



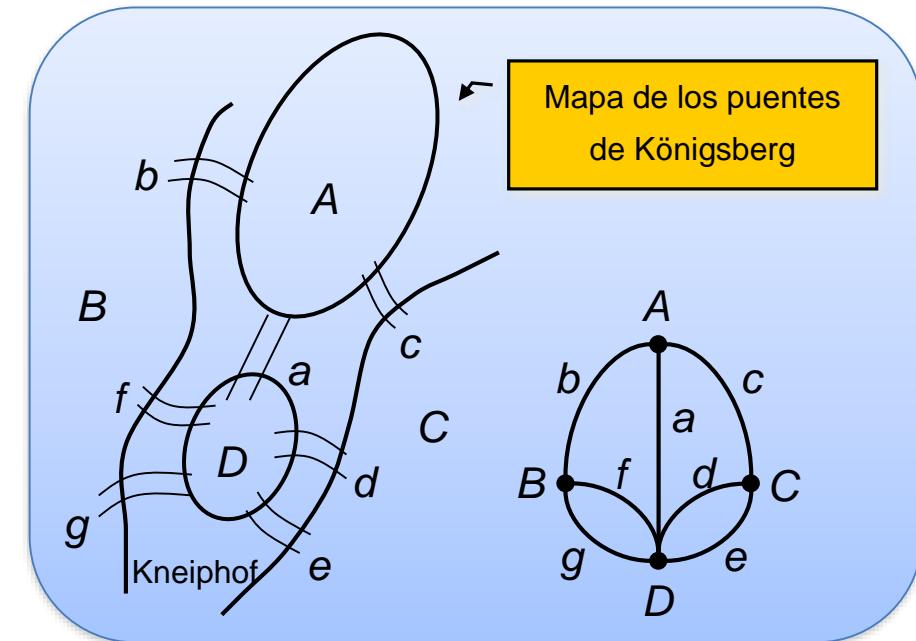
6. Grafos

Tema VI



Introducción

- En problemas originados en ciencias de la computación, matemáticas, ingeniería y muchas otras disciplinas, a menudo es necesario representar **relaciones arbitrarias** entre objetos de datos. Los **grafos dirigidos** y los **no dirigidos** son modelos naturales de tales relaciones.
- Gráficamente, los puntos (**vértices**) representan objetos individuales, posiciones, componentes químicos, etc., conectados por líneas (**ejes**) o flechas (**arcos**) que simbolizan las relaciones.





Definición de grafo

Grafo

Un grafo es un par $G=(V, A)$ formado por dos conjuntos, V (de vértices o nodos) y A (de arcos o ejes).

Cada arco es un par (v, w) con $v, w \in V$.

El grafo es **simple** si sólo hay un arco (v, w) .

Los vértices y los arcos pueden estar **etiquetados**.

Grafo Dirigido

Si el par (v, w) es ordenado, el grafo es un **grafo dirigido** o **dagrafo**.

Se dice que el arco $(v, w) \in A$ va de v a w y que w es **adyacente** a v .

Grafo No Dirigido

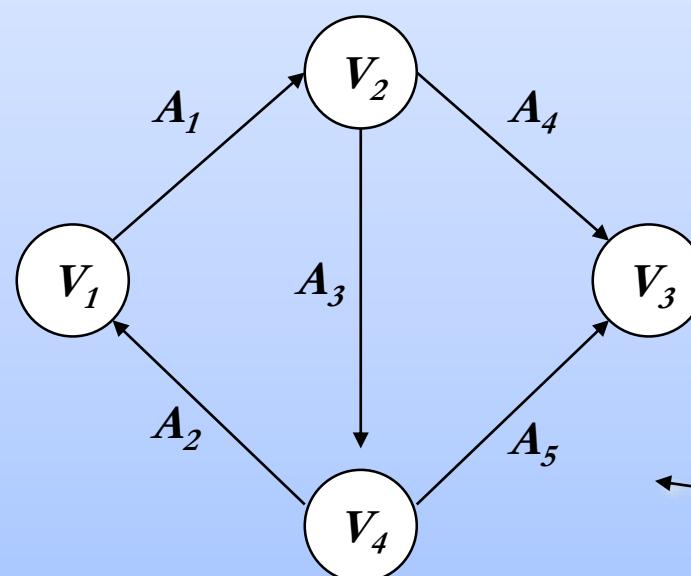
Si el par (v, w) no es ordenado, el grafo es un **grafo no dirigido** o simplemente **grafo**.

En este caso $(v, w) = (w, v)$ y se dice que v y w son **adyacentes** entre sí.

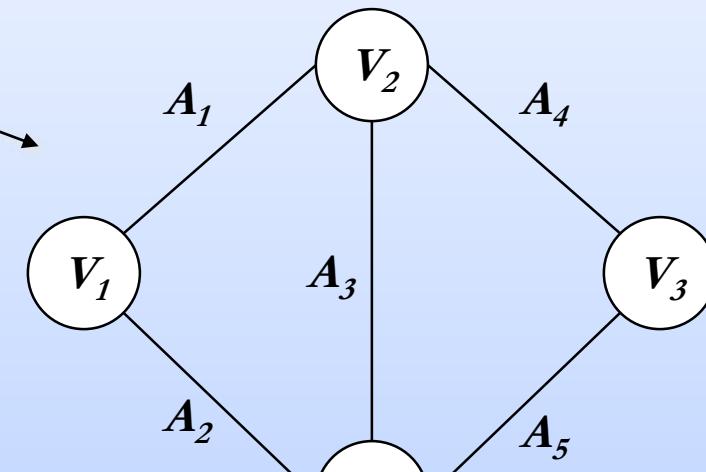


Ejemplos de grafos

Grafo $G=(V, A)$



Digrafo $D=(V, A)$

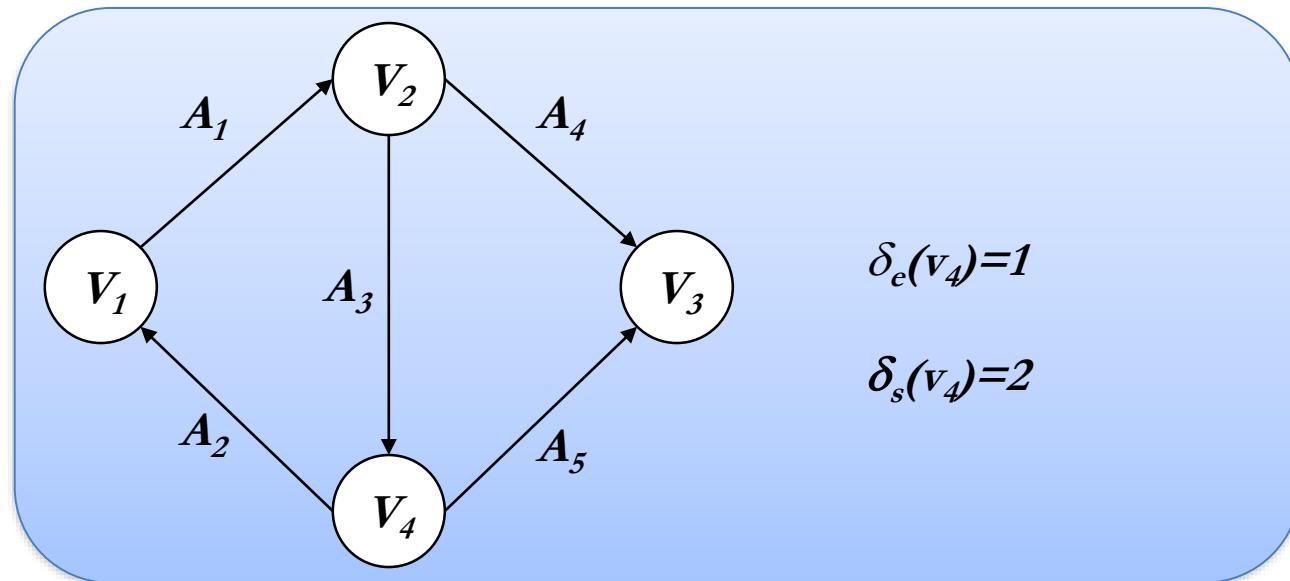


Conceptos de grafos

Grado de un vértice

Sea $G=(V, A)$ un grafo y $v \in V$ un vértice, se define el grado de v , $\delta(v)$, como el número de arcos incidentes en él; es decir, el número de arcos de los que v es extremo.

En digrafos se definen el grado de entrada $\delta_e(v)$ y el grado de salida $\delta_s(v)$.





Conceptos de grafos (II)

Camino de un grafo

Sea G un grafo (o digrafo). Un camino en G es una sucesión de vértices y arcos

$$v_0 \ a_1 \ v_1 \ a_2 \ v_2 \dots \ a_n v_n$$

tal que los extremos (origen y destino si es un digrafo) del arco a_i son v_{i-1} y v_i ($0 \leq i \leq n$).

Al número de arcos del camino se le denomina longitud (orden) de éste; v_0 y v_n son los extremos (origen y destino para un digrafo) del camino y los vértices v_1, v_2, \dots, v_{n-1} se denominan vértices interiores.

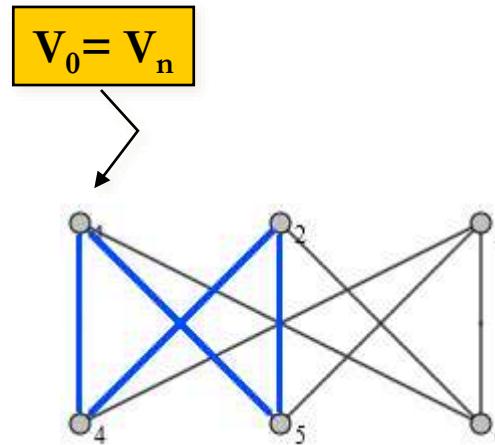
- Un camino es cerrado si $v_0 = v_n$
- Un camino es simple si sus arcos son distintos dos a dos
- Un camino es elemental si sus vértices son distintos dos a dos exceptuando, a lo sumo, sus extremos.
- Un camino de longitud cero es el que va de un vértice a sí mismo.



Conceptos de grafos (III)

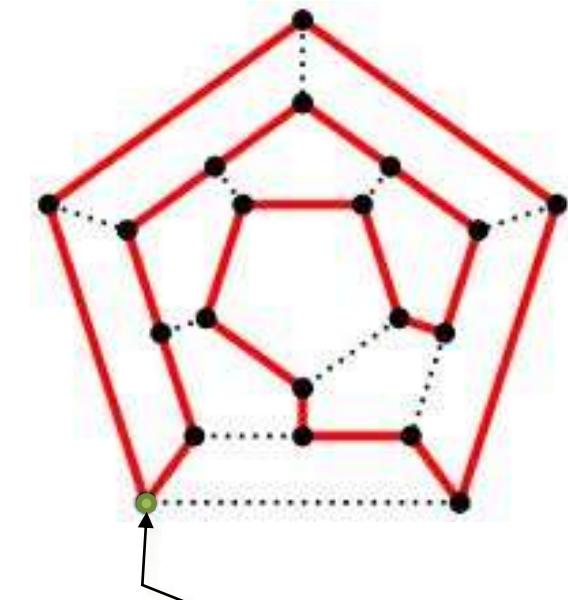
Ciclo de un grafo

Es un camino cerrado, elemental y simple (no se repiten ni vértices ni arcos, salvo el vértice inicial y final)



Ciclo hamiltoniano

Ciclo donde todos los vértices se recorren exactamente una vez



Conceptos de grafos (IV)

Vértices conectados

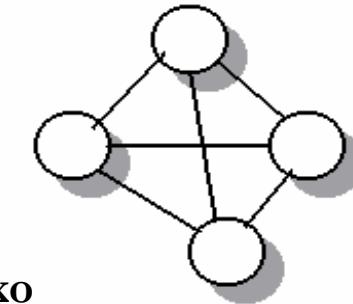
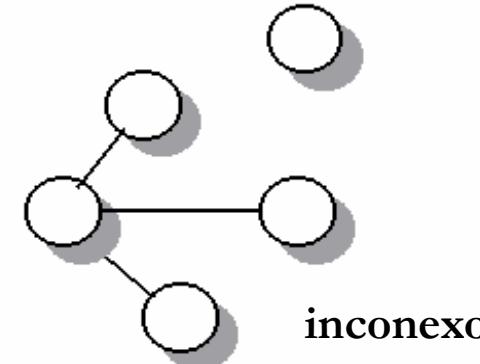
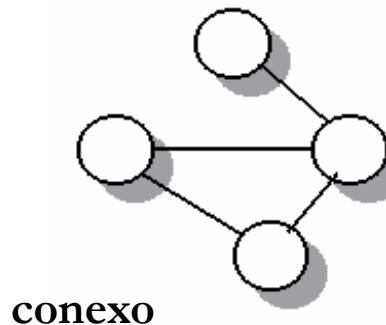
Dos vértices u y v de un grafo G se dice que están conectados cuando existe un camino en G de extremos u y v .

Grafo conexo (conectado)

Un grafo G es conexo si hay un camino entre cada par de vértices del mismo.

Grafo completo

Un grafo G es completo si es conexo y además cada par de vértices están unidos por un arco.



Conceptos de grafos (V)

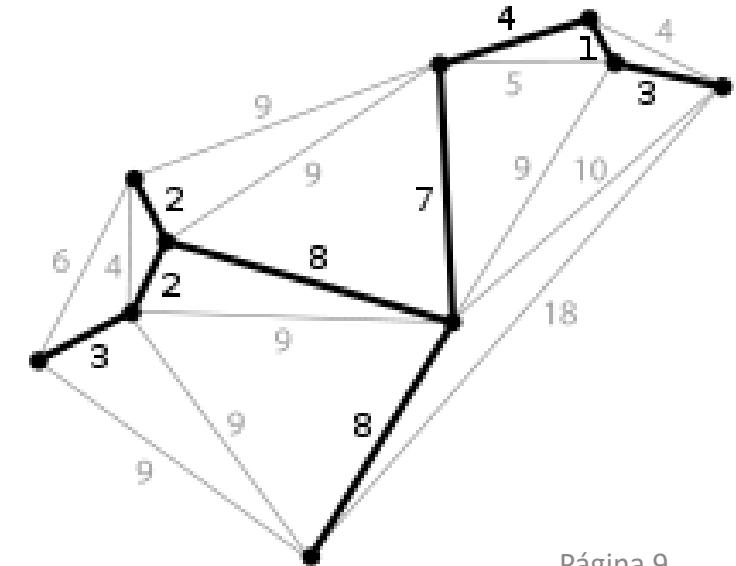
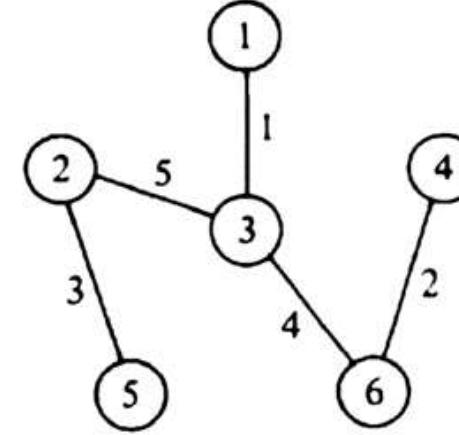
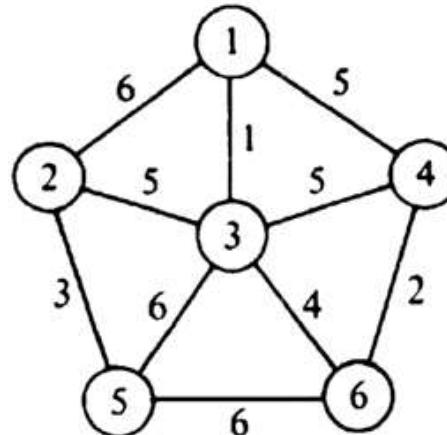
Árbol de expansión (árbol generador)

Un árbol de expansión (*spanning tree*) de un grafo G es un árbol que conecta todos los vértices de V .

Coste de un árbol de expansión

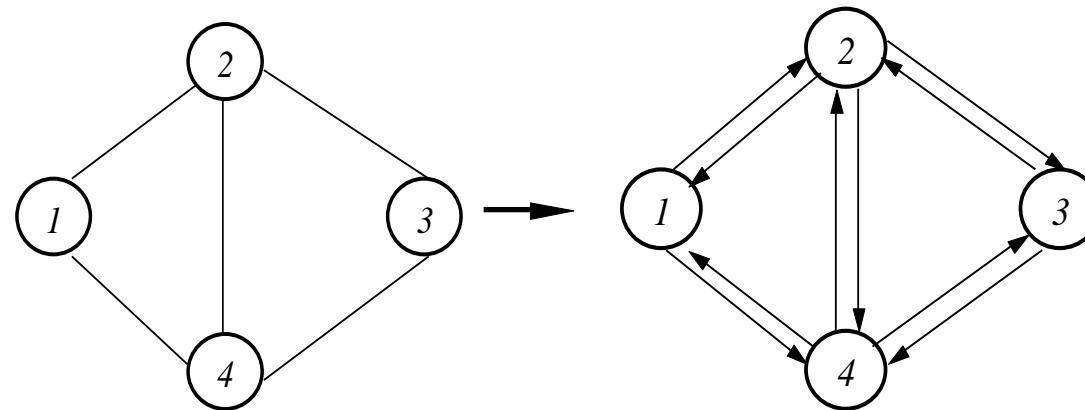
El coste de un árbol de expansión es la suma de los costes de los arcos del árbol.

Un problema típico consiste en la búsqueda del **árbol de expansión de mínimo coste**.



Representaciones de grafos

Desde el punto de vista de la representación no se establecerá ninguna diferencia entre grafos no dirigidos y grafos dirigidos dada la equivalencia que existe entre ambos:

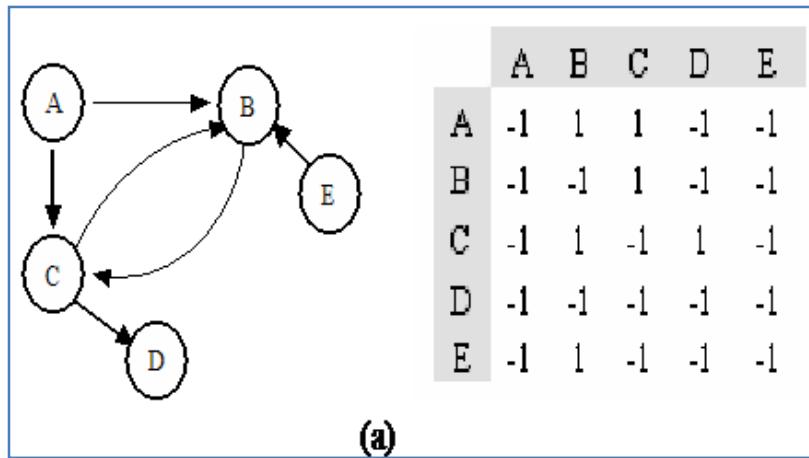


Las dos representaciones de grafos más extendidas son las que se basan en:

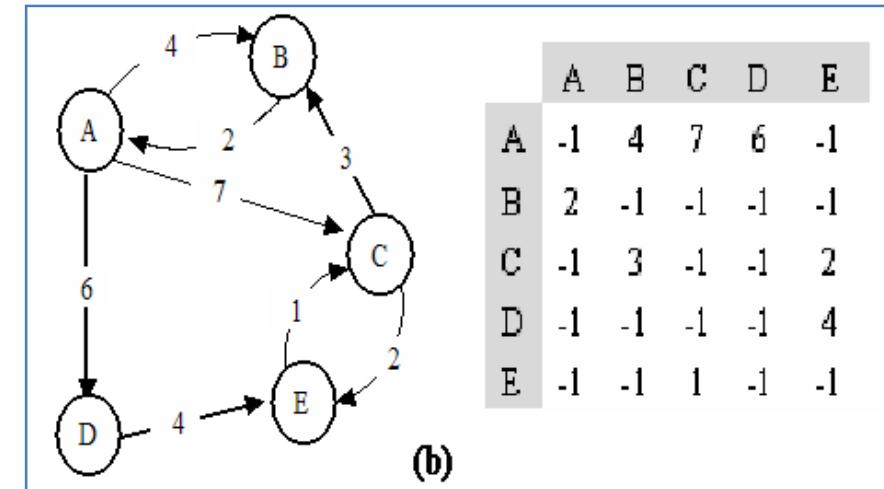
- **Matriz de adyacencias**
- **Listas de adyacencias**

Matriz de adyacencias

- Dado un grafo G con n vértices en V , la matriz de adyacencia para G es una matriz de dimensión nxn donde $a[i,j]$ tiene un valor que indica si existe o no el arco (i,j) .
- Se puede extender fácilmente para el caso de arcos etiquetados



(a)

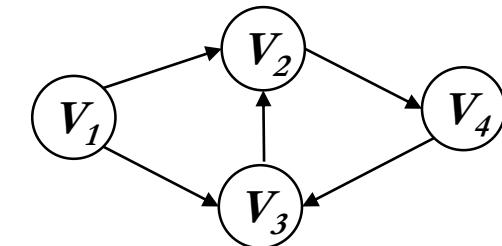
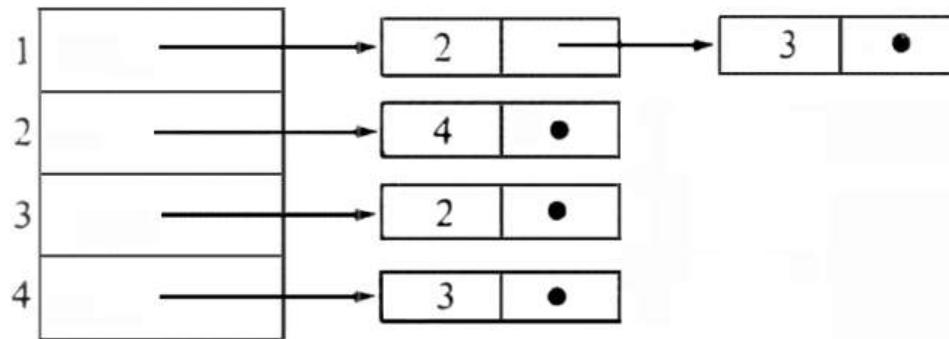


(b)

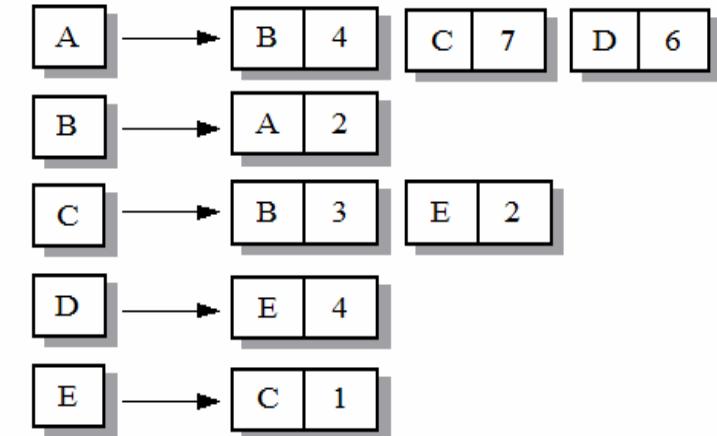
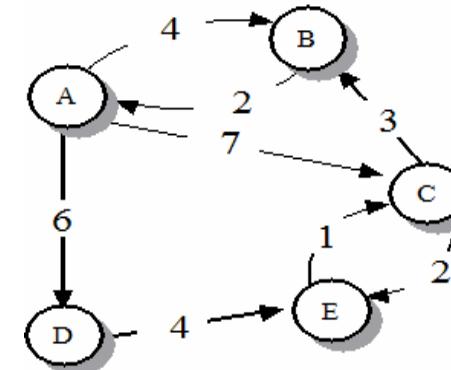
- Requiere un espacio de $O(n^2)$ aunque el grafo tenga menos de n^2 arcos.

Listas de adyacencias

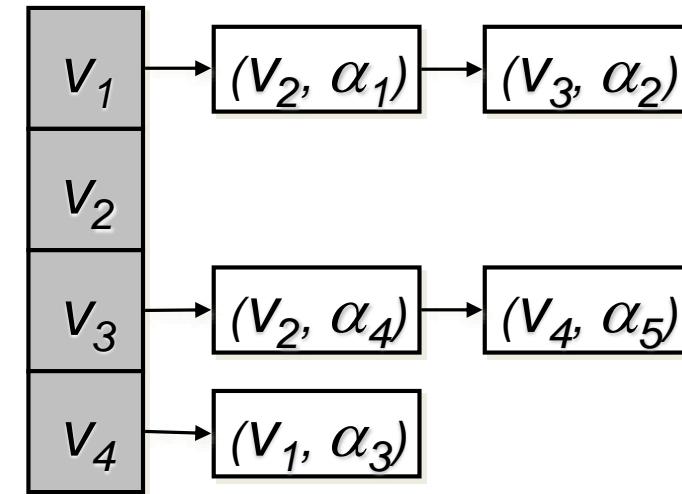
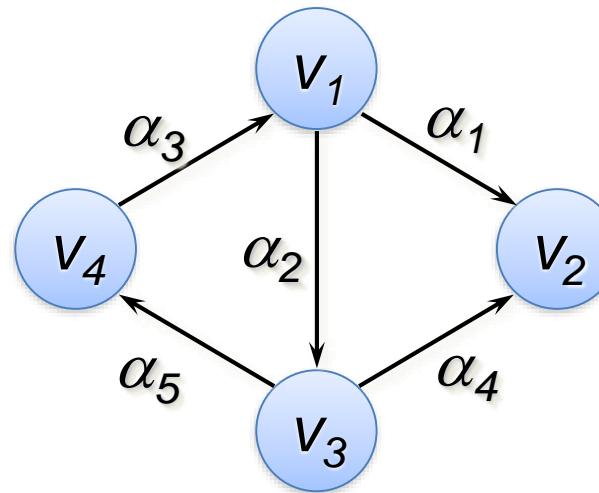
- Dado un grafo G con n vértices en V , la lista de adyacencia para un vértice i es una lista, en algún orden, de todos los vértices adyacentes a i .



- Cada vértice de la lista puede llevar asociado el peso del arco asociado



Listas de adyacencias



```
Vector<List<Integer>> data; // sin pesos
```

```
Vector<Map<Integer, A>> data; // con pesos
```

**Requiere numerar
los vértices**

```
Map<V, Map<V, A>> data;
```

```
SortedMap<V, SortedMap<V, A>> data;
```

**No requiere numerar
los vértices**



Recorridos de grafos

Muchos problemas sobre grafos requieren recorrer todos los vértices y arcos del mismo de forma sistemática. Para ello se utilizan dos formas de búsqueda:

- La búsqueda en profundidad

Generalización del recorrido en preorden en un árbol

- La búsqueda en anchura

Generalización del recorrido por niveles en un árbol





Búsqueda en profundidad

i) Recorrido en profundidad (grafo)

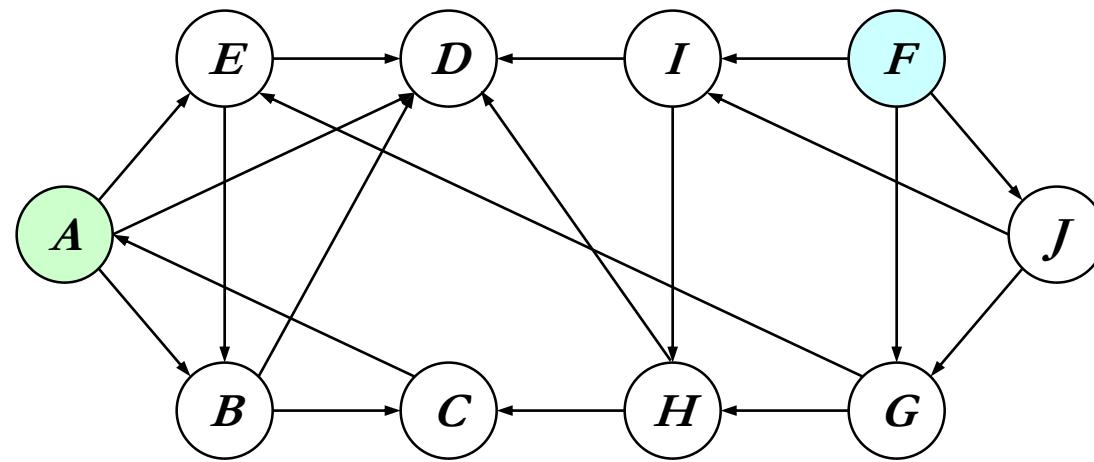
```
acción REP (g:Grafo)
inicio
  para cada vértice v de g hacer
    desmarcar(v)
    predecesor(v, NULO)
  fpara;
  para cada vértice v de g hacer
    si ¬ marcado(v)
    entonces
      BPF(v)
    fsi
  fpara
facción
```

ii) Búsqueda en profundidad (nodo inicial)

```
acción BPF (v:Vértice)
var w:Vértice
inicio
  marcar(v)
  para cada vértice w adyacente a v hacer
    si ¬ marcado(w)
    entonces
      predecesor(w, v)
      BPF(w)
    fsi
  fpara
facción
```

O($\max(n,a)$): **n** es el número de vértices o nodos y **a** es el número de arcos

Ej: Búsqueda en profundidad



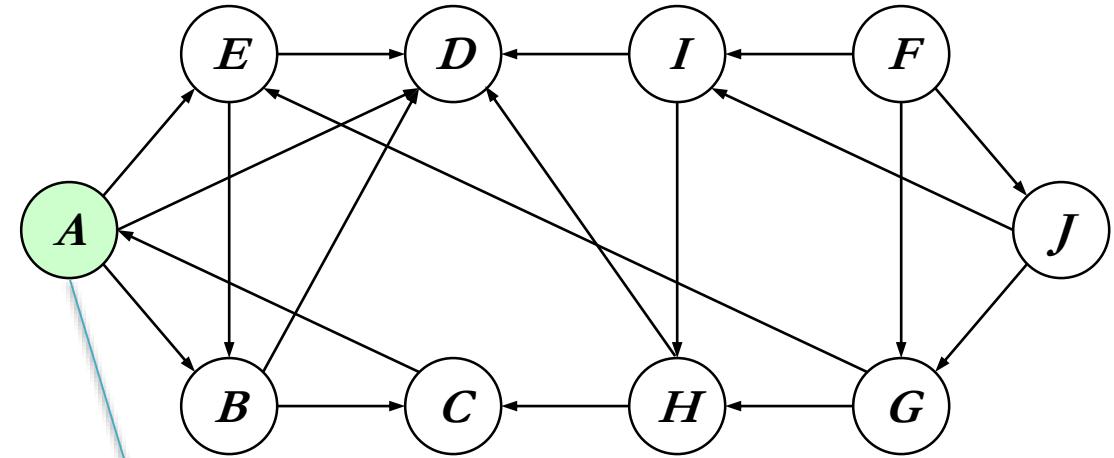
A, B, C, D, E + F, G, H, I, J

Búsqueda en profundidad (i)

```
acción REP (g:Grafo)
para cada vértice v de g hacer
    desmarcar(v)
    predecesor(v, NULO)
fpara
para cada vértice v de g hacer
    si  $\neg$  marcado(v)
    entonces
        BPF(v)
    fsi
fpara
facción
```

Para ir construyendo los árboles que forman el bosque de extensión correspondiente

REP escoge un vértice no marcado para empezar y llama al algoritmo BEP sobre este vértice



Marcados (recorridos):

()

El orden que sigue REP para escoger los vértices es el orden alfabético de las etiquetas (para este ejemplo)

Búsqueda en profundidad (ii)

acción BPF (v:Vértice)

var w:Vértice

marcar(v)

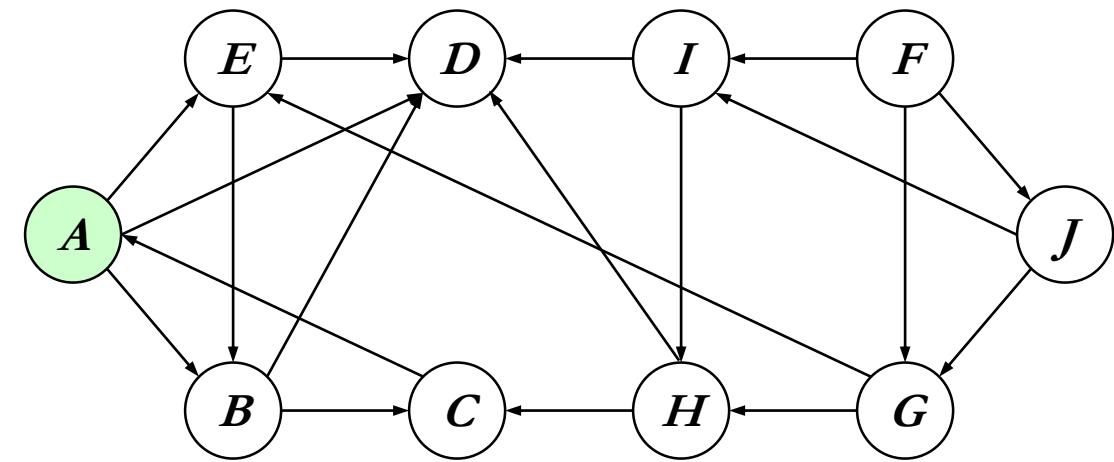
para cada vértice w adyacente a v

si \neg marcado(w)

entonces

predecesor(w, v)

BPF(w)



ady(A)={B, D, E}

Se escoge un vértice adyacente **no marcado**
y se llama recursivamente a BPF sobre este
vértice → B

Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

El orden para escoger los adyacentes es el alfabético en este ejemplo.
Pero podría usarse otro, tal como el coste de los arcos (si hubiera)

Búsqueda en profundidad (iii)

acción BPF (v:Vértice)

var w:Vértice

marcar(v)

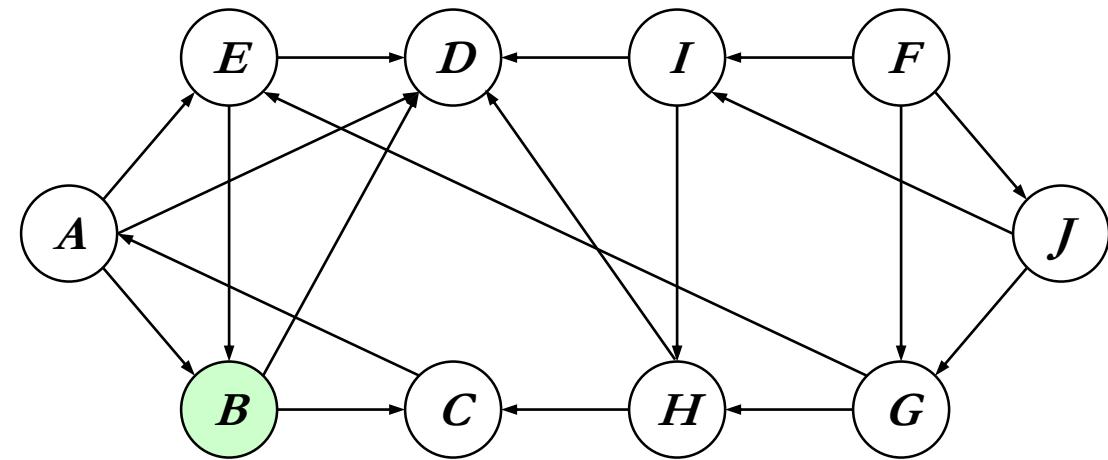
para cada vértice w adyacente a v

si \neg marcado(w)

entonces

predecesor(w, v)

BPF(w)

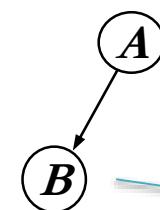


ady(B)={C, D}

Se escoge un vértice adyacente **no marcado** y se llama recursivamente a BPF sobre este vértice → C

Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)



Esto se hace con la orden:
predecesor("B","A")

Búsqueda en profundidad (iv)

acción BPF (v:Vértice)

var w:Vértice

marcar(v)

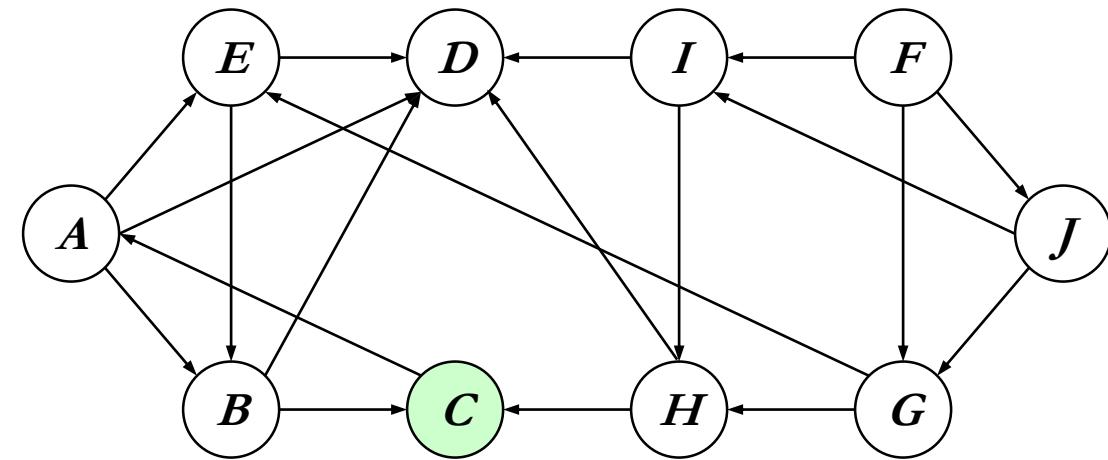
para cada vértice w adyacente a v

si \neg marcado(w)

entonces

predecesor(w, v)

BPF(w)



ady(C)={A}

Como no hay vértices adyacentes sin marcar,
el algoritmo termina

Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

Aquí iría construyéndose
el árbol de extensión tal
como hicimos antes

Búsqueda en profundidad (v)

acción BPF (v:Vértice)

var w:Vértice

marcar(v)

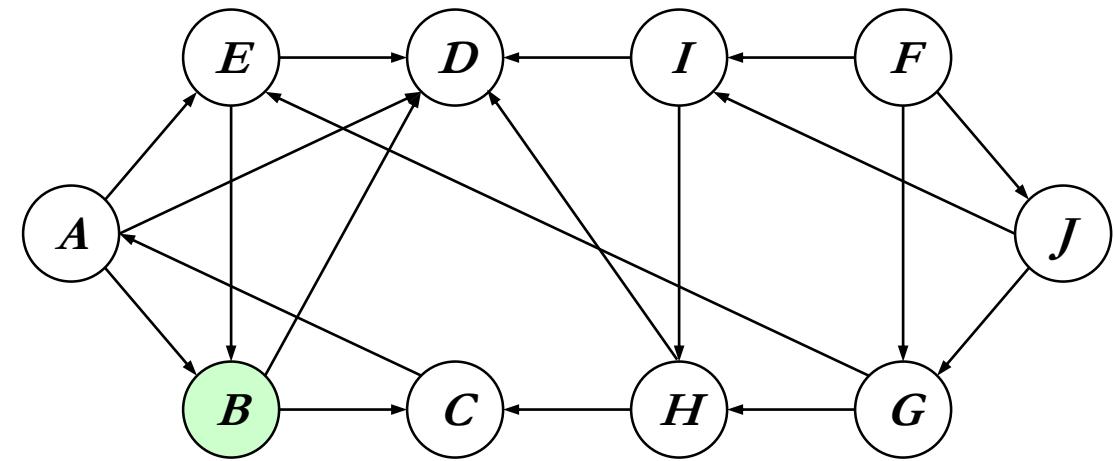
para cada vértice w adyacente a v

si \neg marcado(w)

entonces

predecesor(w, v)

BPF(w)



ady(B)={C, D}

Se escoge un vértice adyacente **no marcado**
y se llama recursivamente a BPF sobre este
vértice → D

Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

Búsqueda en profundidad (vi)

acción BPF (v:Vértice)

var w:Vértice

marcar(v)

para cada vértice w adyacente a v

si \neg marcado(w)

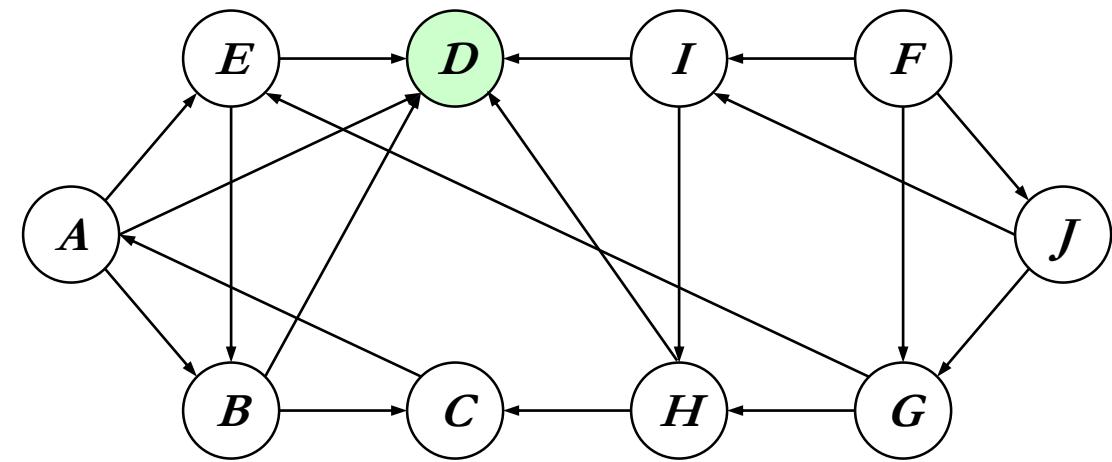
entonces

 predecesor(w, v)

 BPF(w)

ady(D)={ }

No tiene adyacentes. El algoritmo termina



Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

Búsqueda en profundidad (vii)

acción BPF (v:Vértice)

var w:Vértice

marcar(v)

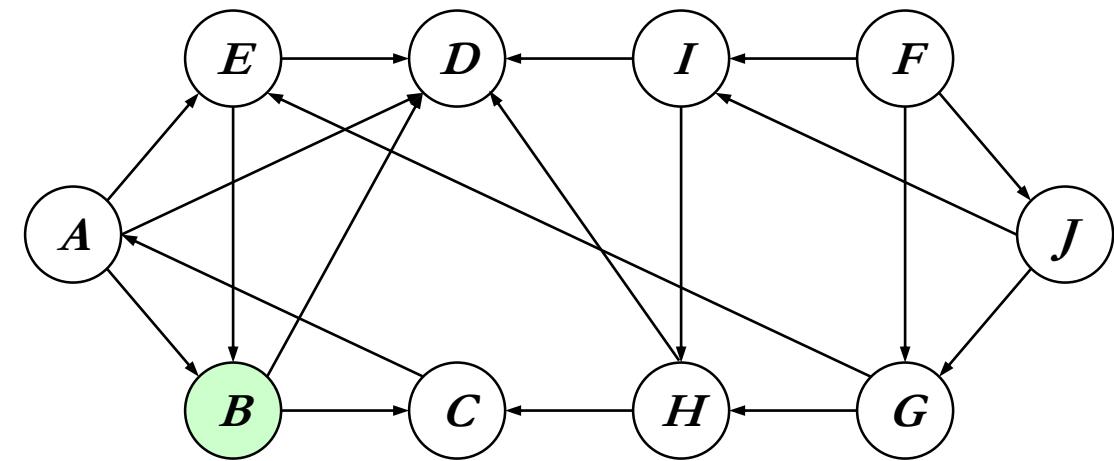
para cada vértice w adyacente a v

si \neg marcado(w)

entonces

predecesor(w, v)

BPF(w)



ady(B)={C, D}

Como no hay vértices más adyacentes sin marcar, el algoritmo termina

Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

Búsqueda en profundidad (viii)

acción BPF (v:Vértice)

var w:Vértice

marcar(v)

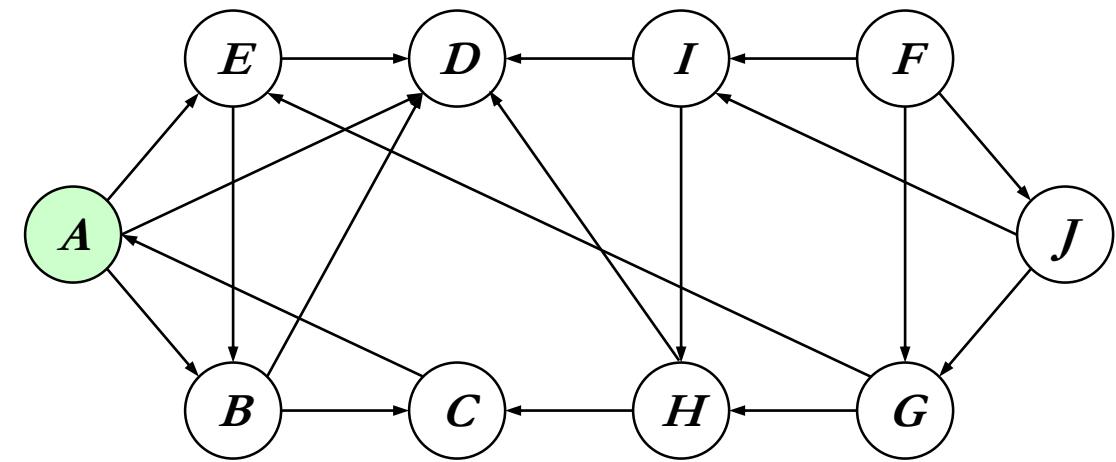
para cada vértice w adyacente a v

si \neg marcado(w)

entonces

predecesor(w, v)

BPF(w)



ady(A)={B, D, E}

Se escoge un vértice adyacente **no marcado**
y se llama recursivamente a BPF sobre este
vértice → E

Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

Búsqueda en profundidad (ix)

acción BPF (v:Vértice)

var w:Vértice

marcar(v)

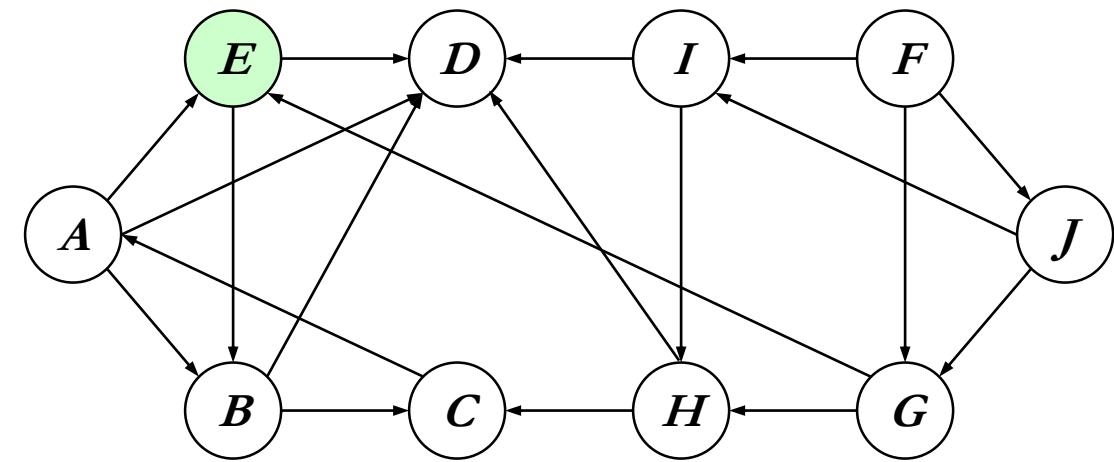
para cada vértice w adyacente a v

si \neg marcado(w)

entonces

predecesor(w, v)

BPF(w)



ady(E)={B, D}

Como no hay más vértices adyacentes sin marcar, el algoritmo termina

Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

Búsqueda en profundidad (x)

acción BPF (v:Vértice)

var w:Vértice

marcar(v)

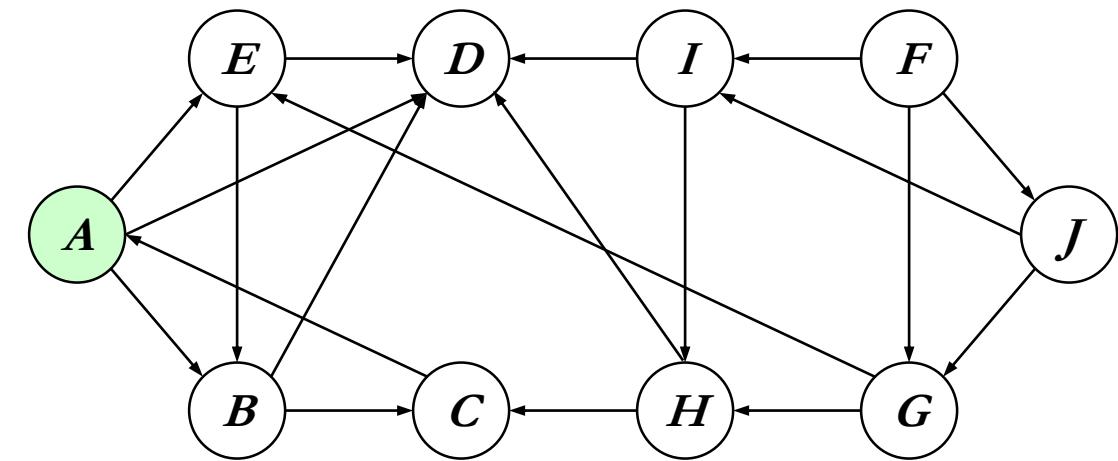
para cada vértice w adyacente a v

si \neg marcado(w)

entonces

predecesor(w, v)

BPF(w)



ady(A)={B, D, E}

Como no hay más vértices adyacentes sin marcar, el algoritmo termina

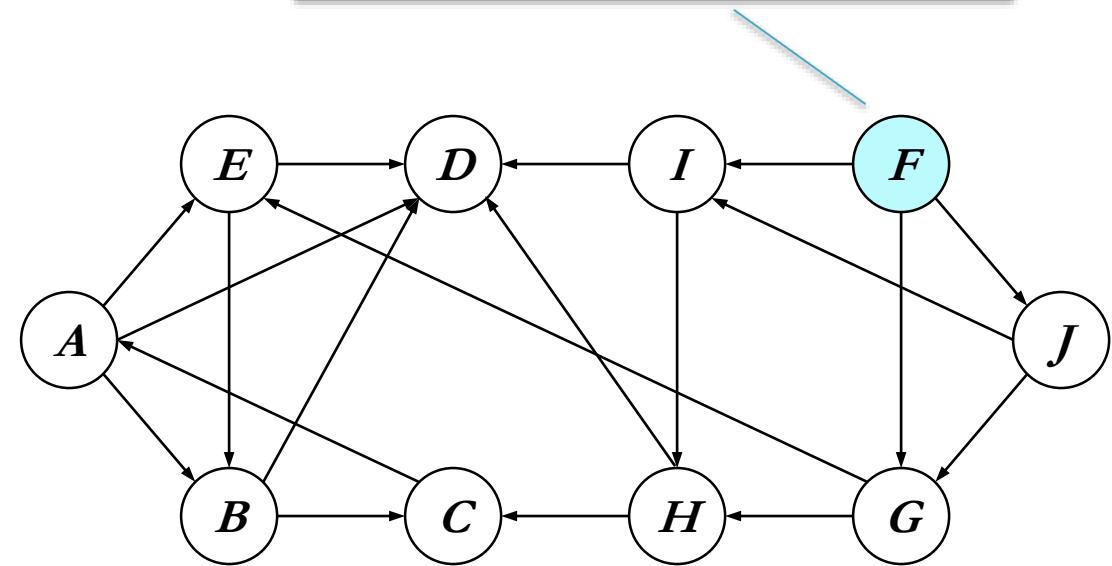
Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

Búsqueda en profundidad (xi)

```
acción REP (g:Grafo)
para cada vértice v de g hacer
    desmarcar(v)
    predecesor(v, NULO)
fpara
para cada vértice v de g hacer
    si ¬ marcado(v)
    entonces
        BPF(v)
    fsi
fpara
facción
```

REP escoge un vértice no marcado para empezar y llama al algoritmo BEP sobre este vértice



Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

(sigue)



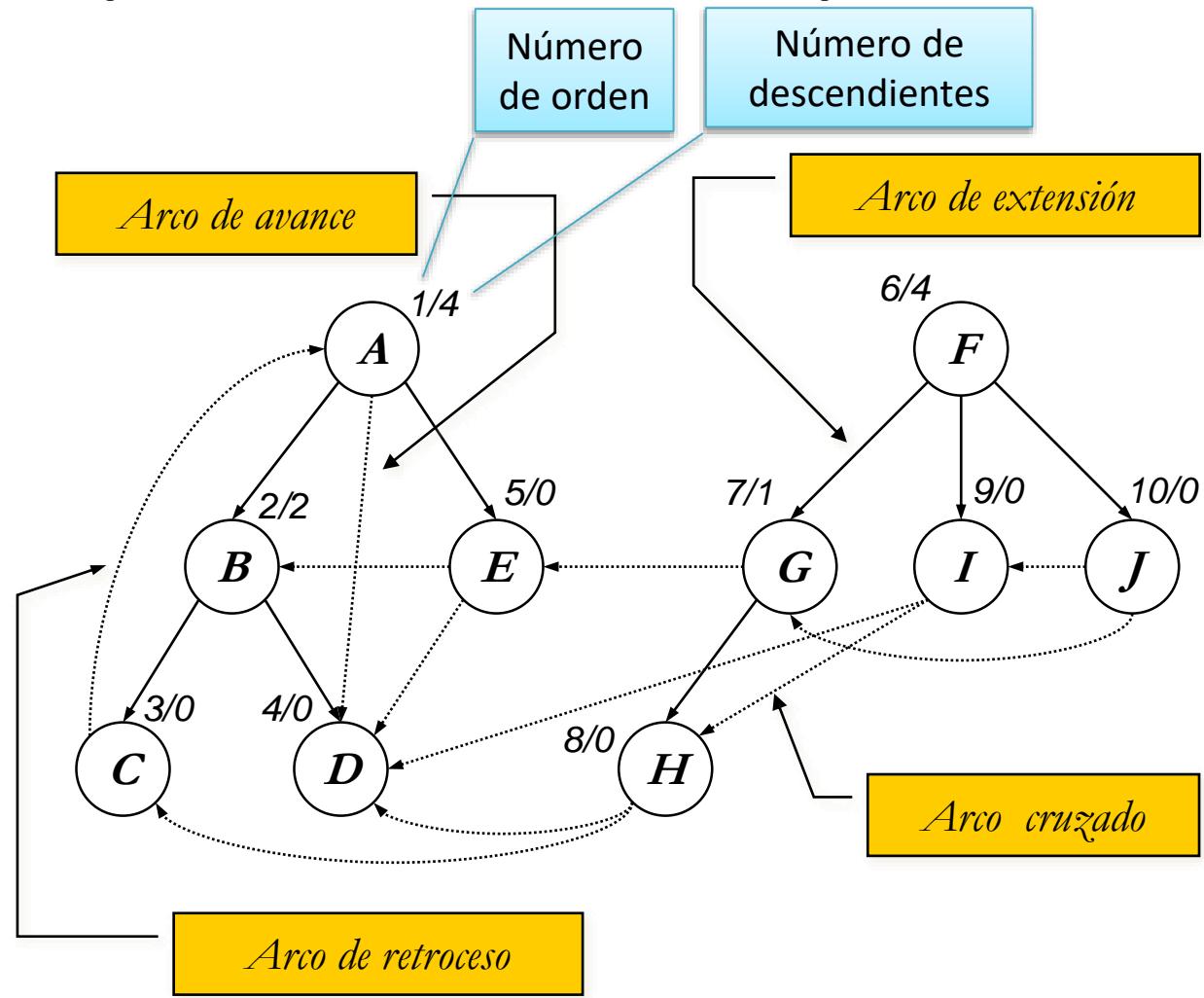
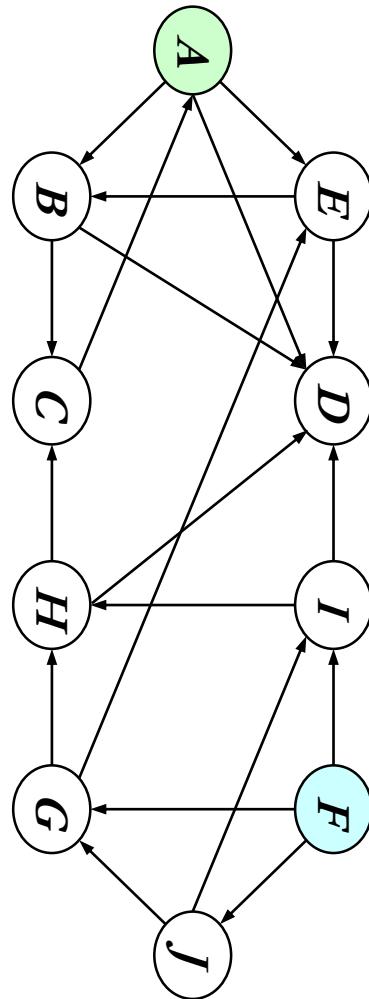
Bosque de extensión profundidad

El recorrido de un digrafo basado en la búsqueda en profundidad, define un bosque (conjunto de árboles ordenados) que se denomina **bosque de extensión en profundidad**.

- Los arcos de los árboles son los arcos que llevan a vértices no visitados, arcos que se denominan **arcos de extensión**.
- Los **arcos de avance** son arcos de no extensión que van desde un vértice a un descendiente propio en el árbol .
- Los **arcos de retroceso** son arcos que van desde un vértice a uno de sus antecesores en el árbol.
- Los arcos que no son de extensión, avance, o retroceso son **arcos cruzados**.



Ej: Bosque de extensión prof.





Clasificación de un arco (u,v) en función del orden en que se visitan los vértices ($num(v)$) y de sus descendientes en el árbol de extensión ($desc(v)$):

- Arco de extensión o avance
 $num(u) < num(v) \leq num(u) + desc(u)$
- Arco de retroceso
 $num(v) < num(u) \leq num(v) + desc(v)$
- Arco cruzado
 $num(v) \leq num(v) + desc(v) < num(u)$

En un grafo no dirigido no existe distinción entre ejes de avance y ejes de retroceso, por lo que se denominarán de la última forma indicada. Además, no pueden existir ejes cruzados, como puede demostrarse fácilmente.

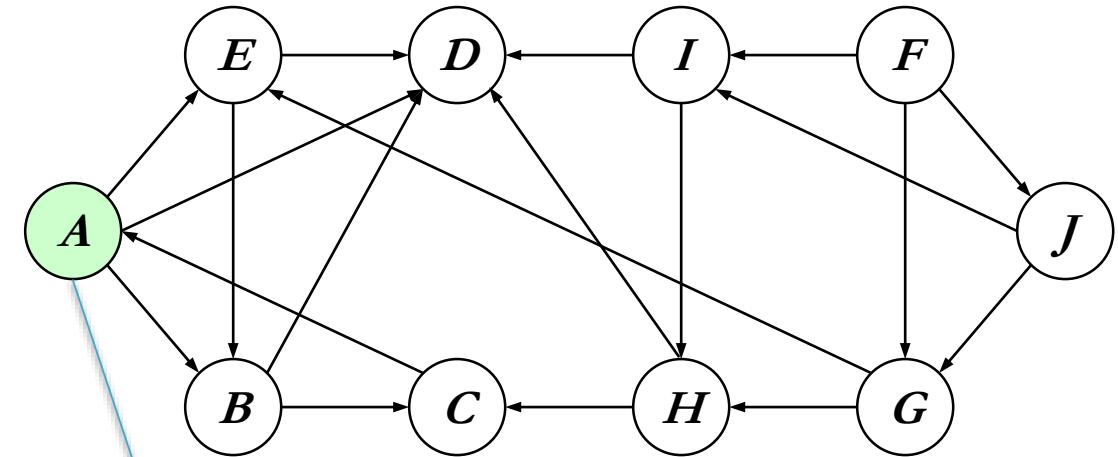


Bosque en profundidad (i)

```
acción REP (g:Grafo)
para cada vértice v de g hacer
    desmarcar(v)
    predecesor(v, NULO)
fpara
para cada vértice v de g hacer
    si  $\neg$  marcado(v)
    entonces
        BPF(v)
    fsi
fpara
facción
```

Para ir construyendo los árboles que forman el bosque de extensión correspondiente

REP escoge un vértice no marcado para empezar y llama al algoritmo BEP sobre este vértice



Marcados (recorridos):

()

Predecesores (árboles)

Ningún nodo tiene predecesor

Bosque en profundidad (ii)

acción BPF (v:Vértice)

var w:Vértice

marcar(v)

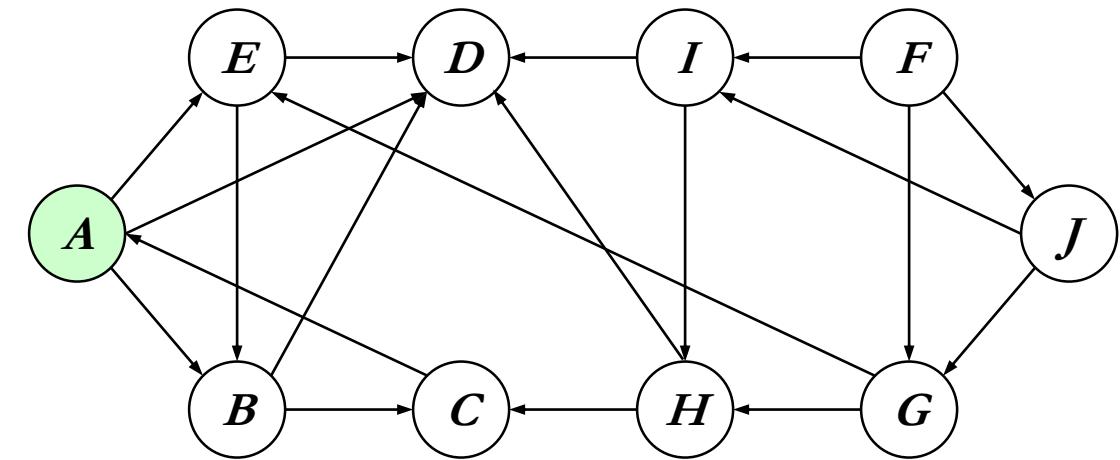
para cada vértice w adyacente a v

si \neg marcado(w)

entonces

predecesor(w, v)

BPF(w)



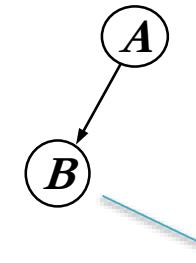
ady(A)={B, D, E}

Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

Predecesores (árboles)

((B,A))



Esto se hace con la orden:
predecesor("B","A")

Bosque en profundidad (iii)

acción BPF (v:Vértice)

var w:Vértice

marcar(v)

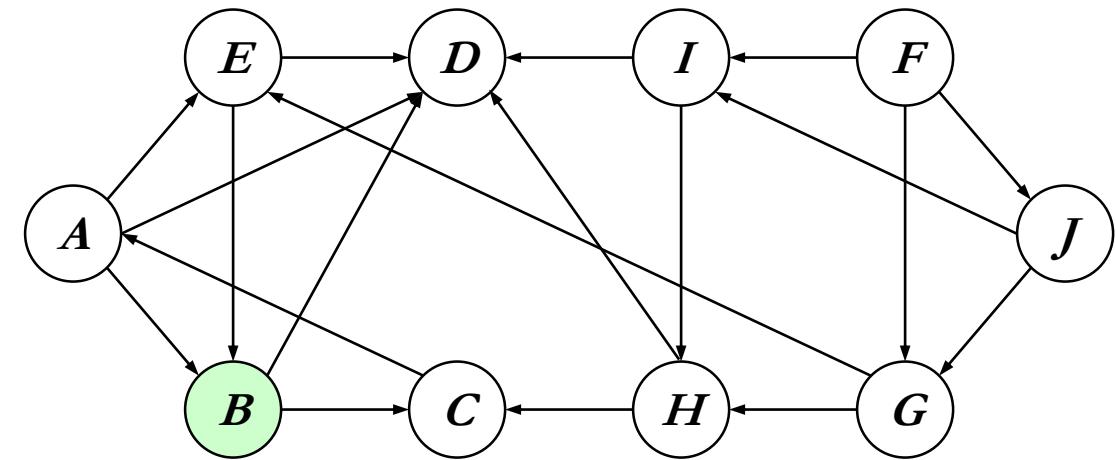
para cada vértice w adyacente a v

si \neg marcado(w)

entonces

predecesor(w, v)

BPF(w)



ady(B)={C, D}

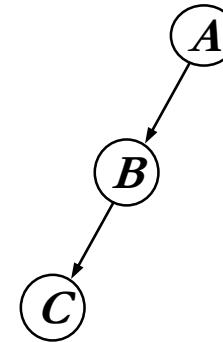


Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

Predecesores (árboles)

((B,A) (C,B))



Bosque en profundidad (iv)

acción BPF (v:Vértice)

var w:Vértice

marcar(v)

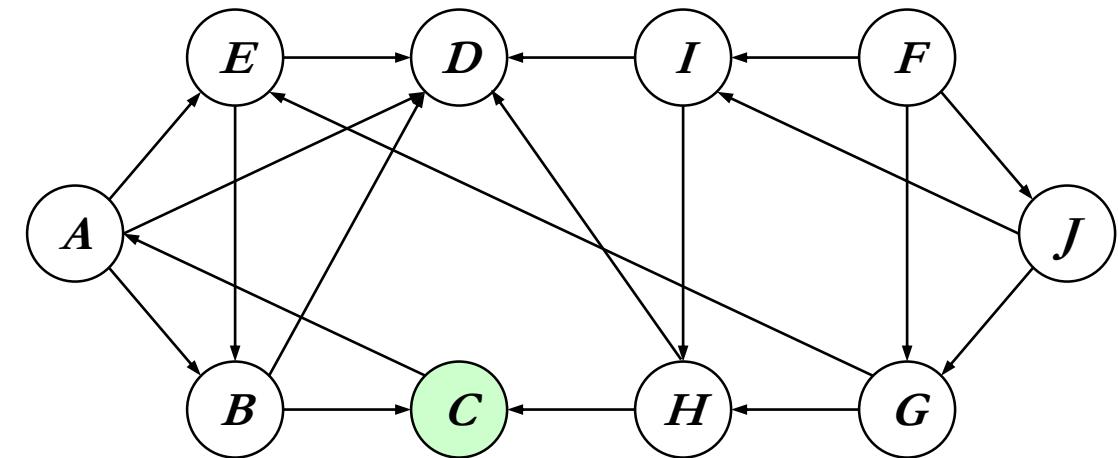
para cada vértice w adyacente a v

si \neg marcado(w)

entonces

predecesor(w, v)

BPF(w)



ady(C)={A}

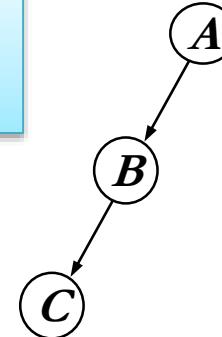
No hay vértices adyacentes sin marcar. El árbol no cambia

Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

Predecesores (árboles)

((B,A) (C,B))



Bosque en profundidad (v)

acción BPF (v:Vértice)

var w:Vértice

marcar(v)

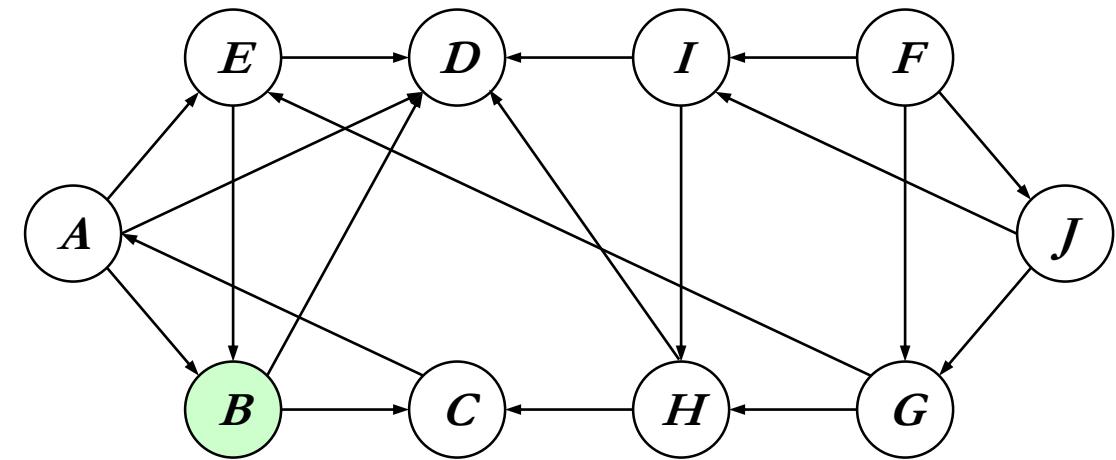
para cada vértice w adyacente a v

si \neg marcado(w)

entonces

predecesor(w, v)

BPF(w)



ady(B)={C, D}

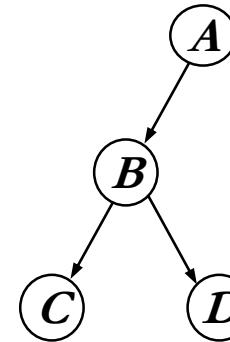


Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

Predecesores (árboles)

((B,A) (C,B) (D,B))



Bosque en profundidad (vi)

acción BPF (v:Vértice)

var w:Vértice

marcar(v)

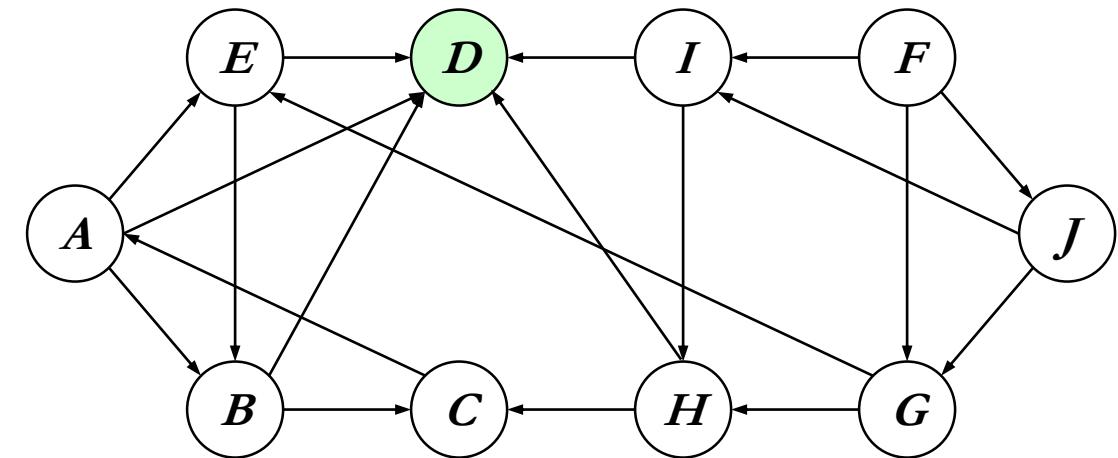
para cada vértice w adyacente a v

si \neg marcado(w)

entonces

predecesor(w, v)

BPF(w)



ady(D)={ }

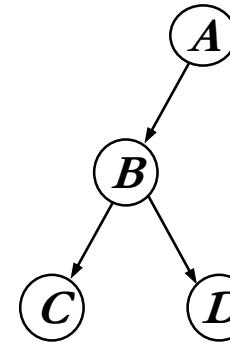
No tiene
adyacentes

Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

Predecesores (árboles)

((B,A) (C,B) (D,B))



Bosque en profundidad (vii)

acción BPF (v:Vértice)

var w:Vértice

marcar(v)

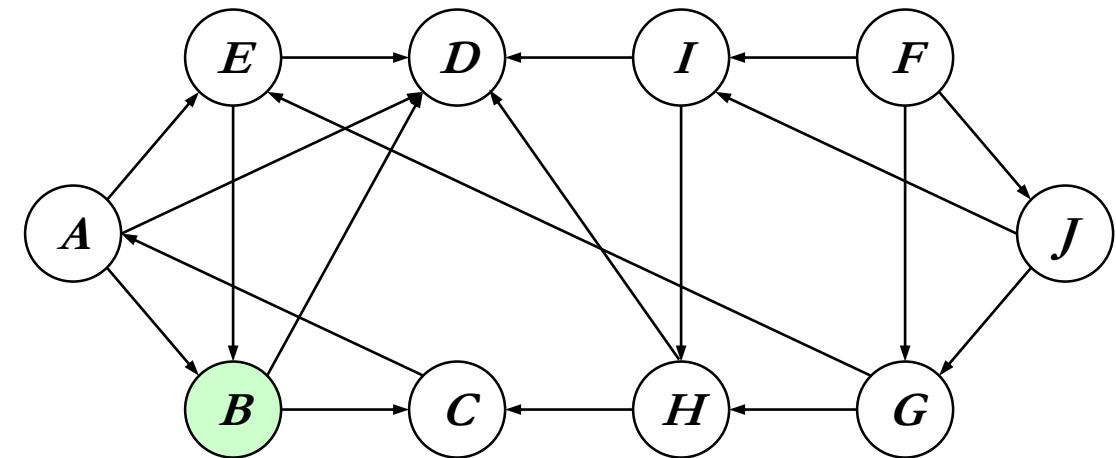
para cada vértice w adyacente a v

si \neg marcado(w)

entonces

predecesor(w, v)

BPF(w)



ady(B)={C, D}

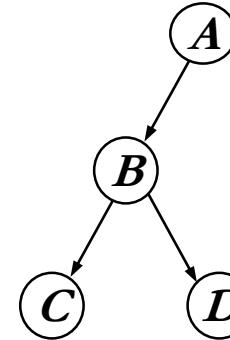
No tiene más
adyacentes

Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

Predecesores (árboles)

((B,A) (C,B) (D,B))



Bosque en profundidad (viii)

acción BPF (v:Vértice)

var w:Vértice

marcar(v)

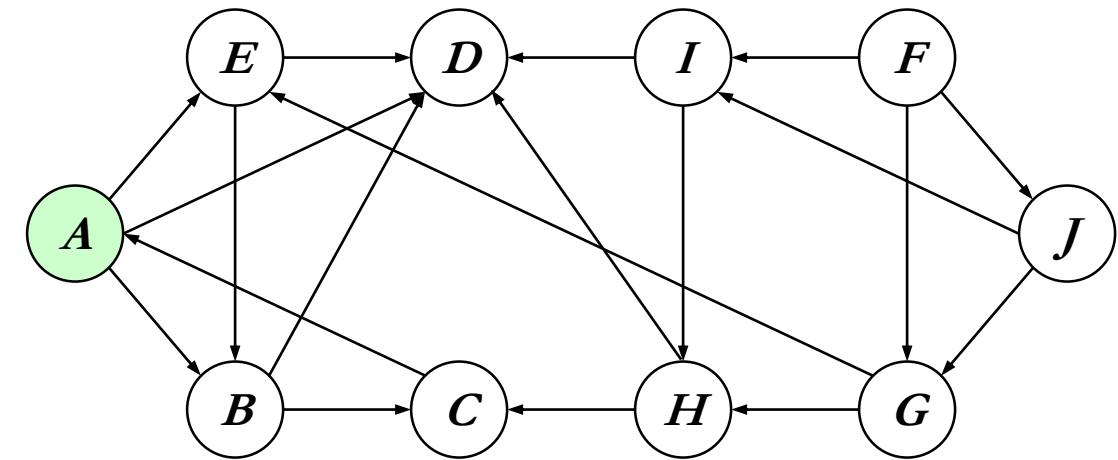
para cada vértice w adyacente a v

si \neg marcado(w)

entonces

predecesor(w, v)

BPF(w)



ady(A)={B, D, E}

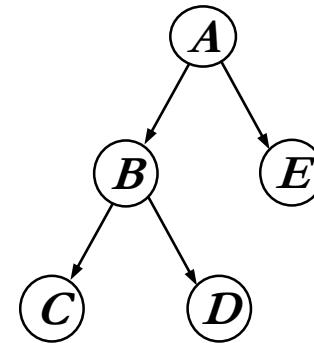


Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

Predecesores (árboles)

((B,A) (C,B) (D,B) (E,A))



Bosque en profundidad (ix)

acción BPF (v:Vértice)

var w:Vértice

marcar(v)

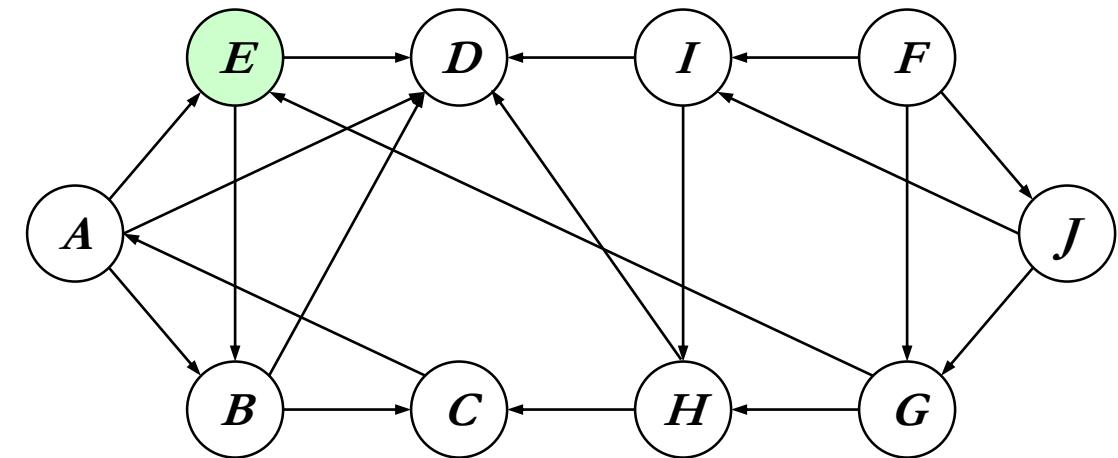
para cada vértice w adyacente a v

si \neg marcado(w)

entonces

predecesor(w, v)

BPF(w)



ady(E)={B, D}

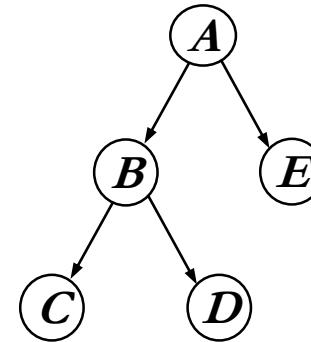
No quedan
adyacentes

Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

Predecesores (árboles)

((B,A) (C,B) (D,B) (E,A))



Bosque en profundidad (x)

acción BPF (v:Vértice)

var w:Vértice

marcar(v)

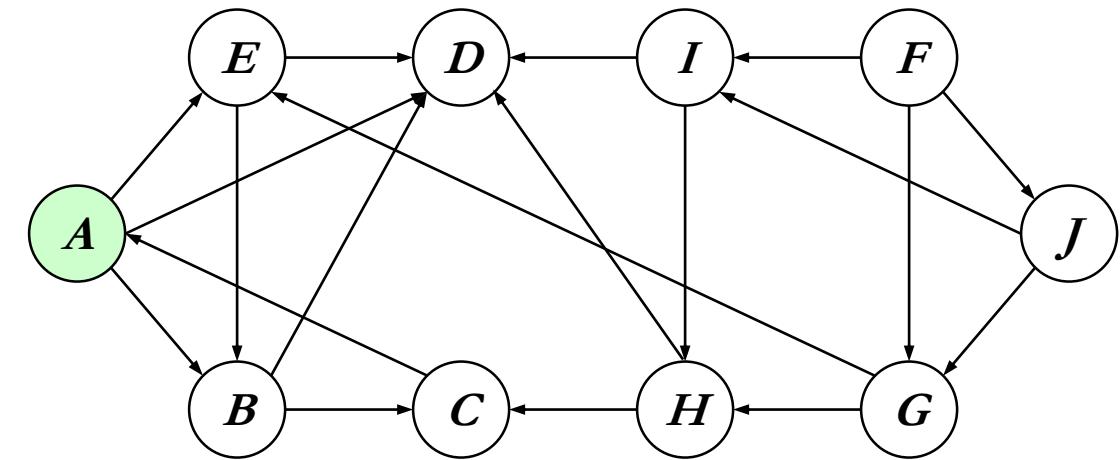
para cada vértice w adyacente a v

si \neg marcado(w)

entonces

predecesor(w, v)

BPF(w)



ady(A)={B, D, E}

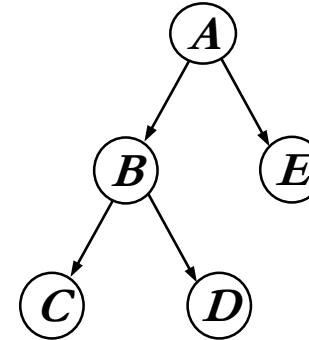
No quedan
adyacentes

Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

Predecesores (árboles)

((B,A) (C,B) (D,B) (E,A))

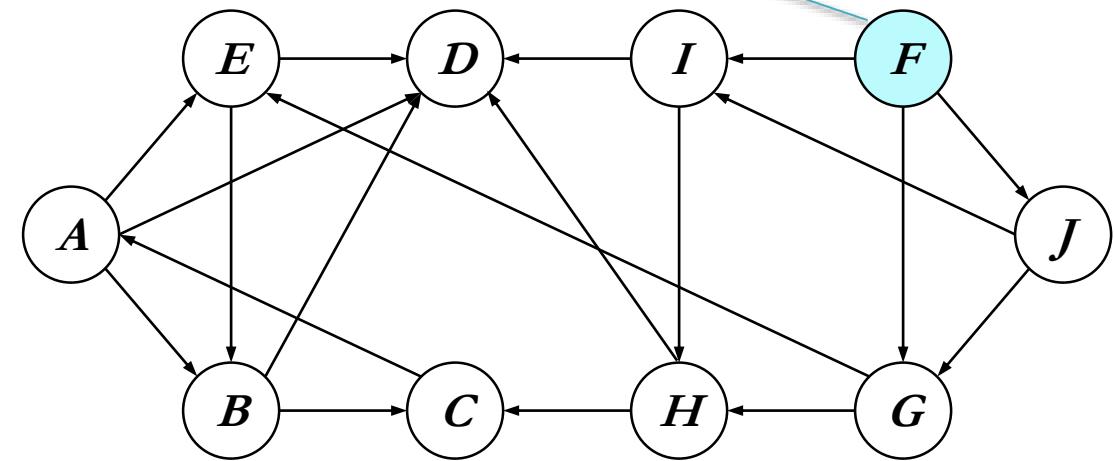


Bosque en profundidad (xi)

```

acción REP (g:Grafo)
para cada vértice v de g hacer
    desmarcar(v)
    predecesor(v, NULO)
fpara
para cada vértice v de g hacer
    si ¬ marcado(v)
    entonces
        BPF(v)
    fsi
fpara
facción
    
```

REP escoge un vértice no marcado

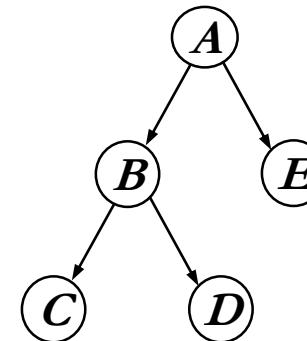


Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

Predecesores (árboles)

((B,A) (C,B) (D,B) (E,A))



Bosque en profundidad (xii)

acción BPF (v:Vértice)

var w:Vértice

marcar(v)

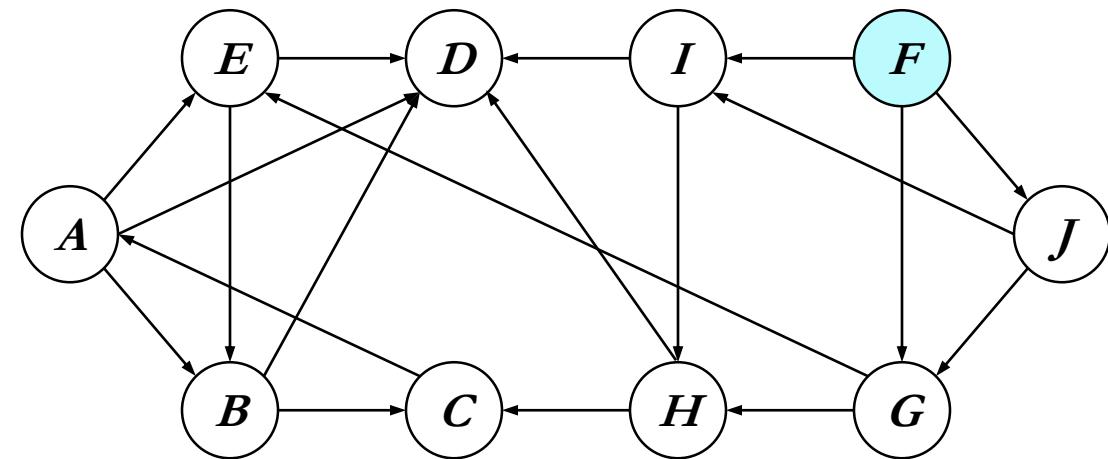
para cada vértice w adyacente a v

si \neg marcado(w)

entonces

predecesor(w, v)

BPF(w)



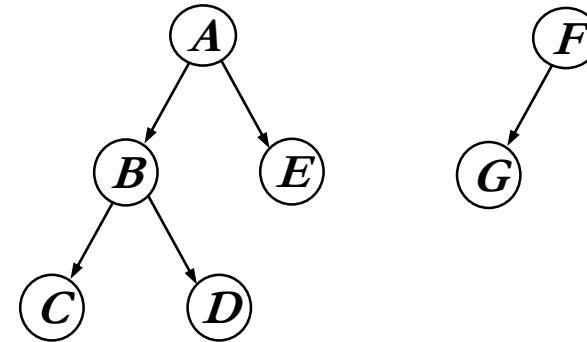
ady(F) = {G, I, J}

Marcados (recorridos):

(A, B, C, D, E + F, G, H, I, J)

Predecesores (árboles)

((B,A) (C,B) (D,B) (E,A) (G,F))



(sigue)



Búsqueda en anchura

i) Recorrido en anchura (grafo)

acción REA (g:Grafo)

inicio

para cada vértice v de g **hacer**

 desmarcar(v)

 predecesor(v, NULO)

fpara;

para cada vértice v de g **hacer**

si \neg marcado(v)

entonces

 BEA(v)

fsi

fpara

facción

O(máx(n,a))

ii) Búsqueda en anchura (nodo inicial)

acción BEA (v:Vértice)

var v, w, u:Vértice

 q:ColaNodos

inicio

 añadir(q,v); marcar(v)

mientras \neg vacía(q) **hacer**

 u \leftarrow borrar(q)

para cada vértice w adyacente a u **hacer**

si \neg marcado(w)

entonces

 marcar(w)

 predecesor(w, u)

 añadir(q, w)

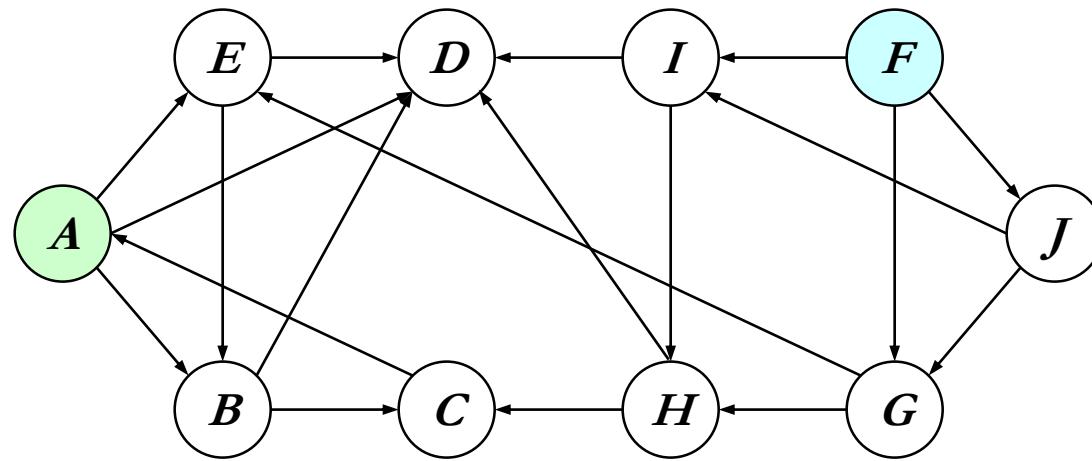
fsi

fpara

fmientras
facción



Ej: Búsqueda en anchura



A, B, D, E, C + F, G, I, J, H

Búsqueda en anchura (i)

acción REA (g:Grafo)

inicio

para cada vértice v de g hacer
desmarcar(v)
predecesor(v, NULO)

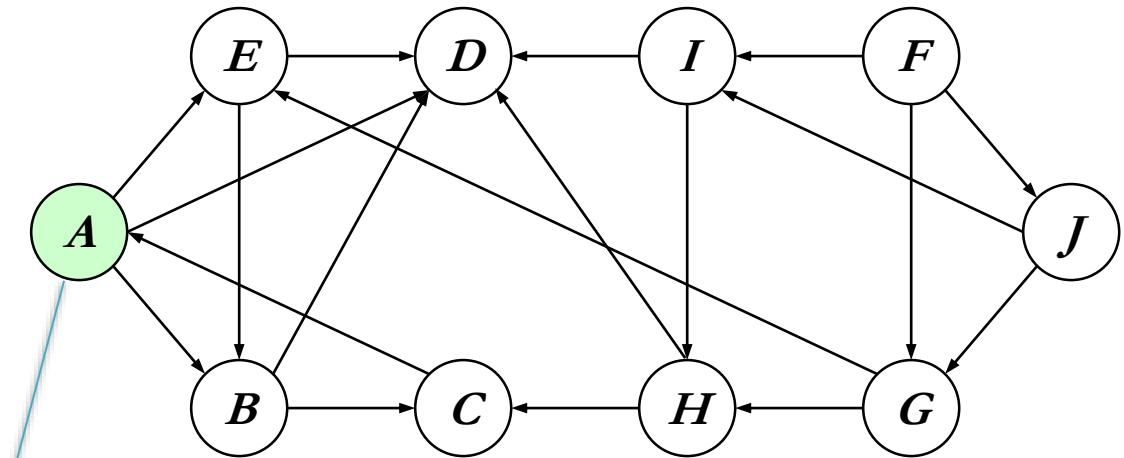
fpara;

para cada vértice v de g hacer
si \neg marcado(v)
entonces
BEA(v)

fsi

fpara
facción

REA escoge un vértice
no marcado para
empezar y llama al
algoritmo BEA sobre
este vértice



Marcados (recorridos):

()

El orden que sigue REA para escoger los vértices es el
orden alfabético de las etiquetas (para este ejemplo)

Búsqueda en anchura (ii)

acción BEA (v:Vértice)

var v, w, u:Vértice

q:ColaNodos

añadir(q,v); marcar(v)

mientras \neg vacía(q) **hacer**

u \leftarrow borrar(q)

para cada vértice w adyacente a u

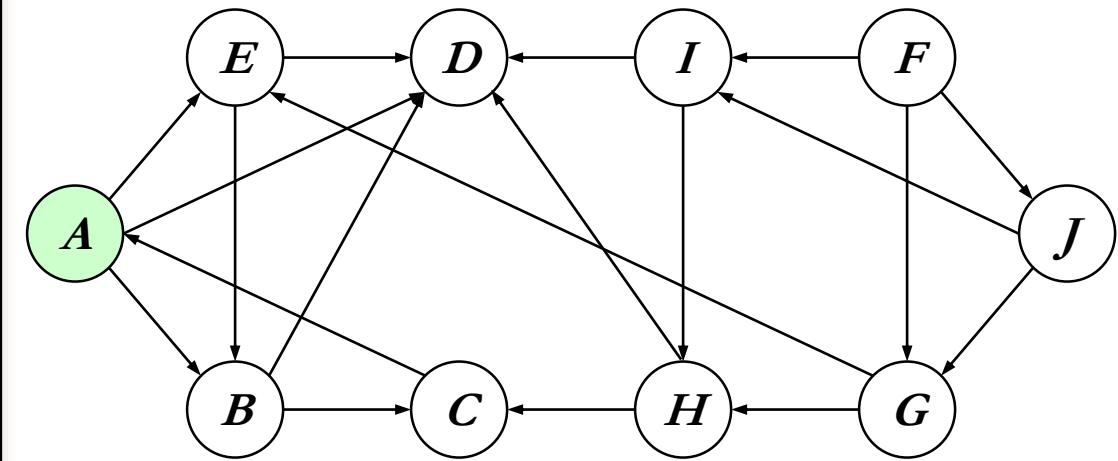
si \neg marcado(w)

entonces

marcar(w)

predecesor(w, u)

añadir(q, w)



Marcados (recorridos):

(A, B, D, E, C + F, G, I, J, H)

q=(A)

u=A adj(A)={B,D,E}

q=(B,D,E)

El orden para escoger los adyacentes es el alfabético en este ejemplo.
Pero podría usarse otro, tal como el coste de los arcos (si hubiera)

Búsqueda en anchura (ii)

acción BEA (v:Vértice)

var v, w, u:Vértice

q:ColaNodos

añadir(q,v); marcar(v)

mientras \neg vacía(q) **hacer**

u \leftarrow borrar(q)

para cada vértice w adyacente a u

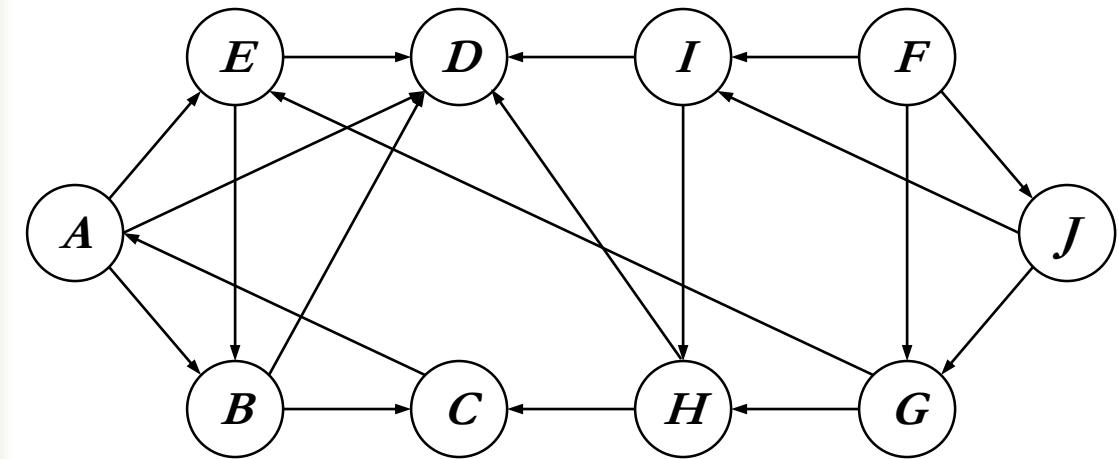
si \neg marcado(w)

entonces

marcar(w)

predecesor(w, u)

añadir(q, w)



Marcados (recorridos):

(A, B, D, E, C + F, G, I, J, H)

q=(B,D,E)

u=B adj(B)={C, D}

q=(D,E,C)

marcado

Búsqueda en anchura (iii)

acción BEA (v:Vértice)

var v, w, u:Vértice

q:ColaNodos

añadir(q,v); marcar(v)

mientras \neg vacía(q) **hacer**

u \leftarrow borrar(q)

para cada vértice w adyacente a u

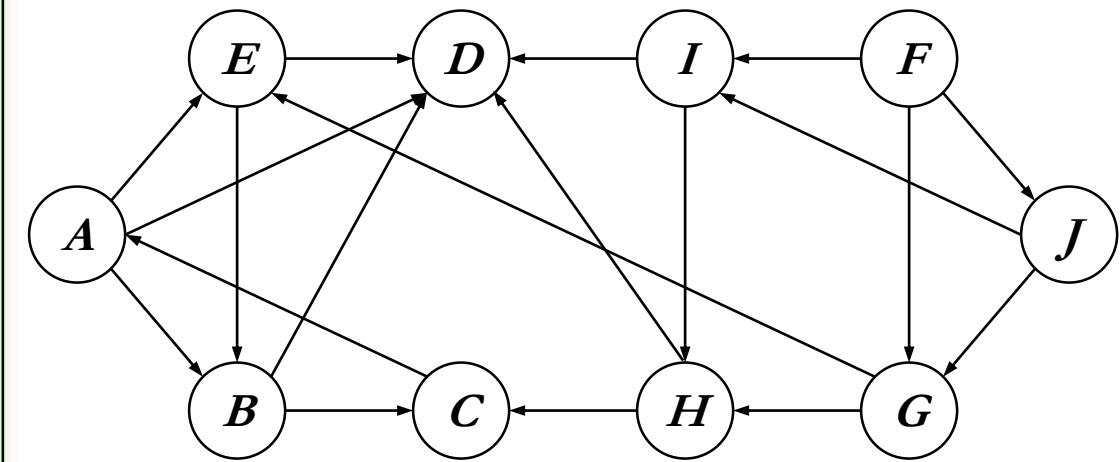
si \neg marcado(w)

entonces

marcar(w)

predecesor(w, u)

añadir(q, w)



Marcados (recorridos):

(A, B, D, E, C + F, G, I, J, H)

q=(D,E,C)

u=D adj(D)={ }

q=(E,C)

Búsqueda en anchura (iv)

acción BEA (v:Vértice)

var v, w, u:Vértice

q:ColaNodos

añadir(q,v); marcar(v)

mientras \neg vacía(q) **hacer**

u \leftarrow borrar(q)

para cada vértice w adyacente a u

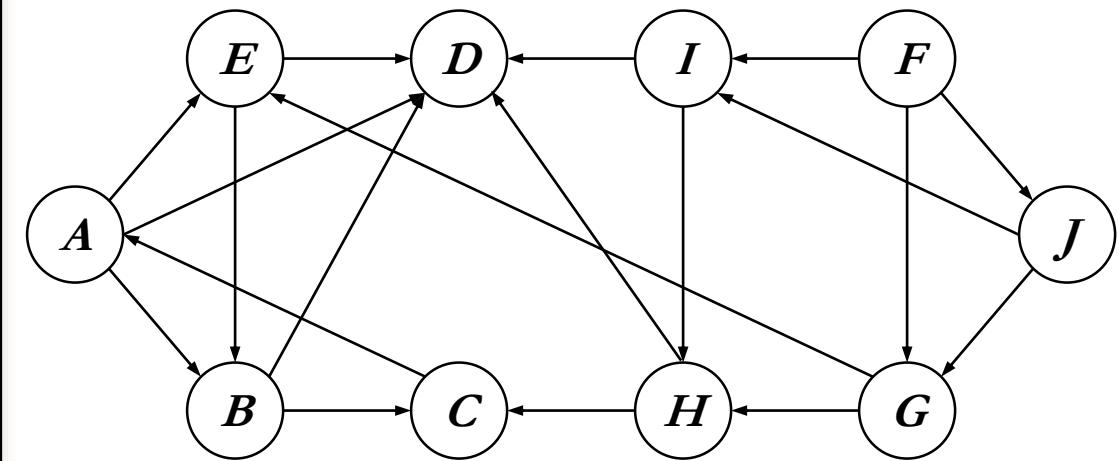
si \neg marcado(w)

entonces

marcar(w)

predecesor(w, u)

añadir(q, w)



Marcados (recorridos):

(A, B, D, E, C + F, G, I, J, H)

q=(E,C)

u=E adj(E)={ B,D}

q=(C)

marcados

Búsqueda en anchura (v)

acción BEA (v:Vértice)

var v, w, u:Vértice

q:ColaNodos

añadir(q,v); marcar(v);

mientras \neg vacía(q) **hacer**

u \leftarrow borrar(q)

para cada vértice w adyacente a u

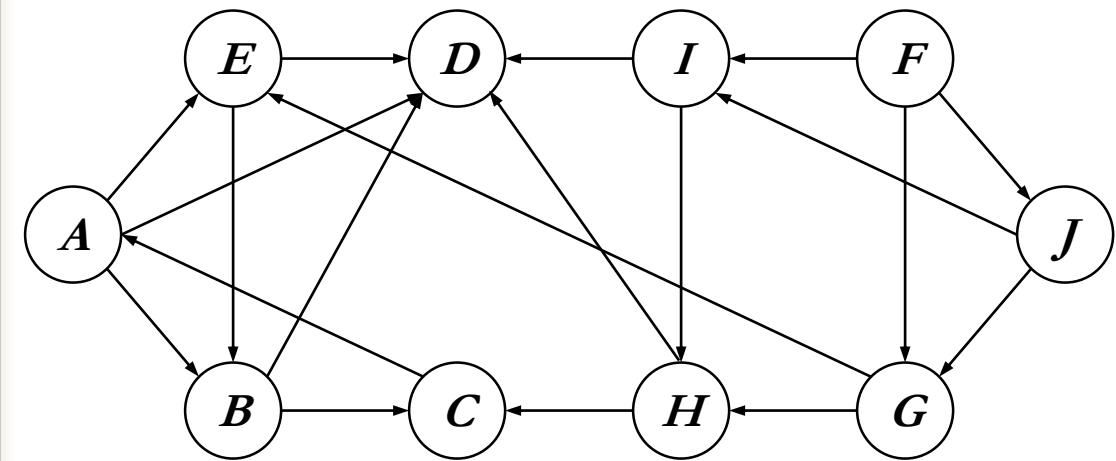
si \neg marcado(w)

entonces

marcar(w)

predecesor(w, u)

añadir(q, w)



Marcados (recorridos):

(A, B, D, E, C + F, G, I, J, H)

q=(C)

u=C adj(C)={ A}

q=()

Cola vacía. Se acaba el algoritmo BEA

Búsqueda en anchura (vi)

acción REA (g:Grafo)

inicio

para cada vértice v de g hacer

desmarcar(v)

predecesor(v, NULO)

fpara;

para cada vértice v de g hacer

si \neg marcado(v)

entonces

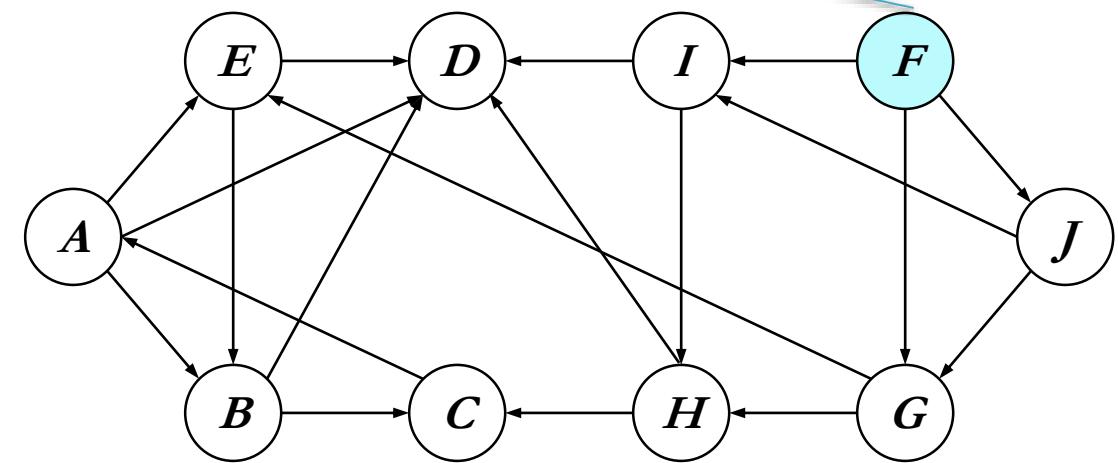
BEA(v)

fsi

fpara

facción

REA escoge otro vértice no marcado y se llama a BEA con él



Marcados (recorridos):

(A, B, D, E, C + F, G, I, J, H)

Búsqueda en anchura (vii)

acción BEA (v:Vértice)

var v, w, u:Vértice

q:ColaNodos

añadir(q,v); marcar(v)

mientras \neg vacía(q) **hacer**

u \leftarrow borrar(q)

para cada vértice w adyacente a u

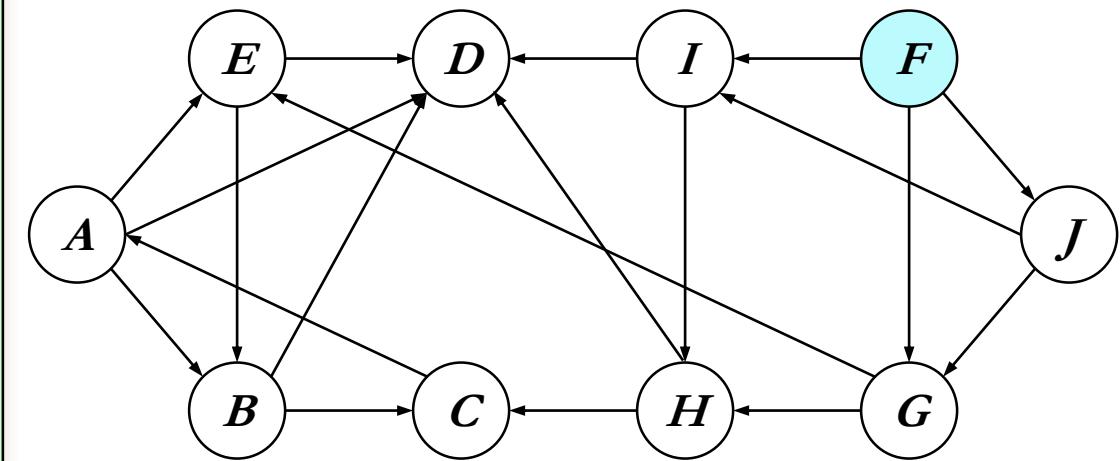
si \neg marcado(w)

entonces

marcar(w)

predecesor(w, u)

añadir(q, w)



Marcados (recorridos):

(A, B, D, E, C + F, G, I, J, H)

q=(F)

u=F adj(F)={G,I,J}

q=(G, I, J)

(sigue)



Bosque de extensión anchura

El recorrido en anchura de un digrafo, también establece una clasificación de los arcos del mismo en: *arcos de árbol o extensión*, *arcos de retroceso* y *arcos cruzados* (no pueden existir *arcos de avance*).

Clasificación de un arco (u,v) en función del nivel en que se encuentran en el árbol de extensión ($nivel(v)$):

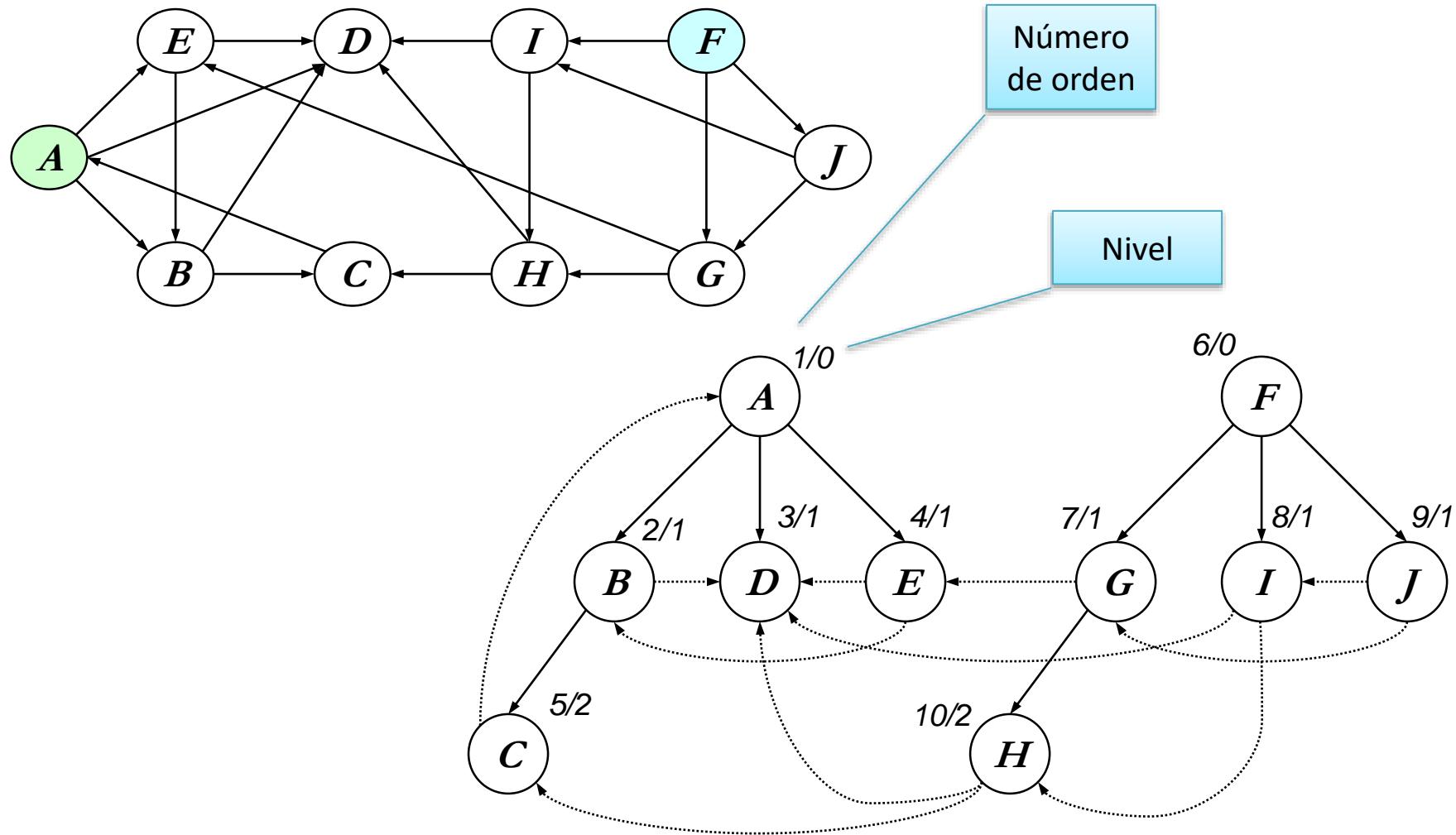
- Arco de extensión, $nivel(v) = nivel(u) + 1$
- Arco de retroceso, $0 \leq nivel(v) < nivel(u)$
- Arco cruzado, $nivel(v) \leq nivel(u) - 1$

En un grafo no dirigido no existe arcos de retroceso y, los arcos (u, v) del mismo cumplen las propiedades siguientes:

- Arco de extensión, $nivel(v) = nivel(u) + 1$
- Arco cruzado, $nivel(v) = nivel(u) - 1$



Ej: Bosque de extensión anch.





Implementación de grafos

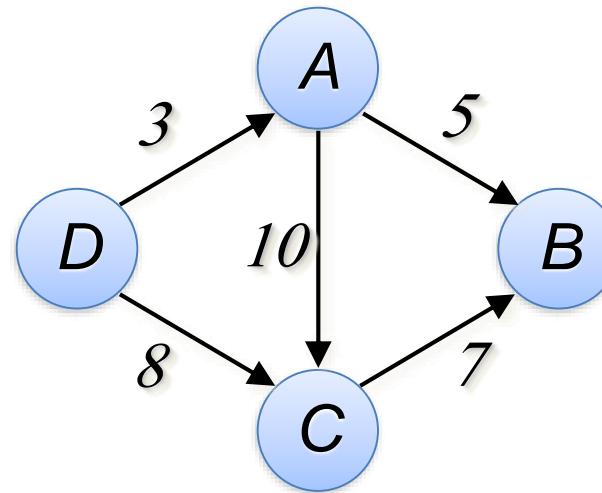
```
public class Graph<Vertex> {  
    private SortedMap<Vertex, SortedMap<Vertex, Double>> adjList;  
    private int numVertex;           número de vértices  
    private int numEdges;          número de arcos  
    ...  
}
```

- Los grafos son dirigidos y con los arcos etiquetados
- La implementación está basada en listas de adyacencias
- Se parametriza el tipo de los vértices y se ordena por ellos
- Los pesos de los ejes son de tipo “Double”



Área de datos

```
public class Graph<Vertex> {  
    private int numVertex;    número de vértices  
    private int numEdges;    número de arcos  
    private SortedMap<Vertex, SortedMap<Vertex, Double>> adjList;  
    ...  
}
```



Arrows point from the code's adjList structure to the graph's edges:

- A blue arrow points from the entry for vertex A to the edge from A to B.
- An orange arrow points from the entry for vertex B to the edge from B to C.
- An orange arrow points from the entry for vertex C to the edge from C to B.

(A, ((B, 5) (C, 10))
B, ()
C, ((B, 7))
D, ((A, 3) (C, 8)))



Constructores

```
private SortedMap<Vertex, SortedMap<Vertex, Double>> adjList;
private int numVertex;           número de vértices
private int numEdges;           número de arcos
```

```
public Graph() {
    adjList=new TreeMap<Vertex, SortedMap<Vertex,Double>> ();
    numVertex=0;
    numEdges=0;
}

public Graph(Graph<Vertex> g) {
    adjList=new TreeMap<Vertex, SortedMap<Vertex,Double>> (g.adjList);
    numVertex=g.numVertex;
    numEdges=g.numEdges;
}
```





Operaciones

```
public boolean hasVertex(Vertex v);
public boolean addVertex(Vertex v);
public boolean removeVertex(Vertex v);
public int degreeIn(Vertex v);
public int degreeOut(Vertex v);

public boolean hasEdge(Vertex from, Vertex to);
public boolean addEdge(Vertex from, Vertex to, Double weight);
public boolean removeEdge(Vertex from, Vertex to);
public Double weightEdge(Vertex from, Vertex to);

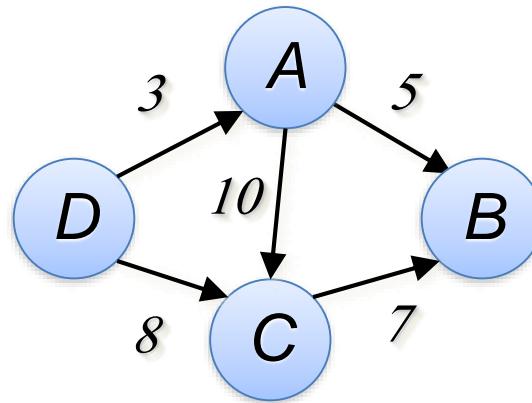
public Collection<Vertex> adjacentsTo(Vertex v);
public Collection<Vertex> getAllVertex();

public int getNumVertex();
public int getNumEdge();
```



```
private SortedMap<Vertex, SortedMap<Vertex, Double>> adjList;
private int numVertex;           número de vértices
private int numEdges;          número de arcos
```

```
public boolean addVertex(Vertex v){
    if (hasVertex(v)) return false; //no se admiten repetidos
    numVertex++;
    //Primero creo una lista_de_adyacentes vacía
    TreeMap<Vertex, Double> aux=new TreeMap<Vertex,Double>();
    adjList.put(v, aux); //meto el par (vertice, lista_de_adyacentes)
    return true;
}
```



(A, ((B, 5) (C, 10))
B, ()
C, ((B, 7))
D, ((A, 3) (C, 8)))

¿ g.addVertex("E") ?

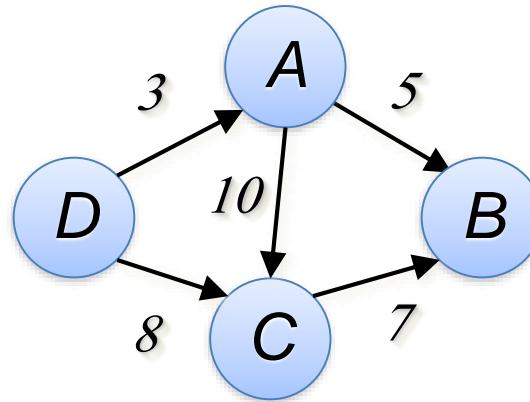




```
private SortedMap<Vertex, SortedMap<Vertex, Double>> adjList;  
private int numVertex;           número de vértices  
private int numEdges;          número de arcos
```

```
public boolean hasVertex(Vertex v)  
{  
    return adjList.containsKey(v);  
}
```

```
public boolean hasEdge(Vertex from, Vertex to){  
    if (hasVertex(from) && hasVertex(to)){  
        SortedMap<Vertex,Double> adjs=adjList.get(from);  
        return adjs.containsKey(to);  
    }  
    else  
        return false;  
}
```



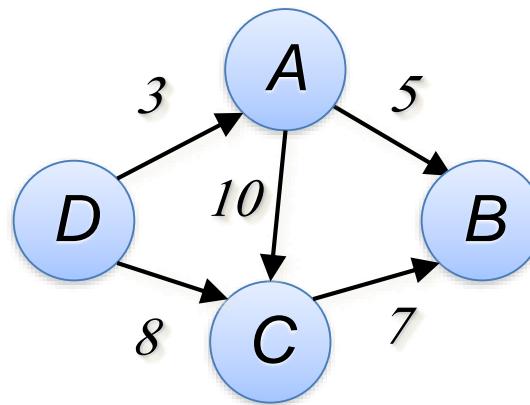
(A, ((B, 5) (C, 10))
B, ()
C, ((B, 7))
D, ((A, 3) (C, 8)))

¿ g.hasEdge("D","C") ?



```
private SortedMap<Vertex, SortedMap<Vertex, Double>> adjList;
private int numVertex;           número de vértices
private int numEdges;           número de arcos
```

```
public boolean addEdge(Vertex from, Vertex to, Double weight){
    if (hasEdge(from,to) || from.equals(to)) return false;
    numEdges++;
    addVertex(from); //puede existir ya y no lo mete de nuevo
    addVertex(to);   //puede existir ya y no lo mete de nuevo
    SortedMap<Vertex,Double> adjs=adjList.get(from);
    adjs.put(to, weight);
    return true; }
```



(A, ((B, 5) (C, 10))
B, ()
C, ((B, 7))
D, ((A, 3) (C, 8)))

¿ g.addEdge("B","C",10) ?

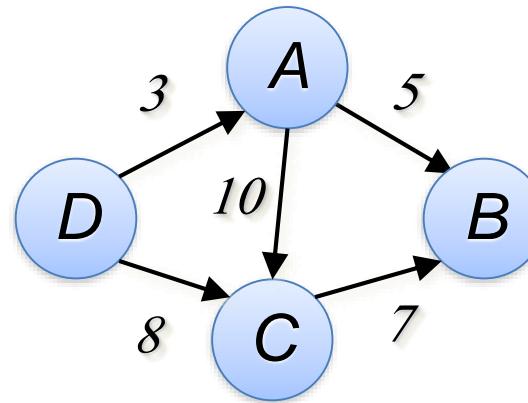




```
private SortedMap<Vertex, SortedMap<Vertex, Double>> adjList;
private int numVertex;           número de vértices
private int numEdges;           número de arcos
```

```
public Double weightEdge(Vertex from, Vertex to){
    if (!hasEdge(from,to))
        return null; //o lanzar una excepción

    SortedMap<Vertex,Double> adjs=adjList.get(from);
    return adjs.get(to);
}
```



(A, ((B, 5) (C, 10))
B, ()
C, ((B, 7))
D, ((A, 3) (C, 8)))

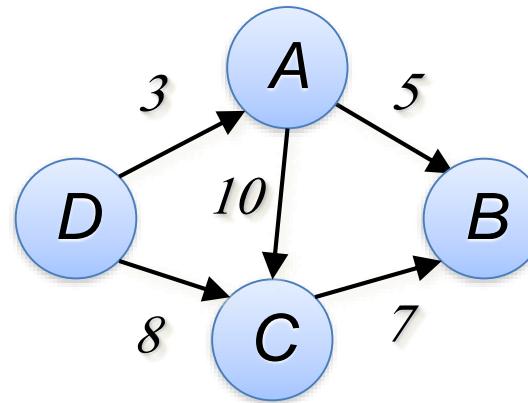
¿ g.weightEdge("D","C") ?





```
private SortedMap<Vertex, SortedMap<Vertex, Double>> adjList;
private int numVertex;           número de vértices
private int numEdges;           número de arcos
```

```
public boolean removeEdge(Vertex from, Vertex to){
    if (!hasEdge(from,to))
        return false;
    SortedMap<Vertex,Double> adjs=adjList.get(from);
    adjs.remove(to);
    numEdges--;
    return true;
}
```



(A, ((B, 5) (C, 10))

B, ()

C, ((B, 7))

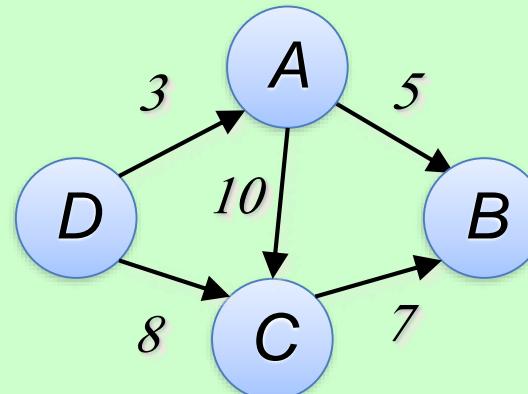
D, ((A, 3) (C, 8)))

¿ g.removeEdge("D","C") ?



```
private SortedMap<Vertex, SortedMap<Vertex, Double>> adjList;
private int numVertex;           número de vértices
private int numEdges;           número de arcos
```

```
public boolean removeVertex(Vertex v){
    if (!hasVertex(v))
        return false;
    //hay que borrarlo de cada dicc de adyacentes y del dicc principal
    numEdges=numEdges-degreeOut(v);
    adjList.remove(v);
    for (Entry<Vertex, SortedMap<Vertex, Double>> entry : adjList.entrySet()){
        SortedMap<Vertex,Double> adjs=entry.getValue();
        if(adjs.containsKey(v)) {
            adjs.remove(v);
            numEdges--;
        }
    }
    numVertex--;
    return true;
}
```



(A, ((B, 5) (C, 10))
B, ()
C, ((B, 7))
D, ((A, 3) (C, 8)))

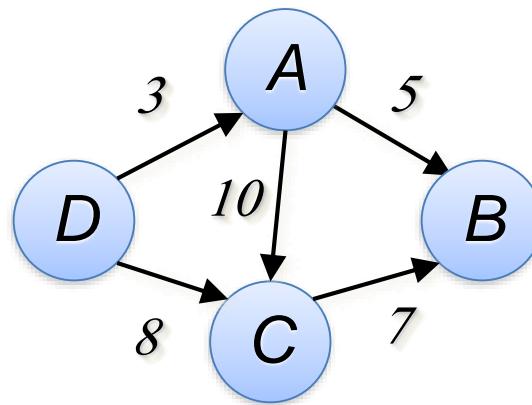
¿ g.removeVertex("C") ?



```
private SortedMap<Vertex, SortedMap<Vertex, Double>> adjList;
private int numVertex;           número de vértices
private int numEdges;           número de arcos
```

```
public Collection<Vertex> getAllVertex()
{
    return adjList.keySet();
}
```

```
public Collection<Vertex> adjacentsTo(Vertex v)
{
    if (!hasVertex(v))
        throw new RuntimeException
            ("Vertice no existe");
    SortedMap<Vertex,Double> adjs=adjList.get(v);
    return adjs.keySet();
}
```



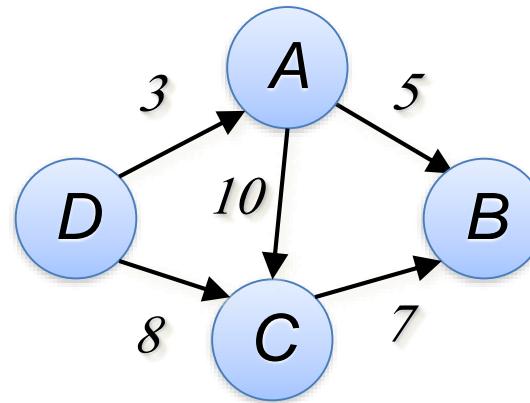
- (A, ((B, 5) (C, 10))
- B, ()
- C, ((B, 7))
- D, ((A, 3) (C, 8)))

¿ g.adjacentsTo("D") ?



```
private SortedMap<Vertex, SortedMap<Vertex, Double>> adjList;
private int numVertex;           número de vértices
private int numEdges;           número de arcos
```

```
public int degreeOut(Vertex v){
    if (!hasVertex(v))
        throw new RuntimeException("Vertice no existe");
    return adjacentsTo(v).size();
}
```



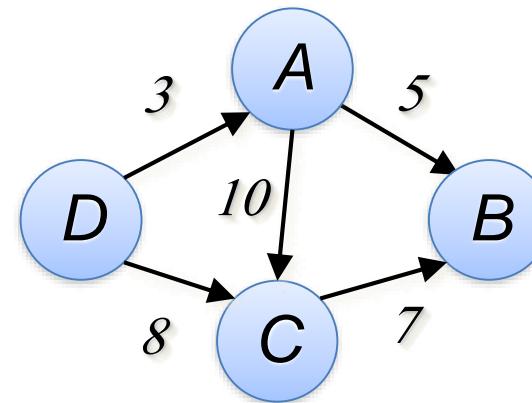
(A, ((B, 5) (C, 10))
B, ()
C, ((B, 7))
D, ((A, 3) (C, 8)))

¿ g.degreeOut("D") ?



```
private SortedMap<Vertex, SortedMap<Vertex, Double>> adjList;
private int numVertex;           número de vértices
private int numEdges;           número de arcos
```

```
public int degreeIn(Vertex v){
    if (!hasVertex(v))
        throw new RuntimeException("Vertice no existe");
    int count=0;
    for (Entry<Vertex, SortedMap<Vertex, Double>> entry : adjList.entrySet()){
        SortedMap<Vertex,Double> adjs=entry.getValue();
        if (adjs.containsKey(v))
            count++;
    }
    return count;
}
```



- (A, ((B, 5) (C, 10))
- B, ()
- C, ((B, 7))
- D, ((A, 3) (C, 8)))

¿ g.degreeIn("C") ?

Implementación de la búsqueda en profundidad (i)

acción REP (g:Grafo)

inicio

para cada vértice v de g hacer
desmarcar(v)
predecesor(v, NULO)

fpara;

para cada vértice v de g hacer
si \neg marcado(v)

entonces

BPF(v)

fsi

fpara

facción

Recorrido en profundidad (grafo)

¡No se incluye en el
código generar el
árbol!

```
public static <Vertex> void REP (Graph<Vertex> g) {  
    HashSet<Vertex> marcados=  
        new HashSet<Vertex>(g.getNumVertex()*2);  
    Collection<Vertex> vertices=g.getAllVertex();  
    for (Vertex v : vertices)  
        if (!marcados.contains(v))  
            BEP(g, v, marcados);  
}
```



Implementación de la búsqueda en profundidad (ii)

acción BPF (v:Vértice)

var w:Vértice

inicio

marcar(v)

para cada vértice w adyacente a v **hacer**

si \neg marcado(w)

entonces

predecesor(w, v)

 BPF(w)

fsi

fpara

facción

¡No se incluye en el código generar el árbol!

Búsqueda en profundidad (nodo inicial)

```
public static <Vertex> void BEP (Graph<Vertex> g, Vertex v,  
Collection<Vertex> marked)  
{  
    System.out.print(v+ " "); //tratar_vertice  
    marked.add(v);  
    Collection<Vertex> adyacentes=g.adjacentsTo(v);  
    for (Vertex w : adyacentes)  
        if (!marked.contains(w))  
            BEP(g, w, marked);  
}
```



Implementación de la búsqueda en anchura (i)

acción REA (g:Grafo)

inicio

para cada vértice v de g **hacer**

 desmarcar(v)

 predecesor(v, NULO)

fpara;

para cada vértice v de g **hacer**

si \neg marcado(v)

entonces

 BEA(v)

fsi

fpara

facción

Recorrido en anchura (grafo)

¡No se incluye en el
código generar el
árbol!

```
public static <Vertex> void REA (Graph<Vertex> g) {  
    HashSet<Vertex> marcados=  
        new HashSet<Vertex>(g.getNumVertex()*2);  
    Collection<Vertex> vertices=g.getAllVertex();  
    for (Vertex v : vertices)  
        if (!marcados.contains(v))  
            BEA(g, v, marcados);  
}
```

acción BEA (v:Vértice)

var v, w, u:Vértice

q:ColaNodos

inicio

añadir(q,v); marcar(v)

mientras \neg vacía(q) **hacer**

u \leftarrow borrar(q)

para cada vértice w adyacente a u **hacer**

si \neg marcado(w)

entonces

marcar(w)

predecesor(w, u)

añadir(q, w)

fsi

fpara

fmientras

facción

¡No se incluye en el código generar el árbol!

Implementación de la búsqueda en anchura(ii)

Búsqueda en anchura(nodo inicial)

```
public static <Vertex> void BEA (Graph<Vertex> g, Vertex v,
Collection<Vertex> marked) {
```

```
LinkedList<Vertex> cola=new LinkedList<Vertex>();
```

```
cola.addLast(v); marked.add(v);
```

```
while (!cola.isEmpty()) {
```

```
Vertex u=cola.removeFirst();
```

```
System.out.print(u); System.out.print(" ");
```

```
Collection<Vertex> adyacentes=g.adjacentsTo(u);
```

```
for (Vertex w : adyacentes)
```

```
if (!marked.contains(w)) {
```

```
marked.add(w);
```

```
cola.addLast(w);
```

```
} } }
```





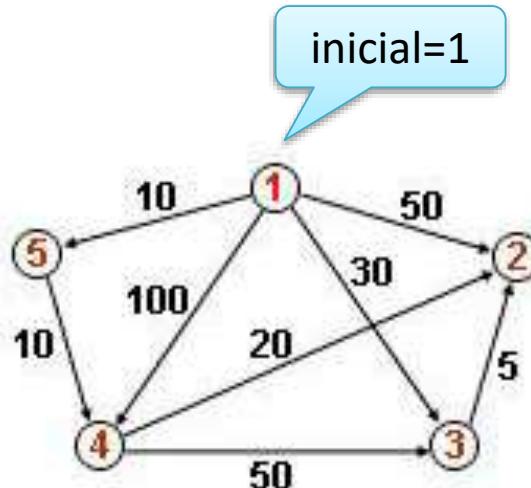
Algoritmo de Dijkstra

```
acción Dijkstra (g:Grafo, inicial:Vertice)
    para cada vértice v de g hacer
        Si (no existe arco entre inicial y v) entonces
            distancia(v) = Infinito
        Si_no
            distancia(v) = peso(inicial, v) ; predecesor(v)=inicial
        fisi
        marcar(inicial)
    mientras que (no_estén_vistos_todos) hacer
        sgte = vértice con menor distancia y que no esté marcado
        marcar(sgte)
    para cada vértice w adyacente a sgte hacer
        si distancia(w)>distancia(sgte)+peso(sgte, w) entonces
            distancia(w) = distancia(sgte)+peso(sgte, w)
            predecesor(w)=sgte
        fisi
    fpara
    fmientras
facción
```



Algoritmo de Dijkstra (ejemplo)

```
acción Dijkstra (g:Grafo, inicial:Vertice)
para cada vértice v de g hacer
    Si (no existe arco entre inicial y v) entonces
        distancia(v) = Infinito
    Si_no
        distancia(v) = peso(inicial, v) ; predecesor(v)=inicial
    fisi
    marcar(inicial)
...
...
```



distancia(2)=50	predecesor(2)=1
distancia(3)=30	predecesor(3)=1
distancia(4)=100	predecesor(4)=1
distancia(5)=10	predecesor(5)=1

marcados={1}

...

mientras que (no_estén_vistos_todos) **hacer**

sgte = vértice con menor distancia y que no esté marcado

marcar(sgte)

para cada vértice w adyacente a sgte **hacer**

si distancia(w)>distancia(sgte)+peso(sgte, w) **entonces**

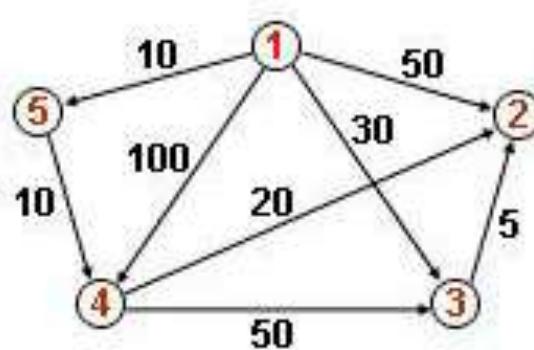
distancia(w) = distancia(sgte)+peso(sgte, w)

predecesor(w)=sgte

fsi

fpara

fmientras



dist(2)=50	pred(2)=1
dist(3)=30	pred(3)=1
dist(4)=100	pred(4)=1
dist(5)=10	pred(5)=1

marcados={1}

sgte=5

w → {4}

dist(2)=50	pred(2)=1
dist(3)=30	pred(3)=1
dist(4)= 20	pred(4)= 5
dist(5)=10	pred(5)=1

marcados={1,5}

dist(4)=100
dist(5)+peso(5,4)=10+10



...

mientras que (no_estén_vistos_todos) **hacer**

sgte = vértice con menor distancia y que no esté marcado

marcar(sgte)

para cada vértice w adyacente a sgte **hacer**

si distancia(w)>distancia(sgte)+peso(sgte, w) **entonces**

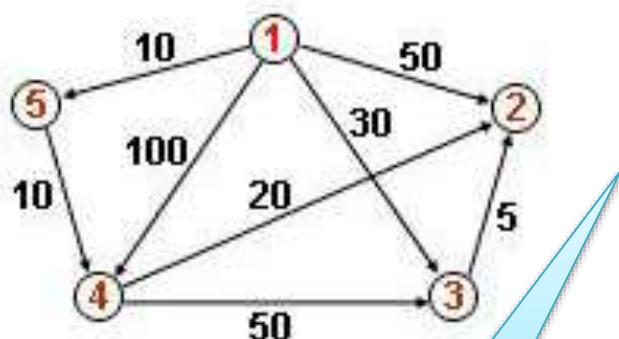
distancia(w) = distancia(sgte)+peso(sgte, w)

predcesor(w)=sgte

fsi

fpara

fmientras



dist(2)=50	pred(2)=1
dist(3)=30	pred(3)=1
dist(4)= 20	pred(4)= 5
dist(5)=10	pred(5)=1

marcados={1,5}

w → {2,3}

dist(2)=40	pred(2)=4
dist(3)=30	pred(3)=1
dist(4)=20	pred(4)=5
dist(5)=10	pred(5)=1

marcados={1,5,4}

dist(2)=50
dist(4)+peso(4,2)=20+20
dist(3)=30
dist(4)+peso(4,3)=20+50



...

mientras que (no_estén_vistos_todos) **hacer**

sgte = vértice con menor distancia y que no esté marcado

marcar(sgte)

para cada vértice w adyacente a sgte **hacer**

si distancia(w)>distancia(sgte)+peso(sgte, w) **entonces**

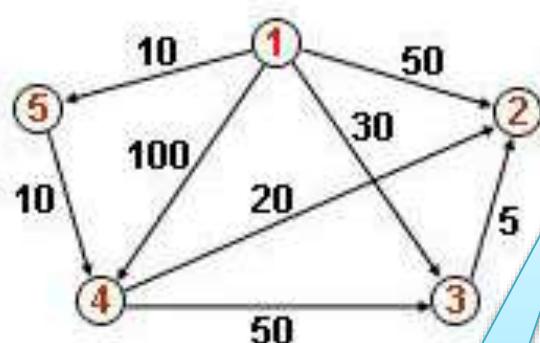
distancia(w) = distancia(sgte)+peso(sgte, w)

predcesor(w)=sgte

fsi

fpara

fmientras



sgte=3

dist(2)= 40	pred(2)= 4
dist(3)=30	pred(3)=1
dist(4)=20	pred(4)=5
dist(5)=10	pred(5)=1

marcados={1,5,4}

w → {2}

dist(2)= 35	pred(2)= 3
dist(3)=30	pred(3)=1
dist(4)=20	pred(4)=5
dist(5)=10	pred(5)=1

marcados={1,5,4,3}

dist(2)=40
dist(3)+peso(3,2)=30+5

...

mientras que (no_estén_vistos_todos) **hacer**

sgte = vértice con menor distancia y que no esté marcado

marcar(sgte)

para cada vértice w adyacente a sgte **hacer**

si distancia(w)>distancia(sgte)+peso(sgte, w) **entonces**

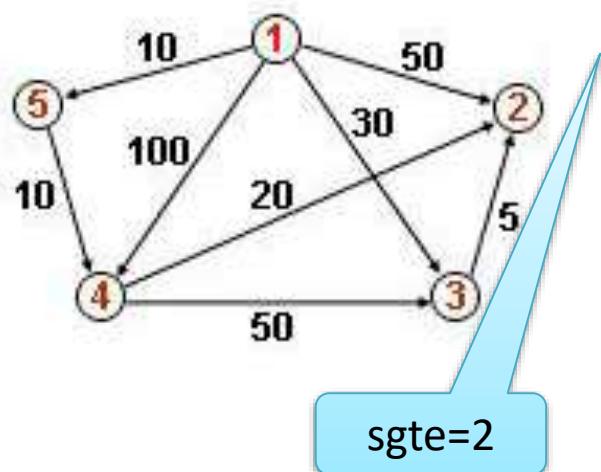
distancia(w) = distancia(sgte)+peso(sgte, w)

predcesor(w)=sgte

fsi

fpara

fmientras



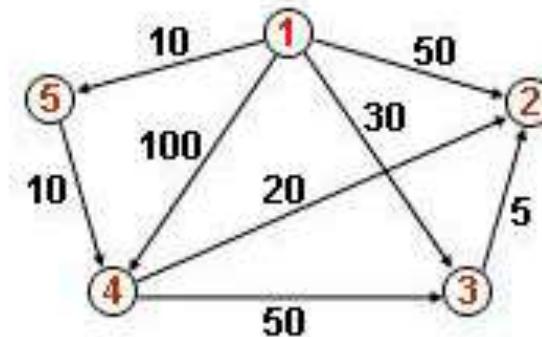
dist(2)= 35	pred(2)= 3
dist(3)=30	pred(3)=1
dist(4)=20	pred(4)=5
dist(5)=10	pred(5)=1

marcados={1,5,4,3}

w → {}

dist(2)= 35	pred(2)= 3
dist(3)=30	pred(3)=1
dist(4)=20	pred(4)=5
dist(5)=10	pred(5)=1

marcados = {1,5,4,3,2}



Fuente:
Transparencias de
Algoritmia

Lo mismo que en la
transparencia anterior
pero de otra manera

$\text{dist}[2..5] = \{ 35, 30, 20, 10 \}$ $\text{pred}[2..5] = \{ 3, 1, 5, 1 \}$

	<u>Coste</u>	<u>Camino</u>
Camino mínimo de 1 a 2:	35	$1 \rightarrow 3 \rightarrow 2$
Camino mínimo de 1 a 3:	30	$1 \rightarrow 3$
Camino mínimo de 1 a 4:	20	$1 \rightarrow 5 \rightarrow 4$
Camino mínimo de 1 a 5:	10	$1 \rightarrow 5$

Guardaremos esta
información de otra
manera (sin necesidad
de numerar los vértices)



```
acción Dijkstra (g:Grafo, inicial:Vertice)
para cada vértice v de g hacer
    Si (no existe arco entre inicial y v) entonces
        distancia(v) = Infinito
    Si_no
        distancia(v) = peso(inicial, v)
        predecesor(v)=inicial;
    fisi
    marcar(inicial)
    ...
```

```
public static <Vertex> void Dijkstra (Graph<Vertex> g, Vertex initial) {
    Map<Vertex,Double> dist=new TreeMap<Vertex,Double>(); //distancia de "initial" al resto
    Map<Vertex,Vertex> pred=new TreeMap<Vertex,Vertex>(); //prededcesores de cada vertice
    Set<Vertex> visited=new HashSet<Vertex>(g.getNumVertex()*2);

    Collection<Vertex> vertices=g.getAllVertex();
    for (Vertex v : vertices)
        if (!g.hasEdge(initial,v))
            dist.put(v, Double.POSITIVE_INFINITY);
        else {
            dist.put(v,g.weightEdge(initial, v));
            pred.put(v,initial);
        }
    visited.add(initial);
    ...
```





```
mientras que (no_estén_vistos_todos) hacer
    sgte = vértice con menor distancia y que no esté marcado
    marcar(sgte)
    para cada vértice w adyacente a sgte hacer
        si distancia(w)>distancia(sgte)+peso(sgte, w) entonces
            distancia(w) = distancia(sgte)+peso(sgte, w)
            predecesor(w)=sgte
        fsi
    fpara
fmientras
```

```
while (visited.size()<g.getNumVertex()){
    Vertex next=dist_min(dist, visited); //hay que hacer esta función!!!
    visited.add(next);
    Collection<Vertex> adyacentes=g.adjacentsTo(next);
    for (Vertex w : adyacentes){
        double dist_w = dist.get(w);
        double dist_next_w = dist.get(next)+ g.weightEdge(next, w);
        if (dist_w > dist_next_w) {
            dist.put(w, dist_next_w);
            pred.put(w,next);
        }
    }
}
```



```
public static <Vertex> void Dijkstra (Graph<Vertex> g, Vertex initial) {  
    Map<Vertex,Double> dist=new TreeMap<Vertex,Double>(); //distancia de "initial" al resto  
    Map<Vertex,Vertex> pred=new TreeMap<Vertex,Vertex>(); //prededecesores de cada vertice  
    Set<Vertex> visited=new HashSet<Vertex>(g.getNumVertex()*2);  
    Collection<Vertex> vertices=g.getAllVertex();  
    for (Vertex v : vertices)  
        if (!g.hasEdge(initial,v))    dist.put(v, Double.POSITIVE_INFINITY);  
        else {  
            dist.put(v,g.weightEdge(initial, v));    pred.put(v,initial); }  
    visited.add(initial);  
    while (visited.size()<g.getNumVertex()){  
        Vertex next=dist_min(dist, visited);    //hay que hacer esta función  
        visited.add(next);  
        Collection<Vertex> adyacentes=g.adjacentsTo(next);  
        for (Vertex w : adyacentes){  
            double dist_w=(double) dist.get(w);  
            double dist_next_w=(double) dist.get(next)+(double) g.weightEdge(next, w);  
            if (dist_w>dist_next_w) {  
                dist.put(w, dist_next_w);  
                pred.put(w,next);  
            }  
        } } }
```





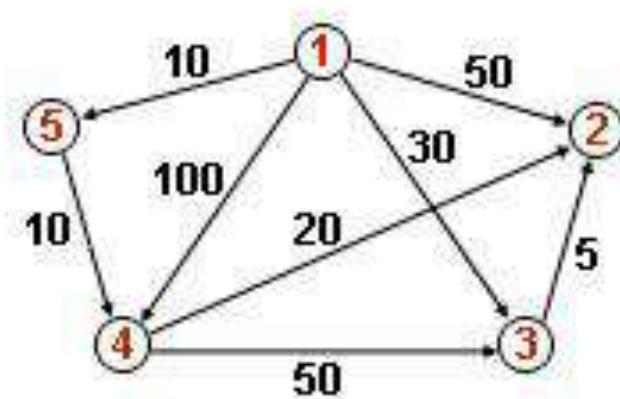
```
public static <Vertex> Vertex dist_min(Map<Vertex,Double> distances,
                                         Set<Vertex> visited)
{
    double minWeight=Double.MAX_VALUE;
    Vertex selected=null;
    for (Entry<Vertex,Double> entry : distances.entrySet()){
        double weight=entry.getValue().doubleValue(); //otra forma de conversión
        Vertex v=entry.getKey();
        if (!visited.contains(v) && weight<minWeight){
            minWeight=weight;
            selected=v;
        }
    }
    return selected;
}
```

- Coste de esta versión: $O(n^2)$
- Es posible hacer implementaciones con coste menor



Para el ejemplo anterior:

```
Graph<String> G = new Graph<String>();  
  
G.addEdge("1", "2", 50);  
G.addEdge("1", "3", 30);  
G.addEdge("1", "4", 100);  
G.addEdge("1", "5", 10);  
G.addEdge("3", "2", 5);  
G.addEdge("4", "2", 20);  
G.addEdge("4", "3", 50);  
G.addEdge("5", "4", 10);  
  
Algoritmos.Dijkstra(G,"1")
```



dist(2)=35	pred(2)=3
dist(3)=30	pred(3)=1
dist(4)=20	pred(4)=5
dist(5)=10	pred(5)=1

Dist= { (2,35) (3,30) (4,20) (5,10) }
Pred= { (2,3) (3,1) (4,5) (5,1) }



Algoritmo de Dijkstra (heap)

```
acción Dijkstra (g:Grafo, inicial:Vertice)
    para cada vértice v de g hacer
        distancia(v) = Infinito
    fpara
    distancia(inicial)=0
    marcar(inicial)
    añadir_a_la_colas ( inicial,distancia(inicial) ) //ordenada por los pesos
    mientras que (cola_no_vacía) hacer
        sgte = mínimo de la cola; borrar_minimo_colas
        marcar(sgte)
        para cada vértice w adyacente a sgte hacer
            si distancia(w)>distancia(sgte)+peso(sgte, w) entonces
                distancia(w) = distancia(sgte)+peso(sgte, w)
                predecesor(w)=sgte
                añadir_a_la_colas ( w, distancia(w))
        fsi
    fpara
    fmientras
facción
```

