Análisis y Diseño de Algoritmos

Postgrado en Computación

Prof. Isabel Besembel Carrera

GRAFOS
FUNDAMENTOS

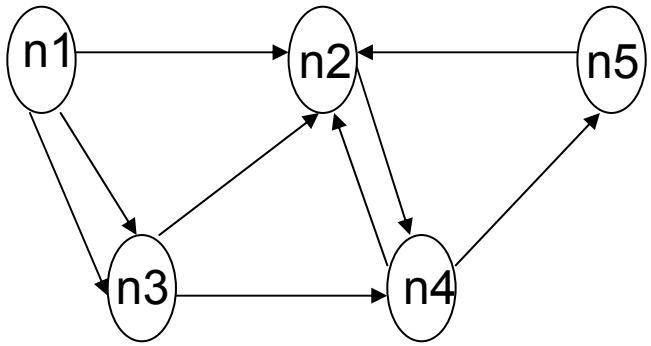
Introducción

- Las estructura de datos no lineales se caracterizan por no existir una relación de adyacencia, entre sus elementos, es decir, un elemento puede estar relacionado con cero, uno o más elementos.
- Entre las múltiples aplicaciones se tienen:
  - Los grafos son estructuras que se utilizan para modelar diversas situaciones tales como: sistemas de aeropuertos, flujo de tráfico, manejo de redes, etc.
  - Los grafos también son utilizados para realizar planificaciones de actividades, tareas del computador, planificar operaciones en lenguaje de máquinas para minimizar tiempo de ejecución.

Introducción

•La estructura no lineal de datos más general es el grafo, donde sus nodos pueden relacionarse de cualquier manera sin una relación de orden predefinida.

•Los grafos pueden ser utilizados como la estructura básica para múltiples aplicaciones en el área de la Computación



Definición

- Un grafo **G (N, A, f)** es un conjunto no vacío de:
  - **N**={n1, n2, ... , n_M) nodos o vértices,
  - **A**={a1, a2, ..., a $_K$} aristas y
  - la función **f : R** $\rightarrow \mathrm{M} \times \mathrm{M}$ que indica los pares de nodos que están relacionados.
- Tipos: según si existe o no dirección en sus relaciones
  - dirigidos o digrafos
  - no dirigidos
- Representación gráfica:
  - círculo o rectángulo para los nodos
  - líneas o flechas para las relaciones, según sea un grafo no dirigido o un digrafo, respectivamente

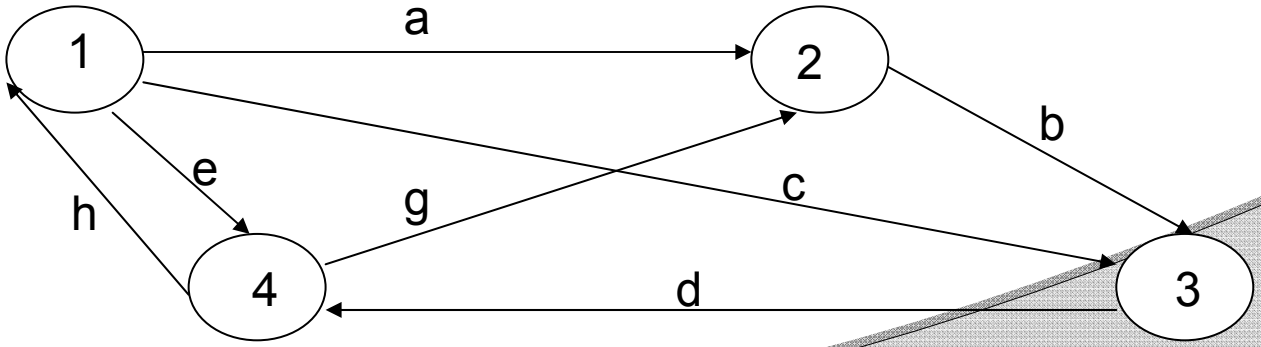
Ejemplo

Digrafo, donde D = {N, A, f} tiene

N = {1, 2, 3, 4},

A = {a, b, c, d, e, g, h} y

f(a) = (1, 2), f(b) = (2, 3), f(c) = (1, 3), f(d) = (3, 4), f(e) = (1, 4), f(g) = (4, 2), f(h) = (4, 1).

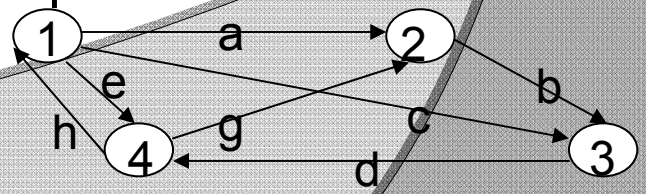


Grafos

- Los elementos de un digrafo D = {N, A, f} son el conjunto de nodos N, el conjunto de arcos A y la función f : A $\rightarrow$ N x N de pares ordenados de nodos que indican de que nodo sale el arco y a que nodo llega ese arco.
- Los elementos de un grafo no dirigido G = {N, L, f} son el conjunto de nodos N, el conjunto de líneas L y la función f : L $\rightarrow$ N x N de pares no ordenados que indican que esos nodos están unidos en ambas direcciones.

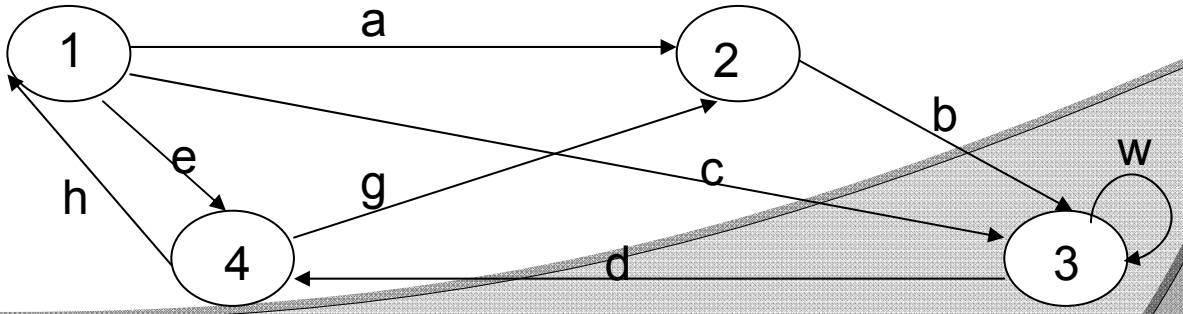
Conceptos básicos

- Se dice que el aj $\in$ A asociado al par de nodos (ni , nj) donde ni , nj $\in$ N, **comienza** en ni y **termina** en nj. Así, ni es el nodo **inicial** y nj es el nodo **terminal**. Ejemplo: si ni = 1, nj = 2 y aj = **a** en el digrafo D, entonces 1 es el nodo inicial de **a** y 2 es el nodo terminal de **a**.
- **Arco incidente**: aj es un arco incidente sobre nk , si nk es el nodo terminal de aj. Ejemplo: **a** es el arco incidente sobre 2, pero no sobre 1.
- **Arcos adyacentes**: ai y aj son arcos adyacentes si ai y aj son incidentes en el mismo nodo. Ejemplo: **c** y **b** son arcos adyacentes, pero **c** y **d** no lo son.
- **Arcos paralelos**: Dos arcos son paralelos si ellos comienzan y terminan en los mismos nodos. Ejemplo: En D no hay arcos paralelos. Si anexamos a D el arco **p** cuya f(p) = (1, 2), entonces **a** y **p** son arcos paralelos.



Conceptos básicos

- **Bucle o lazo:** Si un arco aj comienza en ni y termina en ni, entonces aj es un bucle. Ejemplo: Si f(w) = (3, 3) entonces w es un lazo.
- **Nodos adyacentes:** Dados un par de nodos (ni, nj) que están unidos por el arco aj, se dice que ni es adyacente a nj por aj. Ejemplo: 1 es adyacente a 2 por **a**.



Conceptos básicos

UNIVERSIDAD DE LOS ANDES

- **Grado de incidencia positivo:** El grado de incidencia positivo de un nodo nj es el número de arcos que tienen como nodo inicial a nj. Ejemplo: El grado de incidencia de 1 es igual a 3.
- **Grado de incidencia negativo:** El grado de incidencia negativo de un nodo nj es el número de arcos que terminan en nj. Ejemplo: El grado de incidencia negativo de 1 es igual a 1.
- **Grado de un nodo:** Para digrafos es el grado de incidencia positivo menos el grado de incidencia negativo del nodo. Ejemplo: El grado de 1 es igual a 3 – 1 = 2, el grado del nodo 4 es 2 – 2 = 0. Para grafos no dirigidos es el número de líneas asociadas al nodo.

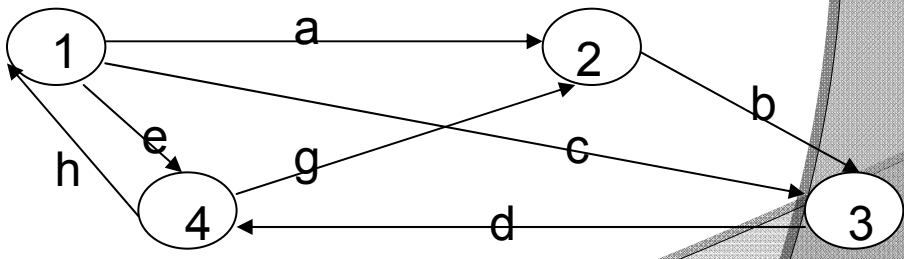
Conceptos básicos

- **Multiplicidad de un par ordenado (**ni**,** nj**):** Dado el digrafo D. Si para un par ordenado de nodos (ni, nj) $\in$ N x N, hay k arcos distintos en A para los que f(aj) = (ni, nj), es decir, hay k arcos que unen ni con nj nj, se dice que el par (ni, nj) tiene multiplicidad k donde k $\geq$ 0. La función de multiplicidad m (ni, nj) = k. Ejemplo: m(1, 1) = 0, m(1, 2) = 1, m(1, 3) = 1, etc.
- **Grafo simple:** Un digrafo D es simple si cada par ordenado (ni, nj) $\in$ N x N tiene como máximo de multiplicidad el valor 1. Ejemplo: El digrafo D de la figura es simple.

Conceptos básicos

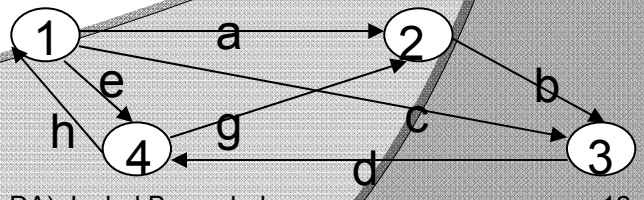
- **Matriz de conexión:** Todo grafo G tiene asociado una matriz M de dos dimensiones de orden |N| x |N|, cuyos elementos representan la multiplicidad del par (ni, nj). Ejemplo: La matriz de conexión del digrafo D es

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



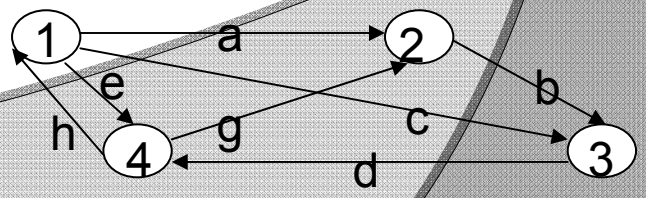
Conceptos básicos

- **Camino:** Un camino C del digrafo D es una secuencia de arcos (a1, a2, ..., an) tal que para todo par (ai, aj) de arcos consecutivos en C, el fin de ai coincide con el inicio de aj. Ejemplo: **b**, **d** es un camino, pero **a**, **c** no lo es.
- **Camino simple:** Es el camino que no recorre el mismo arco dos veces. Ejemplo: **b**, **d**, **g** es un camino simple, pero **b**, **d**, **g**, **b** no es un camino simple.
- **Camino elemental:** Es el camino que no visita un mismo nodo más de una vez. Ejemplo: **b**, **d** es un camino elemental, pero **b**, **d**, **g** no lo es.



Conceptos básicos

- **Circuito:** Es un camino finito en el que el origen del primer arco coincide con el fin del último. Ejemplo: **a**, **b**, **d**, **h** es un circuito, pero **c**, **d**, **g** no lo es.
- **Circuito simple:** Un circuito es simple si a su vez es un camino simple. Ejemplo: **b**, **d, g** es un circuito simple, pero **e**, **h**, **e**, **h** es un circuito no simple.
- **Orden de un camino:** Es el número de arcos que conforman el camino. Ejemplo: Un bucle tiene un orden igual a 1. **b**, **d, g** es un circuito simple de orden 3.



Conceptos básicos

UNIVERSIDAD DE LOS ANDES

- **Nodo accesible:** Un nodo nj es accesible desde otro nodo ni, si
  - 1) ni = nj o
  - 2) existe un camino desde ni hasta nj.
- **Nodo sucesor:** Un nodo nj es sucesor de un nodo ni si existe un camino que va desde ni hasta nj. Ejemplo: 2 es sucesor de 1.
- **Nodo predecesor:** Un nodo ni es predecesor de otro nodo nj si existe un camino que va desde ni hasta nj. Ejemplo: 1 es predecesor de 3.

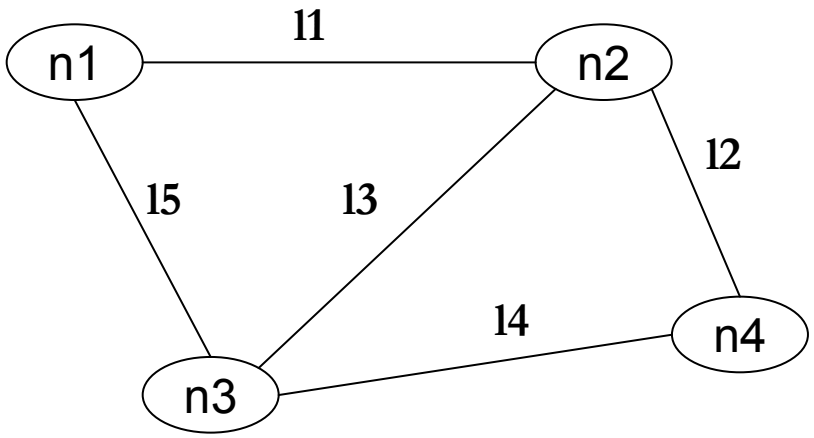
Conceptos básicos

UNIVERSIDAD DE LOS ANDES

- **Grafo conexo:** Un grafo G es conexo si y sólo si sus nodos ni no pueden ser particionados en dos conjuntos no vacíos N1 y N2 en los que sus caminos desde un punto cualquiera a otro estén en el mismo conjunto. Ejemplo: D es un grafo conexo.
- **Subgrafo:** Un subgrafo SG de G es un grafo constituido por un subconjunto SN de N y un subconjunto SA de A que conectan los nodos de SN. Ejemplo: SG = { SN, SA, sf } donde SN $\subset$ N y SA $\subset$ A, SN = {2, 3, 4}, SA = {b, d, g}, sf(b) = (2, 3), sf(d) = (3, 4), sf(g) = (4, 2).
- **Isomorfismo:** Dos grafos G y G' son isomórficos si existe una biyección f: N$\rightarrow$N' tal que (ni, nj) $\in$A $\Leftrightarrow$ (f(ni), f(nj)) $\in$A'.
- **Vecino:** En un dag G, el vecino de un nodo **n** es cualquier nodo adyacente a **n** en la versión no dirigida de G.

Conceptos básicos

- Grafo no dirigido (GND): G = {N, L, f}



GND1

Conceptos básicos

- **Cadena:** Para un grafo no dirigido GND es la secuencia de líneas (l1, l2, ..., ln) tal que el fin l_i coincide con el origen de l_{i+1}. Ejemplo: Para el grafo GND1, l3, l2, l4 es una cadena, pero l1, l2 no lo es.
- **Cadena simple:** Es la cadena donde ninguna línea se repite. Ejemplo: Para el grafo GND1, l3, l2, l4 es una cadena simple, pero l3, l2, l4, l4 no lo es.
- **Cadena elemental:** Es la cadena donde ningún nodo se repite. Ejemplo: Para el grafo GND1, l3, l2, l4 es una cadena elemental, pero l2, l4, l4 no lo es.

Conceptos básicos

- **Ciclo:** Es una cadena finita donde el nodo inicial de la cadena coincide con el nodo terminal de la misma.
- **Ciclo simple:** Es el ciclo que a su vez es una cadena simple. Ejemplo: El grafo GND1 no tiene ciclos simples

Conceptos básicos

UNIVERSIDAD DE LOS ANDES

- **Grafo múltiple:** Es un grafo que contiene alguna arista paralela.
- **Digrafo acíclico:** Es un digrafo que no contiene circuitos. Se le conoce con las siglas **dag**.
- **Grafo o digrafo con peso:** Es un grafo o digrafo que tiene un valor entero o real asignado a cada arista.
- **Grafo completo:** Es un grafo no dirigido donde cada par de nodos es adyacente

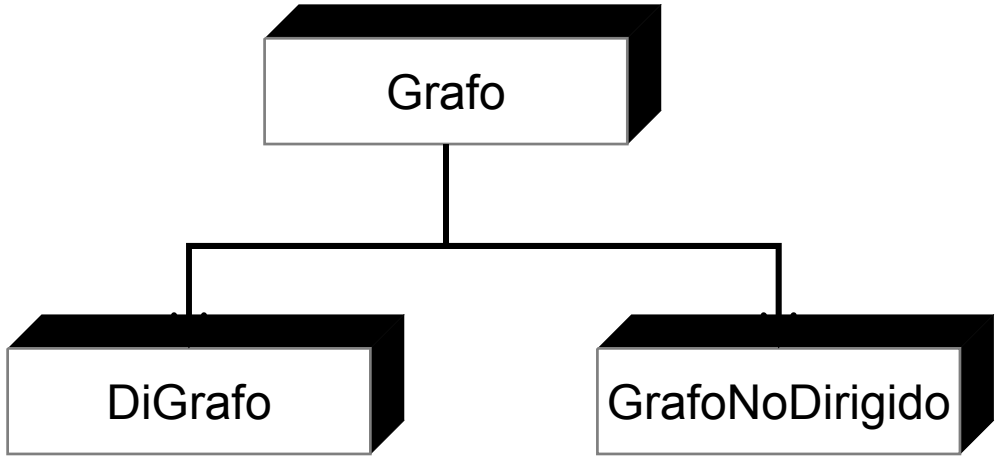
Conceptos básicos

UNIVERSIDAD DE LOS ANDES

- **Grafo bipartito:** Es un grafo no dirigido que puede ser dividido en dos subgrafos sin conexión entre ellos.
- **Multigrafo:** Es un grafo no dirigido que puede tener varias aristas entre nodos y lazos.
- **Hipergrafo:** Es un grafo no dirigido con hiper-aristas que conectan varios nodos. Ejemplo: hipertexto e hipermedia.

Especificación del TAD Grafo

Un grafo se especifica en base a sus funciones básicas, entre las que se pueden mencionar: crearlo, insertar un nodo nuevo o una arista nueva, conocer si está vacío o no, consultar si un nodo existe, etc.



Grafos etiquetados
w: $A \rightarrow \Re$

Especificación del TAD Grafo

Marzo, 2005 Especificación Grafo[TipoEle]		
1	Sintáctica creaGrafo()→Grafo, nuevoNodo(Grafo,TipoEle) →Grafo, nuevaArista(Grafo,TipoEle,TipoEle) →Grafo, eliNodo(Grafo,TipoEle) →Grafo, eliArista(Grafo,TipoEle,TipoEle) →Grafo, conNodo(Grafo,TipoEle) →Lógico, conArista(Grafo,TipoEle,TipoEle) →Lógico, nodoAdyacente(Grafo,TipoEle) →Lista[TipoEle], vacíoGrafo(Grafo) →Lógico, destruyeGrafo()→.	-creaGrafo(): Crea un grafo vacío. -nuevoNodo(): Ingresa un nuevo nodo al grafo -nuevaArista(): Ingresa una nueva arista entre los dos nodos especificados, si existen. -eliNodo(): Elimina un nodo del grafo. -eliArista(): Elimina una arista del grafo entre los nodos especificados. -conNodo(): Regresa verdadero sí existe en el grafo. -conArista(): Regresa verdadero si el arco existe. -nodoAdyacente(): Regresa la lista de los arcos adyacentes a él. -vacíoGrafo(): Regresa verdadero si es el grafo esta vacío -destruyeGrafo(): Destruye el grafo
2	Declaraciones TipoEle: no,{TipoNoDef}	
3	Semántica vacíoGrafo(creaGrafo())=Verdadero vacíoGrafo(nuevoNodo(creaGrafo(),no)=Falso conNodo(creaGrafo(), no)=Falso conNodo(nuevoNodo(creaGrafo(),no),no)=Verdadero conArista(creaGrafo(),no,no)=Falso conArista(nuevaArista(creaGrafo(),no,no),no,no)=Cierto eliNodo(creaGrafo(),no)=creaGrafo()	

Implementación del TAD Grafo

UNIVERSIDAD DE LOS ANDES

- Para las estructuras no lineales de datos se tienen las mismas formas de almacenamiento que para las estructuras lineales, a saber: las que usan memoria continua, denominado aquí método secuencial y las que usan memoria dispersa o método enlazado
- Existen dos métodos para almacenar un grafo G:
  - Matriz de adyacencia: se implementa utilizando el método secuencial o con arreglos y
  - Listas de adyacencia implementada utilizando el método enlazado o listas enlazadas

Implementación de TAD Grafo

UNIVERSIDAD DE LOS ANDES

Marzo, 2005

Grafo[TipoEle]

Clases: Arreglo de Entero, Entero, Tabla, TipoEleTab, Salida[TipoEle], TipoEle

1	Superclase: ninguna	-Grafo(). Constructor. Crea un grafo vacío
2	Estructura: privado:	-nuevoNodo(). Transformador. Inserta un nuevo nodo en el grafo.
3	Operaciones: público: Grafo() nuevoNodo(TipoEle: n) nuevaArista(TipoEle: ni, TipoEle: nf): eliNodo(TipoEle: n) eliArista(TipoEle: ni, TipoEle: nj) conNodo(TipoEle: n): Lógico conArista(TipoEle: ni, TipoEle: nf):Lógico nodoAdyacente(TipoEle: n): Arreglo[100] de TipoEle recorridoEnAmp() recorridoEnProf() rutaOptima(TipoEle: nj): Arreglo[100] de Salida[TipoEle] vacíoGrafo():Lógico numNodos():Entero ~Grafo()	-nuevaArista(). Transformador. Ingresa un nuevo arco al grafo. -eliNodo(). Transformador. Elimina un nodo del grafo -eliArista(). Transformador. Elimina un arco del grafo. -conNodo(). Observador. Regresa verdadero, si existe. -conArista(). Observador. Regresa verdadero si existe el arco. -nodoAdyacente(). Observador. Regresa el arreglo de nodos adyacentes. -recorridoEnAmp(). Observador. Recorre el grafo en amplitud. -recorridoEnProf(). Observador. Recorre el grafo en profundidad -rutaOptima(). Observador. Regresa un arreglo con los nodos que conforman el camino de menos arcos a partir de un nodo inicial -vacioGrafo(). Observador. Regresa verdadero si el grafo está vacío. -numNodos(). Observador. Regresa el número de nodos del grafo. ~Grafo(). Destructor. Destructor del grafo.

Matriz de adyacencia

UNIVERSIDAD DE LOS ANDES

- Es un arreglo de dos dimensiones que representa las conexiones entre pares de nodos o vértices.
- Sea un grafo G con un conjunto de nodos N y un conjunto de aristas A. Suponga que el grafo es de orden **n** (número de nodos del grafo), donde **n** > 0.
- La matriz de adyacencia se representa por un arreglo de tamaño **n** x **n**, donde:

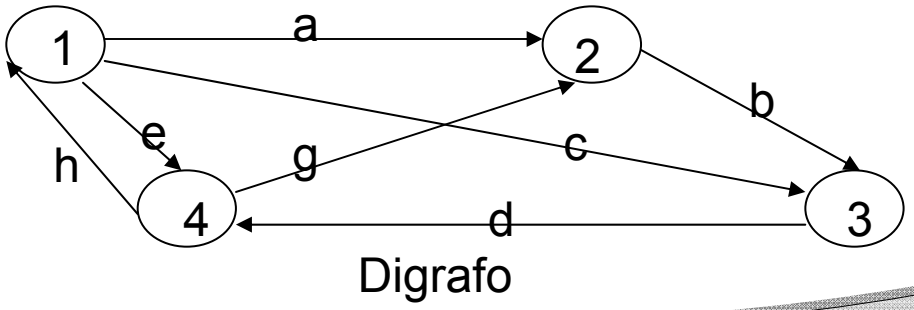
$$M(i,j) = \begin{cases} 1 & \text{si existe un arco(Ni,Nf) en A, Ni es adyacente a Nf} \\ 0 & \text{en caso contrario} \end{cases}$$

- Las columnas y las filas de la matriz representan los nodos del grafo.
- Si existe una arista desde i hacia j (esto es, el nodo i es adyacente al nodo j) se almacena 1, si no existe la arista, se introduce 0
- Si el grafo es no dirigido, la matriz es simétrica M(i,j) = M(j,i).

Matriz de adyacencia

Del nodo \ Al nodo	1	2	3	4
1	0	1	1	1
2	0	0	1	0
3	0	0	0	1
4	1	1	0	0

M del grafo dirigido

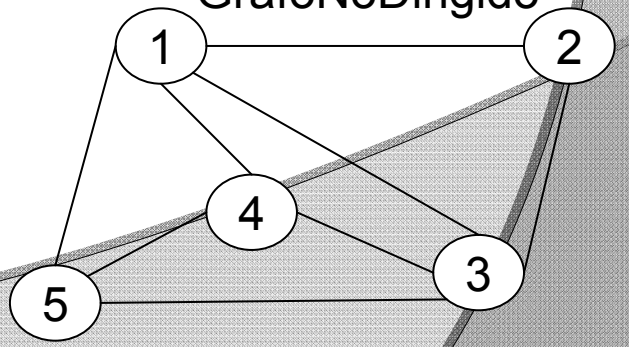


Digrafo

M del grafo no dirigido

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

GrafoNoDirigido

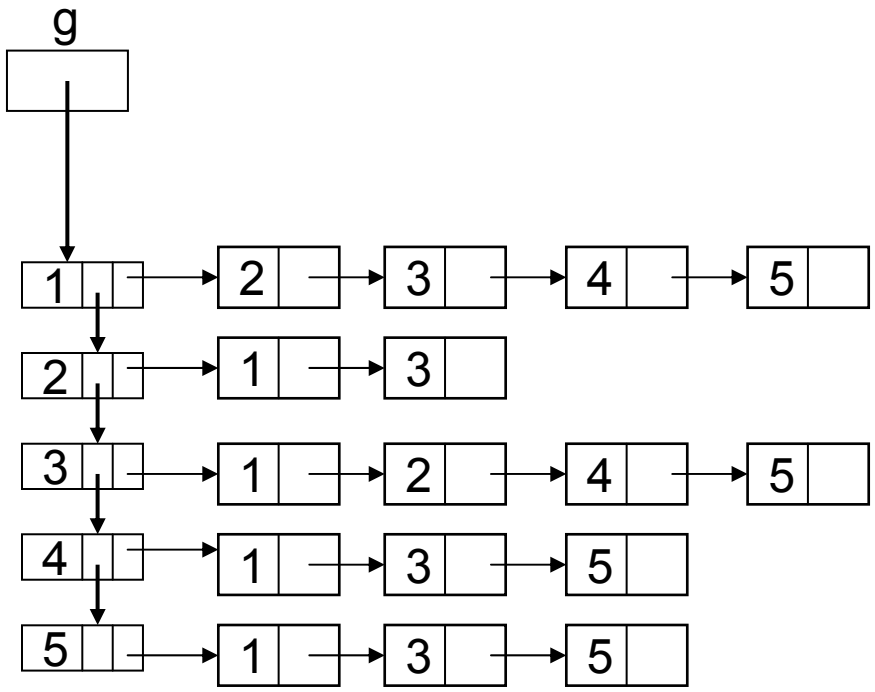


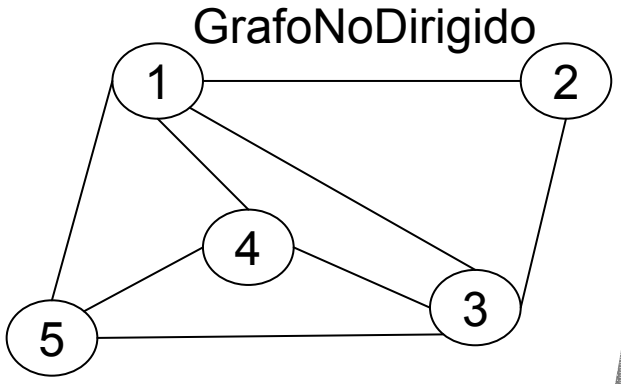
Listas de adyacencia

UNIVERSIDAD DE LOS ANDES

- El segundo método utilizado para representar grafos es útil cuando un grafo posee muchos nodos y pocas aristas (grafos no densos).
- Se utiliza una lista enlazada, llamada directorio, que almacena los nodos N del grafo y asociado a cada nodo del grafo se encuentra una lista enlazada con los nodos que son adyacentes a éste.
- Representa las aristas del grafo.
- Un grafo no dirigido de orden N con A aristas requiere N entradas en el directorio y 2*A entradas de listas enlazadas, excepto si existen bucles, que reduce el número de entradas a la lista en 1

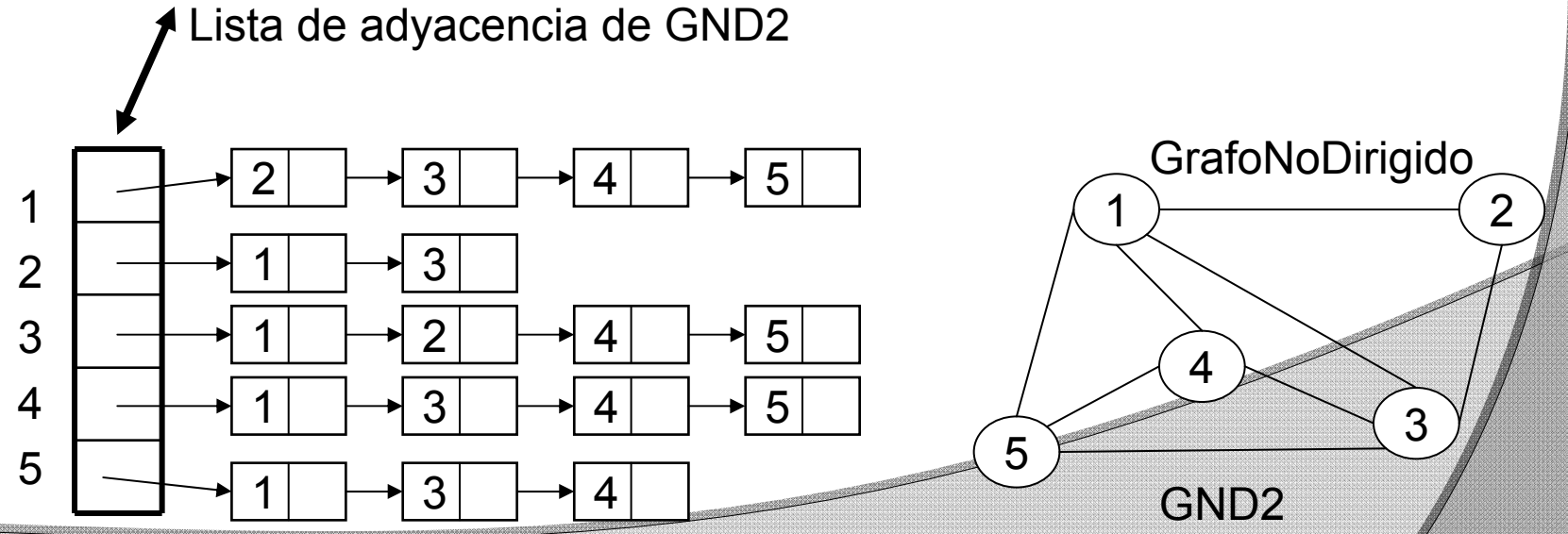
Listas de adyacencia





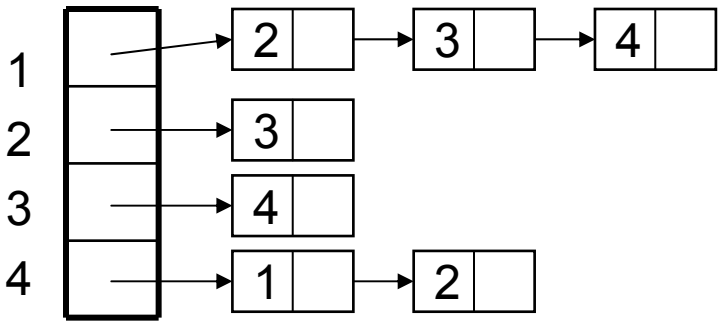
Listas de adyacencia

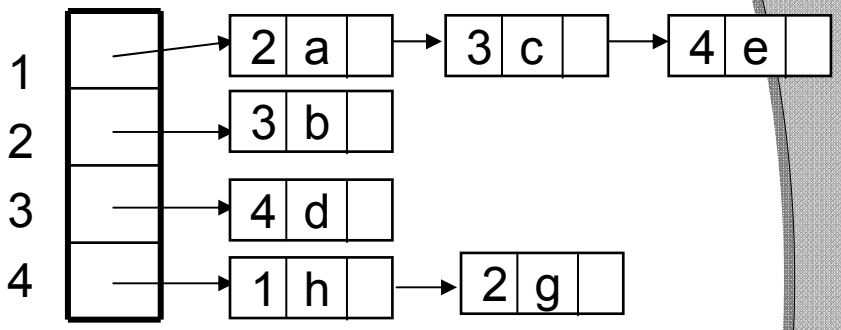
- Un grafo dirigido de orden N con A arcos requiere N entradas en el directorio y A entradas de listas enlazadas
- Una manera alterna de representar grafos utilizando listas de adyacencia es almacenar el directorio en un vector



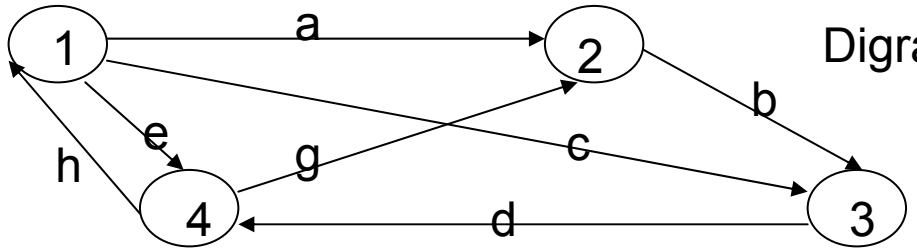
Listas de adyacencia

Lista de adyacencia del Digrafo1



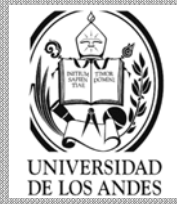


Lista de adyacencia del Digrafo1 etiquetado



Digrafo1

Implantación de un Digrafo usando matriz de adyacencia



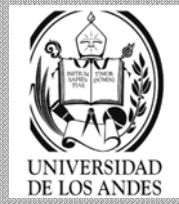
Marzo, 2005

DigrafoMat[TipoEle]

Clases: Arreglo de Entero+, Entero+, Tabla[TipoEleTab], Salida[TipoEle], TipoEle

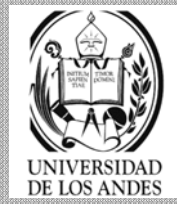
1	Superclase: Grafo	-max: maximo número de elementos del arreglo.
2	Estructura: privado: max: Entero+ =100 pos: Entero+ =0 g: Arreglo[100,100] de Entero + dir: Arreglo[100] de TipoEle t: Tabla[TipoEleTab]	-pos: Posición libre en la matriz. -g: Matriz de adyacencia para representar el digrafo. -dir: Almacena los nombres del cada nodo en la posicion correspondiente en la matriz. -t: Permite guardar el nombre del nodo con su posicion en la matriz.
3	Operaciones: público: DigrafoMat() nuevoNodo(TipoEle: n) nuevoArco(TipoEle: ni, TipoEle: nf) eliNodo(TipoEle: n) eliArco(TipoEle: ni, TipoEle: nj) conNodo(TipoEle: n): Lógico conArco(TipoEle: ni,TipoEle: nf):Lógico nodoAdyacente(TipoEle: n): ArregloDeTipoEle recorridoEnAmp() recorridoEnProf() rutaOptima(TipoEle: nj): ArregloDe Salida[TipoEle] numNodos(): Entero incidenciaEnt(TipoEle: n): Entero incidenciaSal(TipoEle: n): Entero vacíoDigrafo(): Lógico ~DigrafoMat()	-DigrafoMat(). Constructor. Crea un digrafo vacío -nuevoNodo(). Transformador. Inserta un nuevo nodo en el digrafo. -nuevoArco(). Transformador. Ingresa un nuevo arco al digrafo. -eliNodo(). Transformador. Elimina un nodo del digrafo -eliArco(). Transformador. Elimina un arco del digrafo. -conNodo(). Observador Regresa el nodo si existe. -conArco(). Observador. Regresa verdadero si existe el arco. -nodoAdyacente(). Observador. Regresa el arreglo de nodos adyacentes. -recorridoEnAmp(). Observador. Recorrido en amplitud -recorridoEnProf(). Observador. Recorrido en profundidad -rutaOptima(). Observado. Regresa en un arreglo los nodos que conforman el camino con menos arcos. -numNodos(). Observador. Regresa el número de nodos del grafo. -incidenciaEnt(). Observador. Da el grado de incidencia de entrada de un nodo. -incidenciaSal(). Observador. Da el grado de incidencia de salida de un nodo. -vacíoDigrafo(). Observador. Regresa verdadero si el digrafo está vacío. -~DigrafoMat(). Destructor.

Implementación de la clase Salida



Marzo/05	Salida[TipoEle] Clases: Entero+, TipoEle	
1 2 3	Superclase: ninguna Estructura: privado: dis: Entero+ =0 padre: TipoEle Operaciones: público: Salida() Dis(): Entero Dis(Entero: d) Padre(): TipoEle Padre(TipoEle: el)	-dis: Número de arcos recorridos. -padre: Nombre del nodo padre adyacente a él. -Dis(). Observador y transformador. Almacena la distancia del nodo origen a ese nodo. -Padre(): Observador y transformador. Almacena el valor del nodo que lo antecede en el camino desde el nodo origen.

Implantación de un Digrafo usando listas de adyacencia



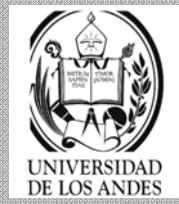
Marzo, 2005

DigrafoLis[TipoEle]

Clases: TipoEle, Nodo1[TipoEle], Lista[Nodo1[TipoEle]], ArregloDe[TipoEle]

1	Superclase: Grafo	-g: Lista de adyacencia para representar el DigrafoLis.
2	Estructura: privado: g: Lista[Nodo1[TipoEle]]	-DigrafoLis(). Constructor. Crea un digrafo vacío
3	Operaciones: público: DigrafoLis() nuevoNodo(TipoEle: n) nuevoArco(TipoEle: ni,TipoEle: nf) eliNodo(TipoEle: n) eliArco(TipoEle: ni, TipoEle: nj) conNodo(TipoEle: n): Lógico conArco(TipoEle: ni, TipoEle: nf): Lógico nodoAdyacente(TipoEle: n): Lista[TipoEle] recorridoEnAmp() recorridoEnProf() rutaOptima(TipoEle: n)Lista[Salida[TipoEle]] numNodos(): Entero incidenciaEnt(TipoEle: n): Entero incidenciaSal(TipoEle: n): Entero vacíoDigrafo(): Lógico ~DigrafoLis()	-nuevoNodo(). Transformador. Inserta un nuevo nodo en el digrafo. -nuevoArco(). Transformador. Ingresa un nuevo arco al digrafo. -eliNodo(). Transformador. Elimina un nodo del digrafo -eliArco(). Transformador. Elimina un arco del digrafo. -conNodo(). Observador. Regresa verdadero sí el nodo existe. Deja el cursor de la lista antes del nodo que lo contiene. -conArco(). Observador. Regresa verdadero si existe el arco. -nodoAdyacente(). Observador. Regresa el arreglo de nodos adyacentes. -recorridoEnAmp(). Observador. Recorre el grafo en amplitud. -recorridoEnProf(). Observador. Recorre el grafo en profundidad. -rutaOptima(). Observador. Regresa en un arreglo los nodos que conforman el camino con menos arcos. -incidenciaEnt(). Observador. Da el grado de incidencia de entrada de un nodo. -incidenciaSal(). Observador. Da el grado de incidencia de salida de un nodo. -vacíoDigrafo(). Observador. Regresa verdadero si el digrafo está vacío. numNodos(). Observador. Regresa el número de nodos del grafo. -~DigrafoLis(). Destructor.

Implantación de la clase Nodo1



Marzo, 2005 Nodo1[TipoEle] Clases: Entero, TipoEle, ApuntadorA, Lista[TipoEle]		
1 2	Superclases: ninguna Estructura: privado: infor: TipoEle padre:TipoEle distancia: Entero+ listaAdya: Lista[TipoEle] pSig: ApuntadorA Nodo1[TipoEle] = Nulo	-infor. Nombre del nodo. -padre. Nombre del nodo padre. -distancia. Distancia para llegar a ese nodo desde el nodo inicio. -listaAdy. Lista de adyacencia de cada nodo. -pSig. Apuntador al siguiente nodo. -Nodo1(): Constructor -Infor(). Observador y transformador. Da la identificación del nodo. -Padre(). Observador y transformador. Da la información del nodo padre en el calculo del camino. -Distancia(). Observador y transformador. Da la distancia al nodo en el cálculo del camino. -ListaAdya(). Observador y transformador. Da la información de la lista de nodos adyacentes. -PSig():Observador y transformador. Da la información del apuntador al siguiente nodo.
3	Operaciones: público: Nodo1() Infor(): TipoEle Infor(TipoEle: e) ListaAdya(): Lista[TipoEle] ListaAdya(Lista[TipoEle]: lad) Padre():TipoEle Padre(TipoEle: p) Distancia():Entero Distancia(Entero: d) PSig():ApuntadorA Nodo1[TipoEle] PSig(ApuntadorA Nodo1[TipoEle]: pn)	

Implementación de las clases Nodo y Salida

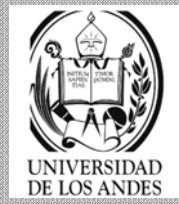
UNIVERSIDAD DE LOS ANDES

Marzo/05	Nodo[TipoEle] Clases: TipoEle, ApuntadorA	
1 2	Superclases: ninguna Estructura: privado: info: TipoEle pSig: ApuntadorA Nodo[TipoEle] = Nulo	-info. Información del nodo adyacente. -pSig. Apuntador al siguiente nodo adyacente. -Nodo(). Constructor. -Info(). Observador y transformador. Da la información del nodo adyacente. -PSig():Observador y transformador. Da la información del apuntador al siguiente nodo adyacente.
3	Operaciones: público: Nodo() Info(): TipoEle Info(TipoEle: e) PSig():ApuntadorA Nodo[TipoEle] PSig(ApuntadorA Nodo[TipoEle]: pn)	

Marzo/05	Salida[TipoEle] Clases: Entero, TipoEle	
1 2 3	Superclase: ninguna Estructura: privado: nNodo: TipoEle padre: TipoEle Operaciones: público: Salida() NNodo(): TipoEle NNodo(TipoEle: n) Padre(): TipoEle Padre(TipoEle: el)	-nNodo: Nombre del nodo. -padre: Nombre de su nodo padre adyacente. -NNodo(). Observador y transformador. Almacena la distancia del nodo origen a ese nodo. -Padre(): Observador y transformador. Almacena el valor del nodo que lo antecede en el camino desde el nodo origen.

Lista[TipoEle] Clases: Entero, Lógico, ApuntadorA, TipoEle, Nodo2[TipoEle]		
1	Superclases: ninguna	-n: Número actual de elementos. -pCab: Apuntador al nodo especial. -pCursor: Apuntador al nodo actual. -Lista(). Constructores. Crea una lista vacía o una lista copia de la lista enviada por parámetro. -~Lista(). Destructor. Destruye la lista. -insertar(). Transformador. Ingresa un nuevo elemento luego del nodo actual. -eliminar(). Transformador. Elimina el elemento después del nodo actual, si la lista no está vacía. -consultar(). Observador. Devuelve el elemento en el nodo actual, si existe. -estaVacía(). Observador. Regresa Verdadero si la lista está vacía. -numEle(). Observador. Regresa el número actual de elementos en la lista. -cursorAlInicio(). Transformador. Coloca el cursor antes del primer nodo. -cursorAlFinal(). Transformador. Coloca el cursor en el último nodo (antes de la cabeza). -cursorAlProximo(). Transformador. Avanza el cursor al próximo nodo. -cursorAlAnterior(). Transformador. Retrocede el cursor al nodo anterior. -despliegue(). Observador. Despliega el contenido de la lista. -union(). Generador. Genera una lista con la unión de las dos listas. -interseccion(). Generador. Genera una lista con los elementos comunes de las dos listas. -fision(). Generador. Genera dos listas con el contenido de la mitad de la lista original cada una. - = (). Observador. Regresa Verdadero si las dos listas son iguales en contenido. - ≠ (). Observador. Regresa Verdadero si las dos listas son diferentes en contenido. - = (). Transformador. Asigna una copia de la lista a la lista resultante.
	Estructura: privado: n : Entero+ = 0 pCab : ApuntadorA Nodo2[TipoEle]= Nulo pCursor : ApuntadorA Nodo2[TipoEle] = Nulo	
3	Operaciones: público: Lista() Lista(Lista[TipoEle]) ~Lista() insertar(TipoEle: e) eliminar() consultar(): TipoEle estaVacía(): Lógico numEle(): Entero+ cursorAlInicio() cursorAlFinal() cursorAlProximo() cursorAlAnterior() despliegue() union(Lista[TipoEle]: l1): Lista[TipoEle] interseccion(Lista[TipoEle]: l1): Lista[TipoEle] fision(Lista[TipoEle]: l1): Lista[TipoEle] = (Lista[TipoEle]: l1): Lógico ≠ (Lista[TipoEle]: l1): Lógico = (Lista[TipoEle]: l1): Lista[TipoEle]	

Implementación de la clase Nodo2



Nodo2[TipoEle] Clases: TipoEle, ApuntadorA		
1	Superclases: ninguna	-pAnt: Dirección del nodo anterior. -info: Información del nodo. -pSig: Dirección del nodo siguiente. -Nodo2(). Constructor. Crea un nodo con un elemento no definido y dos apuntadores nulos. -PAnt(). Transformador. Modifica el apuntador al nodo anterior por la dirección que viene en el parámetro. -Info(). Transformador. Modifica la información del nodo por la que viene en el parámetro. -PSig(). Transformador. Modifica el apuntador al nodo siguiente por la dirección que viene en el parámetro. -PAnt(). Observador.Regresa la dirección del nodo anterior. -Info(). Observador. Devuelve la información contenida en el nodo. -PSig(). Observador. Regresa la dirección del nodo siguiente.
2	Estructura: privado: pAnt : ApuntadorA Nodo2[TipoEle] = Nulo info: TipoEle = {TipoEleNoDef} pSig : ApuntadorA Nodo2[TipoEle] = Nulo	
3	Operaciones: público: Nodo2() PAnt(ApuntadorA Nodo2[TipoEle]: pa) PAnt() : ApuntadorA Nodo2[TipoEle] Info(TipoEle: e) Info(): TipoEle PSig(ApuntadorA Nodo2[TipoEle]: pn) PSig(): ApuntadorA Nodo2[TipoEle]	

Implementación del DigrafoMat

Marzo, 2005	DigrafoMat() {pos: g (i, j) = 0 ∀ i,j}	
1 2	[[g (i, j) = 0] j = 0, 99] i = 0, 99 pos = 0	-g, pos: Definido en DigrafoMat. -i, j:Entero. Contadores para recorrer el arreglo

Constructor de DigrafoMat. O(N^2)

27/04/98	vacíoDigrafo(): Lógico	
{pre: g (i, j) ≥ 0 ∀ i,j}		{pos: g (i, j) ≥ 0 ∀ i,j}
1	regrese (pos = 0)	-pos: Definido en DigrafoMat.

Verifica sí el digrafo está vacío O(1)

Marzo, 2005	nuevoNodo(TipoEle: n)	
{pre: g (i, j) ≥ 0 ∀ i,j}		{pos: g (i, j) ≥ 0 ∀ i,j}
1 2	ele = t.conTabla(n) Si (ele.Clave() ≠ n) entonces ele.Clave(n) ele.Resto(pos) t.insTabla(ele) dir(pos) = n pos = pos+1 fsi	-t, pos, dir: Definido en DigrafoMat. -ele:TipoEleTab. Variable auxiliar.

Ingresa un nuevo nodo al digrafo. O(N).

Implementación del DigrafoMat

Marzo, 2005	nuevoArco(TipoEle: ni, TipoEle nj)	
{pre: g (i, j) ≥ 0 ∀ i,j}		{pos: g (i, j) ≥ 0 ∀ i,j}
1	ele = t.conTabla(ni)	-t, g: Definido en DigrafoMat.
2	ele1 = t.conTabla(nj)	-ele, ele1:TipoEleTab. Variable auxiliar.
3	Si (ele.Clave() = ni) ∧ (ele1.Clave() = nj) entonces i = ele.Resto() j = ele1.Resto() g (i, j) = 1 fsi	-i, j: Entero: variable con la posicion de los nodos en la matriz-

Ingresa un nuevo arco en el digrafo. O(1+N/B)

Marzo, 2005	eliNodo(TipoEle: n)	
{pre: g (i, j) ≥ 0 ∀ i,j}		{pos: g (i, j) ≥ 0 ∀ i,j}
1	ele = t.conTabla(n)	-t,g: definido en DigrafoMat.
2	Si (ele.Clave() = n) t.elimTabla(n) inicio = ele.Resto() dir(pos) = {TipoNoDef} [g (inicio, j) = 0] j = 0, pos-1 [g (j, inicio) = 0] j = 0, pos-1 fsi	-ele:TipoEleTab. Variable auxiliar. -j: Entero: variable con la posicion de los nodos en la matriz

Elimina un nodo del digrafo. O(1+N/B)

Implementación del DigrafoMat

Marzo, 2005	eliArco(TipoEle: ni, TipoEle: nj)	
{pre: g (i, j) ≥ 0 $\forall$ i,j}		{pos: g (i, j) ≥ 0 $\forall$ i,j}
1	ele = t.conTabla(ni)	-t, g: definido en DigrafoMat.
2	ele1 = t.conTabla(nj)	-ele, ele1:TiopEleTab. Variable auxiliar.
3	Si (ele.Clave() = ni) $\wedge$ (ele1.Clave() = nj) entonces i = ele.Resto() j = ele.Resto() g (i, j) = 0 fsi	-i, j: Entero: variable con la posicion de los nodos en la matriz-

Elimina un arco del digrafo. O(N).

Marzo, 2005	conNodo(TipoEle: n): Lógico	
{pre: g (i, j) ≥ 0 $\forall$ i,j}		{pos: g (i, j) ≥ 0 $\forall$ i,j}
1	ele = t.conTabla(n)	-t: definido en DigrafoMat.
2	regrese (ele.Clave() = n)	-ele:TipoEleTab. Variable auxiliar.

Consultar un nodo del digrafo. O(1+N/B)

Implementación del DigrafoMat

Marzo, 2005	conArco(TipoEle: ni, TipoEle nj):Lógico	
{pre: g (i, j) ≥ 0 ∀ i,j}		{pos: g (i, j) ≥ 0 ∀ i,j}
1	ele, ele1, salir = t.conTabla(ni), t.conTabla(nj), Falso	-t,g: definido en DigrafoMat. -ele,ele1:TipoEleTab. Variable auxiliar. -i,j: Entero: variable con la posicion de los nodos en la matriz- -salir: Lógico. Regresa el resultado de la consulta (1 = verdadero)
2	Si (ele.Clave() = ni) ∧ (ele1.Clave() = nj) entonces i = ele.Resto() j = ele.Resto() Si (g (i, j) >0) entonces salir = Verdadero fsi fsi	
3	regresa salir	

Consulta un arco en el digrafo. O(1+N/B)

Marzo, 2005	nodoAdyacente(TipoEle: n):Arreglo[100] de TipoEle	
{pre: g (i, j) ≥ 0 ∀ i,j}		{pos: g (i, j) ≥ 0 ∀ i,j}
1	ele = t.conTabla(n)	-t,g: definido en DigrafoMat. -ele:TipoEleTab. Variable auxiliar. -i,j,k: Entero: variable con la posicion de los nodos en la matriz- -adya: Arreglo[100] de TipoEle. Almacena los nodos que son adyacentes a n.
2	Si (ele.Clave() = n) entonces i = ele.Resto() [Si (g (i, j) >0) entonces adya(k) = dir(k) k = k+ 1 fsi] j = 0, pos-1 fsi	
3	regresa adya	

Busca todos los nodos adyacentes a n. O(N).

Implementación del DigrafoMat

Marzo, 2005	incidenciaEnt(TipoEle: n):Entero	
{pre: g (i, j) ≥ 0 ∀ i,j}		{pos: g (i, j) ≥ 0 ∀ i,j}
1	ele = t.conTabla(n)	-t, g: Definido en DigrafoMat.
2	Si (ele.Clave() = n) entonces	-ele: TipoEleTab. Variable auxiliar.
	j = ele.Resto()	-i, j, k: Entero: variable con la posición de
	[Si (g (i, j) >0) entonces	los nodos en la matriz-
	k = k+1	
	fsi] i = 0, pos-1	
	fsi	
3	regresa k	

Calcula el grado de incidencia de entrada del nodo n. O(N).

Marzo, 2005	incidenciaSal(TipoEle: n):Entero	
{pre: g (i, j) ≥ 0 ∀ i,j}		{pos: g (i, j) ≥ 0 ∀ i,j}
1	ele = t.conTabla(n)	-t, g: Definido en DigrafoMat.
2	Si (ele.Clave() = n) entonces	-ele:TipoEleTab. Variable auxiliar.
	j = ele.Resto()	-i, j, k: Entero: variable con la posicion de los
	[Si (g (i, j) >0) entonces	nodos en la matriz-
	k = k+1	
	fsi] j = 0, pos-1	
	fsi	
3	regresa k	

Calcula el grado de incidencia de salida del nodo n. O(N).

Implementación del DigrafoMat

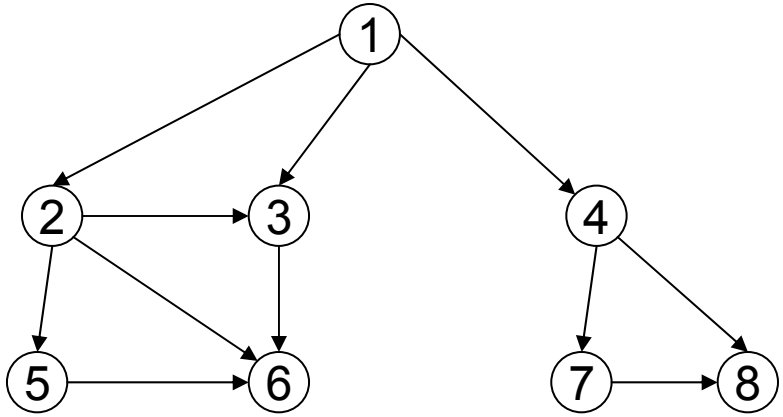
Marzo, 2005		
	numNodos():Entero	
{pre: g (i, j) ≥ 0 ∀ i,j}		{pos: g (i, j) ≥ 0 ∀ i,j}
1	regrese pos	-pos: Definido en DigrafoMat.

Regresa el número de nodos almacenados en el digrafo O(1)

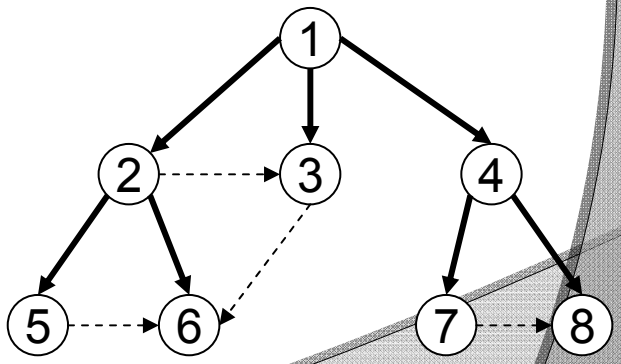
Marzo, 2005		
	Especificación Cola[TipoEle]	
1	Especificación sintáctica: creaCola() Cola, enCola(Cola,TipoEle) Cola, desenCola(Cola) Cola, primero(Cola) TipoEle, vaciaCola(Cola) Lógico, destruyeCola(Cola) .	-creaCola(): Crea una cola vacía. -enCola(): Ingresa un nuevo elemento a la cola por el fin de la misma. -desenCola(): Elimina el elemento que está actualmente en el inicio de la cola. Si la cola está vacía no hace ninguna eliminación. -destruyeCola(): Destruye la cola. -vaciaCola(): Regresa Verdadero si la cola está vacía. -primero(): Devuelve el elemento que se encuentra actualmente en el inicio de la cola. Si la cola está vacía devuelve un valor especial que lo indica.
2	Declaraciones: TipoEle: e, {TipoEleNoDef}	
3	Especificación semántica: vaciaCola(creaCola())=Verdadero vaciaCola(enCola(creaCola(),e))=Falso primero(creaCola())=TipoEleNoDef desenCola(creaCola())=creaCola()	

Búsqueda en amplitud del DigrafoMat

- Usos:
  - Para recorridos parciales de grafos infinitos o tan grandes que son inmanejables
  - Encontrar los caminos más cortos desde un nodo origen hasta los otros nodos

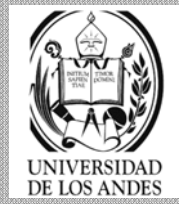


Digrafo DG



Recorrido en amplitud:
arcos con líneas continuas

Búsqueda en amplitud del DigrafoMat



Marzo, 2005

recorridoEnAmp( )

{pre: g (i, j) ≥ 0 ∀ i,j} {pos: marca (i) = Verdadero ∀ i}

1 2	[marca(i) = 'B'] i = 1, pos [Si (¬ marca(i)) entonces busAmp(i, marca) fsi] i = 1, pos	-pos: Definido en DigrafoMat. -marca: Arreglo[100]De Caracter. Marca que indica si el nodo no ha sido visitado ('B'), visitado ('G') o terminado ('N'). -i: Entero: Variable con la posición de los nodos en la matriz

Marzo, 2005

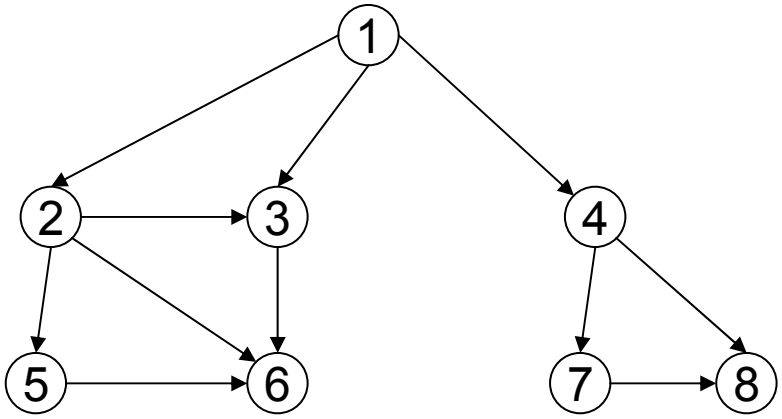
busAmp(Entero+: na, Arreglo[100]De Lógico: mar)

{pre: 1 ≤ na ≤ pos} {pos: marca (i) = Verdadero ∀ i}

1 2 3	mar(na) = 'G' co.enCola(na) (¬ co.vaciaCola( )) [na = co.primero( ) co.desenCola( ) nadya = nodoAdyacente(na) [Si (mar(nadya(i))='B') entonces mar(nadya(i)) = 'G' co.enCola(nadya(i)) fsi] i = 1, pos mar(na)='N']	-pos, nodoAdyacente( ): Definidos en DigrafoMat. -mar: Arreglo[100]De Lógico. Marca que indica si el nodo no ha sido visitado ('B'), visitado ('G') o terminado ('N'), Blanco, Gris o Negro -i: Entero+: Variable con la posición de los nodos en la matriz -co: Cola[Entero+]. Cola que mantiene los nodos no tratados. -nadya: Arreglo[100]De Entero+. Vector auxiliar que contiene los nodos adyacentes -enCola( ), primero( ), desenCola( ), vaciaCola( ). Operaciones de la clase Cola.

$T(n) = O(N + A) = \Theta(máx(N, A))$

Recorrido en amplitud del DigrafoMat



g

0	1	1	1	0	0	0	0
0	0	1	0	1	1	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	1
0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0

Condiciones iniciales

nadya

2	3	4

na = 1

marca

F	F	F	F	F	F	F	F

Estado intermedio luego de tener los nodos adyacentes al nodo 1

marca

V	V	V	V	V	V	V	V

Condiciones finales

Implementación del DigrafoLis

Marzo, 2005	vacíoDigrafo(): Lógico	
{pre: g.n ≥ 0}		{pos: g.n ≥ 0 }
1	regrese g.vacíaLista()	-g: Definido en DigrafoLis.

Verifica sí el digrafo esta vacío O(1)

Marzo, 2005	nuevoNodo(TipoEle: n)	
{pre: g.n ≥ 0}		{pos: g.n ≥ 0 }
1	Si (¬conNodo(n)) entonces ele.Infor(n) g.insLista(ele) fsi	-g: Definido en DigrafoLis. -ele: Nodo1[TipoEle]. Variable auxiliar.

Ingresa un nuevo nodo al digrafo. O(N).

Marzo, 2005	nuevoArco(TipoEle: ni, TipoEle: nj)	
{pre: g.n ≥ 0}		{pos: g.n ≥ 0 }
1	Si (conNodo(nj) ∧ conNodo(ni)) entonces g.conLista().ListaAdy insLista(nj) fsi	-g: Definido en DigrafoLis.

Ingresa un nuevo arco en el digrafo. O(N)

Implementación del DigrafoLis

UNIVERSIDAD DE LOS ANDES

Marzo, 2005

eliNodo(TipoEle: n)

{pre: g.n ≥ 0 } {pos: g.n ≥ 0 }

1	Si (conNodo(n)) entonces g.conLista().ListaAdya().cursorAlInicio() (¬ g.conLista().ListaAdya().vaciaLista()) [g.conLista().ListaAdya().eliLista()] g.eliLista() g.cursorAlInicio() [g.conLista().ListaAdya().cursorAlInicio() i = 1 (g.conLista().ListaAdya().conLista().Info() ≠ n ∧ i ≤ g.conLista().ListaAdya().numEle()) [i = i+1 g.conLista().ListaAdya().cursorAlProximo()] Si (i ≤ g.conLista().ListaAdya().numEle()) entonces g.conLista().ListaAdya().eliLista() fsi g.cursorAlProximo()] j = 1, g.numEle() fsi	-g: Definido en Digrafo-Lis. -i, j: Entero: variable auxiliar.

Elimina un nodo del digrafo. O(|N| + |A|)

Implementación del DigrafoLis

Marzo, 2005	eliArco(TipoEle: ni, TipoEle: nj)	
{pre: g.n ≥ 0 }		{pos: g.n ≥ 0 }

1	Si (conNodo(nj) ∧ conNodo(ni)) entonces
	  i = 1
	  (g.conLista().ListaAdya().conLista().Info() ≠ nj ∧ i ≤ g.conLista().ListaAdya().numEle())
	  [i = i + 1
	  g.conLista().ListaAdya().cursorAlProximo()]
	  Si (i ≤ g.conLista().ListaAdya().numEle()) entonces
	  g.conLista().ListaAdya().eliLista()
	  fsi
	fsi
		-g: Definido en DigrafoLis.
		-i: Entero. Variable con auxiliar.

Elimina un arco del digrafo. O(N).

Marzo, 2005	conNodo(TipoEle: n): Lógico	
{pre: g.n ≥ 0}		{pos: g.n ≥ 0}

1
2
3
 	
4
	g.cursorAlInicio()
	i = 1
	(i ≤ g.numEle()) [Si (g.conLista().Infor() = n) entonces
	  regrese Verdadero
	sino
	  g.cursorAlProximo()
	  i = i + 1
	fsi]
	regrese Falso
		-g: Definido en DigrafoLis.
		-i: Entero: Variable con auxiliar para recorrer los nodos del digrafo.

Consulta un nodo del digrafo. O(N).

Implementación del DigrafoLis

Marzo, 2005	conArco(TipoEle: ni, TipoEle: nj): Lógico	
{pre: g.n ≥ 0 }		{pos: g.n ≥ 0}

1	Si (conNodo(ni)) entonces
	  i = 1
	  (i ≤ g.conLista().ListaAdya().numEle())
	  [Si (g.conLista().ListaAdya().conLista().Info() = nj) entonces
	  regrese Verdadero
	  sino
	  i = i + 1
	  g.conLista().ListaAdya().cursorAlProximo()
	  fsi]
	fsi
		-g: Definido en DigrafoLis.
		-i: Entero: variable auxiliar

2	regrese Falso	

Consulta un arco en el digrafo. O(N)

Marzo, 2005	nodoAdyacente(TipoEle: n): Lista[TipoEle]	
{pre: g.n ≥ 0}		{pos: g.n ≥ 0}
1	Si (conNodo(n)) entonces adya = g.conLista().ListaAdya() fsi	-g: definido en Digrafo. -adya: Lista[TipoEle]. Almacena los nodos que son adyacentes a n.
2	regresa adya	

Busca todos los nodos adyacentes a n. O(N).

Implementación del DigrafoLis

Marzo, 2005	incidenciaEnt(TipoEle: n): Entero	
{pre: g.n ≥ 0}		{pos: g.n ≥ 0}
1	k = 0	-g: Definido en DigrafoLis. -salir: Lógico. Variable auxiliar para controlar la búsqueda. -i: Entero: Variable auxiliar. -k: Entero: Grado de incidencia de entrada del nodo.
2	g.cursorAlInicio() [g.conLista().ListaAdya().cursorAlInicio() i = 1 salir = Falso (i ≤ g.conLista().ListaAdya().numEle() ∧ ¬salir) [Si (g.conLista().ListaAdya().conLista().Info() = n) entonces salir = Verdadero k = k + 1 sino i = i + 1 g.conLista().ListaAdya().cursorAlProximo() fsi] g.cursorAlProximo()] j = 1, g.numEle()	
3	regresa k	

Calcula el grado de incidencia de entrada del nodo n. O(|N|+|A|).

Implementación del DigrafoLis

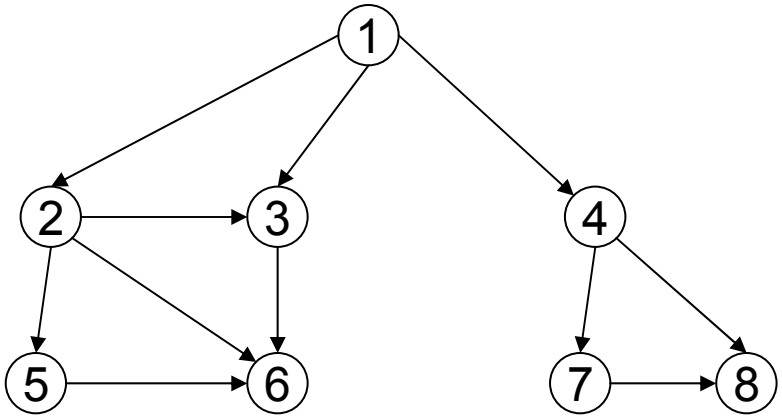
Marzo, 2005	incidenciaSal(TipoEle: n): Entero	
{pre: g.n ≥ 0 }		{pos: g.n ≥ 0}
1 2	Si (conNodo(n)) entonces regrese g.conLista().ListaAdya().numEle() fsi regresa -1	-g: Definido en DigrafoLis.

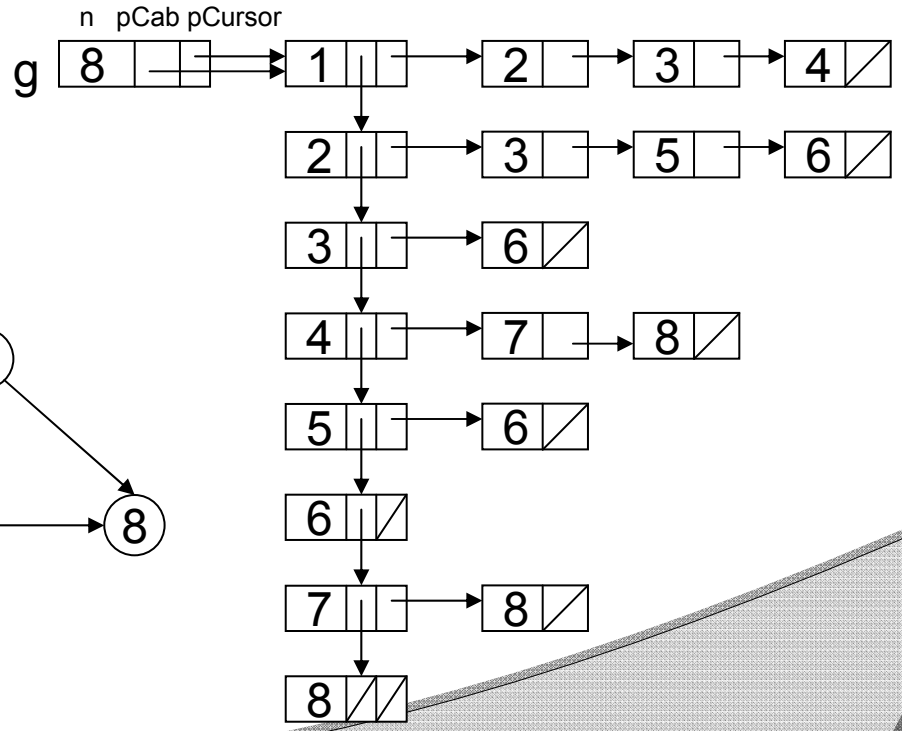
Calcula el grado de incidencia de salida del nodo n. O(N).

Para la búsqueda en amplitud utilizando listas de adyacencia, se utiliza también la clase **Cola**, con el propósito de ir almacenando los nodos no tratados que son adyacentes al nodo actual (na).

Búsqueda en amplitud del DigrafoLis

- Se recorre la lista de adyacencia de cada nodo





Búsqueda en amplitud del DigrafoLis

Marzo, 2005	recorridoEnAmp( )	
{pre: g.n $\geq$ 0 }		{pos: marca (i) = Verdadero $\forall$ i $\leq$ |N|}
1 2	[marca(i) = falso] i = 1, g.numEle( ) [Si ($\neg$ marca(i)) entonces busAmp(i, marca) fsi] i = 1, g.numEle( )	- g: Definido en DigrafoLis -marca: Arreglo[100]De Lógico. Marca que indica si el nodo fue visitado (Verdadero) o no (Falso). -i: Entero: Variable auxiliar para los nodos

$$T(n) = O(N + A) = \Theta(máx(N, A))$$

Búsqueda en amplitud del DigrafoLis

Marzo, 2005

busAmp(Entero+: na, Arreglo[100]De Lógico: mar)

{pre: g.n ≥ 0 } {pos: marca (i) = Verdadero ∀ i}

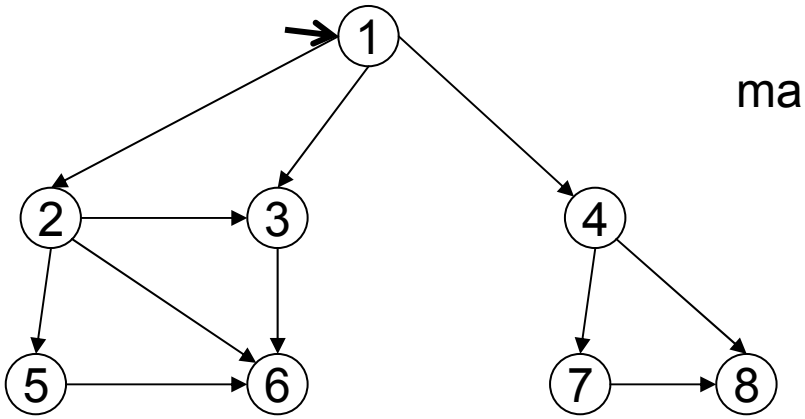
```
1  mar( na ) = Verdadero
2  co.enCola( na )
3  ( ¬ co.vaciaCola( ) ) [ na = co.primero( )
                           co.desenCola( )
                           nadya = nodoAdyacente( na )
                           nadya.cursorAlInicio( )
                           [ Si (¬mar(nadya.conLista().Info())) entonces
                                 mar(nadya.conLista().Info()) = Verdadero
                             fsi
                             co.enCola( nadya.conLista().Info())
                             nadya.cursorAlProximo( )
                           ] i = 1, nadya.numEle( )
                         ]
```

-g, nodoAdyacente( ): Definidos en DigrafoLis
-mar: Arreglo[100]De Lógico. Marca que indica si el nodo fue visitado (Verdadero) o no (Falso).
-i: Entero+: Variable con la posición de los nodos en la matriz
-co: Cola[Entero+]. Cola que mantiene los nodos no tratados.
-nadya: Lista[Entero+]. Lista auxiliar que contiene los nodos adyacentes
-enCola( ), primero( ), desenCola( ), vaciaCola( ). Operaciones de la clase Cola.
-cursorAlInicio( ), numEle( ), conLista( ), cursorAlProximo( ). Definidos en Lista

Ejemplo

recorridoEnAmp

i=1



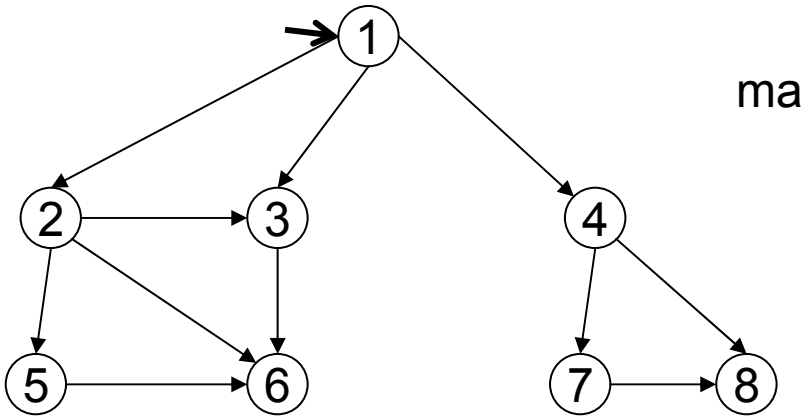
mar

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

Ejemplo

busAmp

na=1



mar

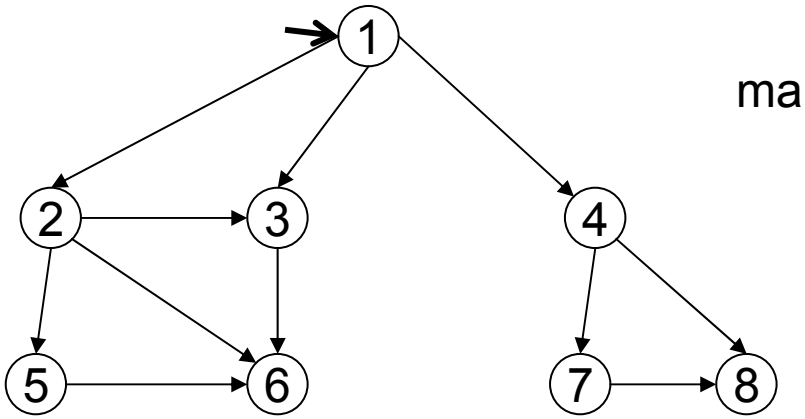
V	F	F	F	F	F	F	F
1	2	3	4	5	6	7	8

Co

1

Ejemplo

busAmp



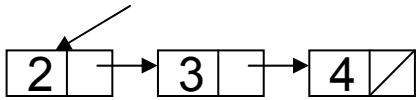
na=1

mar

V	F	F	F	F	F	F	F
1	2	3	4	5	6	7	8

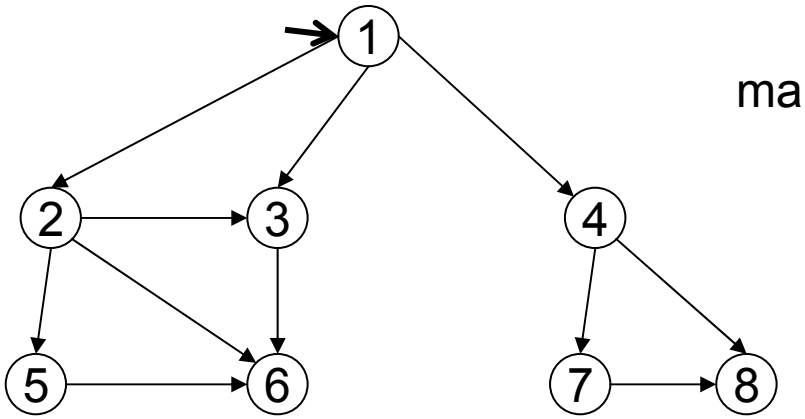
Co

nadya



Ejemplo

busAmp



na=1

mar

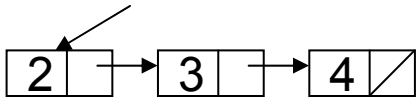
V	V	F	F	F	F	F	F
1	2	3	4	5	6	7	8

Co

2

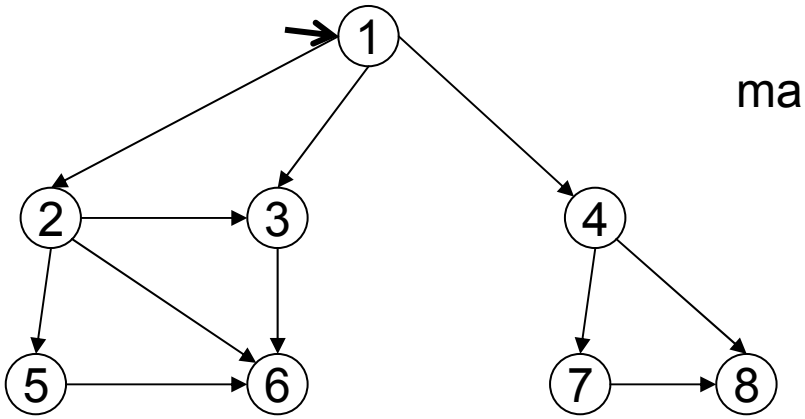
nadya

i=1



Ejemplo

busAmp



na=1

mar

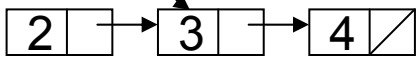
V	V	V	F	F	F	F	F
1	2	3	4	5	6	7	8

Co

2	3

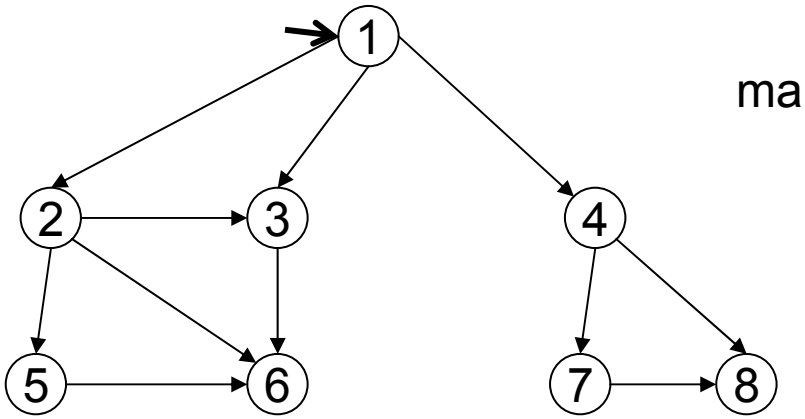
nadya

i=2



Ejemplo

busAmp



na=1

mar

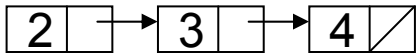
V	V	V	V	F	F	F	F
1	2	3	4	5	6	7	8

Co

2	3	4

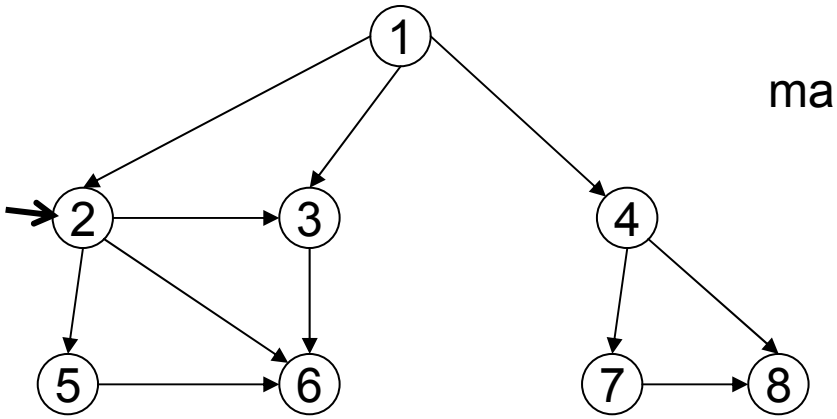
nadya

i=3



Ejemplo

busAmp



na=2

mar

V	V	V	V	F	F	F	F
1	2	3	4	5	6	7	8

Co

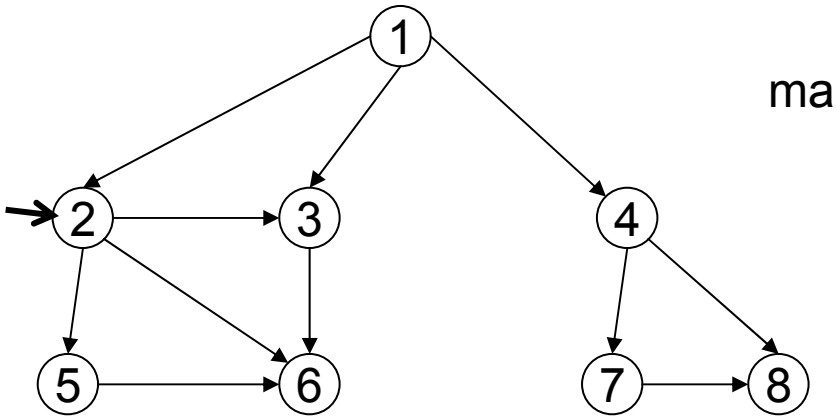
3	4

nadya

i=4

Ejemplo

busAmp



na=2

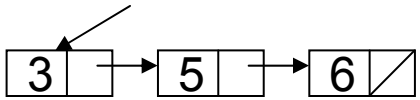
mar

V	V	V	V	F	F	F	F
1	2	3	4	5	6	7	8

Co

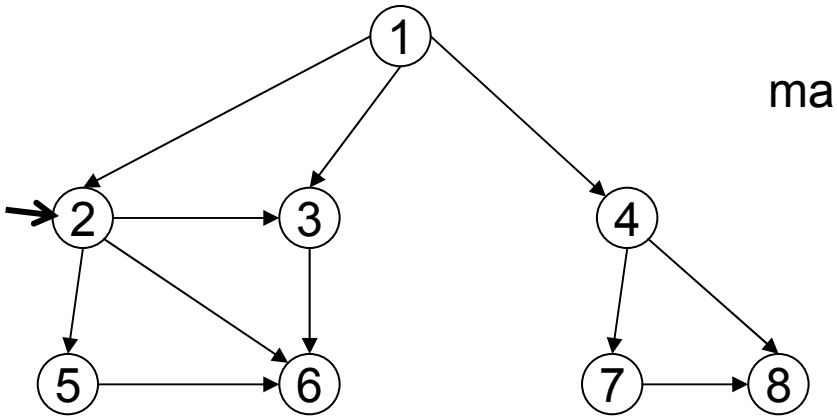
3	4

nadya



Ejemplo

busAmp



na=2

mar

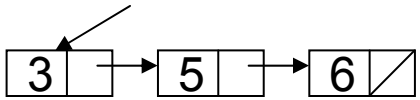
V	V	V	V	F	F	F	F
1	2	3	4	5	6	7	8

Co

3	4	3

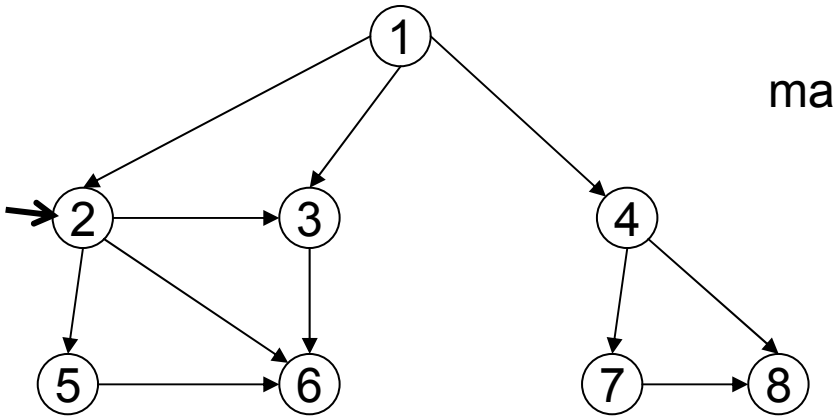
nadya

i=1



Ejemplo

busAmp



na=2

mar

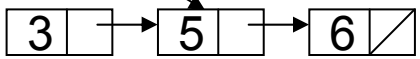
V	V	V	V	V	F	F	F
1	2	3	4	5	6	7	8

Co

3	4	3	5

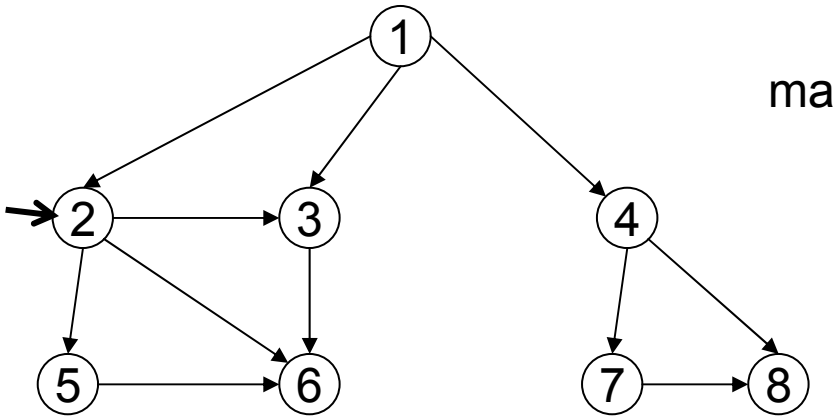
nadya

i=2



Ejemplo

busAmp



na=2

mar

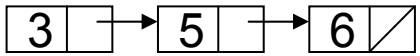
V	V	V	V	V	V	F	F
1	2	3	4	5	6	7	8

Co

3	4	3	5	6

nadya

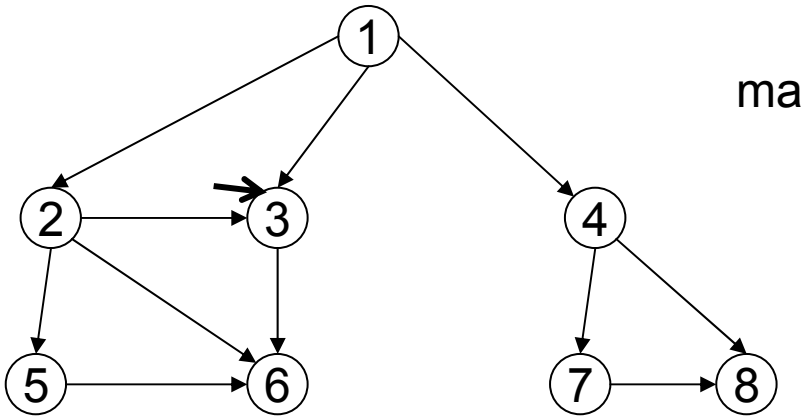
i=3



Ejemplo

busAmp

na=3



mar

V	V	V	V	V	V	F	F
1	2	3	4	5	6	7	8

Co

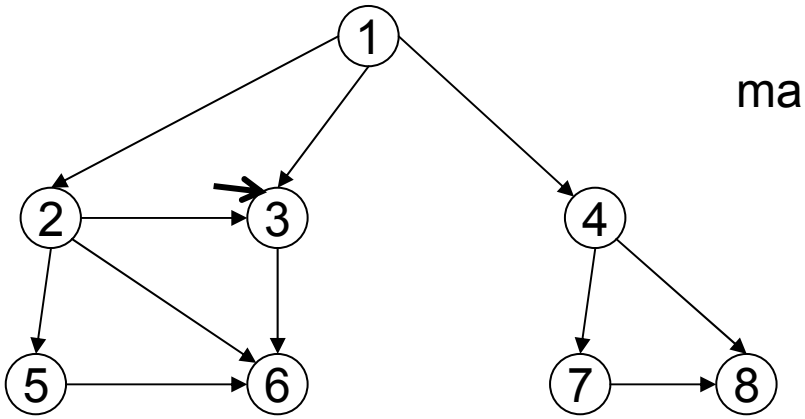
4	3	5	6

nadya

i=4

Ejemplo

busAmp



na=3

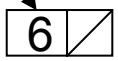
mar

V	V	V	V	V	V	F	F
1	2	3	4	5	6	7	8

Co

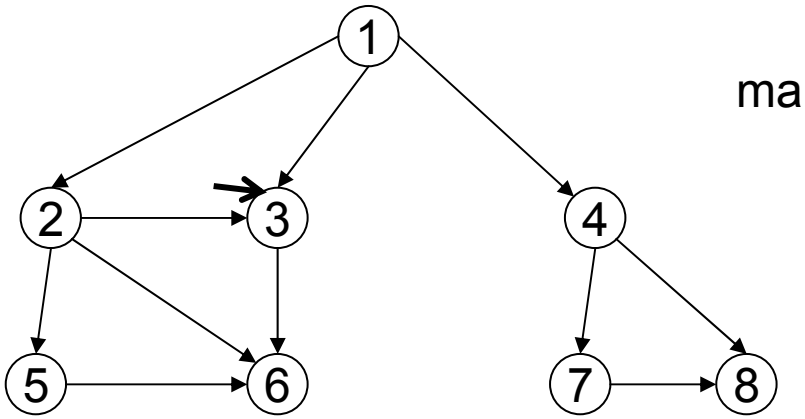
4	3	5	6

nadya



Ejemplo

busAmp



na=3

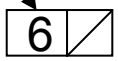
mar

V	V	V	V	V	V	F	F
1	2	3	4	5	6	7	8

Co

4	3	5	6	6

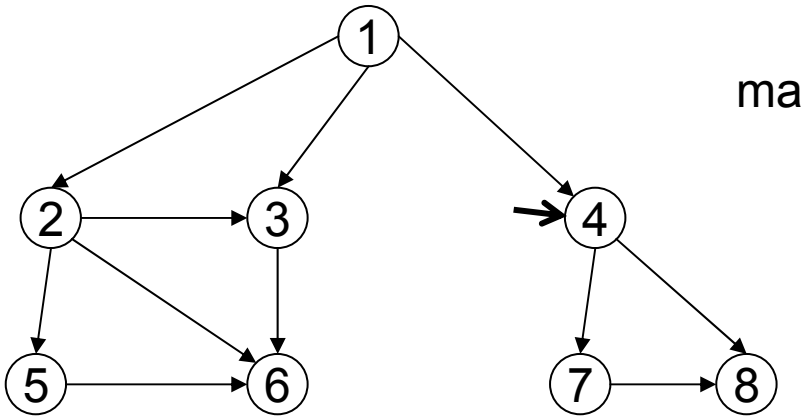
nadya



i=1

Ejemplo

busAmp



na=4

mar

V	V	V	V	V	V	F	F
1	2	3	4	5	6	7	8

Co

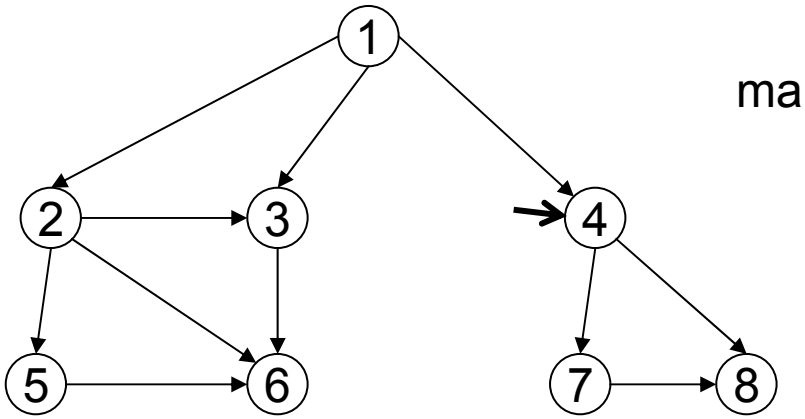
3	5	6	6

nadya

i=2

Ejemplo

busAmp



na=4

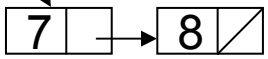
mar

V	V	V	V	V	V	F	F
1	2	3	4	5	6	7	8

Co

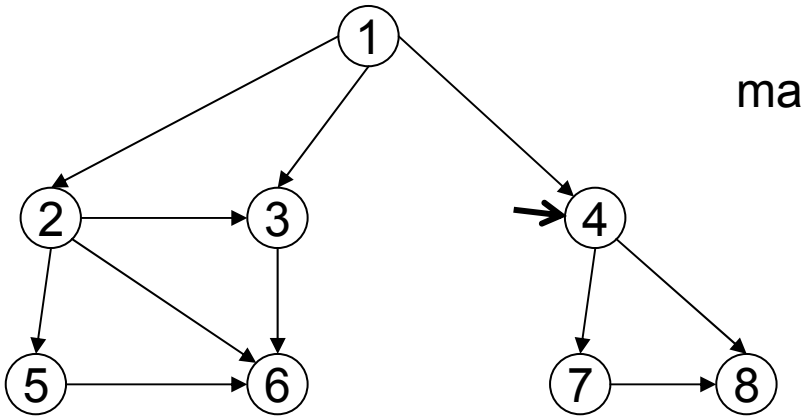
3	5	6	6

nadya



Ejemplo

busAmp



na=4

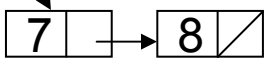
mar

V	V	V	V	V	V	V	F
1	2	3	4	5	6	7	8

Co

3	5	6	6	7

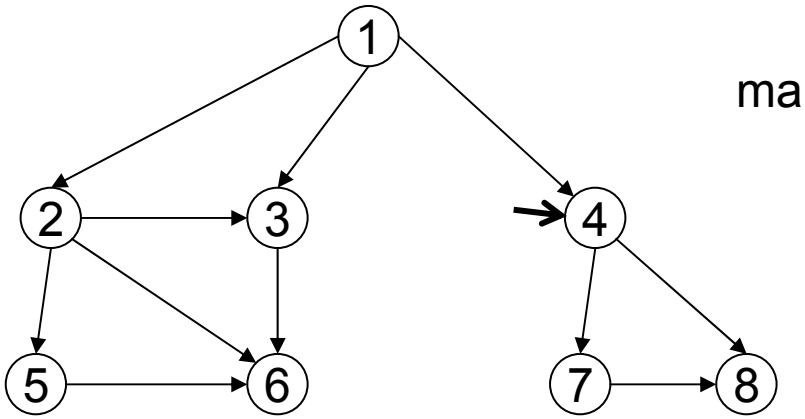
nadya



i=1

Ejemplo

busAmp



na=4

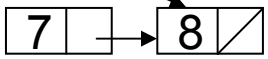
mar

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

Co

3	5	6	6	7	8

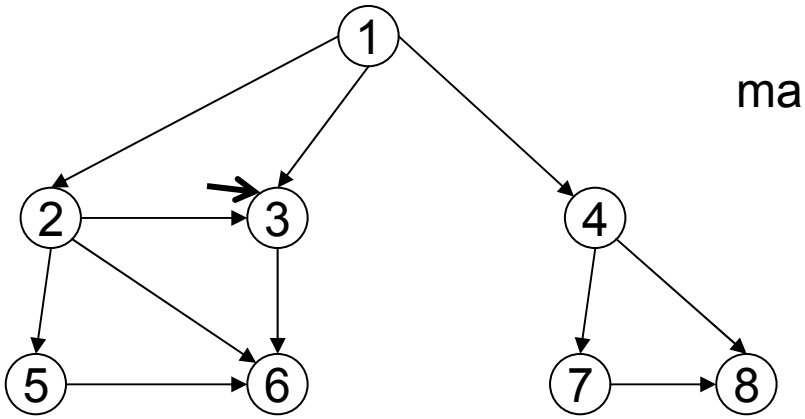
nadya



i=2

Ejemplo

busAmp



na=3

mar

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

Co

5	6	6	7	8

nadya

i=3

Recorridos de árboles

- Árboles binarios:
  - Preorden: raíz-izquierdo-derecho (RID)
  - Enorden: izquierdo-raíz-derecho (IRD)
  - Posorden: izquierdo-derecho-raíz (IDR)
- Pre y posorden se generalizan para los árboles
- Son recursivas por naturaleza
- Exploración de izquierda a derecha o viceversa
- Lema: cada uno de los recorridos, el tiempo T(n) que se necesita para explorar un árbol binario que contiene n nodos se encuentra en $\Theta(n)$

Recorridos de árboles

- Demostración:

  Suponga que visitar un nodo está en O(1), T(0) está acotado superiormente por alguna constante **c**, donde c ≥ T(0)

  Suponga que hay que explorar un árbol de **n > 0** nodos, uno de ellos es la raíz, **g** de ellos están en el subárbol izquierdo y **n-g-1** en el subárbol derecho, entonces

$$T(n) \leq \max_{0 \leq g \leq n-1} (T(g) + T(n-g-1) + c, \ n > 0$$

  Sin importar el orden de exploración. Se demuestra por inducción que $T(n) \leq an+b$, donde a y b son constantes.

  Si $b \geq c$, la hipótesis es verdadera para $n=0$ ya que $c \geq T(0)$

  Paso de inducción: $n > 0$ y la hipótesis cierta para todo $m \in [0, n-1]$

Recorridos de árboles

$$T(n) \leq \max_{0 \leq g \leq m-1} (T(g) + T(n-g-1) + c)$$

$$T(n) \leq \max_{0 \leq g \leq n-1} (ag + b + a(n-g-1) + b + c)$$

$$T(n) \leq an + 3b - a$$

Si a ≥ 2b se tiene T(n) ≤ an+b,

la hipótesis es cierta también para m = n, lo que demuestra que

T(n) ≤ an+b $\forall$n ≥ 0, por lo que T(n) está en O(n)

Además, T(n) está en Ω(n) ya que se visitan todos los nodos

y por ello T(n) está en Θ(n).

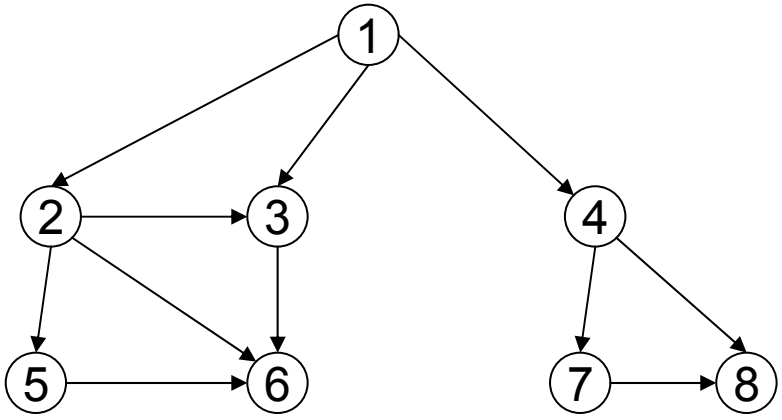
Búsqueda en profundidad

- Sea G={N, A, f} un grafo formado por todos los nodos que se desean visitar.
- Suponga una forma de marcar los nodos visitados
  - Se selecciona un nodo para comenzar v $\in$ N (nodo actual) y se marca como visitado
  - Si hay algún nodo adyacente a v que no ha sido visitado aún, se toma dicho nodo como nodo actual y se invoca recursivamente el proceso de recorrido en profundidad
  - Cuando están marcados como visitados todos los nodos adyacentes a v, se termina el recorrido para v.

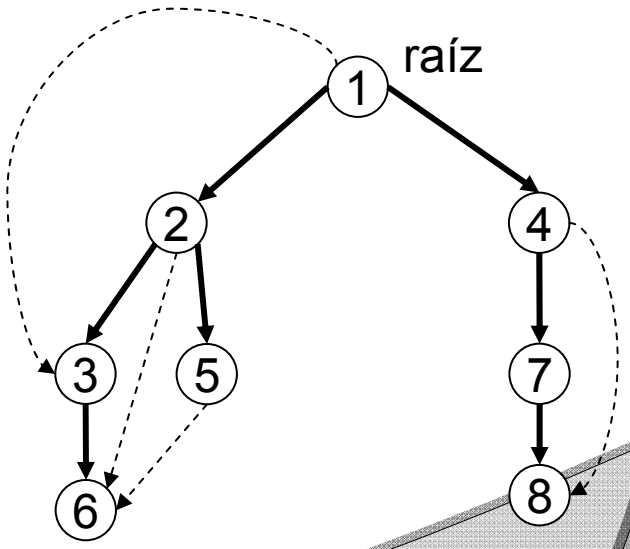
Búsqueda en profundidad

- Si queda algún nodo en G que no ha sido visitado, se repite el proceso tomando un nodo cualquiera como v

- Al recorrido se le asocia un árbol de recubrimiento



Digrafo DG



Recorrido en profundidad: arcos con líneas continuas

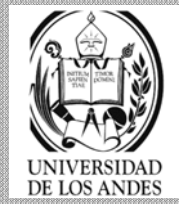
Búsqueda en profundidad del DigrafoMat

Marzo, 2005	recorridoEnProf( )	
{pre: g (i, j) ≥ 0 ∀ i,j}		{pos: marca (i) = Verdadero ∀ i}
1 2	[marca(i) = falso] i = 1, pos [Si (¬ marca(i)) entonces busPro(i, marca) fsi] i = 1, pos	-pos: Definido en DigrafoMat. -marca: Arreglo[100]De Lógico. Marca que indica si el nodo fue visitado (Verdadero) o no (Falso). -i: Entero: Variable con la posición de los nodos en la matriz

Marzo, 2005	busPro(Entero+: na, Arreglo[100]De Lógico: mar)	
{pre: 1 ≤ na ≤ pos}		{pos: marca (i) = Verdadero ∀ i}
1 2 3	mar(na) = Verdadero nadya = nodoAdyacente(na) [Si (¬ mar(nadya(i))) entonces busPro(nadya(i), mar) fsi] i = 1, pos	-pos, nodoAdyacente( ): Definidos en DigrafoMat. -mar: Arreglo[100]De Lógico. Marca que indica si el nodo fue visitado (Verdadero) o no (Falso). -i: Entero+: Variable con la posición de los nodos en la matriz -nadya: Arreglo[100]De Entero+. Vector auxiliar que contiene los nodos adyacentes

$T(n) = \Theta(\text{máx}(N,A))$

Búsqueda en profundidad del DigrafoLis



Marzo, 2005

recorridoEnProf( )

{pre: g.n $\geq$ 0 } {pos: marca (i) = Verdadero $\forall$ i}

1 2	[marca(i) = falso] i = 1, g.numEle( ) [Si ($\neg$ marca(i)) entonces busPro(i, marca) fsi] i = 1, g.numEle( )	- g: Definido en DigrafoLis -marca: Arreglo[100]De Lógico. Marca que indica si el nodo fue visitado (Verdadero) o no (Falso). -i: Entero: Variable auxiliar para los nodos

Marzo, 2005

busPro(Entero+: na, Arreglo[100]De Lógico: mar)

{pre: g.n $\geq$ 0 $\wedge$ 1 $\leq$ na $\leq$ g.n} {pos: marca (i) = Verdadero $\forall$ i}

1 2 3 4	mar(na) = Verdadero nadya = nodoAdyacente(na) nadya.cursorAlInicio( ) [Si($\neg$mar(nadya.conLista().Info())) entonces busPro(nadya.conLista().Info(), mar) fsi nadya.cursorAlProximo( )] i = 1, nadya.numEle( )	-g, nodoAdyacente( ): Definidos en DigrafoLis -mar: Arreglo[100]De Lógico. Marca que indica si el nodo fue visitado (Verdadero) o no (Falso). -i: Entero+: Variable auxiliar -nadya: Lista[Entero+]. Lista auxiliar que contiene los nodos adyacentes -cursorAlInicio( ), numEle( ), conLista( ), cursorAlProximo( ). Definidos en Lista

$T(n) = \Theta(máx(N,A))$

Búsqueda en profundidad

- Si el grafo es conexo, se tendrá un único árbol de recorrido en profundidad
- Si el grafo no es conexo, se tendrá un bosque de árboles, uno por cada componente conexo
- Numera los nodos del árbol en preorden
  - Se coloca en el algoritmo recorridoEnProf
    np = np+1 como paso 1
    prenum(i) = np como paso 2
  - en el algoritmo busPro
    np = 0 como paso 2