_1, f_3 o f_{10} de la salida. Más adelante, se regresará a este punto.

En vez de obtener la función de salida para cada falla individual, es posible obtener los valores de la salida para todas las combinaciones posibles de valores para las variables para cada falla sencilla y ponerlas en una *tabla de fallas*. En la tabla 3 se da esta tabla de fallas para el circuito de la figura 2

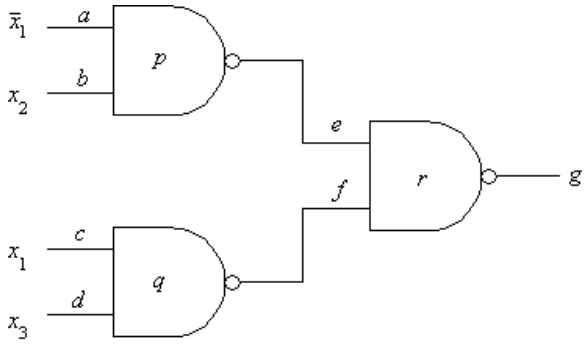


Figura 2. Un circuito de conmutación

Tabla 2 FALLAS SENCILLAS PARA EL CIRCUITO DE LA FIGURA 2

Designación de la falla	Descripción	Función en la salida (g)
f_1	$(a,0)$	x_1x_3
f_2	$(a,1)$	$x_2 + x_1x_3$
f_3	$(b,0)$	x_1x_3
f_4	$(b,1)$	$\bar{x}_1 + x_3$
f_5	$(c,0)$	$\bar{x}_1x_2$
f_6	$(c,1)$	$\bar{x}_1x_2 + x_3$
f_7	$(d,0)$	$\bar{x}_1x_2$
f_8	$(d,1)$	$x_1 + x_2$
f_9	$(e,0)$	1
f_{10}	$(e,1)$	x_1x_3
f_{11}	$(f,0)$	1
f_{12}	$(f,1)$	$\bar{x}_1x_2$
f_{13}	$(g,0)$	0
f_{14}	$(g,1)$	1

Tabla 3 RESULTADOS DE LAS PRUEBAS PARA EL CIRCUITO DE LA FIGURA 2

Test	x_1	x_2	x_3	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}
t_0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	1
t_1	0	0	1	0	0	0	0	1	0	1	0	0	1	0	1	0	0	1
t_2	0	1	0	1	0	1	0	1	1	1	1	1	1	0	1	1	0	1
t_3	0	1	1	1	0	1	0	1	1	1	1	1	1	0	1	1	0	1
t_4	1	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	1
t_5	1	0	1	1	1	1	1	1	0	1	0	1	1	1	1	0	0	1
t_6	1	1	0	0	0	1	0	0	0	0	0	1	1	0	1	0	0	1
t_7	1	1	1	1	1	1	1	1	0	1	0	1	1	1	1	0	0	1

Para circuitos pequeños, las tablas de fallas se pueden calcular o simular con facilidad. Sin embargo, el esfuerzo que se requiere del computador para obtener una tabla de fallas para circuitos más grandes es mucho. Por consiguiente, se formulará un algoritmo que genera, de una manera eficiente, la tabla de fallas.

Definición 1: Una prueba en un circuito combinable es la aplicación de una combinación de símbolos de entrada al circuito.

Se representará una prueba con t_i donde el subíndice *i* representa la elección decimal de su combinación de entrada correspondiente. Como un ejemplo, t_5 denota la prueba, en la Tabla 3, en la que $x_1 = 1$, $x_2 = 0$, y $x_3 = 1$.

Nótese que las columnas f_1 y f_3 en la Tabla 3 son idénticas en cualquier prueba efectuada; esto es, las pruebas efectuadas en el circuito no pueden distinguir entre fallas f_1 y f_3. Este fenómeno lleva a la siguiente definición.

Definición 2: En un circuito combinable, dos fallas f_i y f_j son *indistinguibles* si para cualquier prueba los valores correspondientes en la presencia de f_i sola y de f_j sola son los mismos. De lo contrario, las fallas son *distinguibles*.

Tabla 4 MATRIZ DE FALLAS PARA EL CIRCUITOEN LA FIGURA 2

Test	x_1	x_2	x_3	f_0	f_1	f_2	f_4	f_5	f_6	f_8	f_9	f_{13}
t_0	0	0	0	0	0	0	1	0	0	0	1	0
t_1	0	0	1	0	0	0	1	0	1	0	1	0
t_2	0	1	0	1	0	1	1	1	1	1	1	0
t_3	0	1	1	1	0	1	1	1	1	1	1	0
t_4	1	0	0	0	0	0	0	0	0	1	1	0
t_5	1	0	1	1	1	1	1	0	1	1	1	0
t_6	1	1	0	0	0	1	0	0	0	1	1	0
t_7	1	1	1	1	1	1	1	0	1	1	1	0

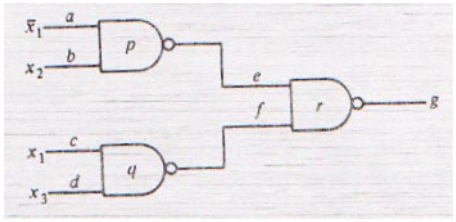
Se presenta una situación interesante cuando una columna f_i para una i, es idéntica a f_0. Esto indica que la presencia de f_i en el circuito no cambia la salida correcta. Esta situación ocurre en circuitos que tienen *redundancia*.

Las columnas f_1, f_3, y f_{10} de la Tabla 3 son indistinguibles; también lo son las columnas f_5, f_7 y f_{12}, y las columnas f_9, f_{11} y f_{14}. Por consiguiente, la Tabla 3 se puede reducir a la Tabla 4 en la que sólo se ha retenido una falla de cada conjunto de fallas indistinguibles. Tal tabla se llama una *matriz de falla*.

Ahora procedemos a formular un algoritmo que puede generar de manera mecánica una matriz de falla.

Procedimiento para la detección de fallas en circuitos combinacionales

Para describir el procedimiento que se sigue en la detección de fallas en los circuitos combinacionales se tomará en cuenta el siguiente ejemplo. Supóngase el siguiente circuito formado a partir de compuertas lógicas de tipo NAND:



Al analizar cuáles son las posibles fallas que se pueden tener en el circuito podemos armar la siguiente tabla:

Falla	Descripción
F1	(a,0)
F2	(a,1)
F3	(b,0)
F4	(b,1)
F5	(c,0)
F6	(c,1)
F7	(d,0)
F8	(d,1)
F9	(e,0)
F10	(e,1)
F11	(f,0)
F12	(f,1)
F13	(g,0)
F14	(g,1)

Se puede notar que tenemos 3 variables de entrada, por lo que se pueden plantear 8 pruebas. La tabla de verdad del circuito queda de la siguiente manera:

Prueba	X1	X2	X3	Resultado
T0	0	0	0	0
T1	0	0	1	0
T2	0	1	0	1
T3	0	1	1	1

T4	1	0	0	0
T5	1	0	1	1
T6	1	1	0	0
T7	1	1	1	1

Ahora, si se presentara alguna de las fallas posibles, los resultados de las tablas de verdad serían:

Prueba	**X1**	**X2**	**X3**	**F0**	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**F8**	**F9**	**F10**	**F11**	**F12**	**F13**	**F14**
T0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	1
T1	0	0	1	0	0	0	0	1	0	1	0	0	1	0	1	0	0	1
T2	0	1	0	1	0	1	0	1	1	1	1	1	1	0	1	1	0	1
T3	0	1	1	1	0	1	0	1	1	1	1	1	1	0	1	1	0	1
T4	1	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	1
T5	1	0	1	1	1	1	1	1	0	1	0	1	1	1	1	0	0	1
T6	1	1	0	0	0	1	0	0	0	0	0	1	1	0	1	0	0	1
T7	1	1	1	1	1	1	1	1	0	1	0	1	1	1	1	0	0	1

Se puede observar que varias fallas dan como resultado la misma tabla de verdad. Por tanto, se eligen aquéllas que no repitan el mismo resultado. Con esto se obtiene la **matriz de fallas** del circuito:

Prueba	**X1**	**X2**	**X3**	**F1**	**F2**	**F4**	**F5**	**F6**	**F8**	**F9**	**F13**
T0	0	0	0	0	0	1	0	0	0	1	0
T1	0	0	1	0	0	1	0	1	0	1	0
T2	0	1	0	0	1	1	1	1	1	1	0
T3	0	1	1	0	1	1	1	1	1	1	0
T4	1	0	0	0	0	0	0	0	1	1	0
T5	1	0	1	1	1	1	0	1	1	1	0
T6	1	1	0	0	1	0	0	0	1	1	0
T7	1	1	1	1	1	1	0	1	1	1	0

Ya teniendo esta matriz, se definirá otra que hará más útil la búsqueda de la falla. Esta matriz revisada se llamará **matriz de detección de fallas** para el circuito. Si para alguna prueba Tj, la salida del circuito en la falla Fi, es diferente de la salida observada para el circuito sin falla, entonces se pone un elemento 1 en la tabla; de lo contrario, se registra un valor 0. Haciendo esto se obtiene:

Prueba	**X1**	**X2**	**X3**	**F1**	**F2**	**F4**	**F5**	**F6**	**F8**	**F9**	**F13**
T0	0	0	0	0	0	1	0	0	0	1	0
T1	0	0	1	0	0	1	0	1	0	1	0
T2	0	1	0	1	0	0	0	0	0	0	1
T3	0	1	1	1	0	0	0	0	0	0	1
T4	1	0	0	0	0	0	0	0	1	1	0

T5	1	0	1	0	0	0	1	0	0	0	1
T6	1	1	0	0	1	0	0	0	1	1	0
T7	1	1	1	0	0	0	1	0	0	0	1

Aun es posible simplificar esta matriz. Para ello se escoge un número de pruebas suficiente para detectar todas las fallas. Con este procedimiento se obtiene:

Prueba	**X1**	**X2**	**X3**	**F1**	**F2**	**F4**	**F5**	**F6**	**F8**	**F9**	**F13**
T1	0	0	1	0	0	1	0	1	0	1	0
T2	0	1	0	1	0	0	0	0	0	0	1
T5	1	0	1	0	0	0	1	0	0	0	1
T6	1	1	0	0	1	0	0	0	1	1	0

Esta matriz es la **matriz de detección de fallas definitiva**. Para aplicarla, se aplica en el circuito cada una de las pruebas indicadas. En caso de que se obtenga un 1 en la salida, se encontró una falla en el circuito; de lo contrario se pasa a la siguiente prueba. Si las pruebas se terminan y nunca hubo un 1 en la salida, el circuito está libre de fallas. Lo anterior se muestra en el siguiente árbol: