# Tema 11. Estructura de datos Grafo

`http://aulavirtual.uji.es`

**José M. Badía, Begoña Martínez, Antonio Morales y José M. Sanchiz**

**{badia, bmartine, morales}@icc.uji.es**

**Estructuras de datos y de la información**

**Universitat Jaume I**

# Índice

# Bibliografía

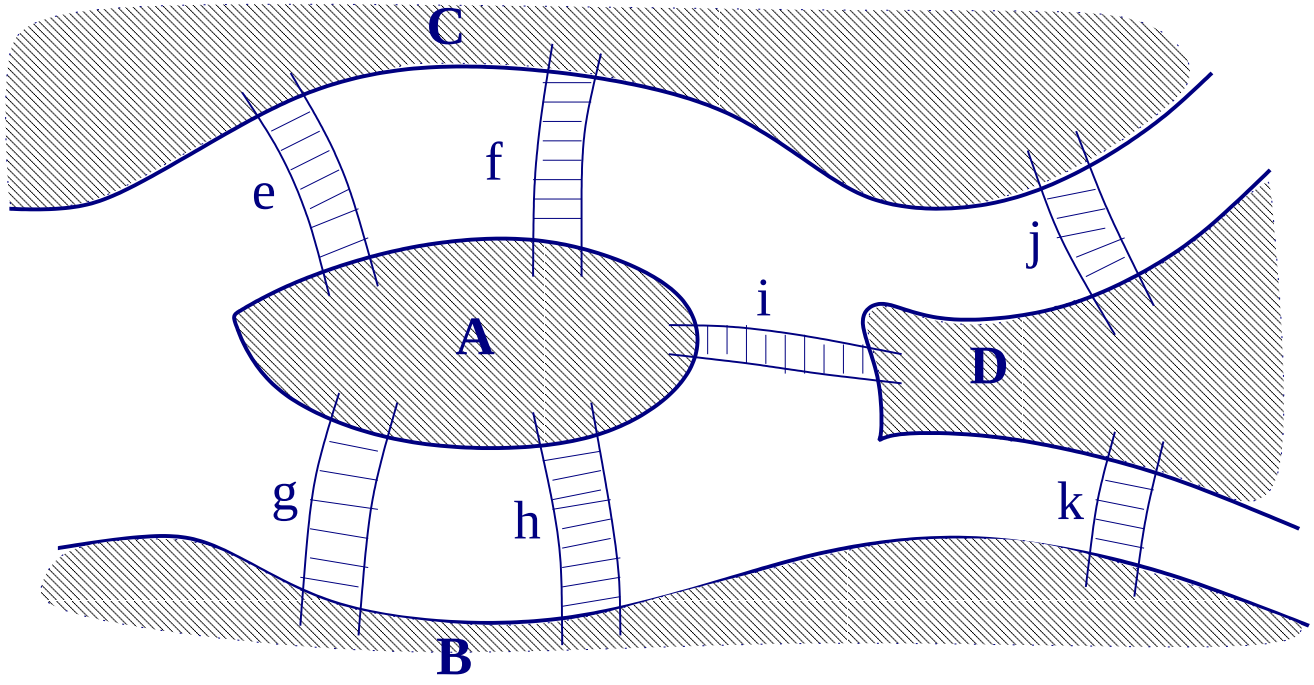

- (Nyhoff ’06), Capítulo 16.
- (Main, Savitch ’01), Capítulo 15.
- *Fundamentals of Data Structures in C++*. E. Horowitz, S. Sahni, D. Mehta. Computer Science Press, 1995. Capítulo 6.
- *Estructuras de Datos, Algoritmos y Programación Orientada a Objetos*. G.L. Heileman. Mc Graw Hill, 1998. Capítulo 14.

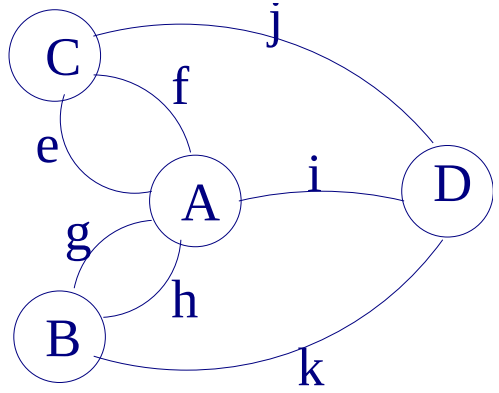
# Objetivos

- Asimilar el concepto de grafo y las definiciones básicas asociadas.
- Conocer los principales problemas que pueden ser modelizados mediante grafos.
- Conocer las operaciones básicas del TAD grafo.
- Saber implementar e interpretar un grafo utilizando matrices de adyacencia.
- Saber implementar e interpretar un grafo utilizando listas de adyacencia.
- Asimilar los algoritmos de recorrido en anchura y en profundidad de un grafo y ser capaz de implementarlos.
- Comprender el concepto de árbol de recubrimiento y árbol de recubrimiento mínimo y saber aplicar los diferentes algoritmos para obtenerlos.
- Asimilar el concepto de camino en un grafo y el de caminos mínimos.
- Ser capaz de calcular la matriz de caminos en un grafo. Conocer y saber aplicar el algoritmo de Warshall y el algoritmo de Dijkstra.

# 1 Introducción

Problema del puente de Königsberg (*Kaliningrado, Rusia*): buscar una ruta que comience en cualquiera de las cuatro zonas de tierra (A, B, C, D), cruce cada uno de los puentes exactamente una sóla vez y vuelva al punto de inicio. L. Euler demostró en 1736 usando un grafo que no había solución.



# 2 Aplicaciones

Los grafos se usan para representar y estudiar problemas en:

- Redes de comunicaciones: Internet, telefonía.
- Redes de transporte: redes de autopistas, tráfico en una ciudad, red de aeropuertos, red ferroviaria, canalizaciones de agua.
- Arquitecturas de computadoras paralelas.
- Circuitos eléctricos.
- Estructuras o compuestos químicos.
- Planificación de tareas.
- Problemas en inteligencia artificial, genética, lingüistica, etc.

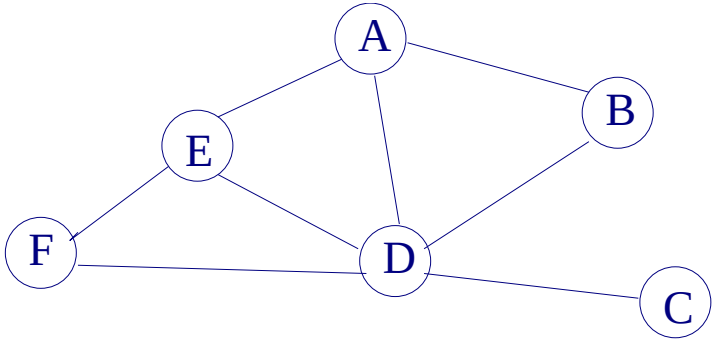
Problemas que se resuelven: conexión, accesibilidad, caminos mínimos, recubrimiento, ordenación y precedencia, búsquedas, etc.

# 3 Definiciones

**Grafo**: $G = (V, E)$

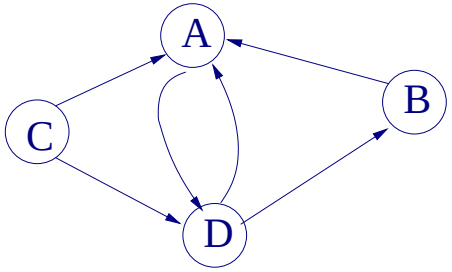
Un grafo se compone de un conjunto finito y no vacío de **nodos** (o vértices) $V$ y conjunto finito de **arcos** (aristas o transiciones) $E$, que conectan pares de nodos.

El conjunto de arcos $E$ son pares $(u, v)$ donde $u$ y $v$ pertenecen al conjunto de nodos $V$.

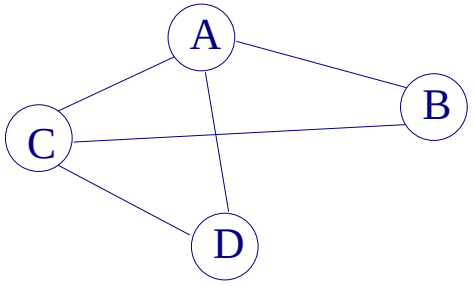


# 3 Definiciones (II)

- **Grafo no dirigido**: Un arco conecta dos nodos en ambos sentidos. Los pares $(u, v)$ de $E$ son no ordenados y por tanto, $(u, v)$ y $(v, u)$ son el mismo arco.
- **Grafo dirigido**: Los pares $\langle u, v \rangle$ son ordenados. $\langle u, v \rangle$ y $\langle v, u \rangle$ son arcos distintos.



grafo dirigido



grafo no dirigido

El máximo número de arcos o aristas de un grafo de $n$ nodos es:

- $n \cdot (n-1)/2$, si es un grafo no dirigido
- $n \cdot (n-1)$, si es un grafo dirigido

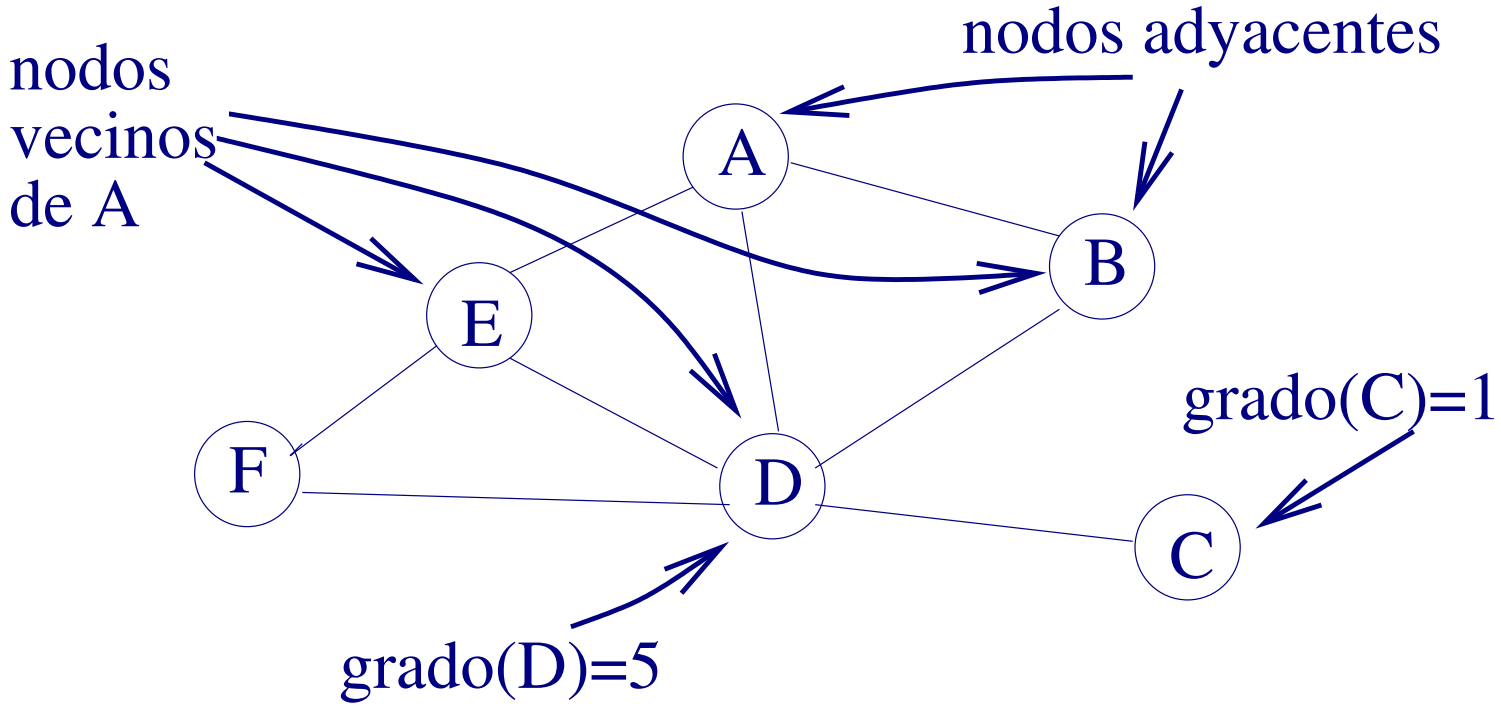
# 3 Definiciones (III)

- Dos nodos son **adyacentes** si hay un arco que los conecta.
  - $(u, v)$ es un arco en un grafo no dirigido, entonces $u$ es **adyacente a** $v$ y $v$ es adyacente a $u$.
  - $\langle u, v \rangle$ es un arco (arista) en un grafo dirigido, entonces $u$ es **adyacente hacia** $v$ y $v$ es **adyacente desde** $u$.
- En un grafo dirigido G,
  - El grado de entrada de un nodo $v$ es el número de nodos adyacentes hacia $v$.
  - El grado de salida de un nodo $v$ es el número de nodos adyacentes desde $v$.
  - El grado de un nodo $v$ es la suma de sus grados de entrada y salida.
- En un grafo no dirigido G
  - El grado de un nodo es el número de arcos incidentes en él.

# 3 Definiciones (IV)

- **Camino** en un grafo $G = (V, E)$: secuencia de nodos del grafo $v_1$, $v_2$, ..., $v_k$, tales que cada par de nodos consecutivos en la secuencia, $v_i$ y $v_{i+1}$ están conectados por un arco. Es decir, los arcos $(v_1, v_2)$, $(v_2, v_3)$ ... $(v_i, v_{i+1})$ ... $(v_{k-1}, v_k)$ pertenecen a $E$.
- **Camino simple**: Es un camino en el que cada nodo aparece una sola vez en la secuencia.
- **Longitud de un camino**: número de arcos que lo componen.
- **Ciclo**: Camino en el que el último y primer nodo coinciden. Para grafos no dirigidos, la longitud del ciclo es mayor que 2.
- En un **grafo etiquetado** se asocian etiquetas a los nodos y/o arcos. Si las etiquetas representan pesos en los arcos entonces se llaman **grafos ponderados**.

# 3 Definiciones (V)

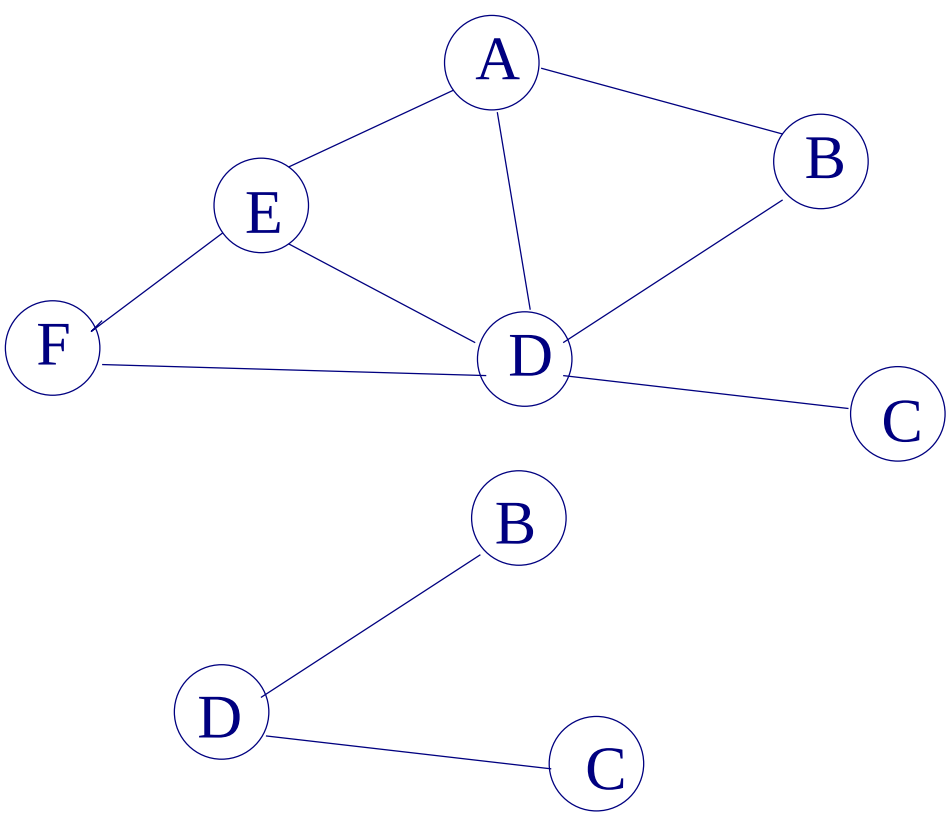


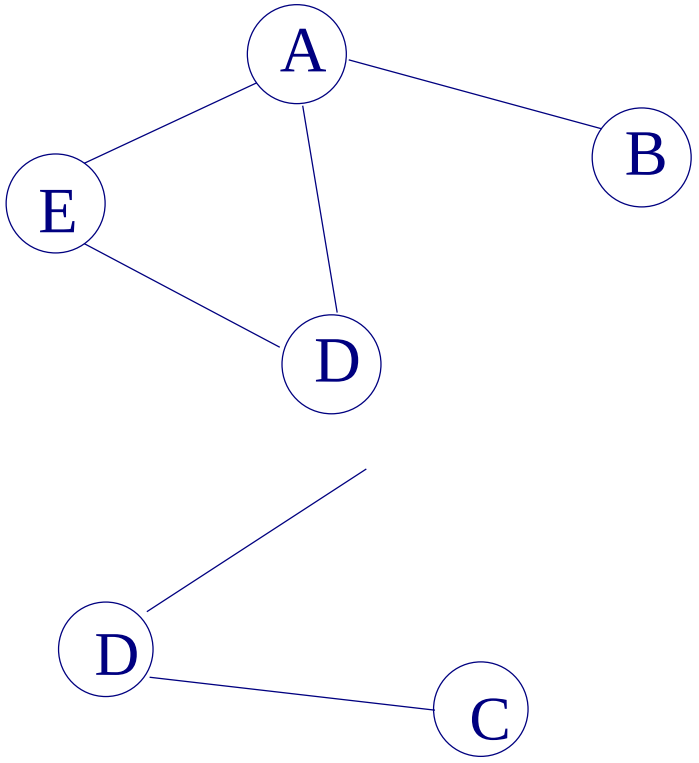

E – D – B – A es un camino simple de longitud 3
E – D – B – A– E es un ciclo

# 3 Definiciones (VI)

- **Subgrafo**: Grafo formado por una parte de los nodos y arcos de otro grafo.

  Sea un grafo $G = (V, E)$, un subgrafo de $G$ es un grafo $G' = (V', E')$ tal que $V' \subseteq V$ y $E' \subseteq E$.



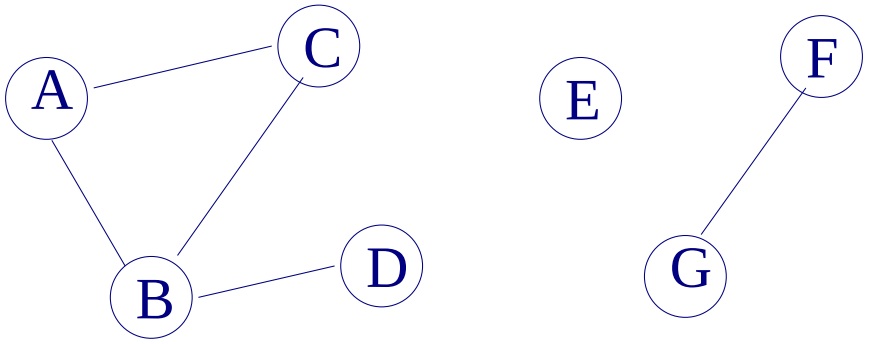




no es un
grafo válido

# 3 Definiciones (VII)

- **Grafo conectado o conexo**: Es un grafo con un camino entre cada par de nodos distintos.
- Las **componentes conexas** de un grafo son el menor número de subgrafos conexos que lo componen.

Ejemplo: grafo no conexo con 3 componentes conexas




- **Árbol**: grafo conexo y sin ciclos.
- Todo grafo conexo con $n$ nodos y $n-1$ arcos no puede contener ciclos y es, por tanto, un árbol.

# 4 El TAD grafo

**TAD** Grafo **Usa** Nodo, Arco, Conjunto, Bool

**Operaciones:**

*//Crea un grafo vacío*

**Crearg:** → **grafo**

*//Añade un nuevo nodo al grafo*

**InsertarNodo: grafo x nodo → grafo**

*//Añade un arco entre dos nodos existentes*

**InsertarArco: grafo x arco → grafo**

*//Indica si un grafo esta vacío*

**EsVacio: grafo → bool**

*//Borra un nodo y todos los arcos que tiene ese nodo*

**BorrarNodo: grafo x nodo → grafo**

*//Borra un arco*

**BorrarArco: grafo x arco → grafo**

*//Buscar un nodo*

**Buscar: grafo x nodo → Bool**

# 4 El TAD grafo (II)

*//Conjunto de nodos de un grafo*

**Nodos: grafo → Conjunto(nodo)**

*//Conjunto de arcos de un grafo*

**Arcos: grafo → Conjunto(arcos)**

*//Conjunto de nodos adyacentes a un nodo*

**Adyacentes: grafo x nodo → Conjunto(nodo)**

**Axiomas:** ∀ g ∈ grafo, ∀ n, m ∈ nodo, ∀ [i,j], [k,l] ∈ arco

1) EsVacio(Crearg) = Verdadero
2) EsVacio(InsertarNodo(g,n)) = Falso
3) EsVacio(InsertarArco(g,[i,j])) = Falso
4) BorrarNodo(Crearg, n) = Error
5) BorrarNodo( InsertarNodo(g, m), n) = Si n=m entonces g
   sino InsertarNodo( BorrarNodo(g, n), m)
6) BorrarNodo( InsertarArco(g,[i,j]), n) =
   Si i=n or j=n entonces BorrarNodo(g, n)
   sino InsertarArco( BorrarNodo(g, n), [i,j])

# 4 El TAD grafo (III)

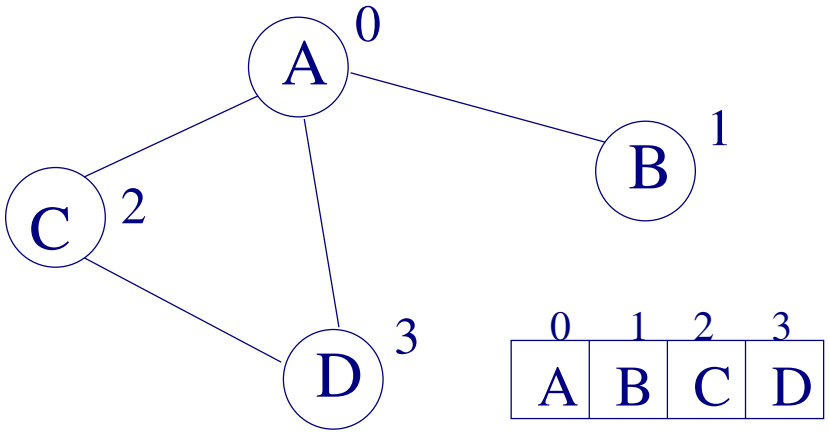
```
7) BorrarArco(Crearg, [i,j]) = Error
8) BorrarArco( InsertarNodo(g, n), [i,j]) =
      Si n=i or n=j entonces Error
      sino InsertarNodo( BorrarArco(g, [i,j]), n)
9) BorrarArco( InsertarArco(g, [k,l]), [i,j]) =
      Si (k=i and l=j) or (k=j and l=i) entonces g
      sino InsertarArco( BorrarArco(g, [i,j]), [k,l])
10) Buscar(Crearg, n) = Falso
11) Buscar( InsertarNodo(g, m), n) = Si n=m entonces Verdadero
      sino Buscar(g, n)
12) Buscar( InsertarArco(g, [i,j]), n) = Buscar(g, n)
13) Nodos(Crearg) = CrearCj
14) Nodos( InsertarNodo(g, n)) = AñadirCj( Nodos(g), n)
15) Nodos( InsertarArco(g, [i,j]) = Nodos(g)
16) Arcos(Crearg) = CrearCj
17) Arcos( InsertarNodo(g, n)) = Arcos(g)
18) Arcos( InsertarArco(g, [i,j]) = AñadirCj( Arcos(g), [i,j])
```

# 4 El TAD grafo (IV)

```
19) Adyacentes( Crearg, n) = Error
20) Adyacentes( InsertarNodo(g, m), n) = Si n=m entonces CrearCj
      sino Adyacentes(g, n)
21) Adyacentes( InsertarArco(g, [i,j]), n) =
      Si i=n entonces InsertarCj( Adyacentes(g, n), j)
      sino Si j=n entonces AñadirCj( Adyacentes(g, n), i)
      sino Adyacentes(g, n)
```

## Matrices de adyacencia

- Numerar los nodos del grafo. Sea $n$ el número de nodos del grafo.
- La matriz de adyacencia es una matriz $n \times n$, **Ady**, en la cual la entrada en la fila $i$ y columna $j$ es 1 (cierto) si el nodo $j$ es adyacente al nodo $i$, y es 0 (falso) si no lo es: $Ady[i][j]$ es 1 si hay un arco del nodo $i$ con el nodo $j$.




| 0 | 1 | 2 | 3 |
|---|---|---|---|
| A | B | C | D |

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 | 1 |
| 3 | 1 | 0 | 1 | 0 |

Si es un grafo no dirigido, la matriz es simétrica.

# 5.1 Matrices de adyacencia

```
template <class tn>
class grafo { // no ponderado
private:
   int numnodos, capacidad;
   bool **Ady; // arcos
   tn *nodos; //etiquetas nodos
public:
   //operaciones del TAD grafo
};
```

**Ventajas**

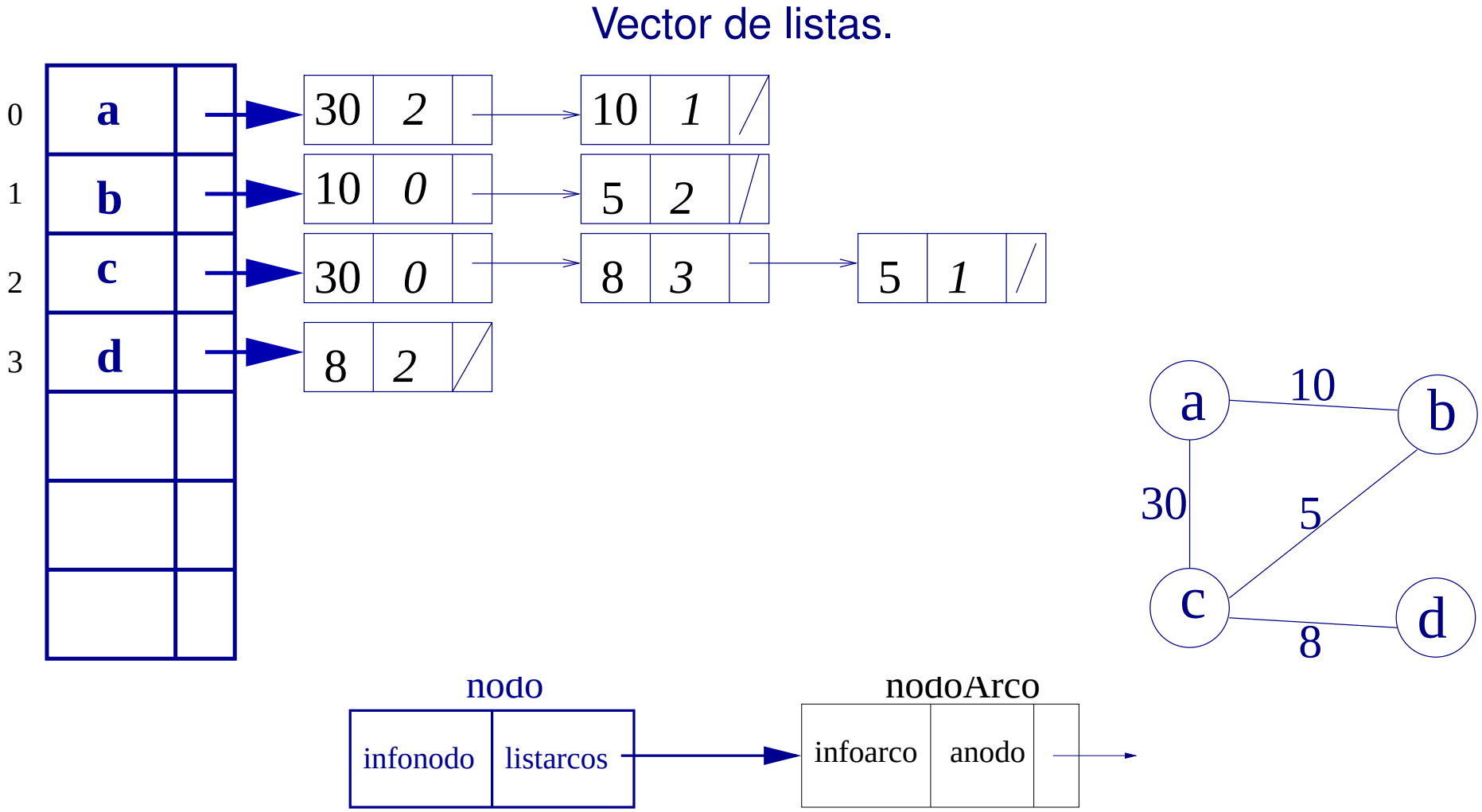
- Fáciles de construir.
- Es sencillo determinar el grado de un nodo.
- Coste temporal acceso a un arco: O(1).

```
template <class tn, class ta>
class grafo { // ponderado
private:
   int numnodos, capacidad;
   ta **Ady; // pesos
   tn *nodos;
public:
   //operaciones del TAD grafo
};
```

**Inconvenientes**

- En grafos dispersos se desperdicia espacio.
- Los recorridos por el grafo son más costosos.

# 5.2 Listas de adyacencia



# 5.2 Listas de adyacencia (II)

```
template <class tn, class ta>
class grafo {
public:
  typedef int apuntador;
private:
    class nodoArco {
    public:
      ta infoarco; //etiqueta del arco, peso
      apuntador anodo;  //índice al nodo adyacente
      nodoArco(const ta & e, apuntador pos);
    }; //fin clase nodoArco
```

# 5.2 Listas de adyacencia (III)

```
class nodo {
public:
  tn infonodo;    //etiqueta del nodo
  list<nodoArco> listarcos;  //lista de arcos de ese nodo
  bool visitado;  //inicialmente a falso
  nodo(const tn & elem=tn());
}; //clase nodo

nodo * graf;    //vector dinámico de nodos
int numnodos;   //número de nodos actual
int capacidad; //talla del vector

apuntador insert_nodo(const nodo &n);
```

# 5.2 Listas de adyacencia (IV)

```
public:
  struct arco {
    apuntador norigen, nfin;
    ta info;
  };
  grafo(int n=256);
  ~grafo();
  grafo(const grafo<tn, ta> & origen);
  grafo<tn,ta> & operator=(const grafo<tn, ta> & origen);
  bool empty() const;
  apuntador insert_elem (const tn & elem);
  void insert_arco (const arco & a);
```

# 5.2 Listas de adyacencia (V)

```
    apuntador find_elem(const tn & elem) const;
    const tn & data_nodo (apuntador p) const;
    tn & data_nodo (apuntador p);
    const ta & data_arco(apuntador ini, apuntador fin) const;
    ta & data_arco(apuntador ini, apuntador fin);
    void erase_nodo (apuntador p);
    void erase_arco (const arco & a);
    void DepthFirst (apuntador p) const;  //rec. en profundidad
    void BreadthFirst (apuntador p) const;  //rec. en anchura
};  //clase grafo
```

# 5.2 Listas de adyacencia (VI)

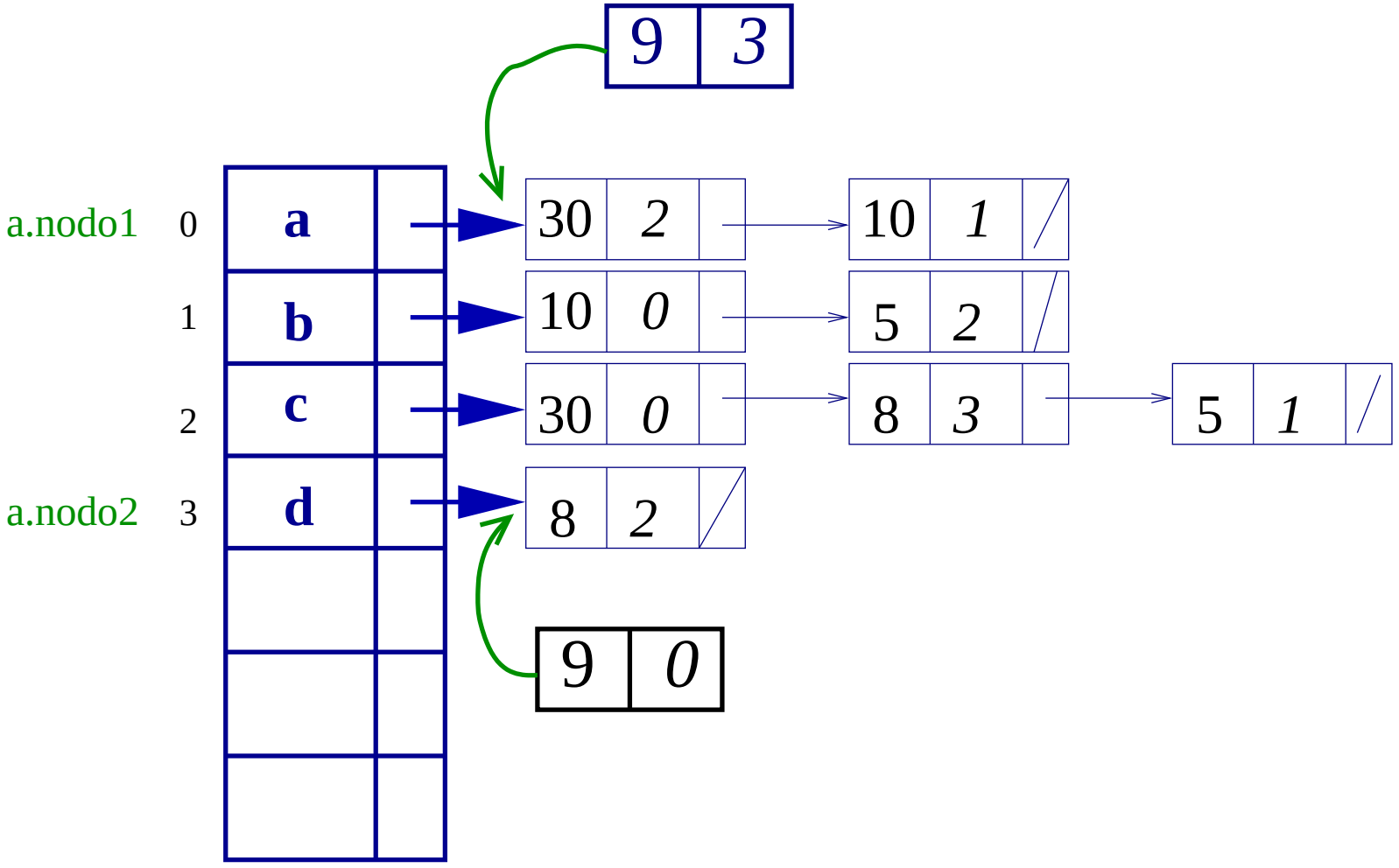
```
template <class tn, class ta>
typename grafo<tn,ta>::apuntador
grafo<tn,ta>::insert_nodo(const nodo & n) {
    int pos;
    if (numnodos < capacidad) {
      pos = numnodos;
      graf[numnodos] = n;
      numnodos++;
    }
    else pos = -1;
    return pos;
}
```

# 5.2 Listas de adyacencia (VII)

```
template <class tn, class ta>
void grafo<tn, ta>::insert_arco (const arco & a) {
    // Implementación para un grafo no dirigido
    nodoArco ar1(a.info, a.nfin);
    graf[a.norigen].listarcos.push_front(ar1);

    nodoArco ar2(a.info, a.norigen);
    graf[a.nfin].listarcos.push_front(ar2);
}
```

# 5.2 Listas de adyacencia (VII)

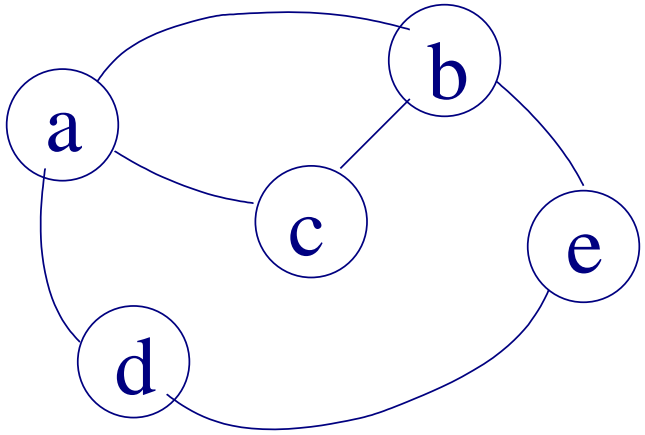


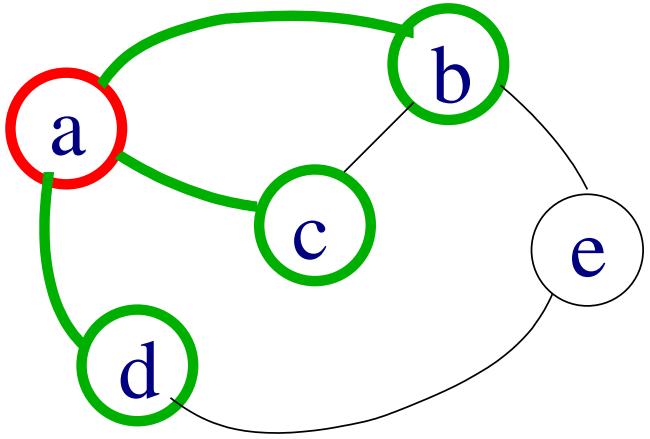
# 6 Recorrido de un grafo

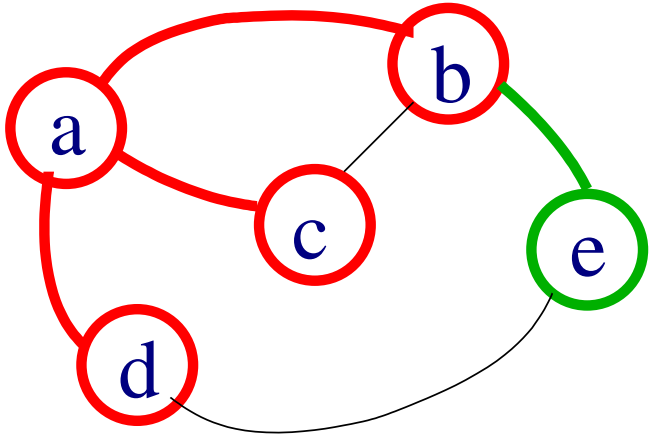
**Recorrer el grafo**: visitar o acceder sólo una vez a cada uno de los nodos.

- Al inicio del recorrido, ningún nodo ha sido visitado.
- Los nodos se van marcando como visitados:
  - Diferentes arcos pueden llevar al mismo nodo.
  - Las marcas como visitado permiten detener el recorrido y evitar un proceso infinito si hay ciclos.
- Dos algoritmos de recorrido:
  1. Recorrido en anchura.
  2. Recorrido en profundidad.
- Los mismos algoritmos se pueden adaptar a la búsqueda de un nodo en el grafo.

# 6.1 Recorrido en anchura




cola

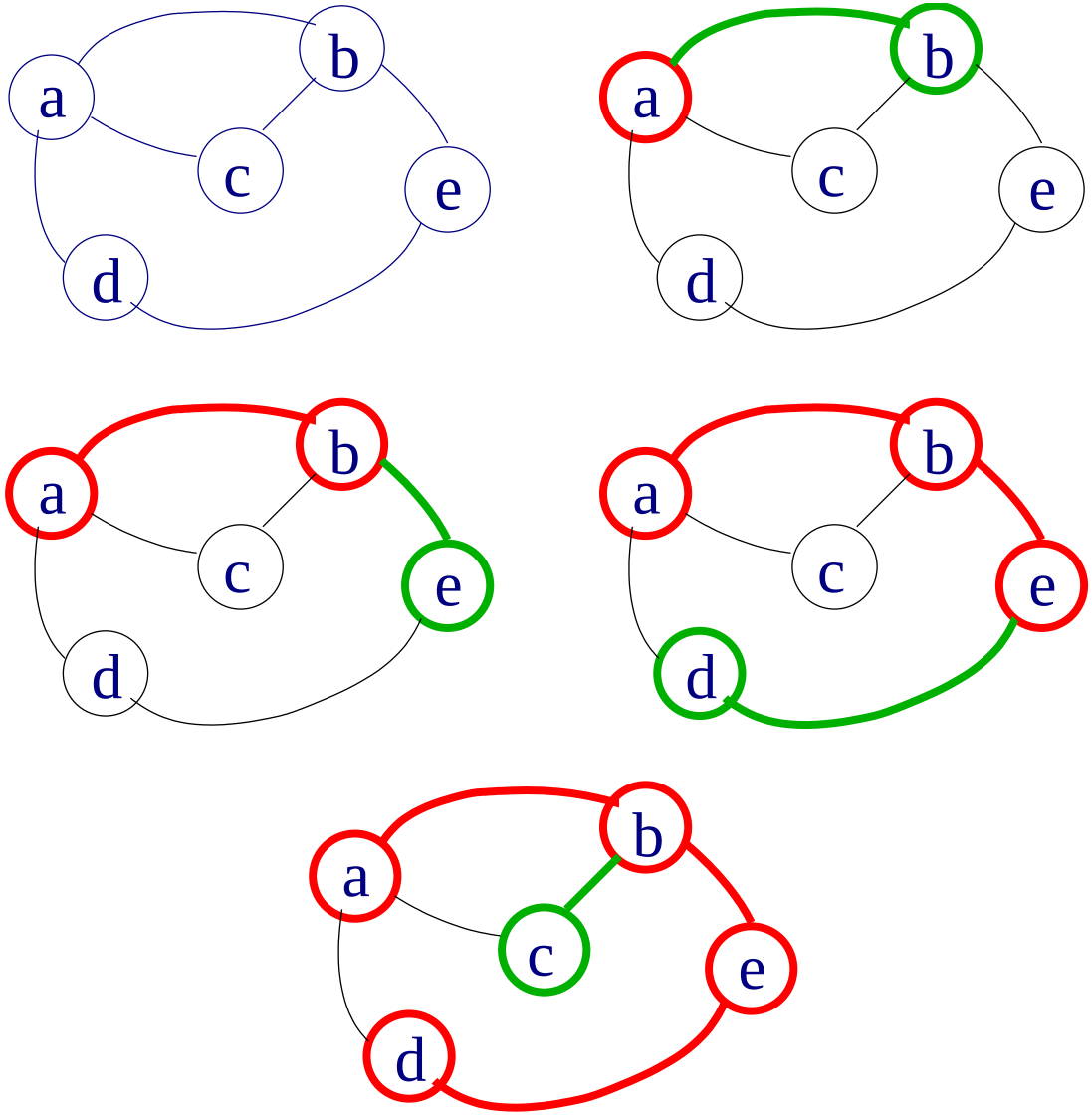
| |
|---|
| a |
| b c d |
| c d e |
| d e |
| e |
| |

# 6.1 Recorrido en anchura (II)

```
template <class tn, class ta>
void grafo<tn,ta>::BreadthFirst (apuntador p) const {
  queue<apuntador> cola;
  visitar(p);
  cola.push(p);
  mientras (!cola.empty()) {
     n = cola.front();  cola.pop();
     para (todos los nodos m adyacentes a n)
       si (no visitado(m)) {
            visitar(m);
            cola.push(m);
       }
  } }
```

Coste: $O(n^2)$ con matriz de adyacencia, $O(n+e)$ con listas de adyacencia

# 6.2 Recorrido en profundidad



# 6.2 Recorrido en profundidad (II)

```
template <class tn, class ta>
void grafo<tn,ta>::DepthFirst(apuntador p) const {
    visitar(p);
    para (cada nodo n adyacente a p)
        si (no visitado(n)) DepthFirst(n);
}
```

Coste: $O(n^2)$ con matriz de adyacencia, $O(n+e)$ con listas de adyacencia

# 7 Árboles de recubrimiento

Un **árbol de recubrimiento** de un grafo G es un subgrafo conexo y acíclico (árbol) de G que contiene todos los nodos y algunos arcos.

- Un grafo conexo de $n$ nodos necesita, al menos, $n-1$ arcos.
- Todo grafo conexo de $n$ nodos y $n-1$ arcos no contiene ciclos y es un árbol (grafo acíclico conexo).

Un grafo tiene distintos árboles de recubrimiento. Depende de:

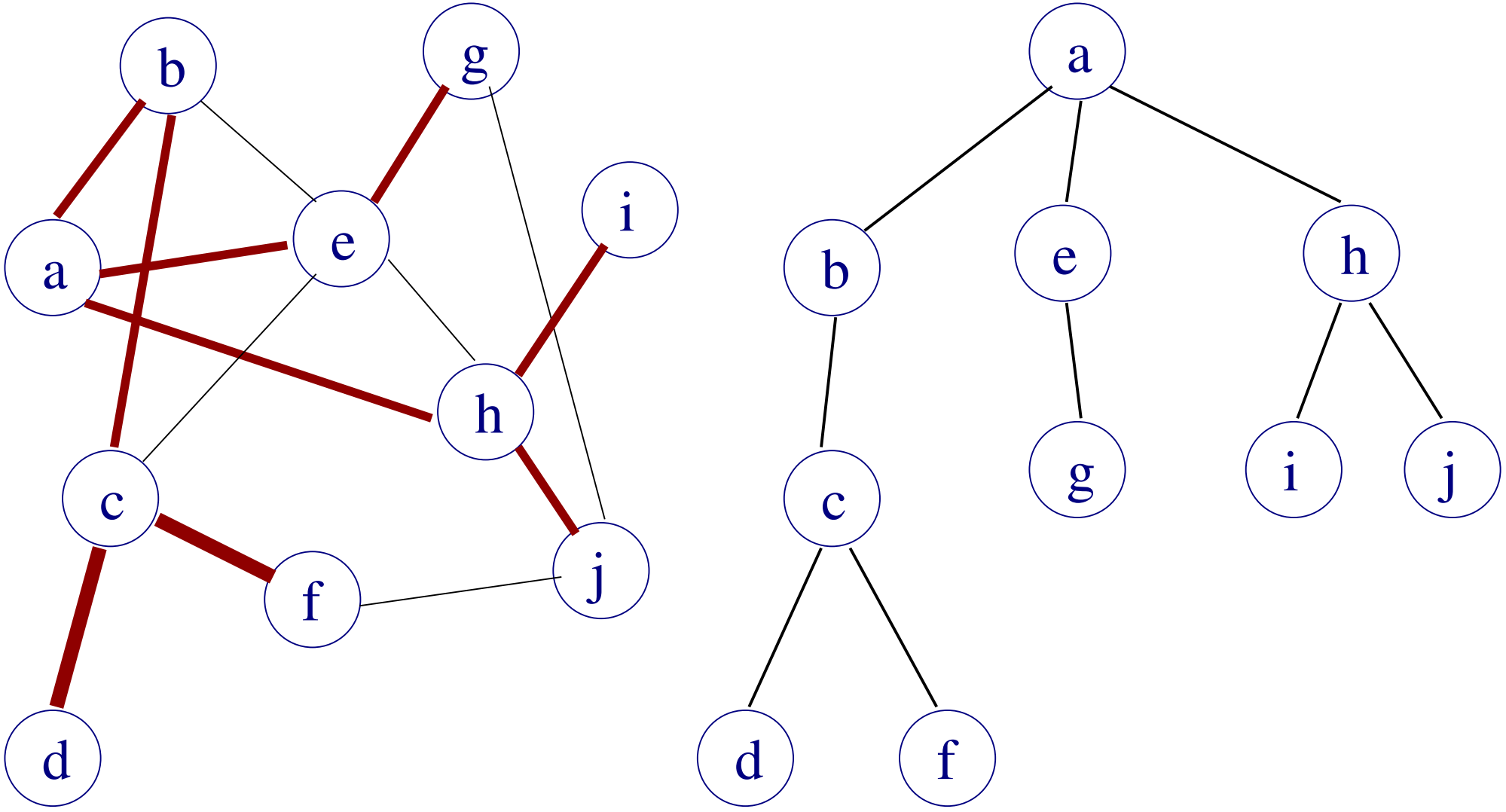
- El algoritmo de recorrido utilizado.
- El nodo origen para el recorrido.

Los algoritmos para construir un árbol de recubrimiento son los de recorrido de grafos. En este caso, hay que marcar también los arcos recorridos para diferenciar qué arcos forman parte del árbol.

# 7.1 Árboles de recubrimiento en anchura

```
template <class tn, class ta>
void grafo<tn,ta>::BFSpanningTree (apuntador p) const {
  queue<apuntador> cola;
  visitar(p);
  cola.push(p);
  mientras (!cola.empty()) {
    n = cola.front();  cola.pop();
    para (todos los nodos m adyacentes a n)
      si (no visitado(m)) {
        visitar(m);
        cola.push(m);
        marcar el arco (n,m);
      }
    }  }
```

# 7.1 Árboles de recubrimiento en anchura (II)



# 8 Árboles de recubrimiento mínimos

Sea G un grafo no dirigido con pesos en los arcos.

- El coste de un árbol de recubrimiento para G es la suma de los costes de cada uno de los arcos que forman parte del árbol de recubrimiento.
- Un árbol de recubrimiento mínimo para G es un árbol de recubrimiento con el menor coste posible.
- Hay dos algoritmos para construir árboles de recubrimiento mínimos:
  1. Algoritmo de Kruskal.
  2. Algoritmo de Prim.

# 8 Árboles de recubrimiento mínimos

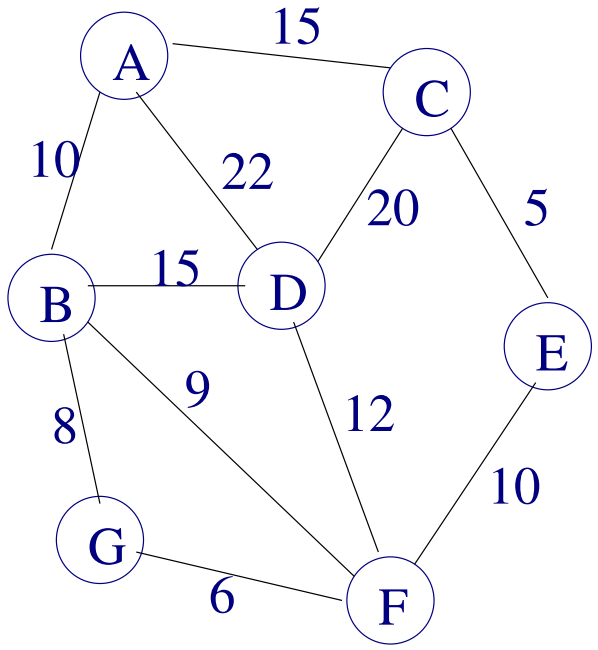
- Kruskal y Prim son ambos algoritmos voraces:
  - La solución óptima se construye por etapas.
  - En cada etapa, se toma la mejor decisión posible en ese momento (según un determinado criterio).
- Ambos algoritmos tienen las mismas restricciones:
  - Usar como mucho $n-1$ arcos del grafo.
  - Usar arcos que no produzcan ciclos.

# 8.1 Algoritmo de Kruskal

`E`: conjunto inicialmente con todos los arcos del grafo.

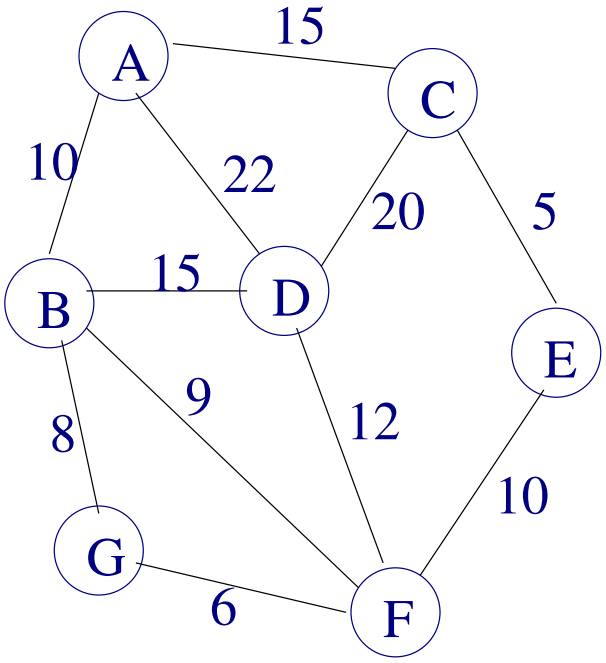
`AR`: conjunto de arcos del grafo que pertenecen al árbol de recubrimiento. Inicialmente está vacío.

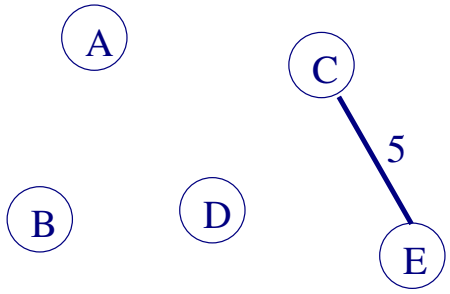
```
void grafo<tn,ta>::Kruskal(conjunto & AR) {
  AR = cjto_vacío;
  while ((AR.size() < n-1) && !E.empty()) {
     arc = E.top();    //arco de menor peso
     E.pop();
     if (arc no crea ciclos en AR)
        AR.insert(arc);
     else  //descartarlo
  }
  if (AR.size() < n-1) cout<<"grafo no conexo" << endl;
}
```

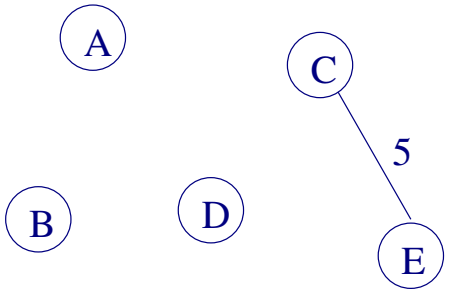


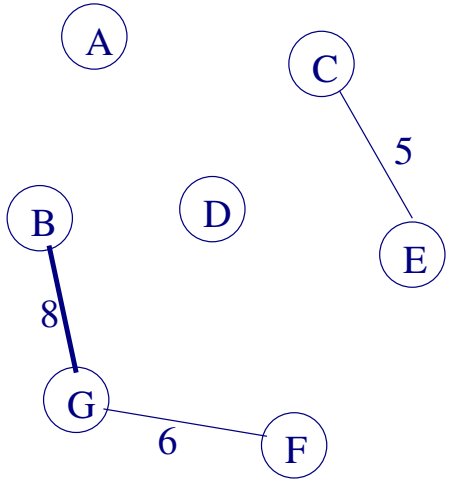
# 8.1 Algoritmo de Kruskal (II)

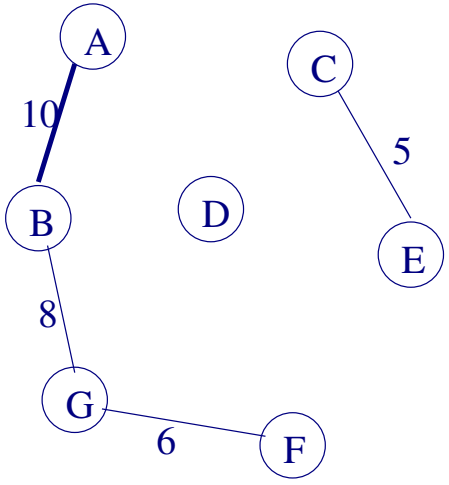
Primeras etapas del algoritmo de Kruskal:



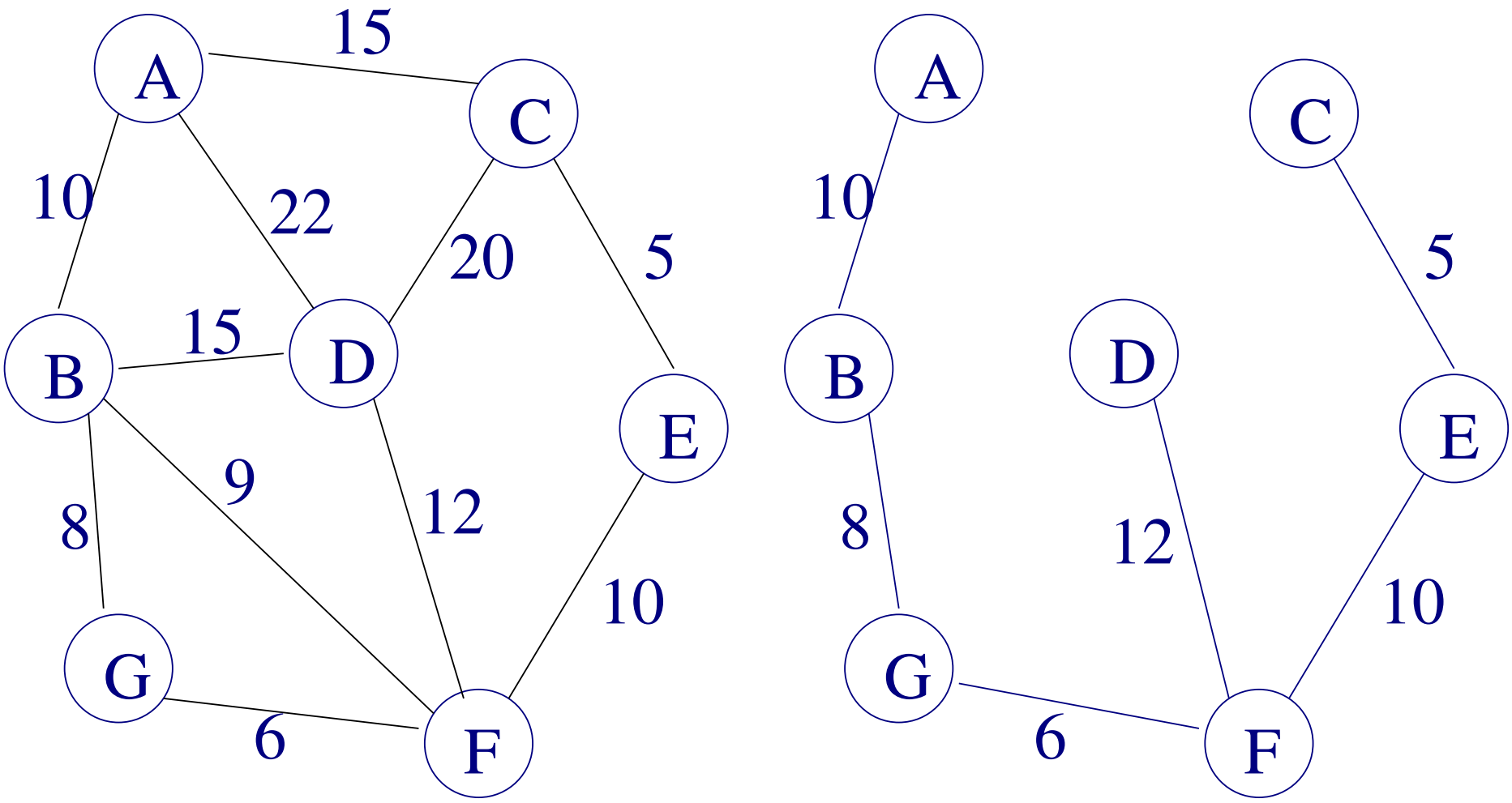



# 8.1 Algoritmo de Kruskal (III)

## Árbol de recubrimiento mínimo



# 8.1 Algoritmo de Kruskal (IV)

Implementación:

- `E` se implementa con un montículo ordenado por los pesos de los arcos. El coste de obtener el arco de menor peso y borrarlo es $O(log_2 n)$.
- Determinar de forma eficiente que un arco no forma ciclos en `AR`:
  - Todos los nodos que pertenezcan a una misma componente conexa de `AR`, están en un mismo conjunto.
  - Para que un arco no forme ciclo, debe unir nodos que pertenezcan a dos conjuntos distintos.

Coste: $O(e \cdot log_2 n)$ con listas de adyacencia

# 9 Caminos en el grafo

Los grafos se usan para representar redes de comunicaciones o de distribución. Por ejemplo, red de carreteras en un país:

- Los nodos representan las ciudades.
- Los arcos representan secciones de carretera. Estos arcos pueden tener un peso asociado, por ejemplo, la distancia entre el par de ciudades que conectan.

Se pueden plantear varios tipos de problemas:

1. ¿Hay un camino entre cada par de nodos?
2. ¿Hay un camino de `A` a `B`?
3. Si hay algún camino de `A` a `B`, ¿cuál es el camino más corto?

# 9.1 Matriz de Caminos. Cierre Transitivo

Sea `G` un grafo no ponderado.

La **matriz de cierre transitivo** o **matriz de caminos**, a la que se denotará como **A**, de un grafo G, es una matriz tal que $A[i][j] = true$ si hay un camino de longitud mayor que 0 del nodo $i$ al nodo $j$. En otro caso, $A[i][j] = false$.

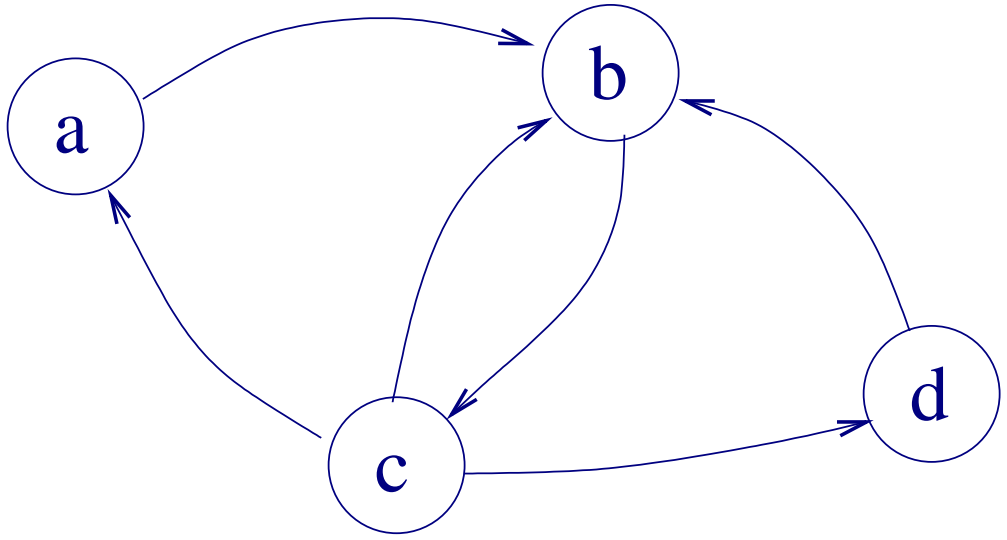
Debido a la existencia de ciclos en un grafo, se pueden dar caminos de longitud infinita.

**Teorema de cierre transitivo**: Sea G un grafo con $n$ nodos. Si entre dos nodos de G, $i$ y $j$, existe un camino de longitud $m > n$, entonces debe existir entre ellos otro camino de longitud $\leq n$, donde la longitud será $n$ si y solo si $i == j$ (es un ciclo).

- En un grafo con $n$ nodos, cualquier camino simple está formado como máximo por $n-1$ arcos.
- Cualquier ciclo tiene longitud menor o igual que $n$.

# 9.1 Matriz de Caminos. Cierre Transitivo (II)

La **matriz de adyacencia**, $Ady$, de un grafo G indica todos los caminos de longitud 1 en ese grafo.




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

- $Ady[i][k] = 1$, hay un camino de longitud 1 entre $i$ y $k$.
- $Ady[k][j] = 1$, hay camino de longitud 1 entre $k$ y $j$.

Si $Ady[i][k]\ AND\ Ady[k][j] = 1$, entonces hay un camino de longitud 2 de $i$ a $j$ (el que pasa por $k$).

# 9.1 Matriz de Caminos. Cierre Transitivo (III)

Sea G un grafo con $n$ nodos. Para cualquier par de nodos $i$, $j$, existe un camino de longitud 2 que los une si se cumple:

```
(Ady[i][1] AND Ady[1][j]) OR (Ady[i][2] AND Ady[2][j]) OR
... OR (Ady[i][n] AND Ady[n][j])
```

Esta expresión, para todos los pares $i$, $j$ de nodos, se calcula como el producto booleano de la matriz de adyacencia por sí misma:

$$A^2 = Ady \times Ady$$

$A^2[i][j] = 1$ (ó $A^2[i][j] = true$) si existe un camino de longitud 2 desde $i$ a $j$.

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

# 9.1 Matriz de Caminos. Cierre Transitivo (IV)

- Caminos de longitud 3: $A^3 = Ady * A^2$.
- ...
- Caminos de longitud $n$: $A^n = Ady * A^{n-1}$.

**Caminos de cualquier longitud: Matriz de Caminos o Cierre Transitivo**:

$$A = Ady\ OR\ A^2\ OR\ A^3\ OR\ \ldots\ OR\ A^{n-1}\ OR\ A^n$$

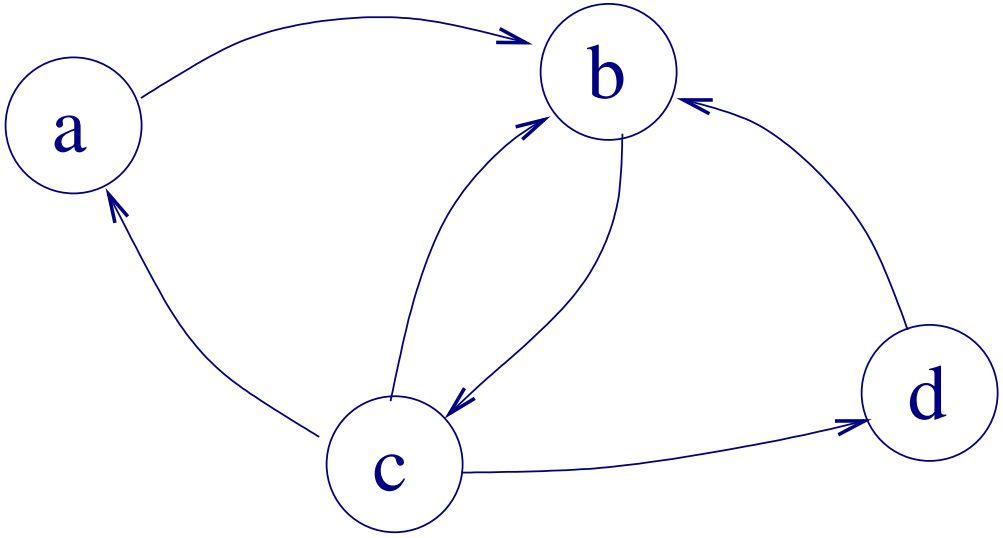
Existe un camino entre el nodo $i$ y el nodo $j$ si:

$$Ady[i][j]\ OR\ A^2[i][j]\ OR\ A^3[i][j]\ OR\ \ldots\ OR\ A^n[i][j]$$

# 9.1 Matriz de Caminos. Cierre Transitivo (V)

En el ejemplo:

$$A = Ady\ OR\ A^2\ OR\ A^3\ OR\ A^4$$




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

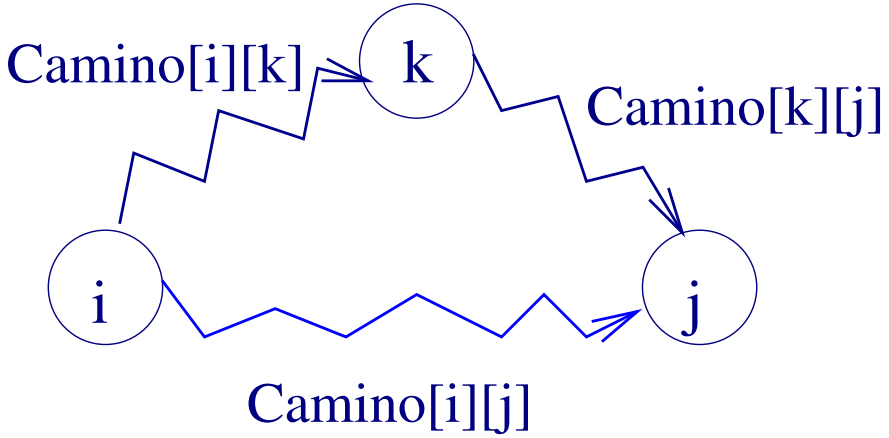
- Un 1 en la matriz indica que existe un camino de longitud menor o igual que 4 entre los nodos correspondientes.
- En este caso, el grafo es conexo.

El **algoritmo de Warshall** construye la matriz de caminos para un grafo.

En la iteración $k$, existe un camino del nodo $i$ al nodo $j$ si y sólo si:

- ya existe un camino de longitud menor entre $i$ y $j$ en la iteración $k-1$, o
- existe un camino de $i$ a $k$ y otro camino de $k$ a $j$.

$$Camino^{k}[i][j] = Camino^{k-1}[i][j]\ OR\ (Camino^{k-1}[i][k]\ AND\ Camino^{k-1}[k][j])$$




El algoritmo parte de la matriz de adyacencia del grafo: $Camino^{-1}$.

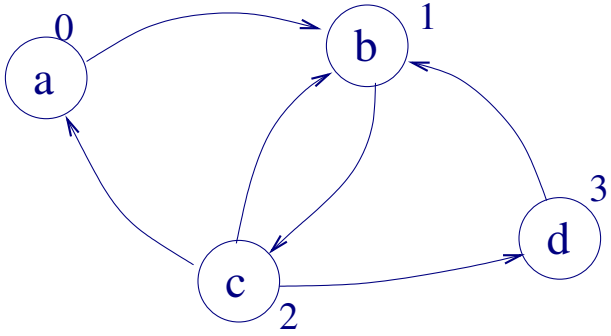
## 9.1.1 Algoritmo de Warshall (II)

- $k = 0$, $Camino^0$: matriz que indica para cada par de nodos, si existe un camino que los una y que pase, como mucho, por el nodo 0. La longitud de los caminos será $\leq 2$.
- $k = 1$, $Camino^1$: matriz que indica para cada par de nodos, si existe un camino que los una y que pase, como mucho, por los nodos numerados hasta 1. La longitud de los caminos será $\leq 3$.
- ...
- $k = n - 1$, $Camino^{n-1}$: matriz que indica para cada par de nodos, si existe un camino que los una y que pase, como mucho, por los nodos numerados hasta $n - 1$. La longitud de los caminos será $\leq n$.

## 9.1.1 Algoritmo de Warshall (III)

```
void grafo<tn,ta>::Warshall (bool** &Camino){
    //calcula la matriz de caminos
    int i, j, k;
    Camino = new (bool *)[n];
    for (i = 0; i < n; i++) Camino[i] = new bool[n];
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++) Camino[i][j] = Ady[i][j];
    for (k = 0; k < n; k++)
      for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
          Camino[i][j]=Camino[i][j] || (Camino[i][k] && Camino[k][j]);
}
```

Coste: $O(n^3)$ con matriz de adyacencia, $O(n \cdot (n+e))$ con listas de adyacencia.




$$Camino^{-1} = Ady = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array} \quad Camino^{0} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

- $Camino^{0}$: Caminos que pasan como mucho por el nodo $a$ (nodo 0). Caminos de longitud menor o igual que 2.

$$Camino^1 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 0 & 1 & \mathbf{1} & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & \mathbf{1} & 1 \\ 0 & 1 & \mathbf{1} & 0 \end{pmatrix} \end{array} \quad Camino^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} \mathbf{1} & 1 & 1 & \mathbf{1} \\ \mathbf{1} & \mathbf{1} & 1 & \mathbf{1} \\ 1 & 1 & 1 & 1 \\ \mathbf{1} & 1 & 1 & \mathbf{1} \end{pmatrix} \end{array}$$

- $Camino^1$: Caminos que pasan los nodos $a$ ó $b$ (nodos $0$ y $1$). Caminos de longitud menor o igual que 3.
- $Camino^2$: Caminos que pasan los nodos $a$, $b$ ó $c$ (nodos $0$, $1$ y $2$). Caminos de longitud menor o igual que 4.
- En este ejemplo, $Camino^3$ no aportaría nada a la matriz de caminos. Se añadiría el nodo $d$ para formar caminos.

# 9.2 Caminos de coste mínimo

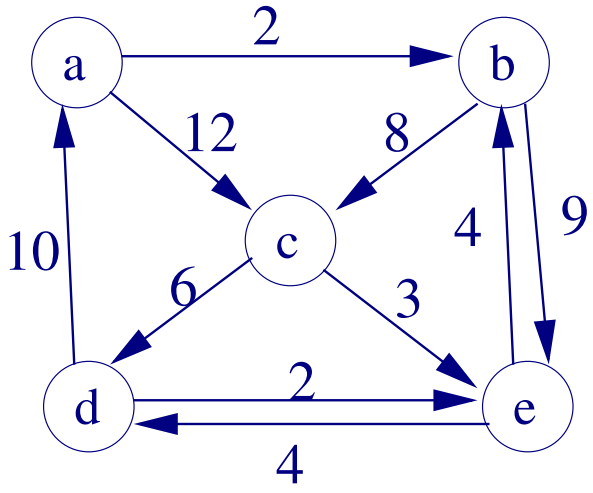
- La matriz de caminos indica si existe camino entre un par de nodos del grafo, sin embargo, no dice nada acerca del coste de dicho camino.
- Camino mínimo entre todos los pares de nodos del grafo: **Algoritmo de Floyd**. Es una generalización del algoritmo de Warshall.
- Camino mínimo desde un nodo a todos los demás: **Algoritmo de Dijkstra**.

Consideraremos grafos ponderados, tanto dirigidos como no dirigidos.

- Transitar por el arco $(i, j)$ tendrá el coste asociado a ese arco: $c(i, j)$.
- El coste de un camino $v_0, v_1, \ldots, v_k$, se calcula como: $\sum_{i=0}^{k-1} c(v_i, v_{i+1})$.
- Dados dos nodos en el grafo, $v_i$ y $v_k$, pueden existir varios caminos.
- Un camino de coste mínimo entre dos nodos es aquel que tiene coste menor o igual que cualquier otro camino entre esos nodos en el grafo.

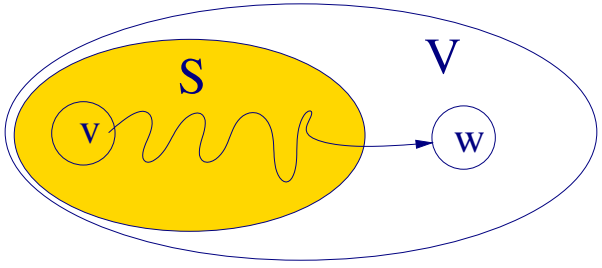
# 9.2 Caminos de coste mínimo (II)

$$Ady = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 0 & 2 & 12 & \infty & \infty \\ \infty & 0 & 8 & \infty & 9 \\ \infty & \infty & 0 & 6 & 3 \\ 10 & \infty & \infty & 0 & 2 \\ \infty & 4 & \infty & 4 & 0 \end{pmatrix} \end{array}$$




- El coste del camino de un nodo $v$ a sí mismo es 0.
- Cuando no hay camino de un nodo a otro o cuando el coste de dicho camino se desconoce, entonces el coste es $\infty$. En la práctica, debe ser un valor finito pero suficientemente elevado.

- Calcula el coste del camino mínimo entre un nodo origen y cada uno de los demás nodos del grafo.
- El grafo será ponderado con **costes positivos en todos los arcos**.
- Sea el grafo $G = (V, E)$ donde $V$ es el conjunto de todos los nodos del grafo.
- Se conocen los costes de los caminos mínimos desde el nodo inicial $v$ hasta todos los nodos de un conjunto $S$, $v \in S$.
- El camino de coste mínimo de $v$ hasta $w$, $w \in V - S$ es el camino cuyo último arco parte de un nodo de $S$ y todos los arcos anteriores pasan sólo por nodos de $S$.



## 9.2.1 Algoritmo de Dijkstra (II)

- $S$ es un conjunto de nodos. Inicialmente, $S = \{v\}$, siendo $v$ el nodo origen desde el que se calcularán los caminos mínimos.
- Se parte de la matriz de adyacencia del grafo.
- En cada iteración, se añade a $S$ el nodo $w$, $w \in V - S$, tal que $w$ tenga el camino de coste mínimo desde $v$.
- Se actualizan los costes de todos los caminos a nodos adyacentes a $w$ que no estén en $S$.
- $dist$ es un vector que contiene para cada nodo del grafo, en una determinada iteración, el coste del camino mínimo desde $v$ a ese nodo.

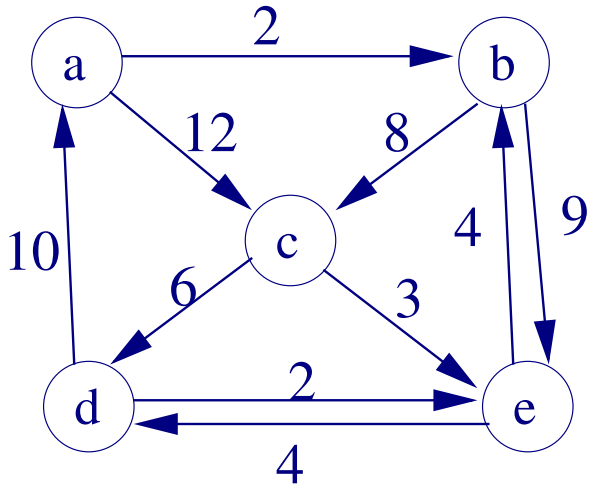
## 9.2.1 Algoritmo de Dijkstra (III)

```
void grafo<tn,ta>::Dijkstra (apuntador v, float dist[]) {
   S.insert(v);
   for (i = 0; i < n; i++) //para cada nodo, inicializar dist
     if (i != v) dist[i] = Ady[v][i];
   while (S.size() < n-1) {
     tomar w en V-S, tal que dist[w]=min{dist[i]} para todo i en V-S;
     S.insert(w);
     para cada nodo r adyacente a w
       if (!S.find(r))
           if (dist[r] > dist[w]+Ady[w][r])
               dist[r] = dist[w]+Ady[w][r];
   }
}
```

Coste: $O(n^2)$ con matriz de adyacencia, $O((n+e) \cdot log(n))$ con listas de adyacencia y montículo.

# 9.2.1 Algoritmo de Dijkstra (IV)

$$Ady = \begin{pmatrix} 0 & 2 & 12 & \infty & \infty \\ \infty & 0 & 8 & \infty & 9 \\ \infty & \infty & 0 & 6 & 3 \\ 10 & \infty & \infty & 0 & 2 \\ \infty & 4 & \infty & 4 & 0 \end{pmatrix}$$




- Iter. 0: $S = \{a\}, L = [2\ 12\ \infty\ \infty\ ]$
- Iter. 1: $S = \{a,b\}, L = [2\ 10\ \infty\ 11\ ]$
- Iter. 2: $S = \{a,b,c\}, L = [2\ 10\ 16\ 11\ ]$
- Iter. 3: $S = \{a,b,c,e\}, L = [2\ 10\ 15\ 11\ ]$

¿Cómo se aplicaría el algoritmo de Dijkstra para obtener el camino mínimo desde un nodo origen a un nodo destino?