Grafos:
búsqueda y ordenamiento topológico

Jose Aguilar

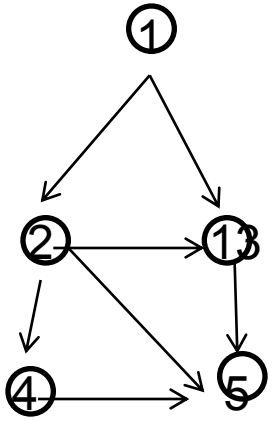
Introducción

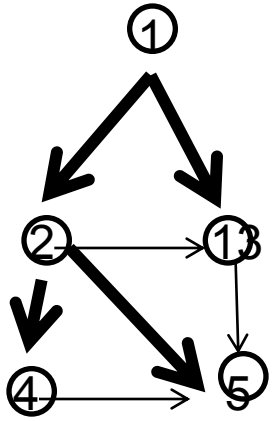
- *Recorrido*: procedimiento sistemático de exploración de un grafo mediante la visita a todos sus vértices y aristas.

- Un recorrido es eficiente si visita todos los vértices y aristas en un tiempo proporcional a su número, esto es, en tiempo lineal.

Interés búsqueda

- Recorridos parciales de grafos infinitos o tan grandes que son inmanejables
- Encontrar un nodo desde un nodo origen





Búsqueda en grafos

- Tipos de Búsqueda
  - **búsqueda ciega (Búsquedas sin contar con información)**: no utiliza información sobre el problema. Normalmente, se realiza una búsqueda exhautiva.
  - **Búsqueda heuristica (Búsqueda respaldada con información)**: usan información sobre el problema como costos, etc. Se posee información muy valiosa para orientar la búsqueda

Estrategias de Búsqueda

Búsqueda No Informada (Ciega)

1. Búsqueda por amplitud
2. Búsqueda de costo uniforme
3. Búsqueda por profundidad
4. Búsqueda limitada por profundidad
5. Búsqueda por profundización iterativa
6. Búsqueda bidireccional

Búsqueda Informada (Heurística)

1. Búsqueda avara
2. Búsqueda A*
3. Búsqueda A*PI
4. Búsqueda A*SRM

Muchas otras heurísticas!!!

Criterios de rendimiento

Las estrategias de búsqueda se evalúan según los siguientes criterios:

- **Completitud:** si garantiza o no encontrar la solución si es que existe.

¿La estrategia garantiza encontrar una solución, si ésta existe?

- **Complejidad temporal:** cantidad de tiempo necesario para encontrar la solución.

¿Cuánto tiempo se necesitará para encontrar una solución?

- **Complejidad espacial:** cantidad de memoria necesaria para encontrar la solución.

¿Cuánta memoria se necesita para efectuar la búsqueda?

- **Optimalidad: si** se encontrará o no la mejor solución en caso de que existan varias.

¿Con esta estrategia se encontrará una solución de la más alta calidad, si hay varias soluciones?

Búsqueda ciega

- La búsqueda consiste en **escoger uno de los nodos posibles**.
- Si hay varios nodos que se pueden expandir, la elección de cual se expande primero se hace según una **estrategia de búsqueda**.
- El proceso de búsqueda se concibe como la construcción de un **recorrido del grafo**.
- Las técnicas **se diferencian** en el orden en que expanden los nodos.

Búsqueda no informada o ciega

Estrategia de búsqueda (a quien expandir)

- Por extensión o amplitud

  Origen, sus vecinos, etc. Completa pero no optima

- Uniforme

  Expande nodo más barato

- Por Profundidad

  Expande un nodo hasta que no se pueda más y regresa. Completa pero no optima

- Por Profundidad limitada. Ni completa ni optima

- Profundidad Iterativa

  Prueba diferentes profundidades. Completa y optima

- Bidireccional

Grafos

Búsqueda en amplitud

Algoritmo BÚSQUEDA ANCHURA (Breadth First Search)

- Se comienza en el vértice inicial (vértice con índice 1), se marca como **vértice activo**, y **se visitan** en orden creciente de índice **todos los vecinos** del vértice activo antes de pasar al siguiente.

- **Hasta que todos los vértices hayan sido visitados**, en cada paso se van visitando en orden creciente de índice todos los vecinos del vértice activo.

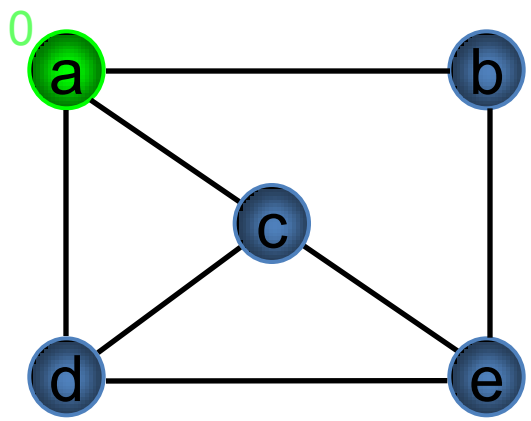
- Cuando se han visitado todos los vecinos del vértice activo, **se toma como nuevo vértice activo el primer vértice *X* visitado después del actual vértice activo** en el desarrollo del algoritmo.

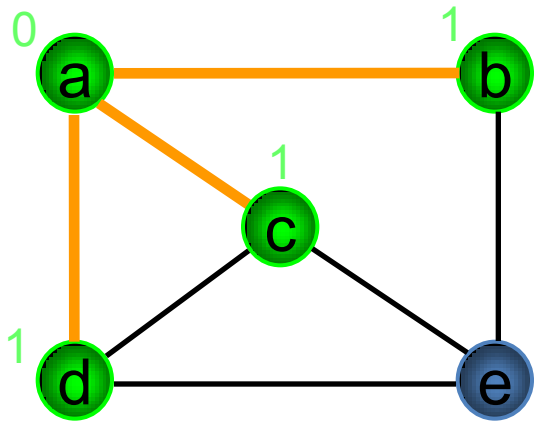
Algoritmo BÚSQUEDA ANCHURA (Breadth First Search)

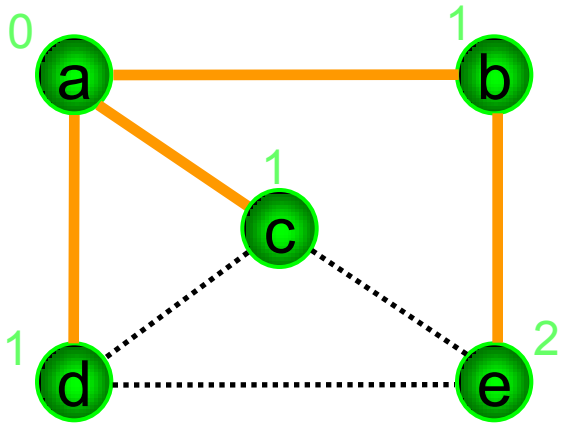
Sea G = (V, A) un grafo conexo,
V' = V un conjunto de vértices,
A' un vector de arcos inicialmente vacío y
P un vector auxiliar inicialmente vacío

1. Se introduce el vértice inicial en P y se elimina del conjunto V'.
2. Mientras V' no sea vacío.
    1. Se toma el primer elemento de P como vértice activo.
    2. Si el vértice activo tiene algún vértice adyacente que se encuentre en V'
        - Se toma el de menor índice.
        - Se inserta en P como último elemento.
        - Se elimina de V'.
        - Se inserta en A' el arco que le une con el vértice activo.
    3. Si el vértice activo no tiene adyacentes en V' se elimina de P.

Búsqueda primero en anchura







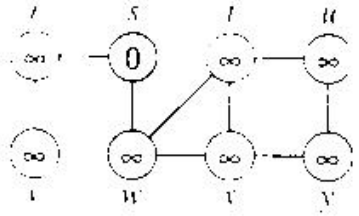
Algoritmo BPA

```
recorre_grafo_bpa()
{
  for cada vértice v
    marca[v]=SINVISITAR;
  for cada vértice v
     if (marca[v]==SINVISITAR)
            BPA(v);
}
```

```
BPA(v)
{
  marca[v] = VISITADO;
  InsertaCola(v, C)
  while not EsVacíaCola(C) {
     u = SuprimirCola(C);
     for cada nodo y adyacente a u {
            if (marca[y]==SINVISITAR) {
                    marca[y] = VISITADO;
                    InsertaCola(y, C);
            }
     }
  }
}
```

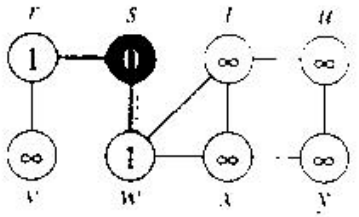
- Orden de complejidad del recorrido en anchura:
  - Con lista de adyacencia: O(n + e).
  - Con matriz de adyacencia: O(n^2).

(a)



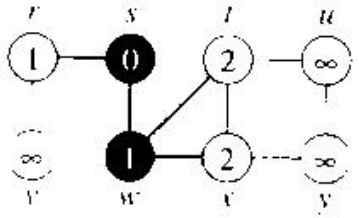
V' | s |
0

(b)



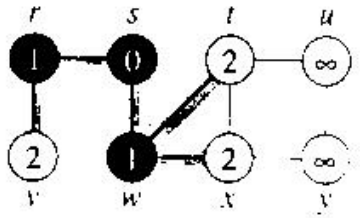
V' | w | r |
1 1

(c)



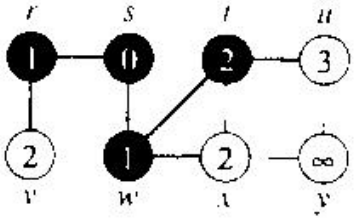
V' | r | t | x |
1 2 2

(d)



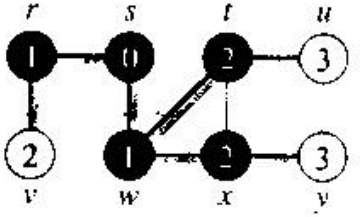
V' | t | x | y |
2 2 2

(e)



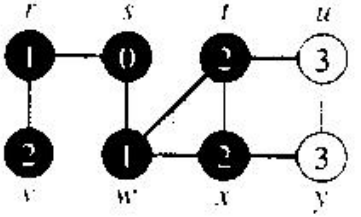
V' | x | v | u |
2 2 3

(f)



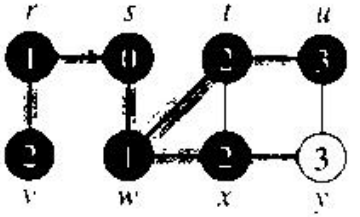
V' | v | u | y |
2 3 3

(g)



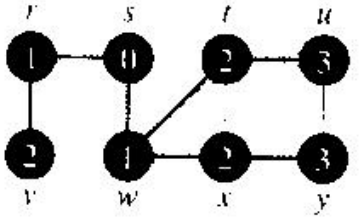
V' | u | y |
3 3

(h)



V' | y |
3

(i)

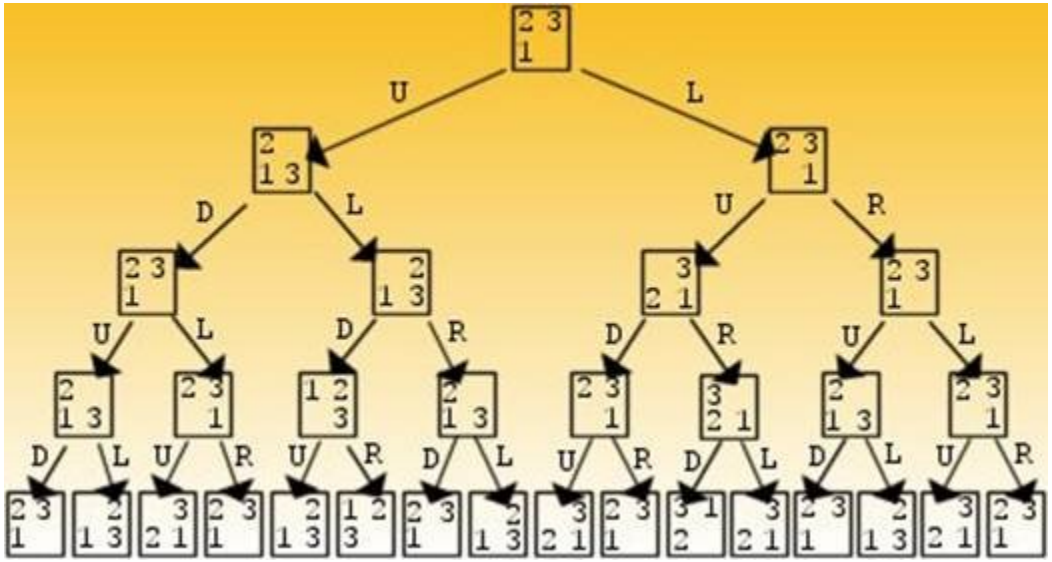


V' ∅

```
BFS(grafo G, nodo_fuente s)
{
  // recorremos todos los vértices del grafo inicializándolos a NO_VISITADO,
  // distancia INFINITA y padre de cada nodo NULL
  for u ∈ V[G] do
  {
      estado[u] = NO_VISITADO;
      distancia[u] = INFINITO;    /* distancia infinita si el nodo no es alcanzable */
      padre[u] = NULL;
  }
   estado[s] = VISITADO;
   distancia[s] = 0;
   Encolar(Q, s);
   while !vacia(Q) do
   {
     // extraemos el nodo u de la cola Q y exploramos todos sus nodos adyacentes
      u = extraer(Q);
      for v ∈ adyacencia[u] do
     {
        if estado[v] == NO_VISITADO then
        {
          estado[v] = VISITADO;
          distancia[v] = distancia[u] + 1;
          padre[v] = u; Encolar(Q, v);
        }
       }
    }
}
```

Árbol de Búsqueda

- La **raíz** del árbol de búsqueda corresponde al **estado inicial**
- Los **nodos hoja** del árbol de búsqueda o no se han expandido, o no tienen sucesores, por lo que al expandirlos generaron un conjunto vacío



Árboles de búsqueda

Componentes de los árboles de búsqueda:

- El **estado** al que corresponde cada nodo,
- El **nodo padre (estado inicial)**,
- La **estrategia** que se aplica para generar cada nodo,
- La **profundidad** del nodo (distancia hasta la raíz),
- El **costo** desde el estado inicial hasta un nodo dado.

Algoritmo General de búsqueda ciega

- Iniciar árbol de búsqueda con el edo. inicial

- Si no hay candidato para expandir entonces
  - **parar**
- de lo contrario
  - escoger nodo para expandir según estrategia
  - Si nodo es edo. objetivo entonces
    - **parar**
  - de lo contrario
    - expandir nodo y añadir nuevo nodo al recorrido

Búsqueda por amplitud o extensión

Todos los nodos que están en la profundidad *d* del árbol de búsqueda se expanden antes de los nodos que estén en la profundidad *d+1*.

- Si son varias las soluciones, este tipo de búsqueda permitirá siempre encontrar primero el estado meta más próximo a la raíz.

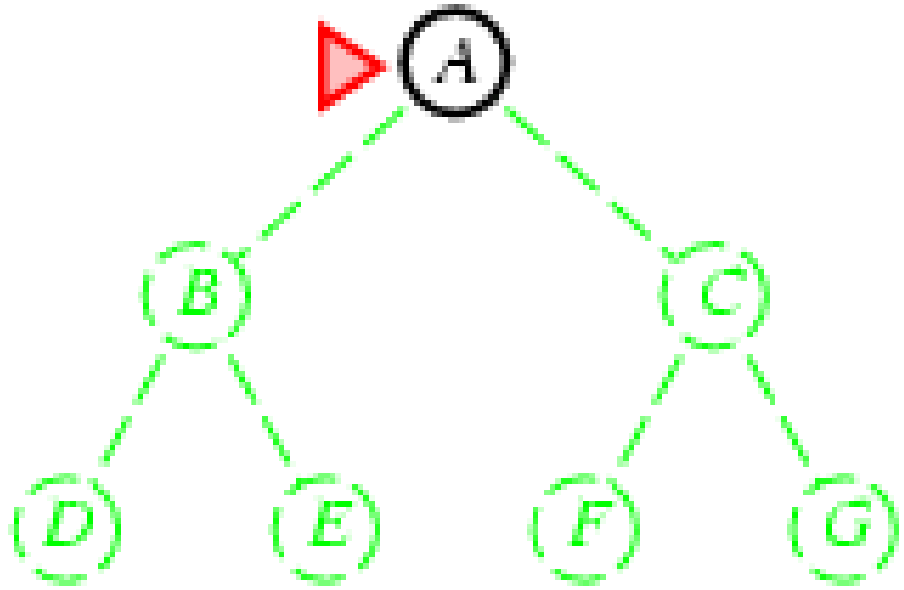
- El tiempo y la cantidad de memoria necesaria crece exponencialmente con respecto a la profundidad.

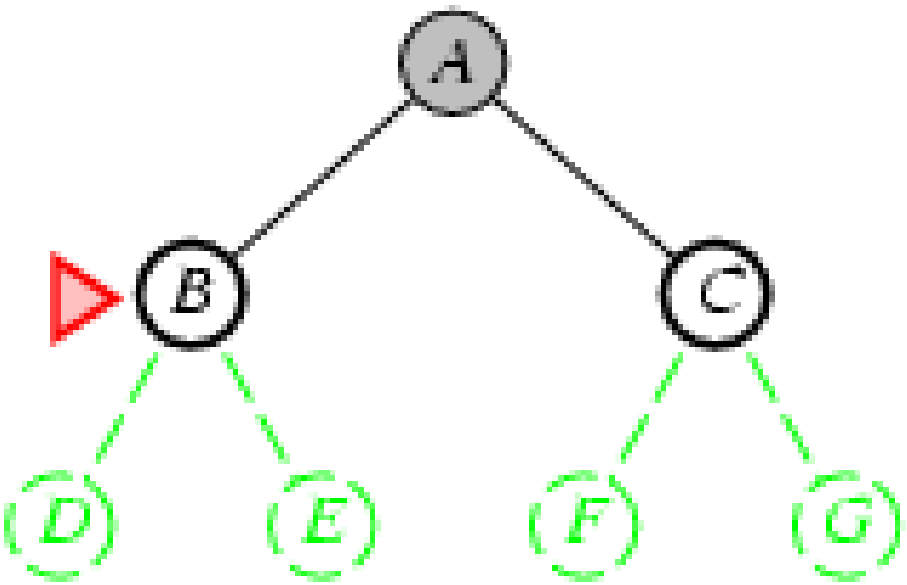
Es sub-óptima y completa.

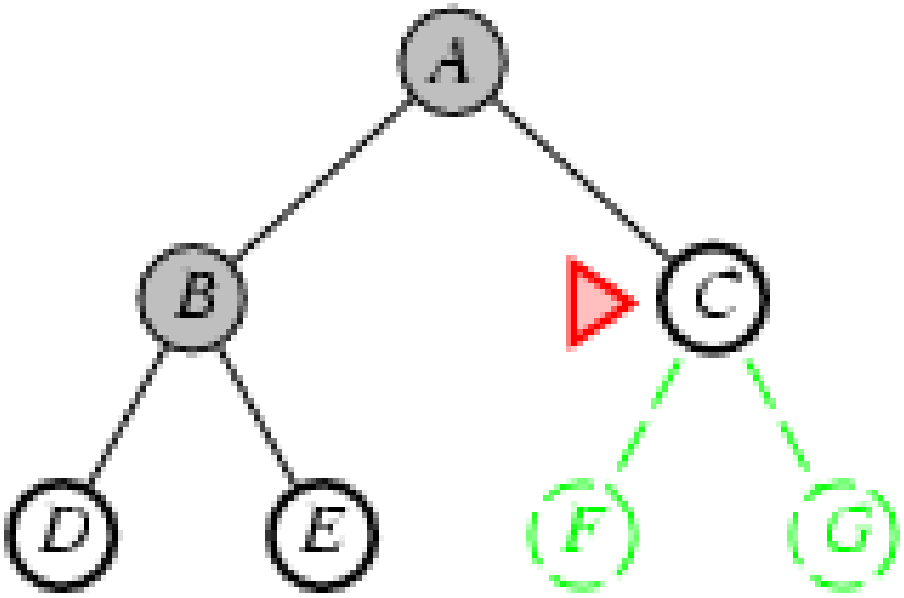
Búsqueda primero en anchura

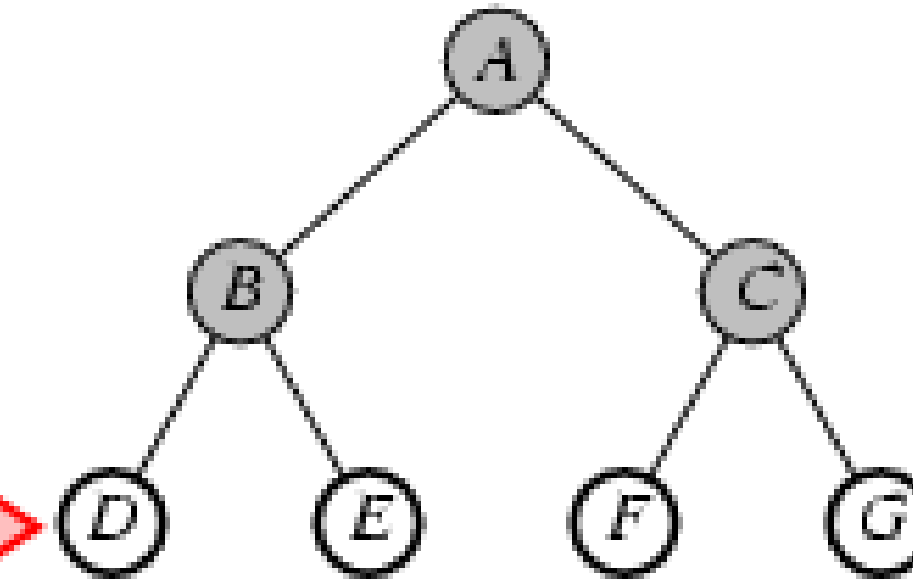
- El recorrido BPA es una generalización del recorrido por niveles de un árbol.
- Se pueden identificar dos tipos de aristas durante el recorrido:
  - *Aristas de descubrimiento*: son aquellas aristas que conducen al descubrimiento de nuevos vértices.
  - *Aristas de cruce*: son aristas que conducen a un vértice ya visitado.
- Las aristas de descubrimiento forman un árbol de cubrimiento de los componentes conectados del vértice inicial s.

Por extensión o amplitud









UNIVERSIDAD DE LOS ANDES
MERIDA VENEZUELA

búsqueda en amplitud matriz

		Versión 1.0
	recorridoEnAmp( )	
{pre: g (i, j) ≥ 0 ∀ i,j}		{pos: marca (i) = Verdadero ∀ i}
1 2	[marca(i) = falso] i = 1, pos [Si (¬ marca(i)) entonces busAmp(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

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

UNIVERSIDAD DE LOS ANDES
MERIDA VENEZUELA

búsqueda en amplitud matriz

Versión 1.0

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

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

1 2 3	mar(na) = Verdadero co.enCola(na) (¬ co.vaciaCola( )) [na = co.primero( ) co.desenCola( ) nadya = nodoAdyacente(na) [Si (¬ mar(nadya(i))) entonces mar(nadya(i)) = Verdadero fsi co.enCola(nadya(i))]i = 1, pos]	-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: Arreglo[100]De Entero+. Vector auxiliar que contiene los nodos adyacentes -enCola( ), primero( ), desenCola( ), vaciaCola( ). Operaciones de la clase Cola.

UNIVERSIDAD DE LOS ANDES
MERIDA VENEZUELA

búsqueda en amplitud lista

Versión 1.0

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

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

1 2 3	mar(na) = Verdadero co.enCola(na) (¬ 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

Algoritmo para grafo completamente conectado

```
void BFS(int v)
{  //v es el nodo de inicio del recorrido
   list<int> cola;//cola de adyacentes
   list<int>::iterator nodo_actual, aux, fin;
   visitado[v] = 1;//marcamos como visitado el nodo de inicio
   cola.push_back(v);//metemos inicio a la cola
   while(!cola.empty())
  {   nodo_actual = cola.front();//sacar nodo de la cola
      cola.pop_front();
      aux = grafo[nodo_actual].begin();//posicionar iteradores para //lista de ady
      fin = grafo[nodo_actual].end();
      while(aux != fin)
       {   //recorrer todos los nodos ady a nodo actual
         if(!visitado[*aux])
         {   //añadir a la cola solo los no visitados
            visitado[*aux] = 1;//marcarlos como visitados
            cola.push_back(*aux);//añadirlos a la cola
            //aqui podriamos añadir codigo para hacer algo mientras //recorremos el grafo
         }
         aux++;//avanzar al siguiente adyacente del nodo actual
      }
   }
}
```

Algoritmo para grafo que no esta completamente conectado

```
void BFS2(){
int i;
for(i = 0; i < nvert; i++)
    if(!visitado[i])
       BFS(i);
}
```

Grafos

Búsqueda en profundidad

Algoritmo BÚSQUEDA EN PROFUNDIDAD (Depth First Search)

- Se comienza en el **vértice inicial** (vértice con índice 1) que se marca como **vértice activo**.

- Hasta que todos los vértices hayan sido visitados, **en cada paso se avanza al vecino con el menor índice** siempre que se pueda, **pasando este a ser el vértice activo.**

- **Cuando todos los vecinos al vértice activo hayan sido visitados, se retrocede** al vértice *X* desde el que se alcanzó el vértice activo y se prosigue siendo ahora *X* el vértice activo.

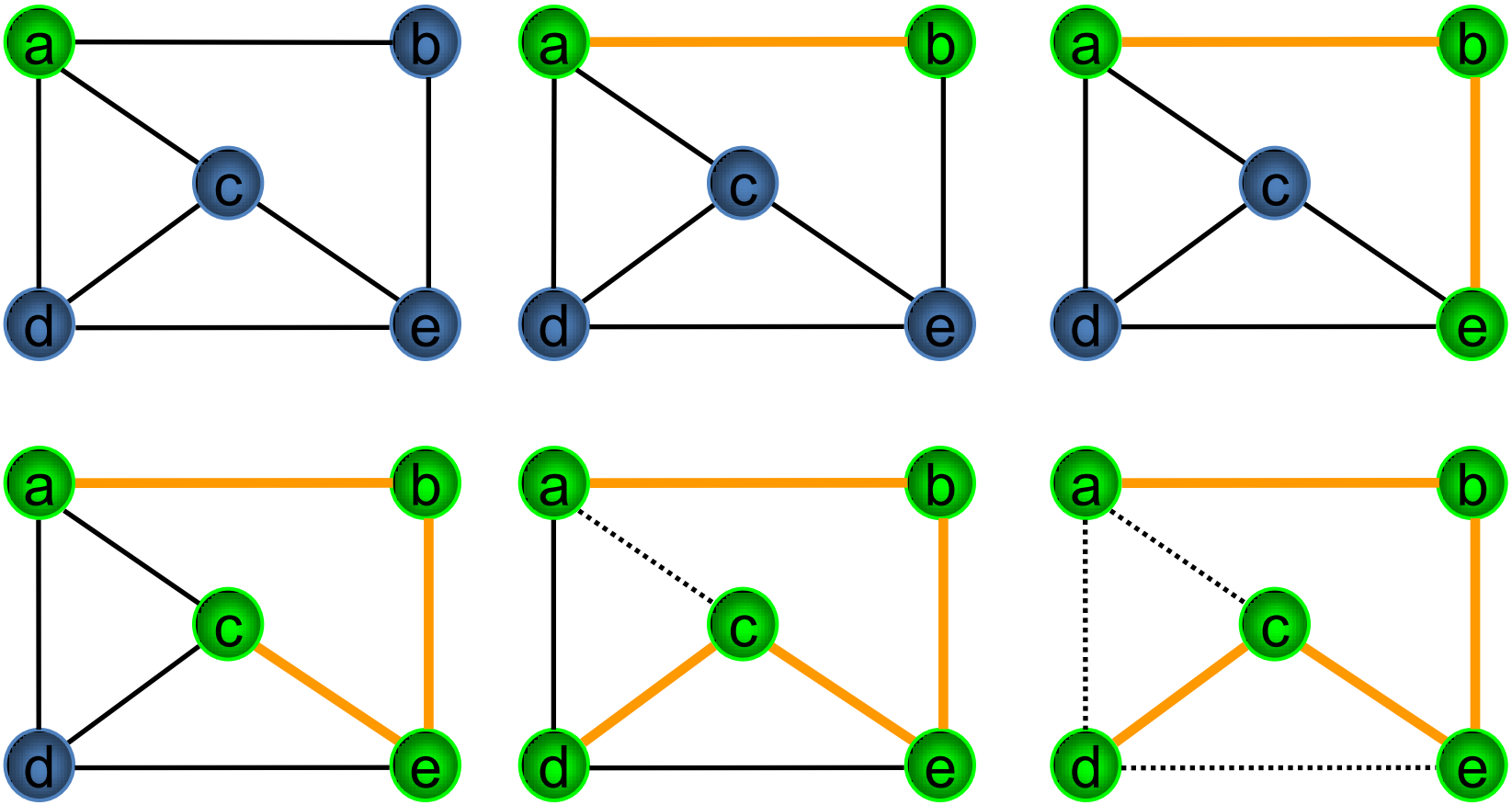
Algoritmo BÚSQUEDA EN PROFUNDIDAD (Depth First Search)

```
DFS(grafo G)
PARA CADA vertice u ∈ V[G] HACER
    estado[u] ← NO_VISITADO
    padre[u] ← NULO
tiempo ← 0
PARA CADA vertice u ∈ V[G] HACER
    SI estado[u] = NO_VISITADO ENTONCES
        DFS_Visitar(u,tiempo)
```

```
DFS_Visitar(nodo u, int tiempo)
estado[u] ← VISITADO
tiempo ← tiempo + 1
d[u] ← tiempo
PARA CADA v ∈ Vecinos[u] HACER
   SI estado[v] = NO_VISITADO ENTONCES
        padre[v] ← u
        DFS_Visitar(v,tiempo)
estado[u] ← TERMINADO
tiempo ← tiempo + 1
f[u] ← tiempo
```

Búsqueda primero en profundidad

- *Animación*

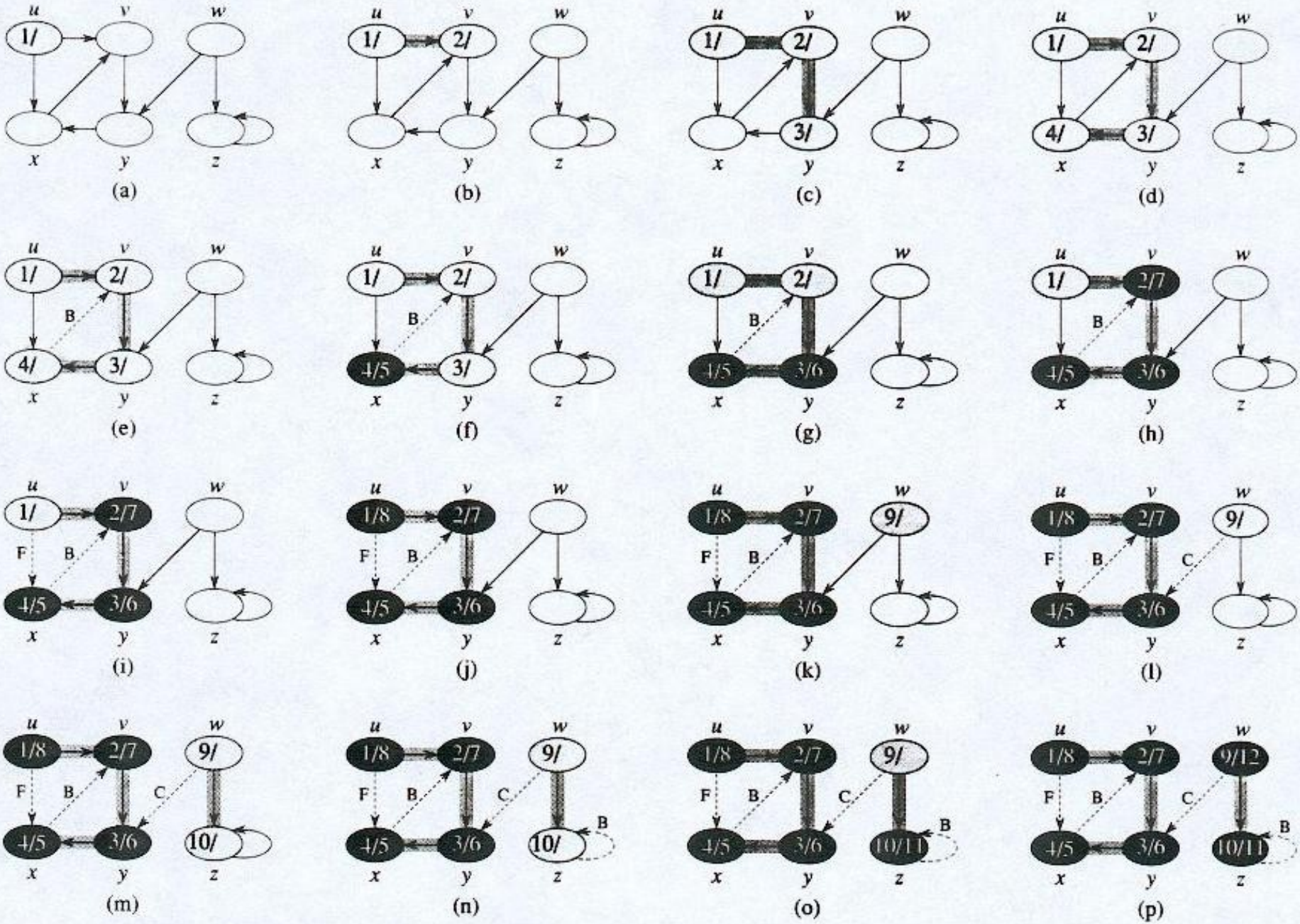


Algoritmo recursivo BPP

```
recorre_grafo_bpp()
{
  for cada vértice v
    marca[v]=SINVISITAR;
  for cada vértice v
     if (marca[v]==SINVISITAR)
            BPP(v);
}
```

```
BPP(u)
{
  marca[u]=VISITADO;
  for cada vértice v adyacente a u
     if (marca[v]==SINVISITAR)
            BPP(v);
}
```

- Orden de complejidad del recorrido en profundidad:
  - Con lista de adyacencia, se recorre cada elemento de lista una vez, O(n + e).
  - Con matriz de adyacencia, para cada nodo se buscan sus adyacentes, O(n^2).


u v w
1/
x y z
(a)
u v w
1/ 2/
x y z
(b)
u v w
1/ 2/ 3/
x y z
(c)
u v w
1/ 2/ 4/ 3/
x y z
(d)
u v w
1/ 2/ B 4/ 3/
x y z
(e)
u v w
1/ 2/ B 4/5 3/
x y z
(f)
u v w
1/ 2/ B 4/5 3/6
x y z
(g)
u v w
1/ 2/7 B 4/5 3/6
x y z
(h)
u v w
1/ 2/7 F B 4/5 3/6
x y z
(i)
u v w
1/8 2/7 F B 4/5 3/6
x y z
(j)
u v w
1/8 2/7 9/ F B 4/5 3/6
x y z
(k)
u v w
1/8 2/7 9/ F B C 4/5 3/6
x y z
(l)
u v w
1/8 2/7 9/ F B C 4/5 3/6 10/
x y z
(m)
u v w
1/8 2/7 9/ F B C 4/5 3/6 10/ B
x y z
(n)
u v w
1/8 2/7 9/ F B C 4/5 3/6 10/11 B
x y z
(o)
u v w
1/8 2/7 9/12 F B C 4/5 3/6 10/11 B
x y z
(p)

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

Búsqueda por profundidad en árbol

Siempre se expande uno de los nodos que se encuentren en los mas profundo del árbol.

Solo si la búsqueda conduce a un callejón sin salida, se revierte la búsqueda y se expanden los nodos de niveles menos profundos.

Esta búsqueda,

- o se queda atorada en un bucle infinito, y nunca es posible regresar al encuentro de una solución,
- o encuentra una ruta de solución más larga que la solución óptima.

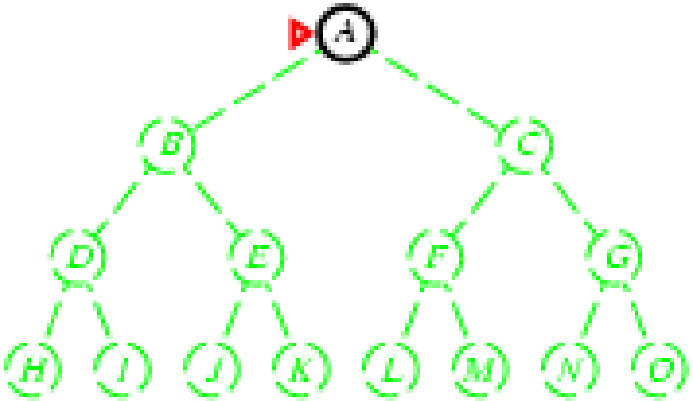
El tiempo necesario crece exponencialmente con respecto a la profundidad, mientras que el espacio requerido en memoria lo hace en forma lineal

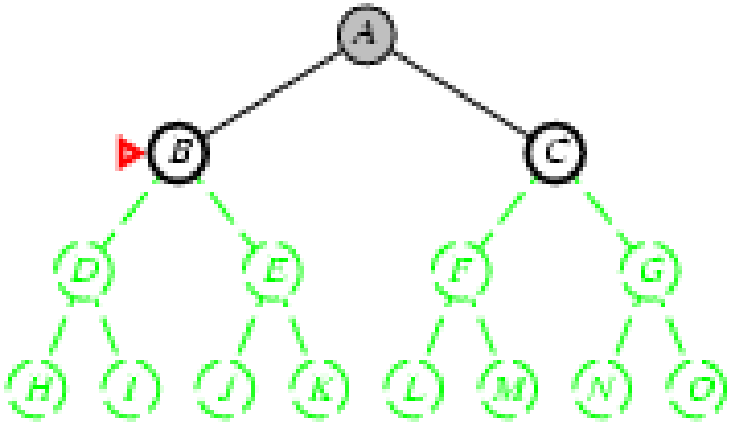
No es óptima pero completa.

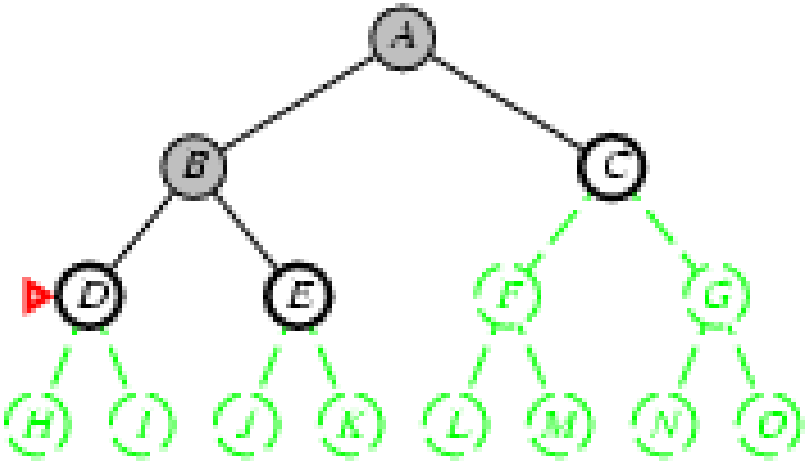
Búsqueda primero en profundidad

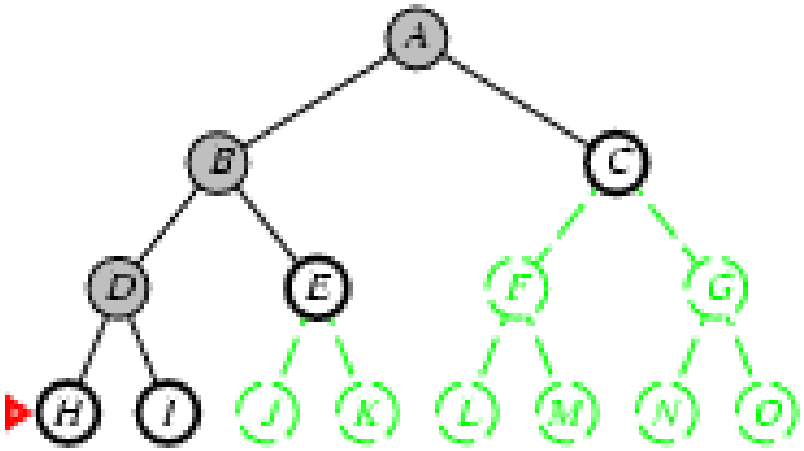
- El recorrido BPP es una generalización del recorrido preorden de un árbol.
- Se pueden identificar cuatro tipos de aristas durante el recorrido:
  - *Aristas de descubrimiento*: son aquellas aristas que conducen al descubrimiento de nuevos vértices. También se les llama *aristas de árbol*.
  - *Aristas de retorno*: son las aristas que conducen a vértices antecesores ya visitados en el árbol.
  - *Aristas de avance*: son las aristas que conducen a vértices descendientes en el árbol.
  - *Aristas de cruce*: son aristas que conducen a un vértice que no es ancestro de otro o a vértices que están en diferentes árboles.
- Las aristas de descubrimiento forman un árbol de cubrimiento de los componentes conectados del vértice inicial s.

Por profundidad

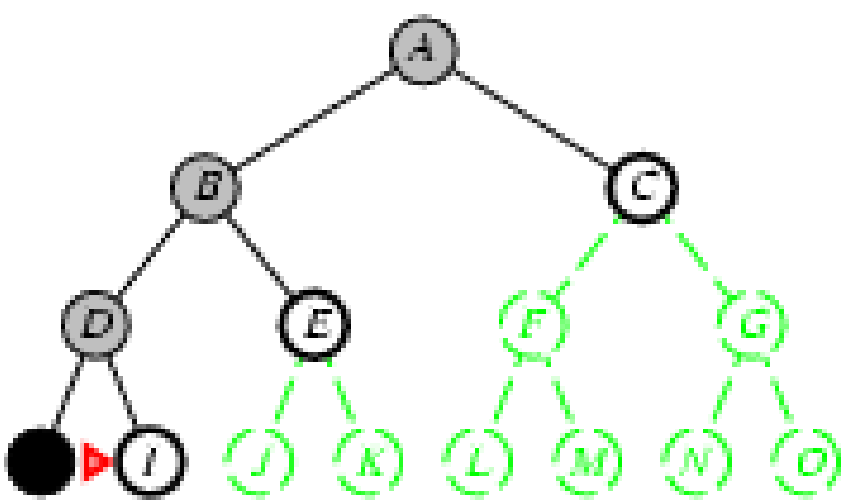


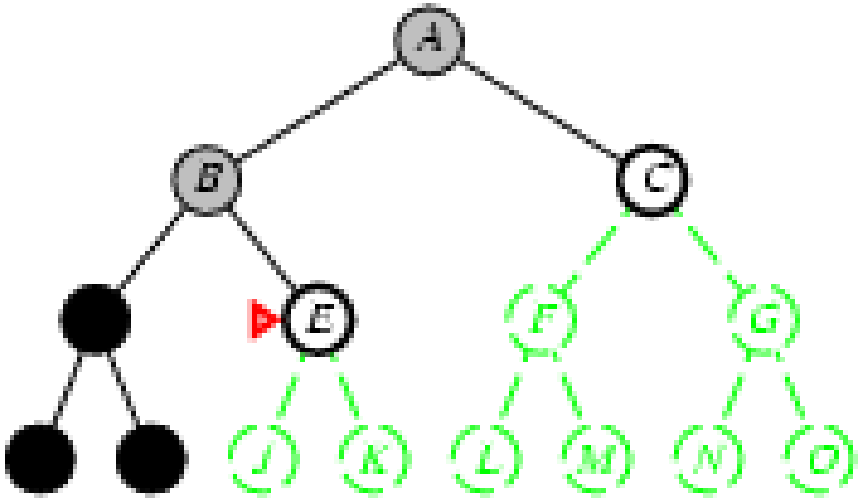


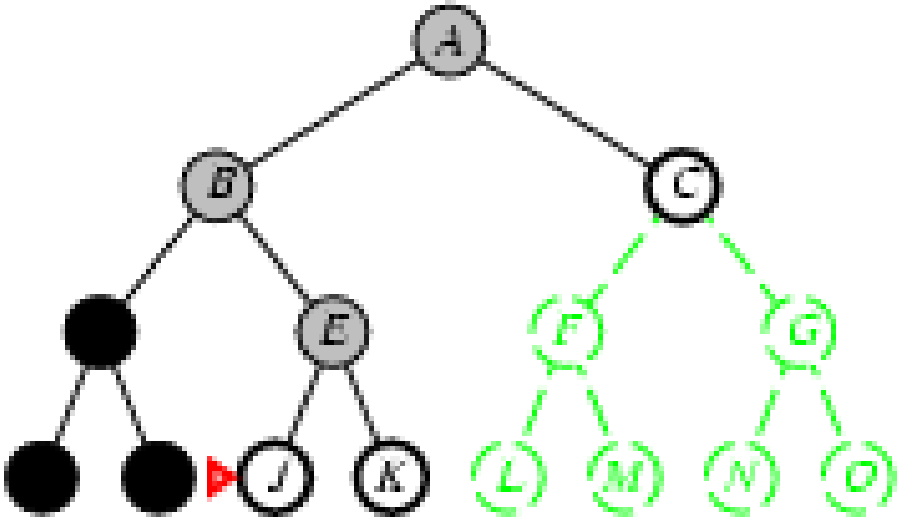


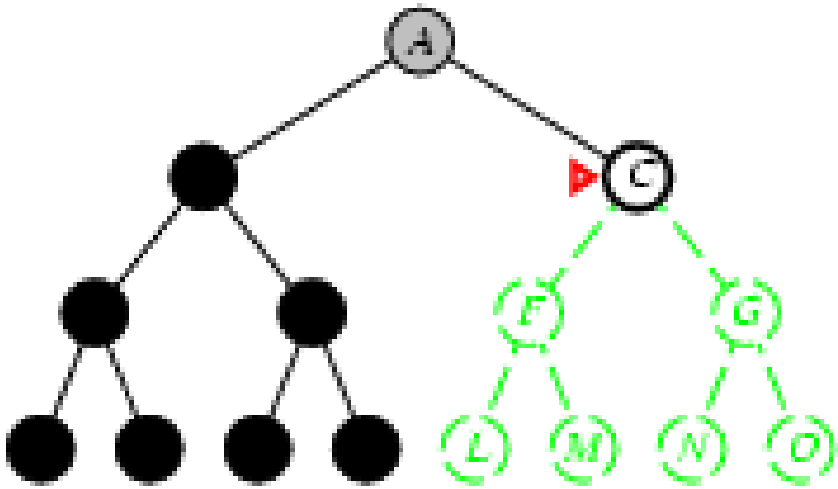


Por profundidad









Árboles binarios

Recorridos Árboles binarios:

- Preorden: raíz-izquierdo-derecho (RID)
- Enorden: izquierdo-raíz-derecho (IRD)
- Posorden: izquierdo-derecho-raíz (IDR)

Pre y posorden

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 Θ(n)

Árboles binarios

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

UNIVERSIDAD DE LOS ANDES
MERIDA VENEZUELA

búsqueda en profundidad matriz

		Versión 1.0
	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

$$T(n) = \Theta(\max(N,A))$$

UNIVERSIDAD DE LOS ANDES
MERIDA VENEZUELA

búsqueda en profundidad matriz

Versión 1.0

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

UNIVERSIDAD DE LOS ANDES
MERIDA VENEZUELA

búsqueda en profundidad lista

		Versión 1.0
	recorridoEnProf( )	
{pre: g.n ≥ 0 }		{pos: marca (i) = Verdadero ∀ i}
1 2	[marca(i) = falso] i = 1, g.numEle( ) [Si (¬ 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

$$T(n) = \Theta(\max(N,A))$$

UNIVERSIDAD DE LOS ANDES
MERIDA VENEZUELA

búsqueda en profundidad lista

Versión 1.0

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

{pre: g.n ≥ 0 ∧ 1 ≤ na ≤ g.n} {pos: marca (i) = Verdadero ∀ i}

1 2 3	mar(na) = Verdadero nadya = nodoAdyacente(na) nadya.cursorAlInicio( ) [Si(¬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

// versión para grafos que no están completamente conectados

```
void DFS2()
{  int i;
    for(i = 0; i < nvert; i++)//buscar un nuevo nodo de inicio que no ha sido visitado
       if(!visitado[i])
          DFS(i);
}
```

UNIVERSIDAD DE LOS ANDES
MERIDA VENEZUELA

búsqueda en profundidad lista

Versión 1.0

puntosDeArticulacion( )

{pre: |N| > 0 } {pos: |N| > 0 ∧ G' = G ∧ padre ⊃ bosque en profundidad }

1 2 3	recorridoEnProf() Recorrer T en posorden y para cada nodo na se calcula masalto(na) [Si (na = raíz ∧ na no tiene mas de 1 hijo) entonces na es un punto de articulación sino Si (na ≠ raíz ∧ na tiene un hijo x / masalto(x) ≥ prenum(na)) entonces na es un punto de articulación fsi] na ∈ N	-na, x: Entero. Nodo actual y nodo hijo del actual. -recorridoEnProf(). Realiza la búsqueda en profundidad calculando prenum de cada nodo - Se numeran los nodos de G con prenum haciendo el recorrido en profundidad (preorden) -calcula el valor de masalto para c/nodo masalto(v) = mín(prenum(v), prenum(w), masalto(x)) Donde, v) es el nodo actual; (w) Es el nodo antecesor de v alcanzable por una arista que no pertenece a las aristas del árbol del recorrido en profundidad; (x) Todo hijo de v desde los cuales se puede recorrer hasta un antecesor de v

Algoritmo para determinar grafo conexo usando búsqueda en profundidad

```
bool Graph::is_connected()
{               If (_n <= 1)
                return true;
vector<bool> visit(_n);
vector<bool>::iterator iter;
for(iter = visit.begin(); iter != visit.end(); iter++)
                *iter = false;
set<int> forvisit;
set<int>::iterator current;
forvisit.insert(0);
while (!forvisit.empty())
{               current = forvisit.begin();
                if (!visit[*current])
                { for (int i = 0; i < _n; i++)
                   { if (_graph[*current][i] == 1 && !visit[i])
forvisit.insert(i); } }
                visit[*current] = true;
                forvisit.erase(current); }
bool result;
for (iter = visit.begin(); iter != visit.end(); iter++) result = result && *iter;
                return result;
```

Usos de los Recorridos

- Ambos recorridos se pueden usar para resolver los siguientes problemas:
  - Probar que G es conectado.
  - Obtener un árbol de expansión de G.
  - Obtener los componentes conectados de G.
  - Obtener un camino entre dos vértices dados de G, o indicar que no existe tal camino.

- El recorrido BPP se usa para:
  - Obtener un ciclo en G, o indicar que G no tiene ciclos.

- El recorrido BPA se usa para:
  - Obtener para cada vértice v de G, el número mínimo de aristas de cualquier camino entre s y v.

Grafos

Ordenamiento topológico

Ordenamiento topológico

Resuelve el problema de encontrar el orden en que se deben llevar a cabo una serie de actividades cuando existen requisitos de actividades previas a realizar como puede ser el currículo de una carrera en una universidad.

Muchas actividades requieren de dicha planeación

Ordenación

El orden topológico tiene sentido sólo en grafos acíclicos dirigidos (DAG).

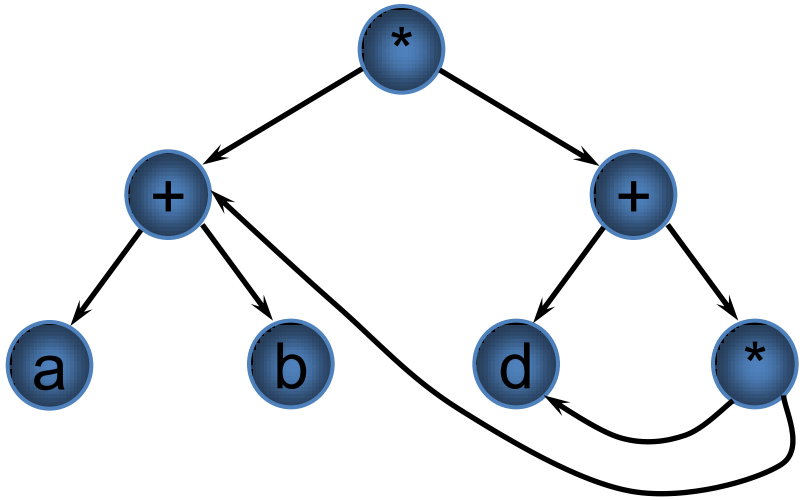
Orden topológico de un DAG G=(V,E) es un orden lineal de todos los vértices tal que si G contiene el arco (u,v), entonces ***u*** aparece antes que ***v*** en el orden.

Cuando se tienen muchas actividades que dependen parcialmente unas de otras, este orden permite definir un **orden de ejecución sin conflictos**.

Gráficamente se trata de poner todos los nodos en una línea de manera que sólo haya arcos hacia delante.

Digrafos acíclicos

- Es un grafo dirigido que no tiene ciclos.
- Representan relaciones más generales que los árboles pero menos generales que los digrafos.
- *Ejemplo*: representar estructuras sintácticas de expresiones aritméticas con subexpresiones comunes y el orden parcial de un conjunto.



(a+b)*(d+d*(a+b))

Orden parcial R en un conjunto S, relación binaria que cumple:

- $\forall$ elemento a de S, (a R a) es falso.
- $\forall$ a, b, c de S, si (a R b) y (b R c) entonces (a R c).

Ordenación

Reflexividad: Una relación R es reflexiva si X R X para todo X en S.

Simetría: Una relación R es simétrica si X R Y implica Y R X para todo X y Y.

Antisimetría: Una relación R es antisimétrica, si X R Y y Y R X implica X = Y, para todo X y Y.
La relación ≤ es antisimétrica, $x \leq p$ y $p \leq x$ implica que $x=p$

Transitividad: Una relación R es transitiva si X R Y y Y R Z implica que X R Z, para todo X, Y, Z.

Un orden parcial (toda relación antisimétrica y transitiva) tiene un digrafo acíclico

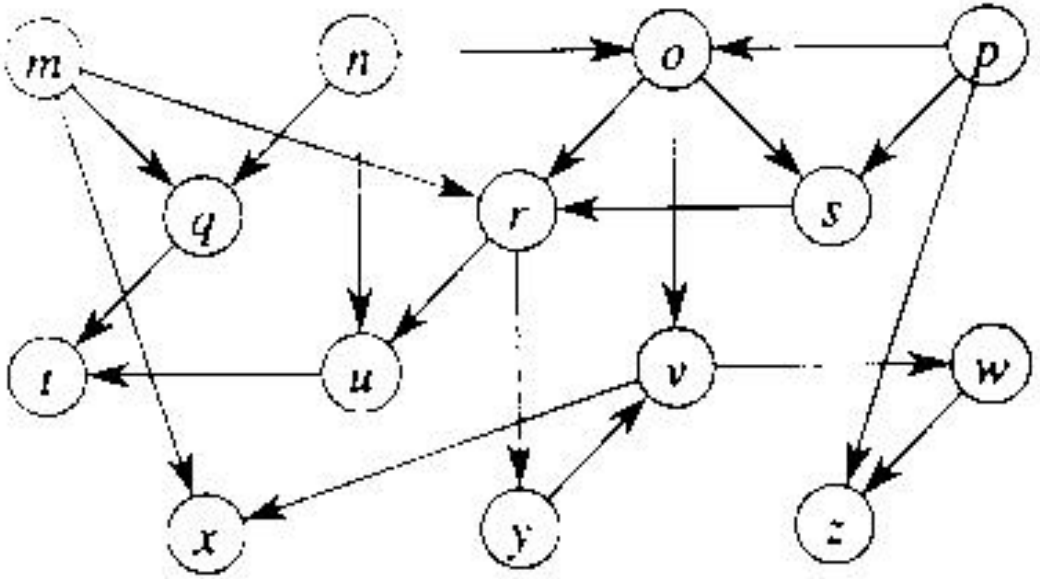
Ordenación

Un orden topológico de los nodos de un digrafo G={N, A} es una secuencia (n1, n2, …, nk) tal que N= {n1, n2, …, nk} y para todo (ni, nj) en N, ni precede a nj en la secuencia.

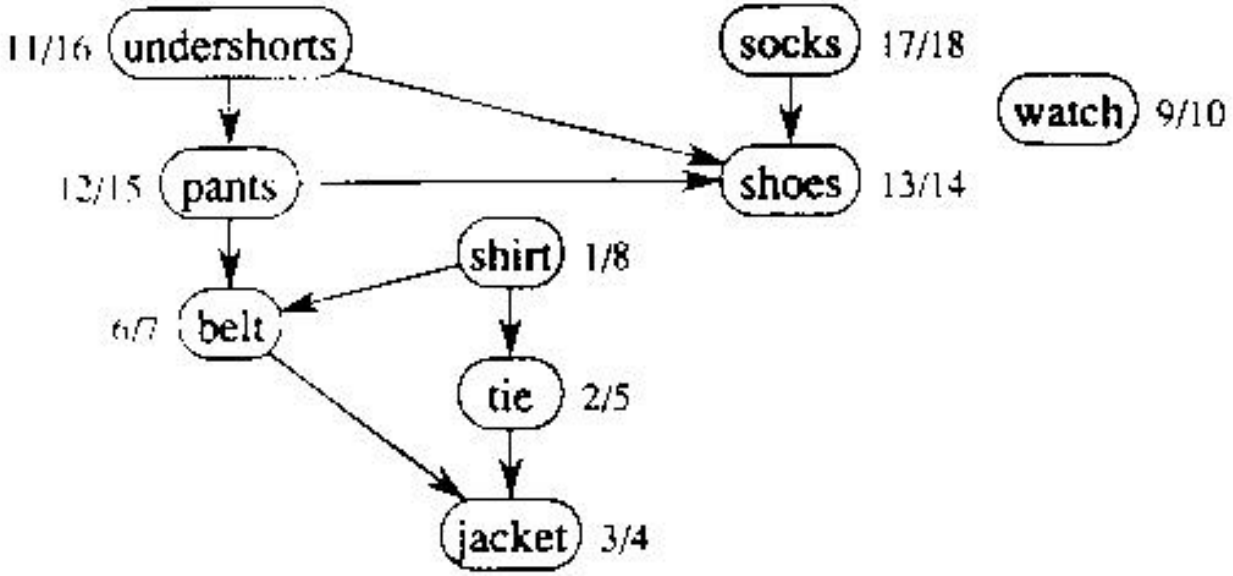
Aplicaciones:

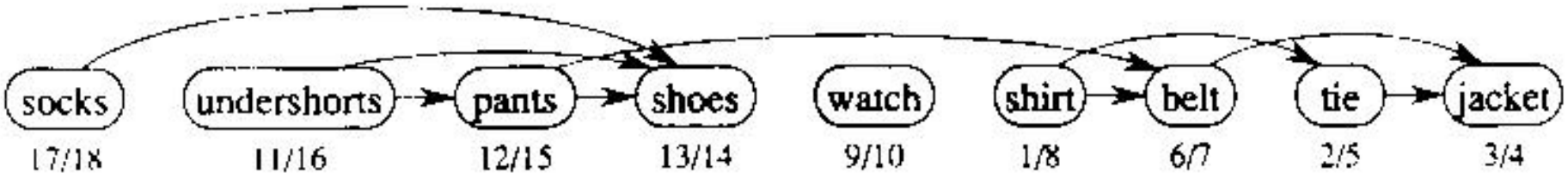
- Modelado de actividades para resolver problemas de planificación o programación de las mismas siguiendo un criterio de minimización o maximización.

- Evaluación de expresiones aritméticas
  (- b + sqrt(b * b - 4 * a * c)) / 2 * a
  (- b - sqrt(b * b - 4 * a * c)) / 2 * a



¿Cuál es el orden topológico?





¿Es éste el único orden topológico?

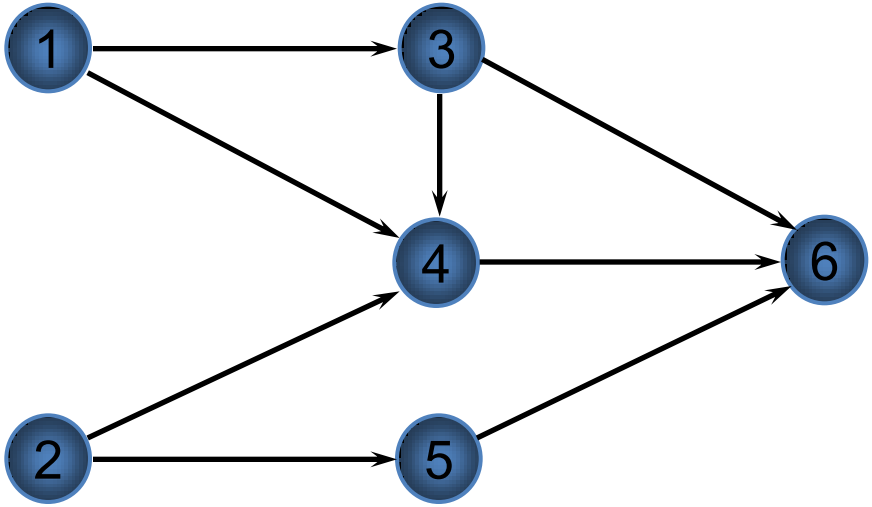
Orden topológico

- *Ordenamiento topológico de un digrafo acíclico*: orden lineal de los vértices colocándolos a lo largo de una línea horizontal de tal manera que todas las aristas tengan una dirección de izquierda a derecha.
- Ejemplo: las tareas de un proyecto de construcción.
- Algoritmo: usar una versión modificada de BPP.

```
orden_topologico(v)      /* orden inverso */
{
  marca[v]=VISITADO;
  for cada vértice w en lista_adyacencia(v)
          if (marca[w]==SINVISITAR)
                orden_topologico(w);
  imprime(v);
}
```

Orden topológico

- *Ejemplo*

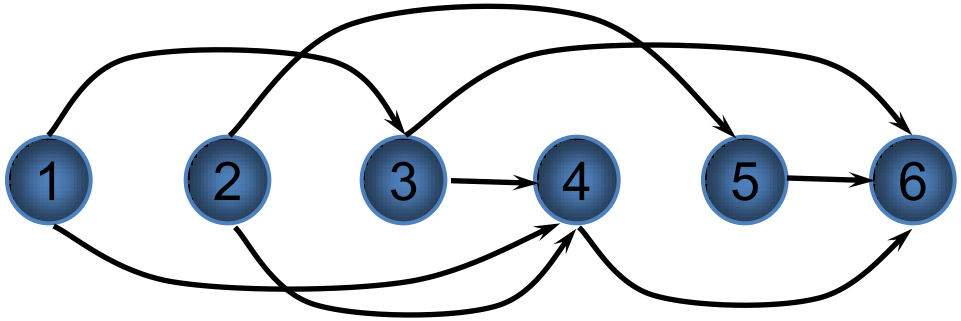


- *Orden topológico:*

1 2 3 4 5 6

1 3 2 4 5 6

2 1 5 3 4 6



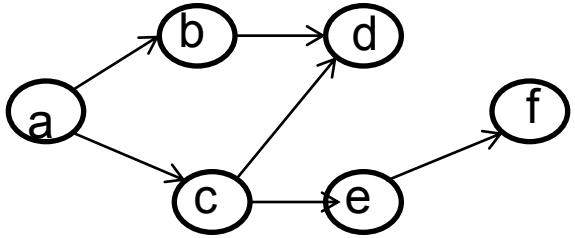
Algoritmo genérico de ordenamiento topológico

Versión 1.0

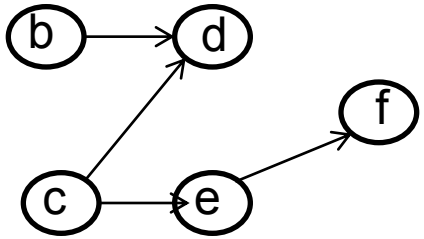
ordenTopologico( ): Lista

{pre: n > 0 } {pos: n = 0 }

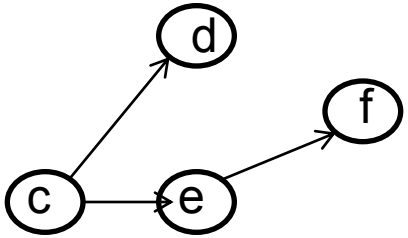
1 2 3	(n > 0) [v = nodo con gin = 0 elimine v y todas las aristas que salen de v s.inserLista(v)] regrese s	-v: Entero. Nodo del digrafo. -inserLista(). Definida en Lista. -s. Lista. Lista que contiene el orden topológico de los nodos del grafo. - gin: grado de incidencia negativo



v=a, s={ }, n=6



v=b, s={a}, n=5



v=c, s={a, b}, n=4

...

ORDENAMIENTO_TOPOLOGICO(G)

DFS(G) y calcular f[v] para cada v en G
Conforme se termina de visitar un vértice insertar al frente de una lista
Devolver la lista como resultado

```
DFS(G)
Para cada nodo u en G
        color[u] = blanco
        padre[u] = NULO
tiempo = 0
Para cada nodo u en G
        Si color[u] = blanco DFS-VISIT(u)

DFS-VISIT(u)
color[u] = gris //se acaba de visitar el nodo u por primera vez
tiempo = tiempo + 1
d[u] = tiempo
Para cada nodo v que se adyacente a u //explorar aristas (u,v)
        si color[v] = blanco
                padre[v] = u
                DFS-VISIT(v)
color[u] = negro //se termino de visitar al nodo u y todos sus adyacentes
tiempo =  tiempo + 1
f[u] = tiempo
```

Componentes fuertemente Conexas

Un componente fuertemente conexo de un grafo G=(V,E) es el máximo conjunto de vértices U subconjunto de V tal que para cada par de vértices u, v en U, existan caminos desde u a v y viceversa.

- Algoritmo que descubre todos las componentes fuertemente conexos.
- Para ello define el grafo traspuesto de G, G^T= (V,E^T), donde E^T={(u,v) tal que (v,u) pertenece a E}. En otras palabras, invierte el sentido de todas los arcos.

Algoritmo Strongly_Connected_Components(G)

1.- Llamar a DFS(G) para obtener el tiempo de término f[u], para cada vértice u;
2.- Calcular G^T;
3.- Llamar a DFS(G^T), pero en el lazo principal de DFS, considerar los vértices en orden decreciente de f[u].
4.- La salida son los vértices de cada árbol del paso 3. Cada árbol es una componente fuertemente conexo separada.