Tema 4. Grafos

4.1. Introducción, notación y definiciones

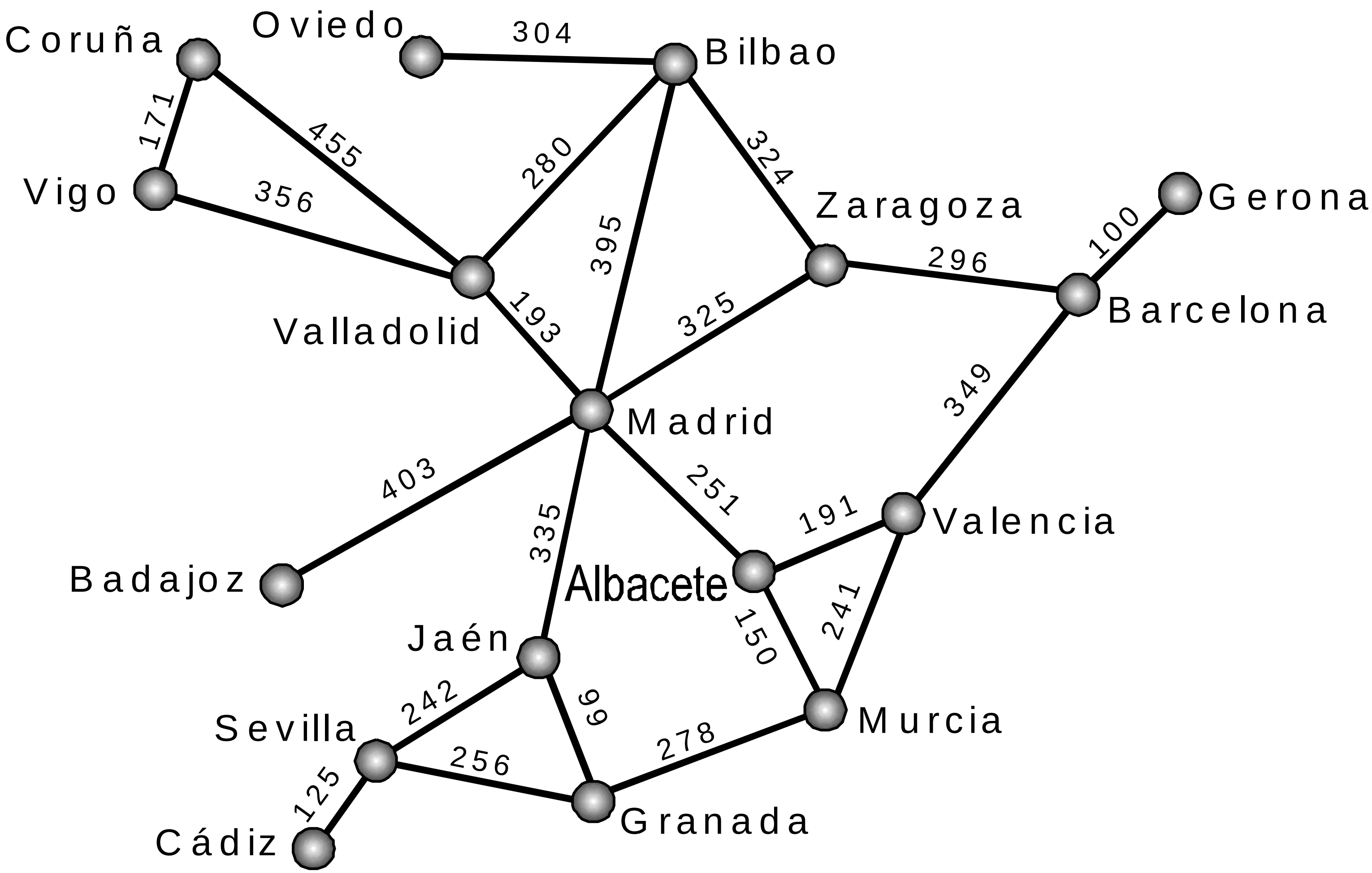
4.2. Representación de grafos

4.3. Problemas y algoritmos sobre grafos

- 4.3.1. Recorridos sobre grafos
- 4.3.2. Árboles de expansión mínimos
- 4.3.3. Problemas de caminos mínimos
- 4.3.4. Algoritmos sobre grafos dirigidos
- 4.3.5. Algoritmos sobre grafos no dirigidos
- 4.3.6. Otros problemas con grafos

4.1.1. Ejemplos de grafos

- **Ejemplo:** grafo de carreteras entre ciudades.



4.1.1. Ejemplos de grafos

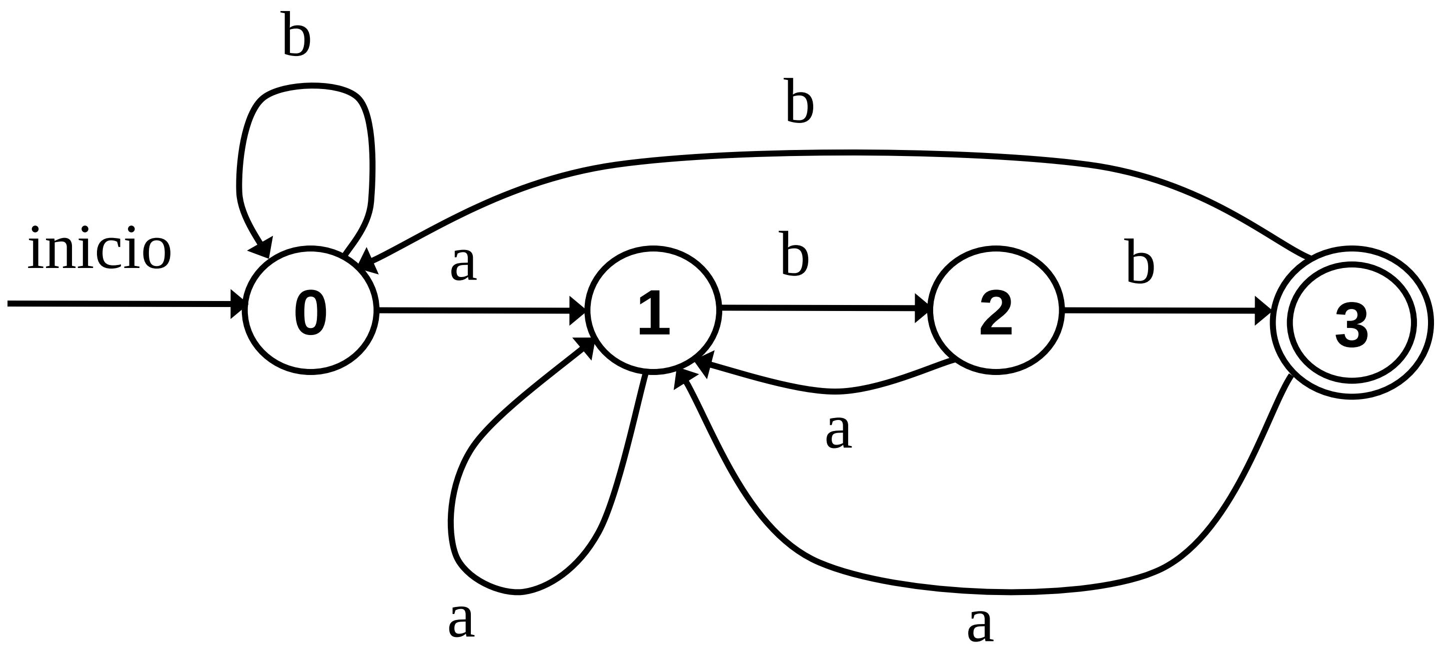
- **Ejemplo:** grafo de carreteras entre ciudades.

Problemas

- ¿Cuál es el camino más corto de Murcia a Badajoz?
- ¿Existen caminos entre todos los pares de ciudades?
- ¿Cuál es la ciudad más lejana a Barcelona?
- ¿Cuál es la ciudad más céntrica?
- ¿Cuántos caminos distintos existen de Sevilla a Zaragoza?
- ¿Cómo hacer un tour entre todas las ciudades en el menor tiempo posible?

4.1.1. Ejemplos de grafos

- **Ejemplo:** grafo de transiciones de un AFD.



4.1.1. Ejemplos de grafos

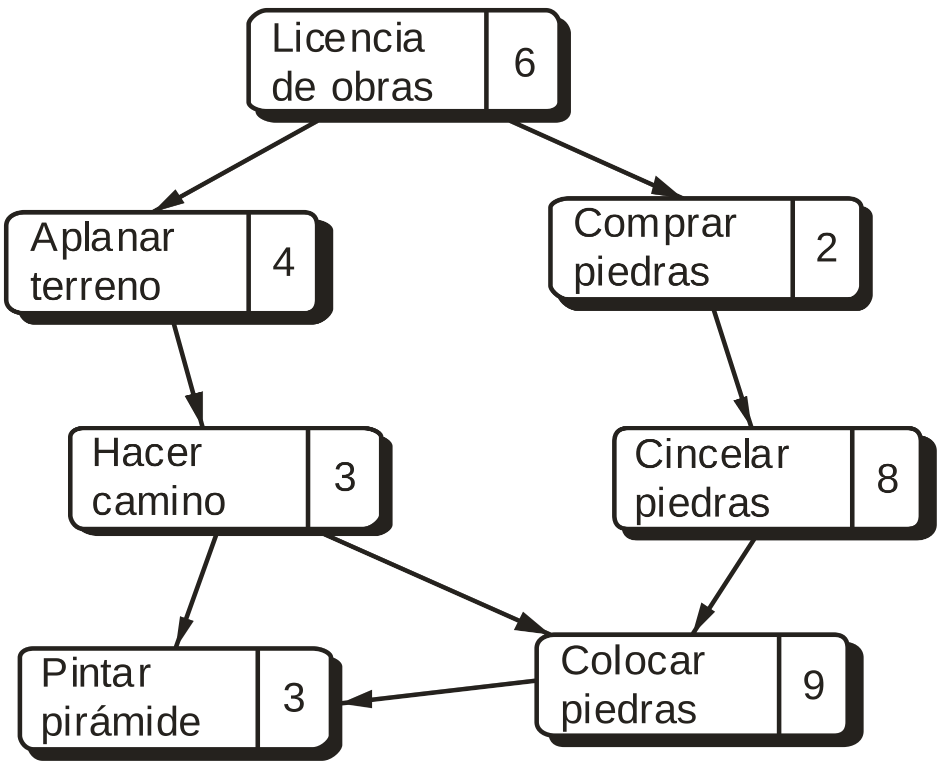
- **Ejemplo:** grafo de transiciones de un AFD.

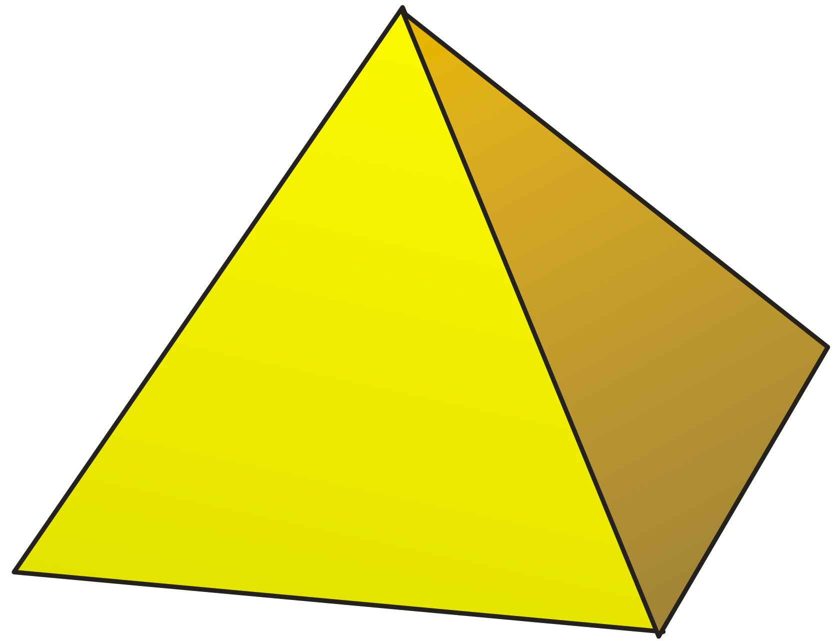
Problemas

- ¿La expresión: a b b a b a b b b a, es una expresión válida del lenguaje?
- ¿Cuál es la expresión válida más corta?
- Transformar el grafo en una expresión regular y viceversa.

4.1.1. Ejemplos de grafos

- **Ejemplo:** grafo de planificación de tareas.





4.1.1. Ejemplos de grafos

- **Ejemplo:** grafo de planificación de tareas.

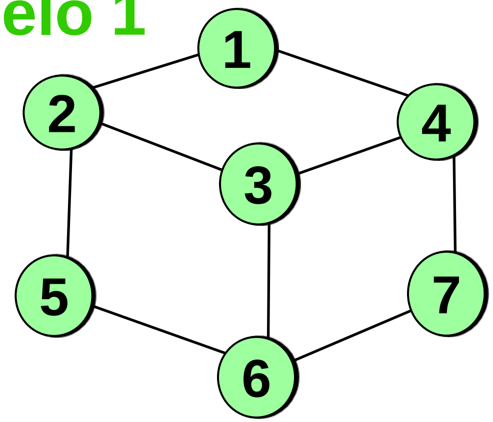
Problemas

- ¿En cuanto tiempo, como mínimo, se puede construir la pirámide?
- ¿Cuándo debe empezar cada tarea en la planificación óptima?
- ¿Qué tareas son más críticas (es decir, no pueden sufrir retrasos)?
- ¿Cuánta gente necesitamos para acabar las obras?

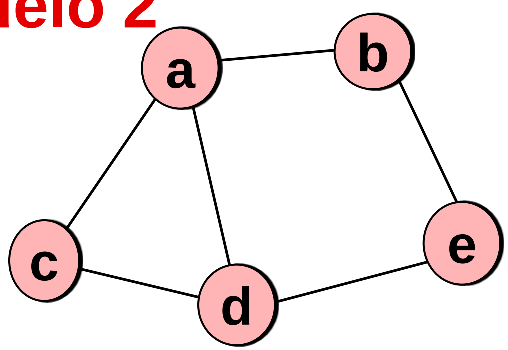
4.1.1. Ejemplos de grafos

- **Ejemplo:** grafo asociado a un dibujo de líneas.

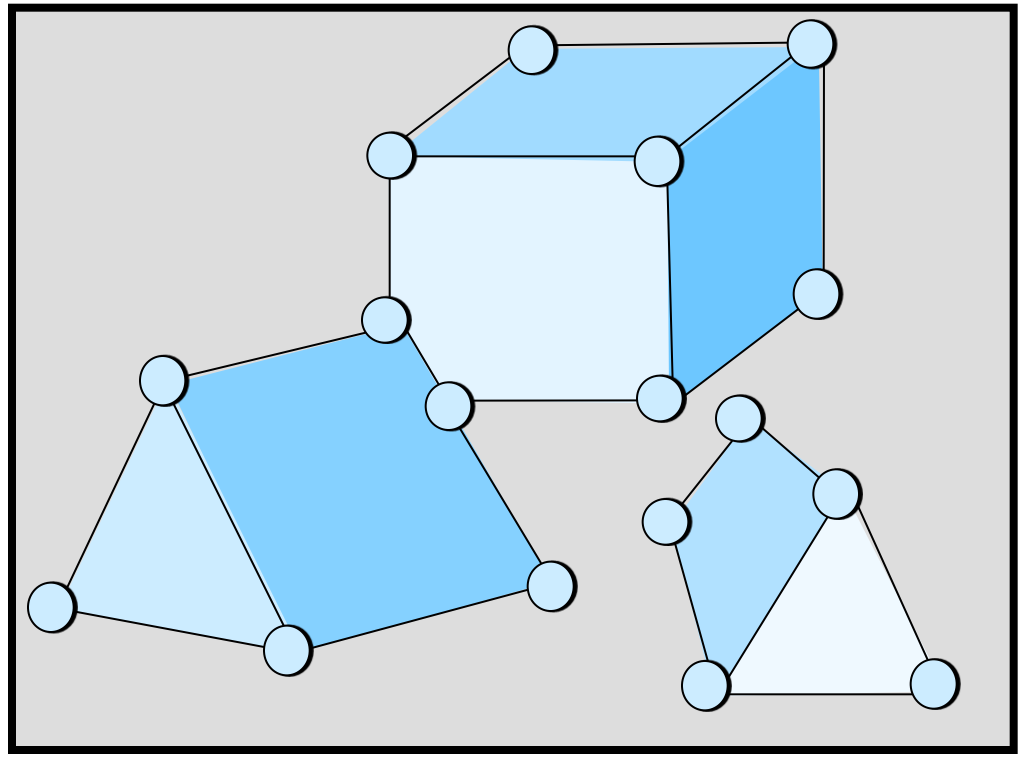
Modelo 1



Modelo 2



Escena



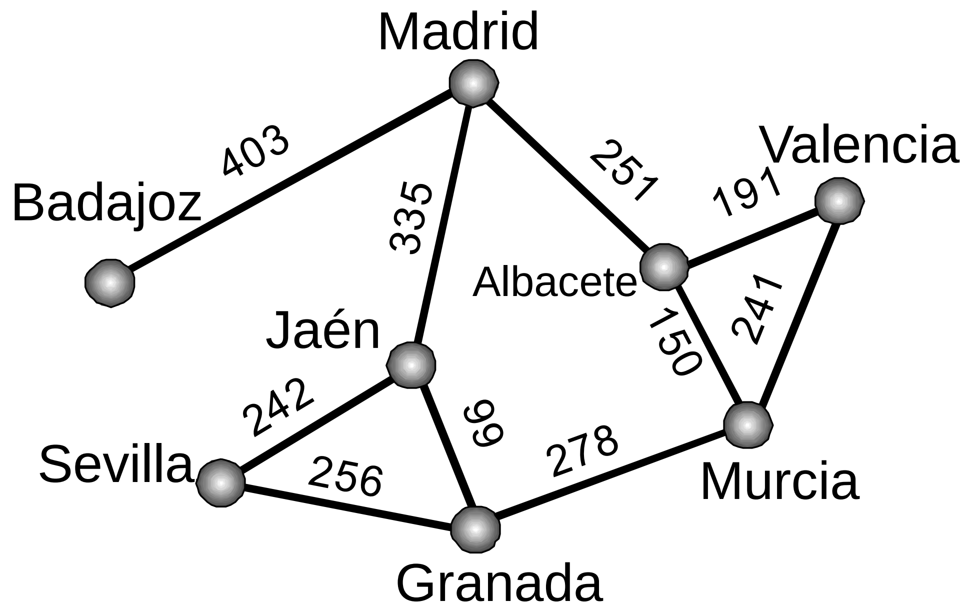
4.1.1. Ejemplos de grafos

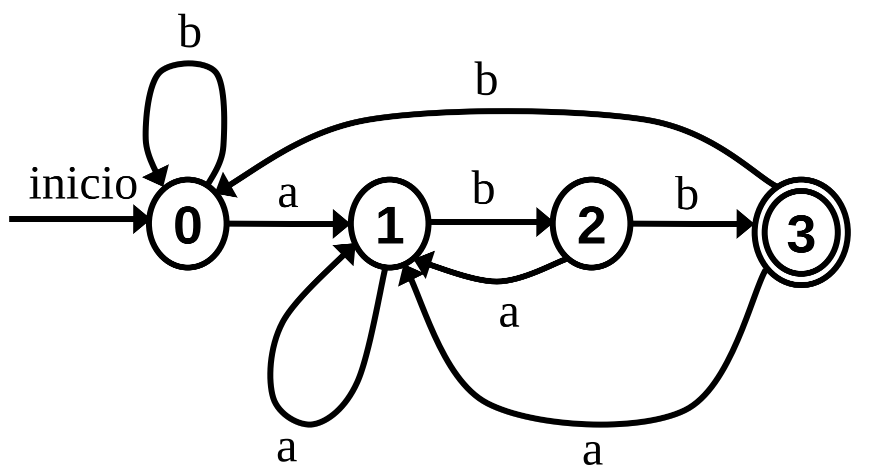
- **Ejemplo:** grafo de asociado a un dibujo de líneas.

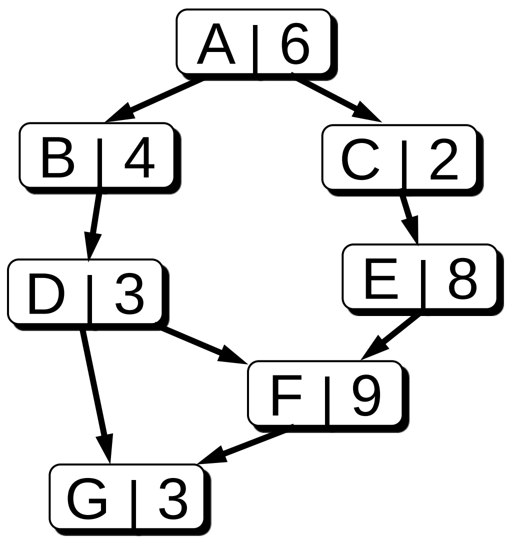
Problemas

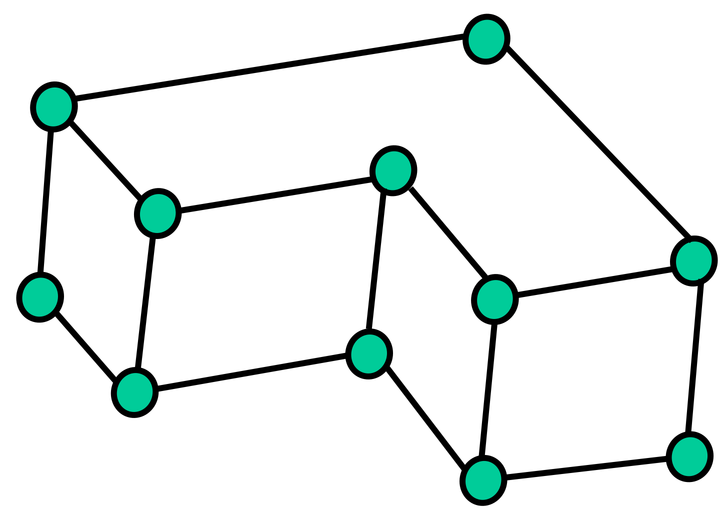
- ¿Cuántos grupos hay en la escena?
- ¿Qué objetos están visibles en la escena y en qué posiciones?
- ¿Qué correspondencia hay entre puntos del modelo y de la escena observada?
- ¿Qué objetos son isomorfos?

4.1.1. Ejemplos de grafos







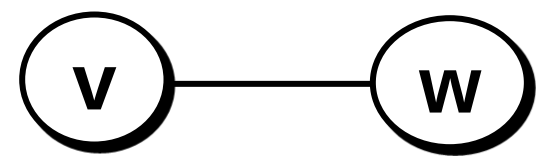


4.1. Introducción y definiciones

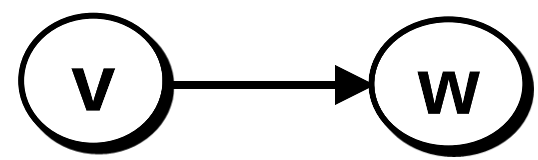
- **Un grafo G** es una tupla G= (V, A), donde V es un conjunto no vacío de **vértices** o **nodos** y A es un conjunto de **aristas** o **arcos**.
- Cada **arista** es un par (v, w), donde v, w $\in$ V.

Tipos de grafos

- **Grafo no dirigido.**
Las aristas no están ordenadas:
(v, w) = (w, v)



- **Grafos dirigidos (o digrafos).**
Las aristas son pares ordenados:
<v, w> $\neq$ <w, v>



<v, w> $\Rightarrow$ w = cabeza de la arista, v = cola.

4.1.2. Terminología de grafos

- **Nodos adyacentes a un nodo v:** todos los nodos unidos a v mediante una arista.
- En grafos dirigidos:
  - **Nodos adyacentes a v:** todos los w con $<v, w> \in A$.
  - **Nodos adyacentes de v:** todos los u con $<u, v> \in A$.

- Un grafo está **etiquetado** si cada arista tiene asociada una etiqueta o valor de cierto tipo.
- **Grafo con pesos:** grafo etiquetado con valores numéricos.
- **Grafo etiquetado:** G= (V, A, W), con **W**: $A \rightarrow$ TipoEtiq

4.1.2. Terminología de grafos

- **Camino de un vértice w_1 a w_q:** es una secuencia $w_1, w_2, ..., w_q \in V$, tal que todas las aristas $(w_1, w_2), (w_2, w_3), ..., (w_{q-1}, w_q) \in A$.
- **Longitud de un camino:** número de aristas del camino = nº de nodos -1.
- **Camino simple:** aquel en el que todos los vértices son distintos (excepto el primero y el último que pueden ser iguales).
- **Ciclo:** es un camino en el cual el primer y el último vértice son iguales. En grafos no dirigidos las aristas deben ser diferentes.
- Se llama **ciclo simple** si el camino es simple.

4.1.2. Terminología de grafos

- Un **subgrafo** de G=(V, A) es un grafo G'=(V', A') tal que $V' \subseteq V$ y $A' \subseteq A$.
- Dados dos vértices v, w, se dice que están **conectados** si existe un camino de v a w.
- Un grafo es **conexo** (o **conectado**) si hay un camino entre cualquier par de vértices.
- Si es un grafo dirigido, se llama **fuertemente conexo**.
- Una **componente (fuertemente) conexa** de un grafo G es un subgrafo maximal (fuertemente) conexo.

4.1.2. Terminología de grafos

- Un grafo es **completo** si existe una arista entre cualquier par de vértices.
- Para **n** nodos, ¿cuántas aristas tendrá un grafo completo (dirigido o no dirigido)?

- **Grado de un vértice v:** número de arcos que inciden en él.
- Para grafos dirigidos:
  - **Grado de entrada de v:** nº de aristas con <x, v>
  - **Grado de salida de v:** nº de aristas con <v, x>

4.1.3. Operaciones elementales con grafos

- Crear un **grafo vacío** (o con **n** vértices).
- **Insertar** un nodo o una arista.
- **Eliminar** un nodo o arista.
- **Consultar** si existe una arista (obtener la etiqueta).
- **Iteradores** sobre las aristas de un nodo:

  para todo nodo *w* adyacente a *v* **hacer**

  acción sobre *w*

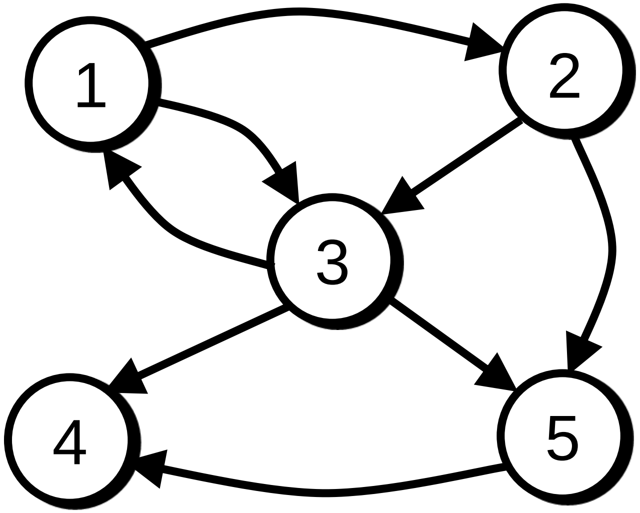
  para todo nodo *w* adyacente de *v* **hacer**

  acción sobre *w*

Mucho menos frecuente

4.2. Representación de grafos

- **Representación de grafos**:
  - Representación del conjunto de nodos, V.
  - Representación del conjunto de aristas, A.

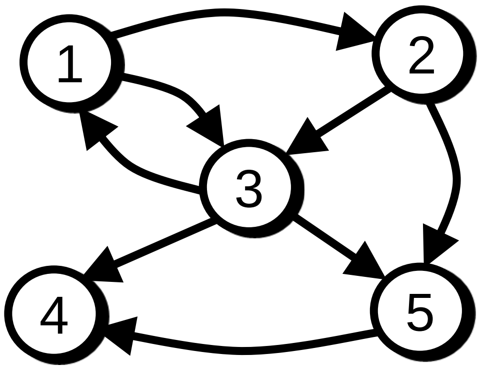


- **Ojo:** las aristas son relaciones “muchos a muchos” entre nodos...

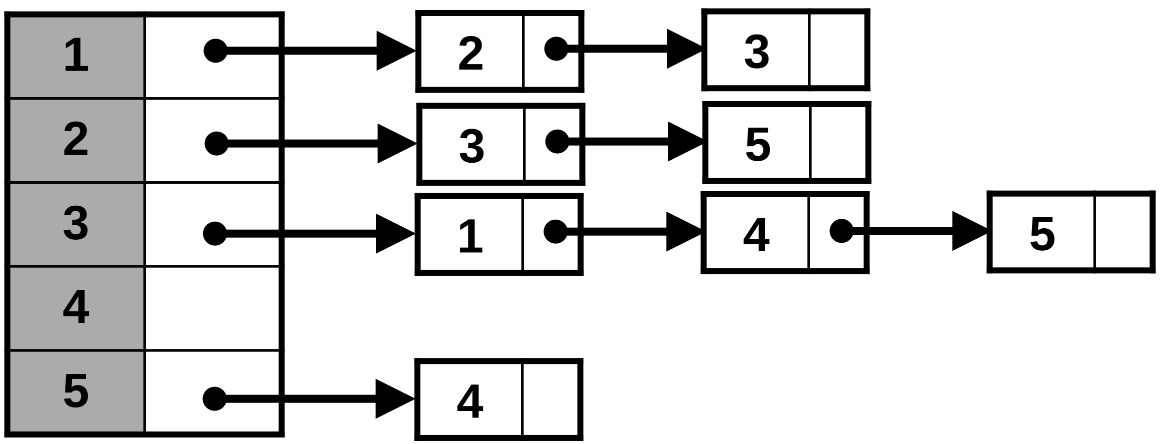
4.2. Representación de grafos

- **Representación del conjunto de aristas, A.**
  - Mediante **matrices de adyacencia**

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



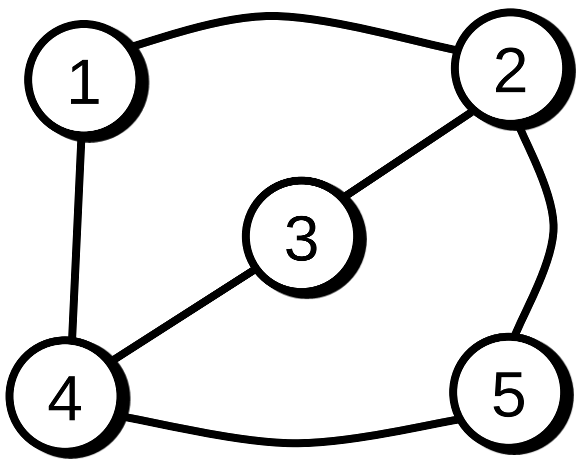
  - Mediante **listas de adyacencia**



4.2.1. Matrices de adyacencia

tipo GrafoNoEtiq= **array** [1..n, 1..n] **de** 0..1

- Sea M de tipo GrafoNoEtiq, G= (V, A).
- M[v, w] = cierto $\Leftrightarrow$ (v, w) $\in$ A



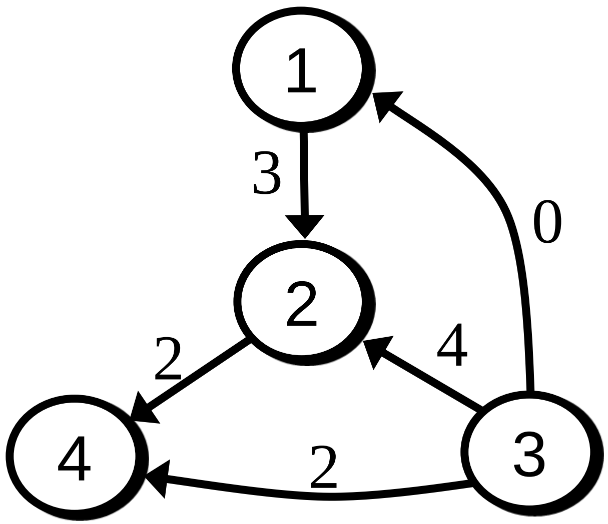
M	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	0	1
3	0	1	0	1	0
4	1	0	1	0	1
5	0	1	0	1	0

- Grafo no dirigido → matriz simétrica: M[i, j] = M[j, i].
- **Resultado:** se desperdicia la mitad de la memoria.

4.2.1. Matrices de adyacencia

- **Grafos etiquetados:**

  tipo GrafoEtiq[E]= **array** [1..n, 1..n] **de** E
- El tipo E tiene un valor infinito, para el caso de no existir arista.



M	1	2	3	4
1	∞	3	∞	∞
2	∞	∞	∞	2
3	0	4	∞	2
4	∞	∞	∞	∞

- ¿Cómo serían los iteradores: **para todo** adyacente a, y adyacente de? ¿Y contar número de aristas?
- ¿Cuánto es el tiempo de ejecución?

4.2.1. Matrices de adyacencia

Uso de memoria

- k_2 bytes/etiqueta
- **Memoria usada:** k_2n^2

Ventajas

- Representación y operaciones muy sencillas.
- Eficiente para el acceso a una arista dada.

Inconvenientes

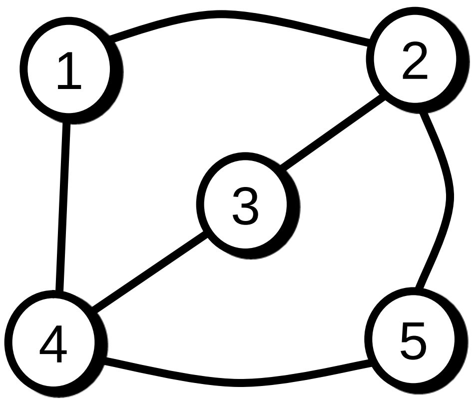
- El número de nodos del grafo no puede cambiar.
- Si hay muchos nodos y pocas aristas ($a \ll n^2$) se desperdicia mucha memoria (matriz *escasa*).

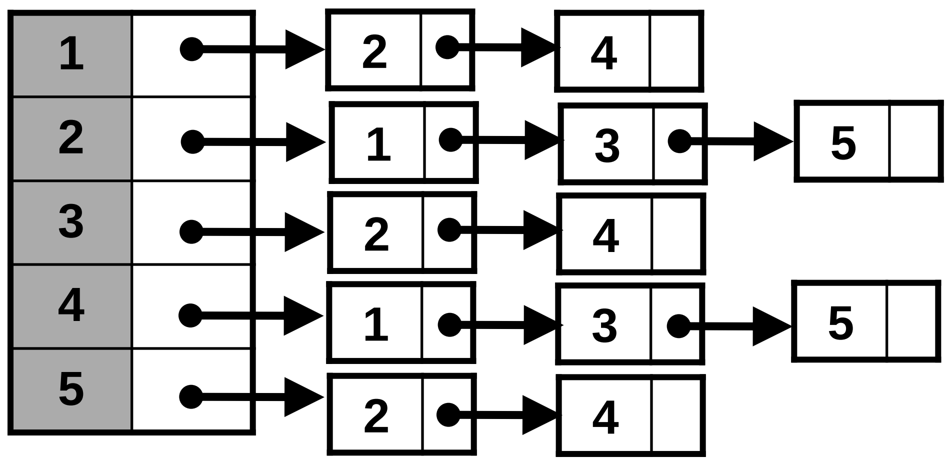
4.2.2. Listas de adyacencia

tipo Nodo= entero (1..n)

tipo GrafoNoEtiq= **array** [1..n] **de** Lista[Nodo]

- Sea R de tipo GrafoNoEtiq, G= (V, A).
- La lista R[v] contiene los w tal que (v, w) $\in$ A.



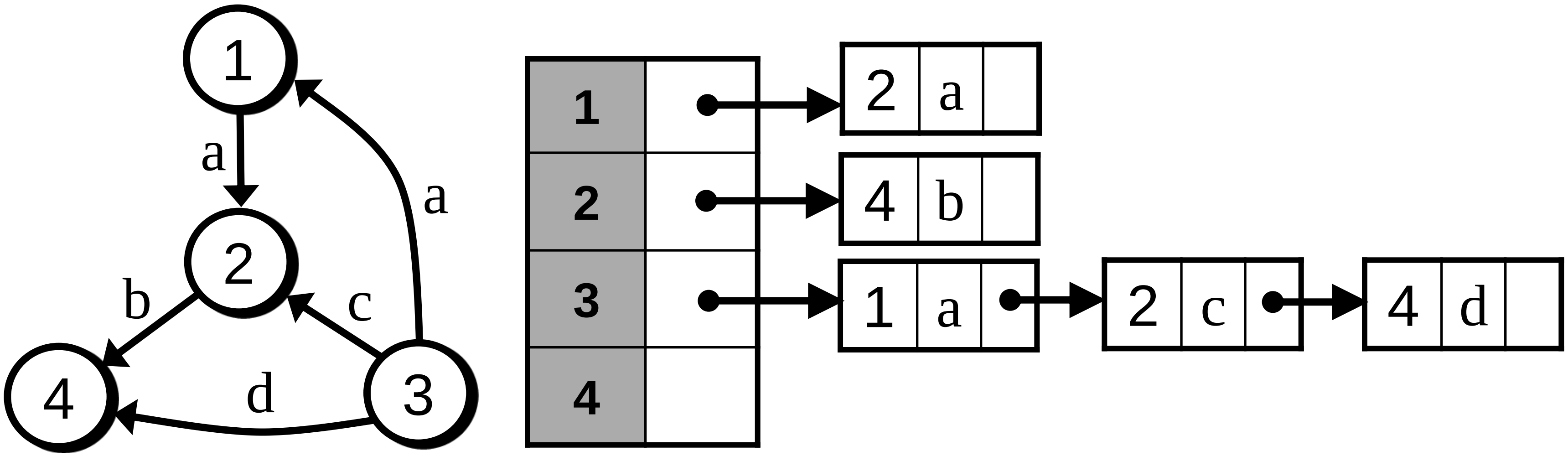


- Grafo no dirigido → las aristas están repetidas.
- **Resultado:** también se desperdicia memoria.

4.2.2. Listas de adyacencia

- **Grafos etiquetados:**

  tipo GrafoEtiq[E]= **array** [1..n] **de** Lista[Nodo,E]



- ¿Cómo serían los iteradores: **para todo** adyacente a, y adyacente de? ¿Y contar número de aristas?
- ¿Cuánto es el orden de complejidad? Se suponen: **n** nodos y **a** aristas.

4.2.2. Listas de adyacencia

Uso de memoria

- k_1 bytes/puntero, k_2 bytes/etiqueta o nodo
- **Memoria usada:** $k_1(n+a) + 2k_2a$
- Con matrices de adyacencia: k_2n^2
- ¿Cuál usa menos memoria?

Ventajas

- Más adecuada cuando $a<<n^2$.

Inconvenientes

- Representación más compleja.
- Es ineficiente para encontrar las aristas que llegan a un nodo. Alternativa: usar estructuras de listas múltiples.

4.3. Problemas y algoritmos sobre grafos

4.3.1. Recorridos sobre grafos

4.3.2. Árboles de expansión mínimos

4.3.3. Problemas de caminos mínimos

4.3.4. Algoritmos sobre grafos dirigidos

4.3.5. Algoritmos sobre grafos no dirigidos

4.3.6. Otros problemas con grafos

4.3.1. Recorridos sobre grafos

- Idea similar al recorrido en un árbol.
- Se parte de un nodo dado y se visitan los vértices del grafo de manera ordenada y sistemática, *moviéndose* por las aristas.

- **Tipos de recorridos**:
  - **Búsqueda primero en profundidad.** Equivalente a un recorrido en preorden de un árbol.
  - **Búsqueda primero en amplitud o anchura.** Equivalente a recorrer un árbol por niveles.

- Los recorridos son una **herramienta** útil para resolver muchos problemas sobre grafos.

4.3.1. Recorridos sobre grafos

- El recorrido puede ser tanto para grafos dirigidos como no dirigidos.
- Es necesario llevar una cuenta de los nodos visitados y no visitados.

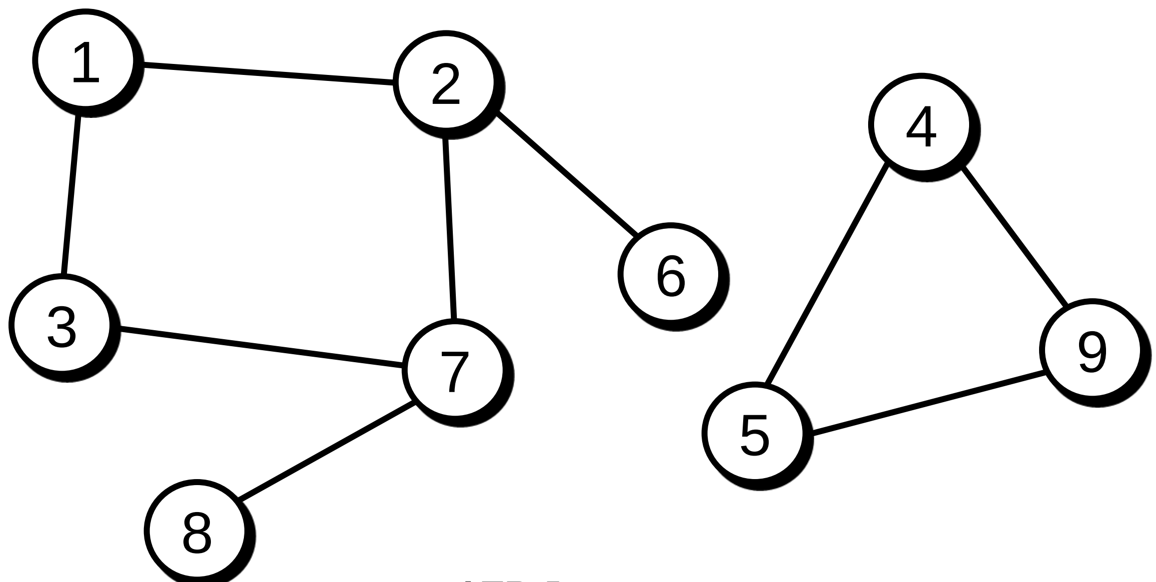
```
var
    marca: array [1, ..., n] de (visitado, noVisitado)

operación BorraMarcas
    para i:= 1, ..., n hacer
        marca[i]:= noVisitado
```

4.3.1.1. Búsqueda primero en profundidad

```
operación bpp (v: nodo)
    marca[v]:= visitado
    para cada nodo w adyacente a v hacer
        si marca[w] == noVisitado entonces
            bpp(w)
    finpara

operación BúsquedaPrimeroEnProfundidad
    BorraMarcas
    para v:= 1, ..., n hacer
        si marca[v] == noVisitado entonces
            bpp(v)
    finpara
```

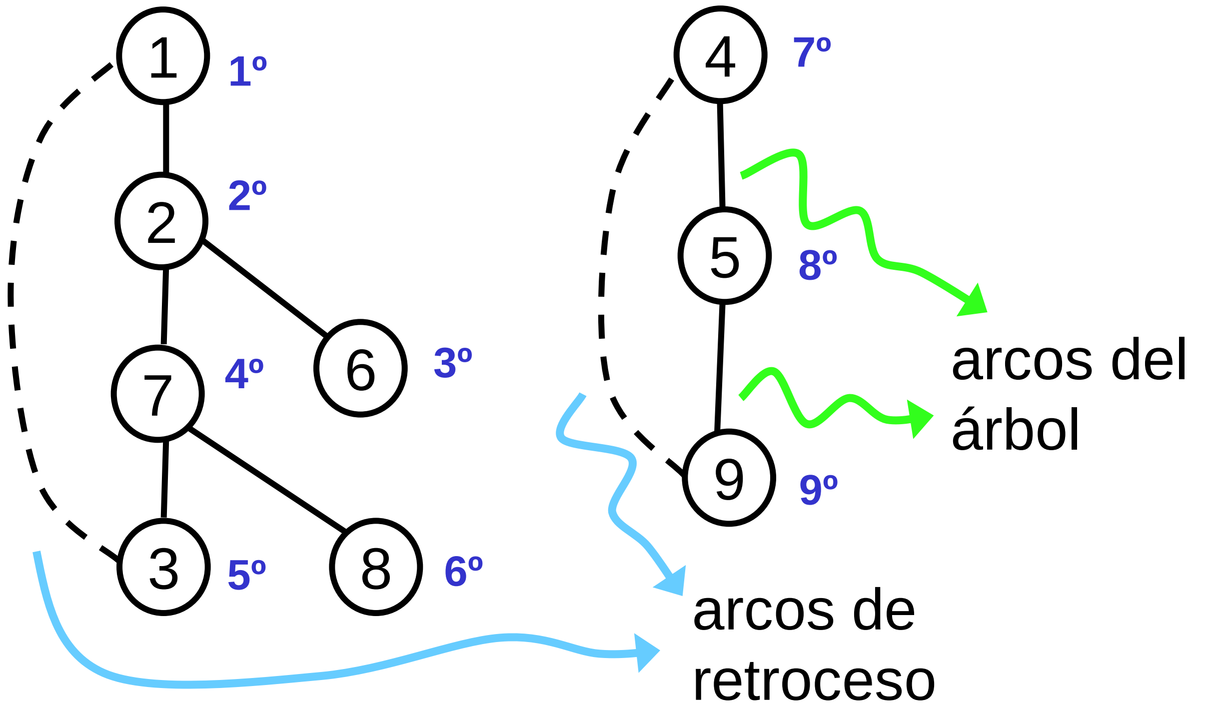
4.3.1.1. Búsqueda primero en profundidad

- El recorrido **no es único:** depende del nodo inicial y del orden de visita de los adyacentes.
- El orden de visita de unos nodos a partir de otros puede ser visto como un árbol: **árbol de expansión en profundidad asociado al grafo**.
- Si aparecen varios árboles: **bosque de expansión en profundidad**.
- **Ejemplo.** Grafo no dirigido.



4.3.1.1. Búsqueda primero en profundidad

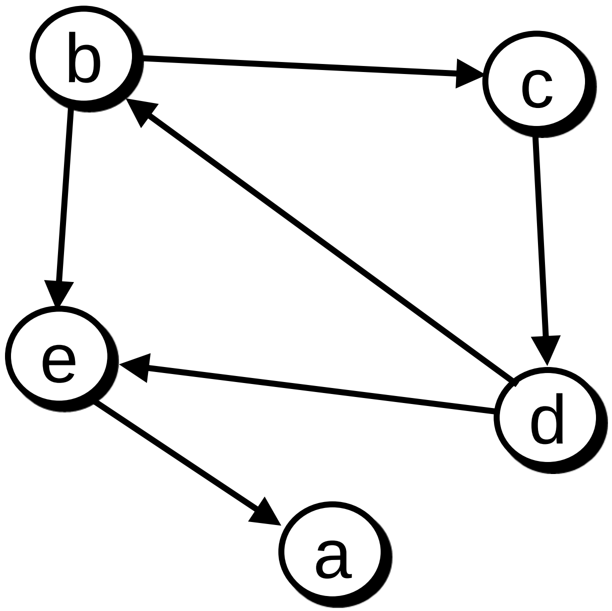
- **Bosque de expansión en profundidad**



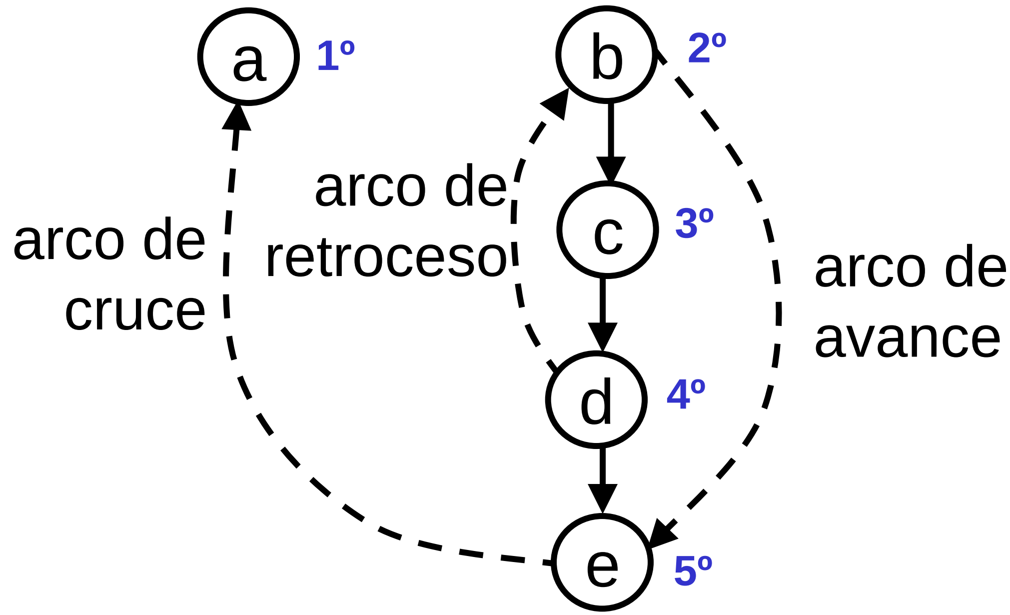
- **Arcos de retroceso:** si marca[v] == noVisitado ...
  → se detectan cuando la condición es falsa.

4.3.1.1. Búsqueda primero en profundidad

- **Ejemplo:** grafo dirigido.



Bosque de expansión



- ¿Cuánto es el tiempo de ejecución de la bpp?
- Imposible predecir las llamadas en cada ejecución.
- **Solución:** medir el "trabajo total realizado".

4.3.1.2. Búsqueda primero en amplitud (o anchura)

- **Búsqueda en amplitud** empezando en un nodo v:
  - Primero se visita v.
  - Luego se visitan todos sus adyacentes.
  - Luego los adyacentes de estos y así sucesivamente.
- El algoritmo utiliza una **cola de vértices**.
- Operaciones básicas:
  - Sacar un elemento de la cola.
  - Añadir a la cola sus adyacentes no visitados.

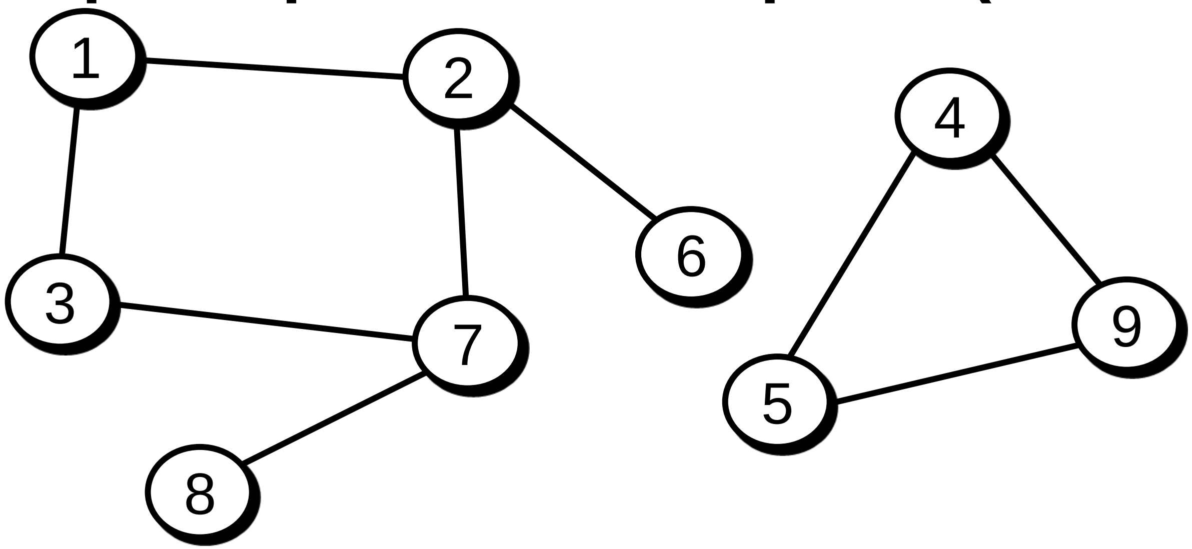
```
operación BúsquedaPrimeroEnAmplitud
   BorraMarcas
   para v:= 1, ..., n hacer
         si marca[v] = noVisitado entonces
               bpa(v)
```

4.3.1.2. Búsqueda primero en amplitud (o anchura)

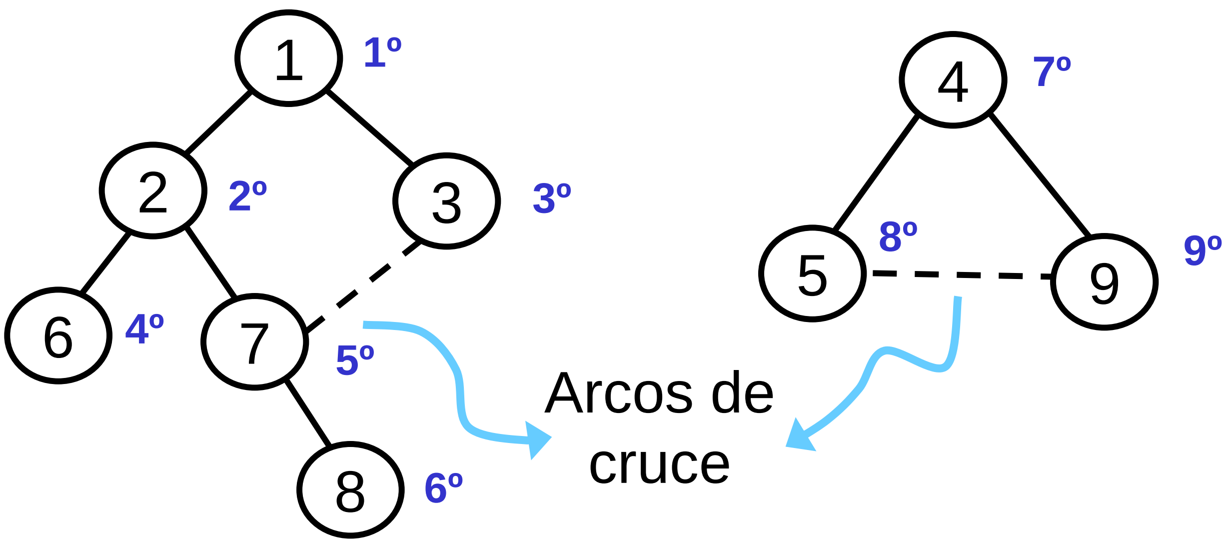
```
operación bpa (v: Nodo)
var  C: Cola[Nodo]
         x, y: Nodo

    marca[v]:= visitado
    InsertaCola (v, C)
    mientras NOT EsVacíaCola (C) hacer
        x:= FrenteCola (C)
        SuprimirCola (C)
        para cada nodo y adyacente a x hacer
            si marca[y] == noVisitado entonces
                marca[y]:= visitado
                InsertaCola (y, C)
            finsi
        finpara
    finmientras
```

4.3.1.2. Búsqueda primero en amplitud (o anchura)

- **Ejemplo:** grafo no dirigido.

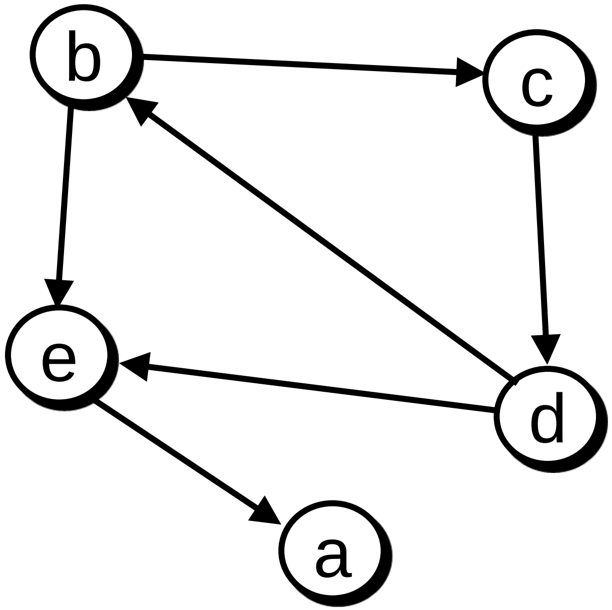


- **Bosque de expansión en amplitud**

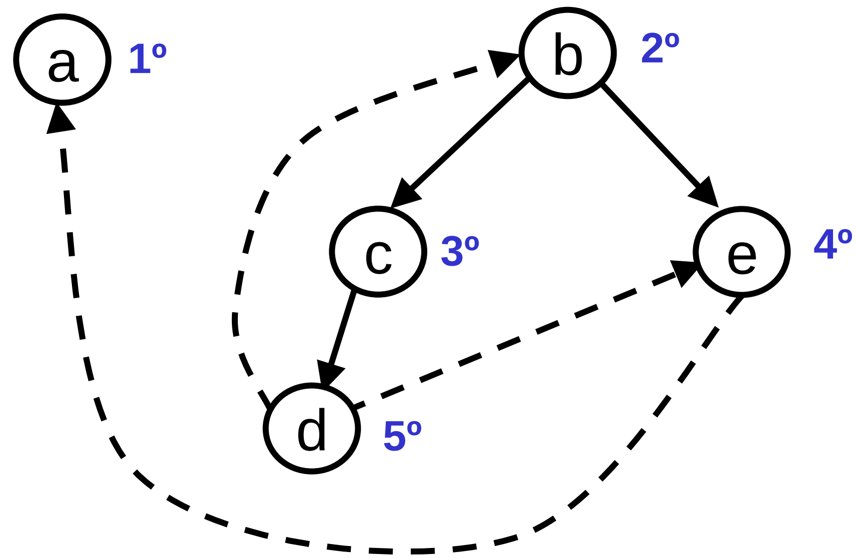


4.3.1.2. Búsqueda primero en amplitud (o anchura)

- **Ejemplo:** grafo dirigido.



Bosque de expansión



- ¿Cuánto es el tiempo de ejecución de la bpa?
- ¿Cómo comprobar si un arco es de avance, cruce, etc.?
- **Solución:** construir el bosque explícitamente.

4.3.1. Recorridos sobre grafos

- Construcción explícita del bosque de expansión:

  - usamos una estructura de **punteros al padre**.

    marca: array [1, ..., n] **de** entero

- **marca[v]** vale: -1 si v no está visitado

  0 si está visitado y es raíz de un árbol

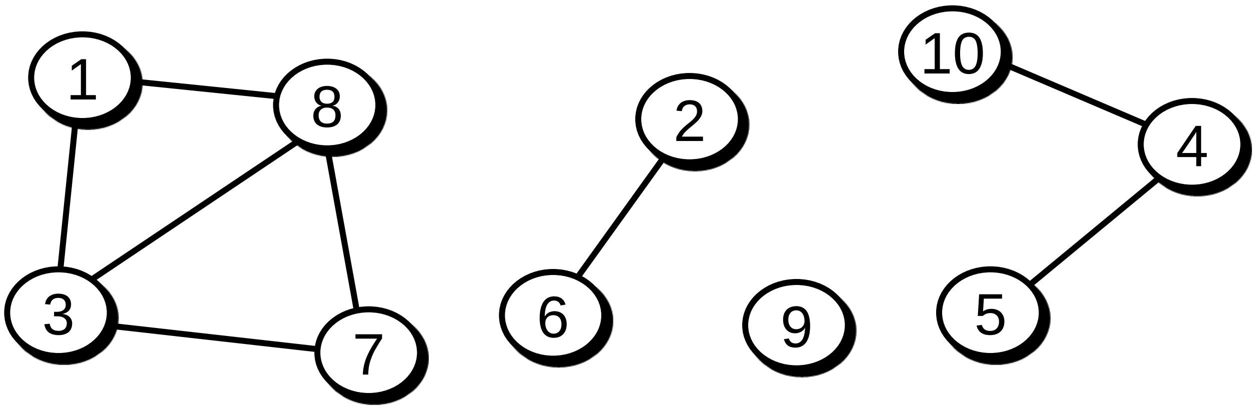
  En otro caso indicará cuál es el padre de v

  – Arco de avance <v, w>: w es descendiente de v en uno de los árboles del bosque.
  – Arco de retroceso <v, w>: v es descendiente de w.
  – Arco de cruce <v, w>: si no se cumple ninguna de las anteriores.

- Modificar BorraMarcas, bpp y bpa, para construir el bosque de expansión.

4.3.1.3. Ejemplos de aplicación de los recorridos

- **Problema 1:** encontrar los componentes conexos de un grafo no dirigido.



- **Problema 2: prueba de aciclicidad.** Dado un grafo (dirigido o no dirigido) comprobar si tiene algún ciclo o no.

4.3.1.3. Ejemplos de aplicación de los recorridos

- **Prueba de aciclicidad**
  - **Grafo no dirigido.** Hacer una bpp (o bpa). Existe algún ciclo si y sólo si aparece algún arco que no es del árbol de expansión.
  - **Grafo dirigido.** Hacer una bpp (o bpa). Existe un ciclo si y sólo si aparece algún arco de retroceso.

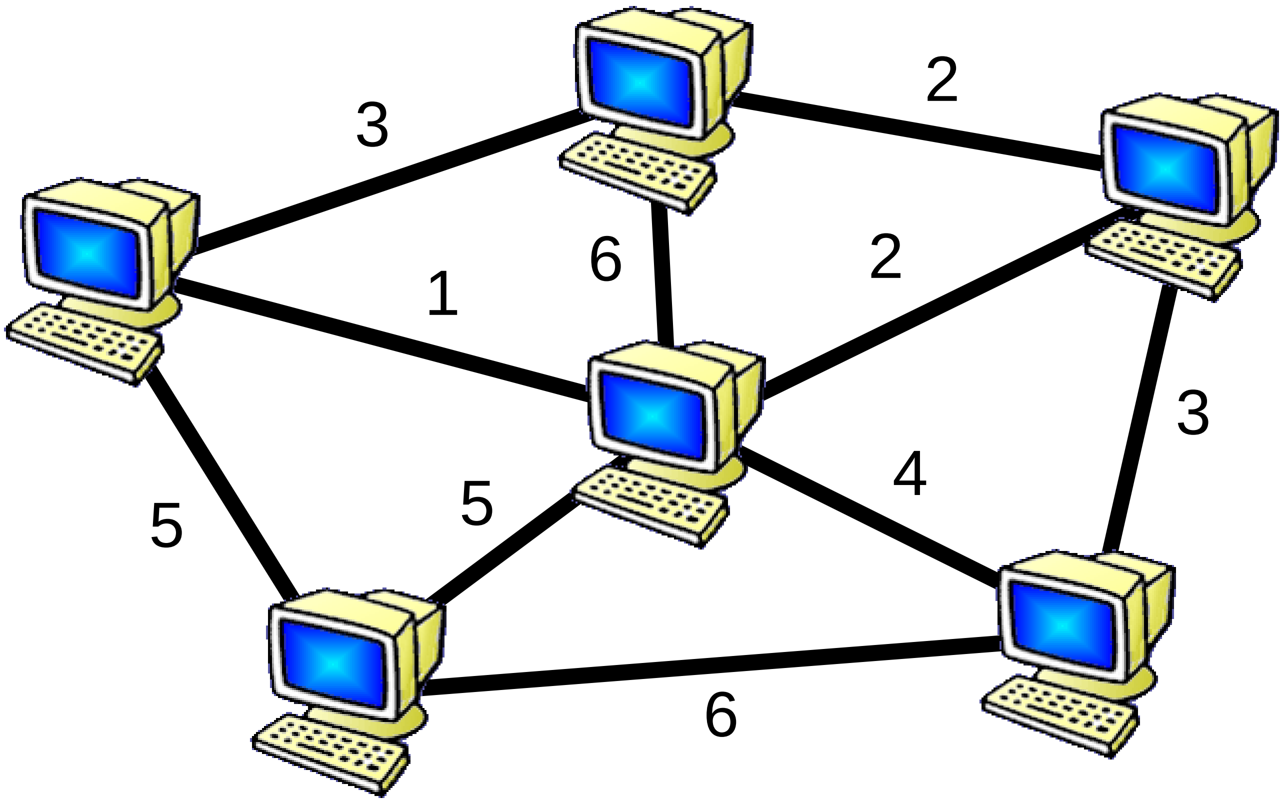
- Orden de complejidad de la prueba de aciclicidad: igual que los recorridos.
  - Con matrices de adyacencia: **O(n^2)**.
  - Con listas de adyacencia: **O(a+n)**.

4.3.2. Árboles de expansión de coste mínimo

- **Definición:** un **árbol de expansión** de un grafo G=(V, A) no dirigido y conexo es un subgrafo G'=(V, A') conexo y sin ciclos.
- **Ejemplo:** los árboles de expansión en profundidad y en amplitud de un grafo conexo.
- En grafos con pesos, el **coste del árbol de expansión** es la suma de los costes de las aristas.

- **Problema del árbol de expansión de coste mínimo:** dado un grafo no dirigido con pesos, encontrar el árbol de expansión de menor coste.

4.3.2. Árboles de expansión de coste mínimo



- **Problema:** conectar todos los ordenadores con el menor coste total.
- **Solución:** algoritmos clásicos de Prim y Kruskal.

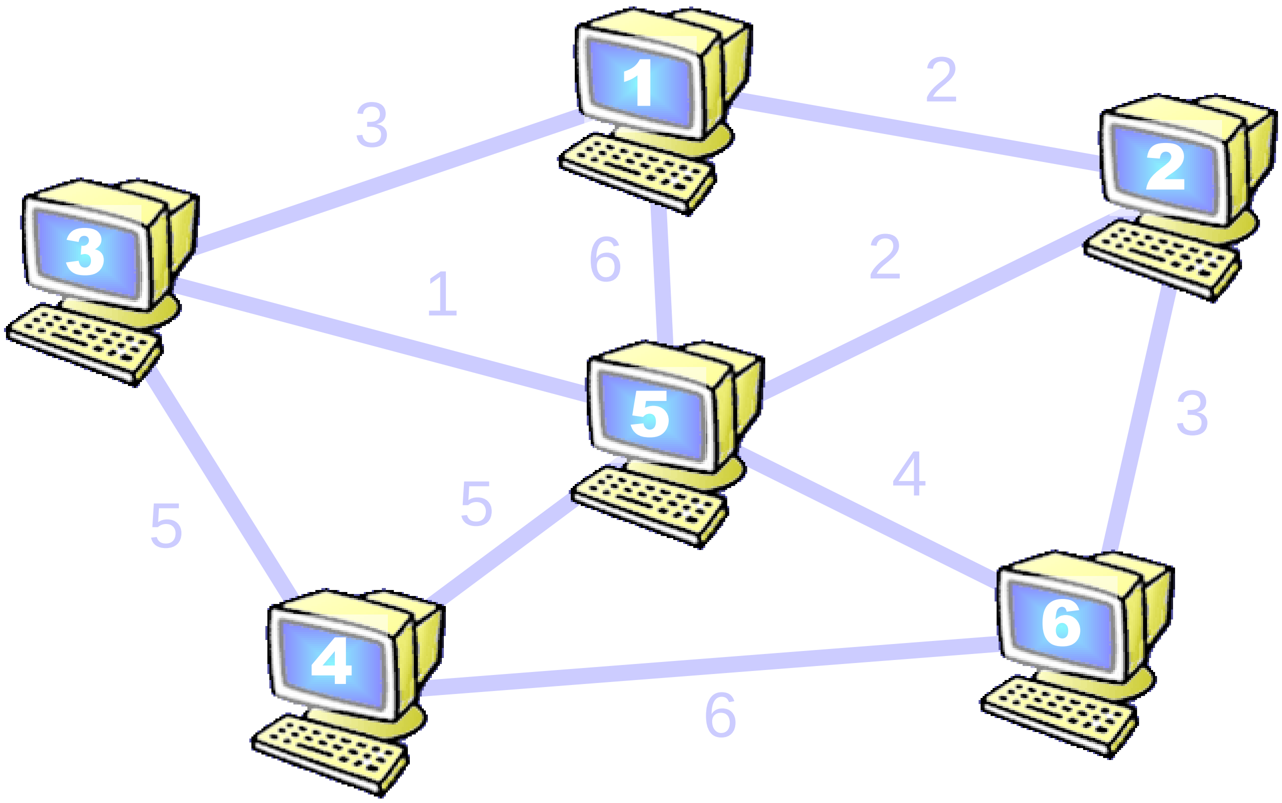
4.3.2.1. Algoritmo de Prim

Esquema:

1. Empezar en un vértice cualquiera **v**. El árbol consta inicialmente sólo del nodo **v**.
2. Del resto de vértices, buscar el que esté más próximo a **v** (es decir, con la arista (**v**, **w**) de coste mínimo). Añadir **w** y la arista (**v**, **w**) al árbol.
3. Buscar el vértice más próximo a cualquiera de estos dos. Añadir ese vértice y la arista al árbol de expansión.
4. Repetir sucesivamente hasta añadir los **n** vértices.

4.3.2.1. Algoritmo de Prim

- **Ejemplo de ejecución del algoritmo.**



4.3.2.1. Algoritmo de Prim

- La solución se construye **poco a poco**, empezando con una solución “vacía”.

- Implícitamente, el algoritmo maneja los **conjuntos**:
  - **V**: Vértices del grafo.
  - **U**: Vértices añadidos a la solución.
  - **V-U**: Vértices que quedan por añadir.

- ¿Cómo implementar eficientemente la búsqueda: encontrar el vértice de **V-U** más próximo a alguno de los de **U**?

4.3.2.1. Algoritmo de Prim

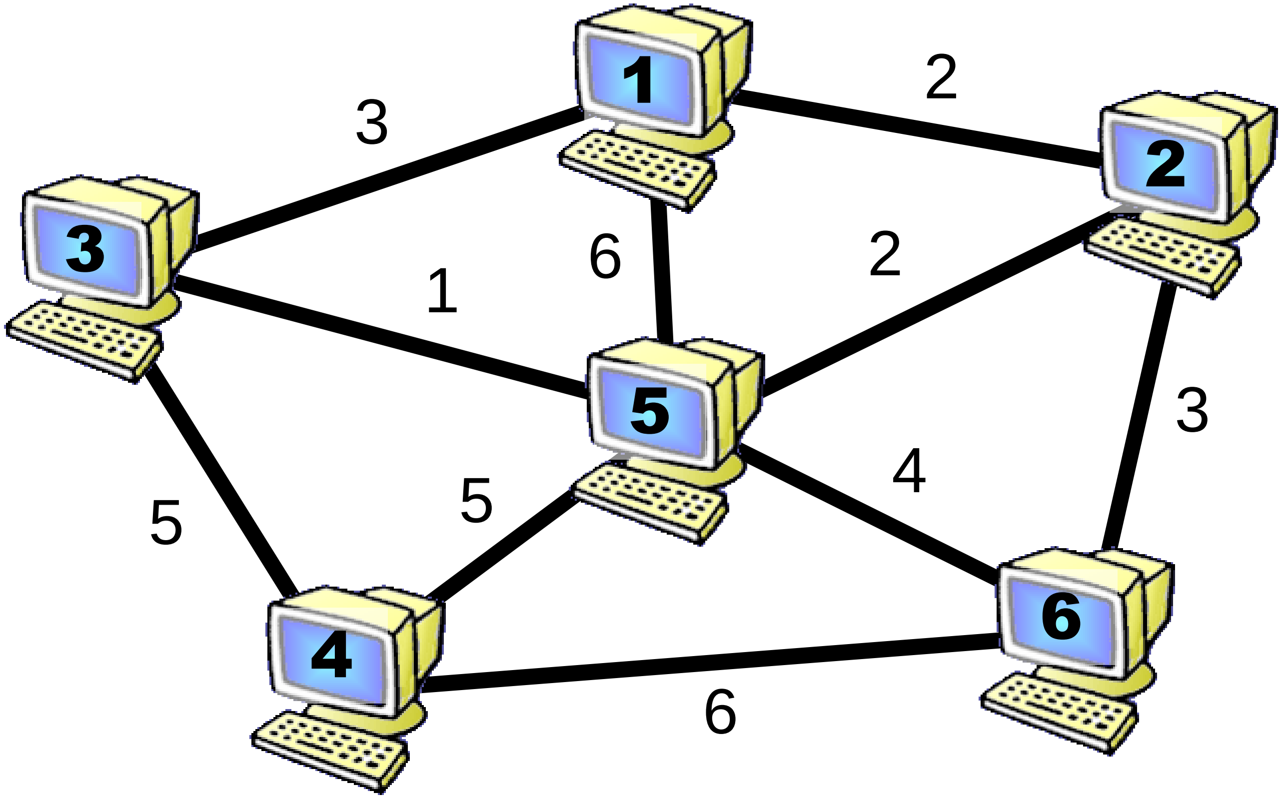
- Se usan dos arrays:
  - **MAS_CERCANO**: Para cada vértice de **V-U** indica el vértice de **U** que se encuentra más próximo.
  - **MENOR_COSTE**: Indica el coste de la arista más cercana.

Estructura del algoritmo de Prim: C[v, w] Matriz de costes

1. Inicialmente **U**= {1}. MAS_CERCANO[v]= 1.
   MENOR_COSTE[v]= C[1, v], para v= 2..n
2. Buscar el nodo **v**, con MENOR_COSTE mínimo.
   Asignarle un valor muy grande (para no volver a cogerlo).
3. Recalcular MAS_CERCANO y MENOR_COSTE de los nodos de **V-U**. Para cada w de **V-U**, comprobar si C[v, w] es menor que MENOR_COSTE[w].
4. Repetir los dos puntos anteriores hasta que se hayan añadido los **n** nodos.

4.3.2.1. Algoritmo de Prim

- **Ejemplo:** mostrar la ejecución del algoritmo sobre el grafo.



- ¿Dónde está almacenado el resultado del algoritmo?
- ¿Cuál es el orden de complejidad del algoritmo?

4.3.2.2. Algoritmo de Kruskal

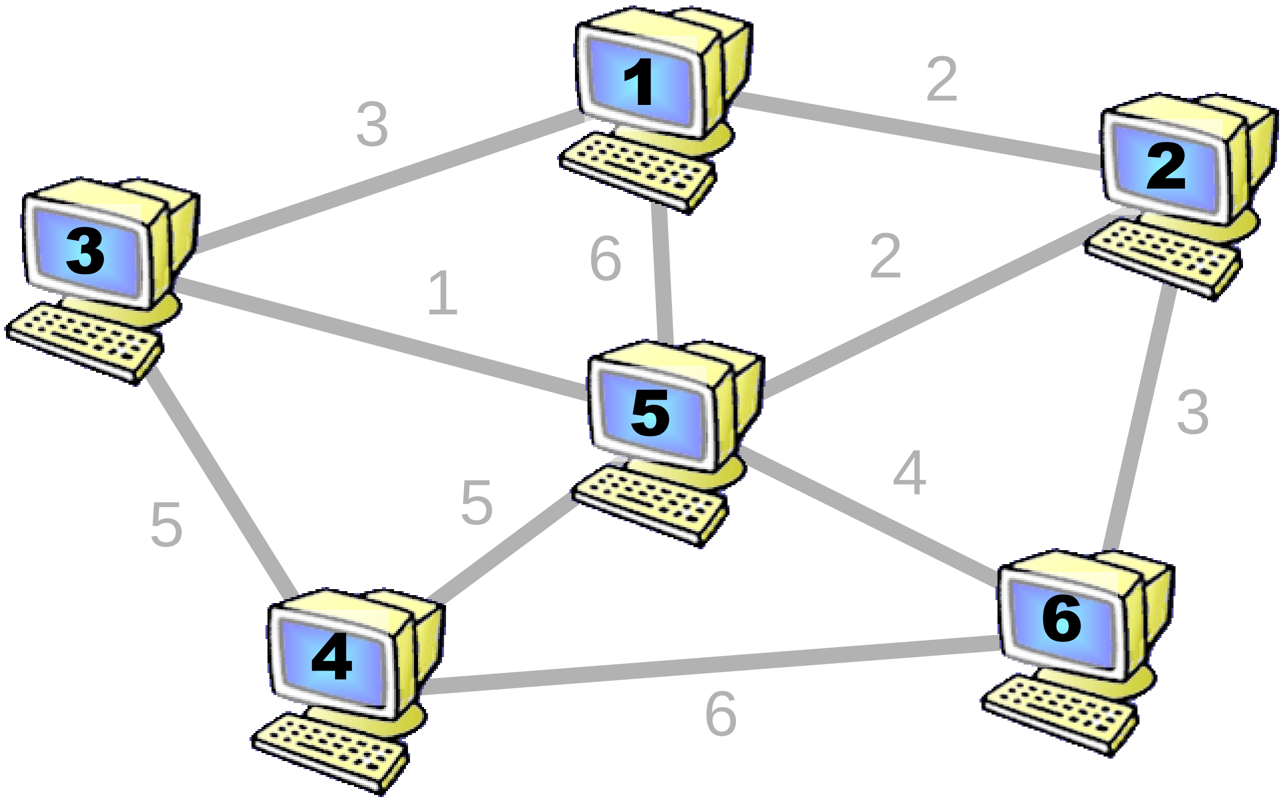
Esquema: G= (V, A)

1. Empezar con un grafo sin aristas: **G’**= (V, Ø)
2. Seleccionar la arista de menor coste de **A**.
3. Si la arista seleccionada forma un ciclo en **G’**, eliminarla. Si no, añadirla a **G’**.
4. Repetir los dos pasos anteriores hasta tener **n-1** aristas.

- ¿Cómo saber si una arista **(v, w)** provocará un ciclo en el grafo **G’**?

4.3.2.2. Algoritmo de Kruskal

- **Ejemplo:** mostrar la ejecución del algoritmo en el siguiente grafo.



4.3.2.2. Algoritmo de Kruskal

Implementación del algoritmo

- **Necesitamos:**
  - Ordenar las aristas de **A**, de menor a mayor: **O(a log a).**
  - Saber si una arista dada **(v, w)** provocará un ciclo.
- ¿Cómo comprobar rápidamente si **(v, w)** forma un ciclo?
- Una arista **(v, w)** forma un ciclo si **v** y **w** están en la misma componente conexa.
- La relación “estar en la misma componente conexa” es una **relación de equivalencia**.

4.3.2.2. Algoritmo de Kruskal

- Usamos la estructura de **relaciones de equivalencia** con punteros al padre:
  - Inicialización: crear una relación de equivalencia vacía (cada nodo es una componente conexa).
  - Seleccionar las aristas **(v, w)** de menor a mayor.
  - La arista forma ciclo si: **Encuentra(v)**=**Encuentra(w)**
  - Añadir una arista **(v, w)**: **Unión(v, w)** (juntar dos componentes conexas en una).
- Mostrar la ejecución sobre el grafo de ejemplo.
- ¿Cuál es el orden de complejidad del algoritmo?

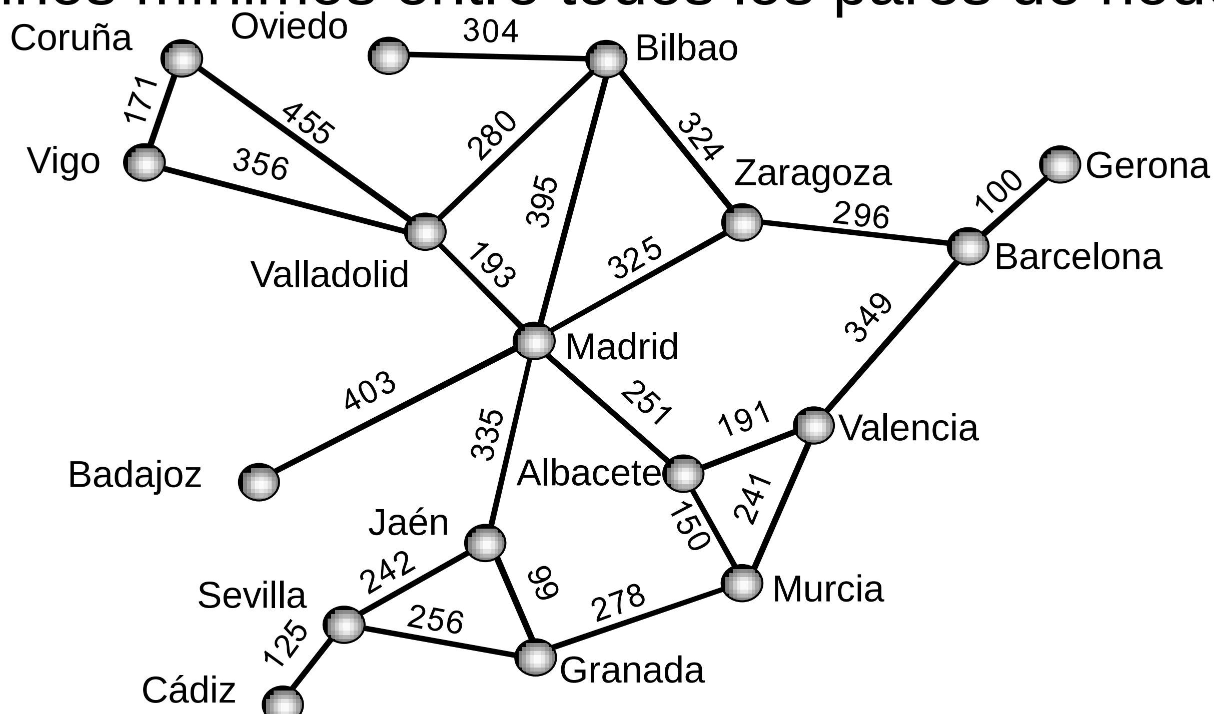
4.3.2. Árboles de expansión mínimos

Conclusiones

- Ambos algoritmos (**Prim** y **Kruskal**) encuentran siempre la solución óptima.
- La solución obtenida será la misma, o no...
- La estructura de los dos algoritmos es muy parecida:
  - Empezar con una solución "vacía".
  - Añadir en cada paso un elemento a la solución (Prim: un nodo; Kruskal: una arista).
  - Una vez añadido un elemento a la solución, no se quita (no se "deshacen" las decisiones tomadas).

4.3.3. Problemas de caminos mínimos

- **Coste de un camino:** suma de los costes de las aristas por las que pasa.
- **Problemas de caminos mínimos:**
  - Camino mínimo entre dos nodos, **v** y **w**.
  - Caminos mínimos entre un nodo **v** y todos los demás.
  - Caminos mínimos entre todos los pares de nodos.



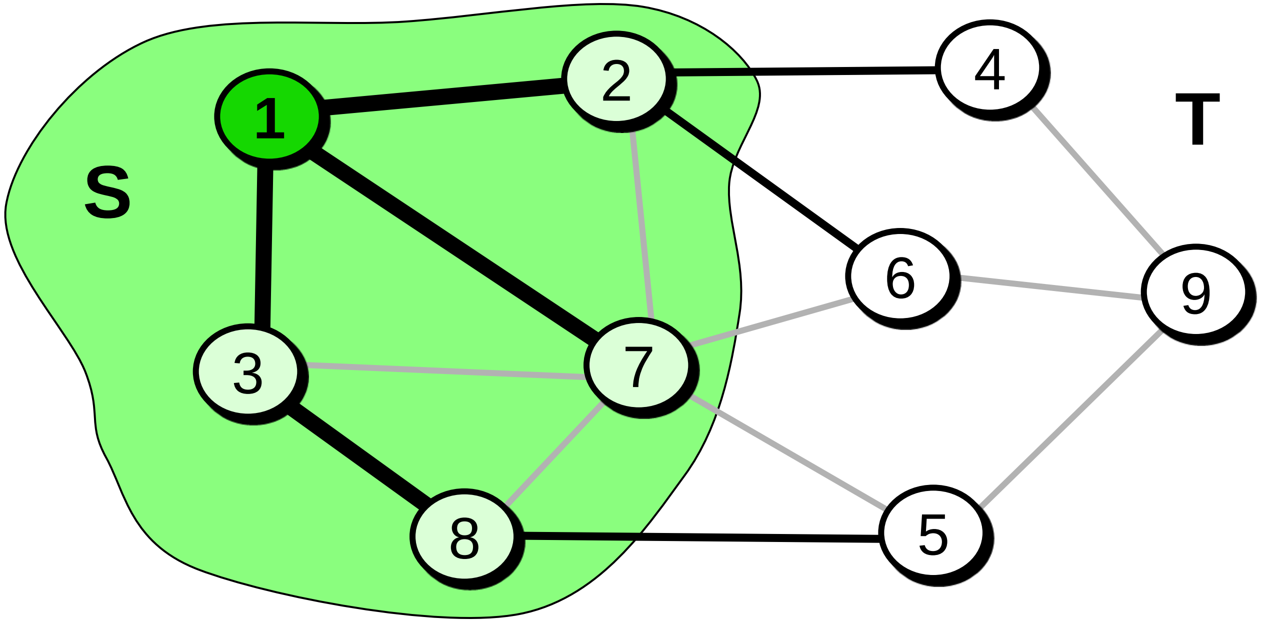
4.3.3.1. Caminos mínimos desde un origen Algoritmo de Dijkstra

- Supongamos un grafo **G**, con pesos positivos y un nodo origen **v**.

- El algoritmo trabaja con dos conjuntos de nodos:
  - **Escogidos: S**. Nodos para los cuales se conoce ya el camino mínimo desde el origen.
  - **Candidatos: T**. Nodos pendientes de calcular el camino mínimo, aunque conocemos los caminos mínimos desde el origen pasando por nodos de **S**.

4.3.3.1. Caminos mínimos desde un origen

- **Camino especial:** camino desde el origen hasta un nodo, que pasa sólo por nodos escogidos, **S**.



- **Idea:** en cada paso, coger el nodo de **T** con menor distancia al origen. Añadirlo a **S**.
- Recalcular los caminos mínimos de los demás candidatos, pudiendo pasar por el nodo cogido.

4.3.3.1. Caminos mínimos desde un origen

Algoritmo de Dijkstra

- **Inicialización: S**= {1}, **T**= {2, ..., n}, caminos especiales mínimos = caminos directos.
- **Repetir** n-1 veces:
  - Seleccionar el nodo **v** de **T** con el camino especial más corto.
  - **Proposición:** el camino mínimo para este nodo **v**, coincide con su camino especial.
  - Recalcular los caminos especiales para los nodos de **T**, pudiendo pasar por **v**.

4.3.3.1. Caminos mínimos desde un origen
Implementación del algoritmo de Dijkstra

- Suponemos que el origen es el nodo 1.
- **D: array** [2..n] **de** real. **D[v]** almacena el coste del camino especial mínimo para el nodo **v**.
- **P: array** [2..n] **de** entero. **P[v]** almacena el último nodo en el camino especial mínimo de **v**.

- **Inicialización:** D[v]:= C[1, v], P[v]:= 1
- **Nodo seleccionado:** nodo de **T** con mínimo D[v]
- **Actualización:** para todos los **w** de **T** hacer

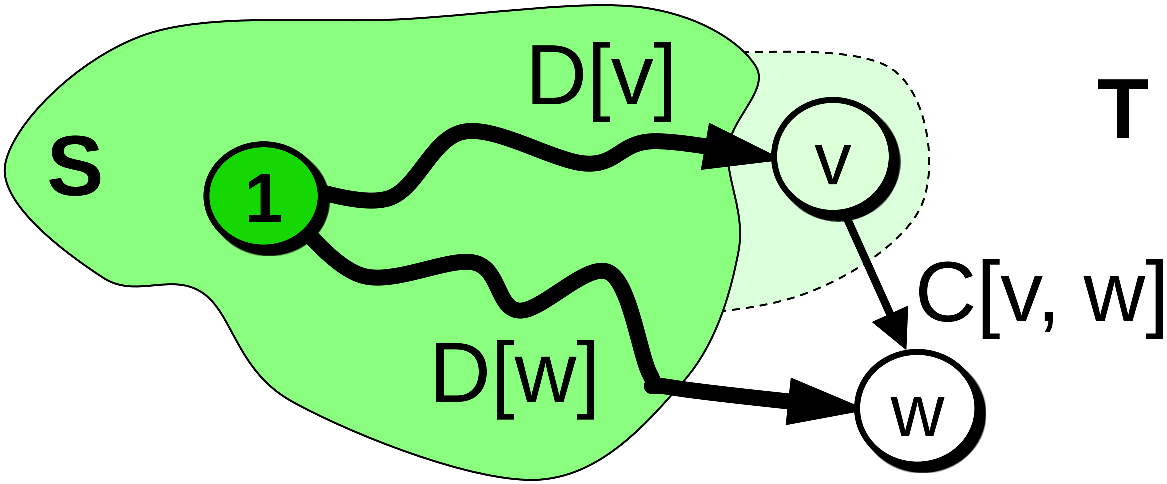
  si D[v] + C[v, w] < D[w] **entonces**

  D[w]:= D[v] + C[v, w]

  P[w]:= v

  finsi

4.3.3.1. Caminos mínimos desde un origen



- **Camino especial para w:**
  - Sin pasar por **v**: D[w]
  - Pasando por **v**: D[v] + C[v,w]
  - Nos quedamos con el menor.
- Si el menor es pasando por **v** entonces: **P[w]= v**.
- Camino especial para **w**:

  $1 \rightarrow ... \rightarrow P[P[P[w]]] \rightarrow P[P[w]] \rightarrow P[w] \rightarrow w$

4.3.3.1. Caminos mínimos desde un origen Algoritmo de Dijkstra

- **Entrada:**

  C: array [1..n, 1..n] **de** real → Matriz de costes

- **Salida:**

  D: array [2..n] **de** real → Costes de caminos mínimos

  P: array [2..n] **de** entero → Nodos de paso

- Datos para **cálculos intermedios:**

  S: array [2..n] **de** booleano → Nodos escogidos

- **Inicialización:**

```
para v:= 2, ..., n hacer
    D[v]:= C[1, v]
    P[v]:= 1
    S[v]:= FALSE
finpara
```

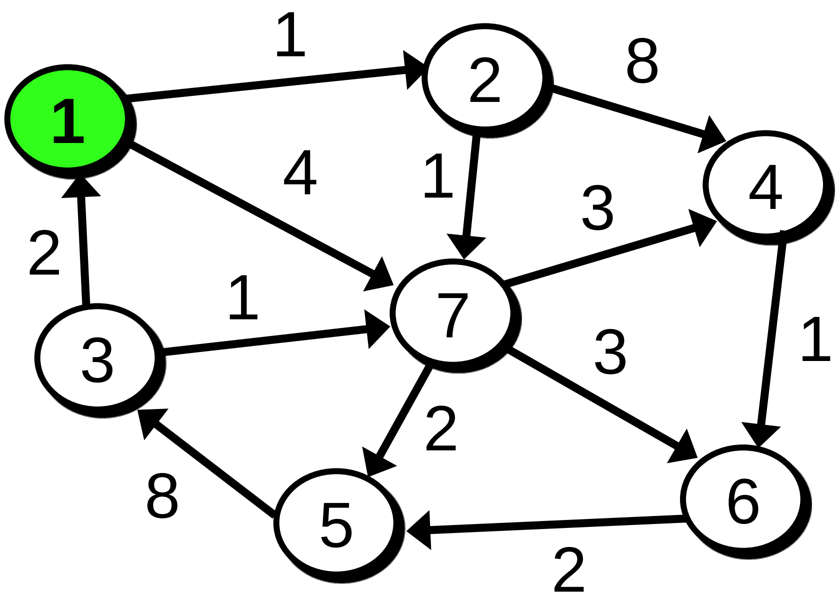
4.3.3.1. Caminos mínimos desde un origen

Algoritmo de Dijkstra

```
para i:= 1, ..., n-1 hacer
    v:= nodo con S[v]==FALSE y mínimo D[v]
    S[v]:= TRUE
    para cada nodo w adyacente a v hacer
        si (NOT S[w]) AND (D[v]+C[v,w]<D[w]) entonces
            D[w]:= D[v] + C[v, w]
            P[w]:= v
        finsi
    finpara
finpara
```

4.3.3.1. Caminos mínimos desde un origen

- **Ejemplo:** mostrar la ejecución del algoritmo de Dijkstra sobre el siguiente grafo.



Inicialmente

Nodo	S	D	P
2	F	1	1
3	F	∞	1
4	F	∞	1
5	F	∞	1
6	F	∞	1
7	F	4	1

- A partir de las tablas, ¿cómo calcular cuál es el camino mínimo para un nodo **v**?

4.3.3.1. Caminos mínimos desde un origen

Eficiencia del algoritmo de Dijkstra

- Con matrices de adyacencia:
  - Inicialización: **O(n)**
  - Ejecutar **n-1** veces:
    - Buscar el nodo con mínimo D[v] y S[v] falso: **O(n)**
    - Actualizar los valores de los candidatos: **O(n)**
  - En total: **O(n^2)**
- Con listas de adyacencia:
  - Seguimos teniendo un **O(n^2)**
  - Podemos modificar la implementación y conseguir un **O((a+n)·log n)**. Será adecuada cuando **a** << **n^2**.

4.3.3.2. Caminos mínimos entre todos los pares

- **Problema:** calcular los caminos mínimos entre todos los pares de nodos del grafo.

Posibilidades

- Aplicar el algoritmo de Dijkstra **n** veces, una por cada posible nodo origen:
  - Con matrices de adyacencia: $\mathbf{O(n^3)}$
  - Con listas de adyacencia: $\mathbf{O((a+n)\cdot n\cdot \log n)}$
- Aplicar el algoritmo de Floyd:
  - Con listas o matrices: $\mathbf{O(n^3)}$
  - Pero más sencillo de programar...

4.3.3.2. Caminos mínimos entre todos los pares

- **Entrada:**

  C: array [1..n, 1..n] **de** real → Matriz de costes

- **Salida:**

  D: array [1..n, 1..n] **de** real → Costes caminos mínimos

Algoritmo de Floyd

D:= C
para k:= 1, ..., n **hacer**
    para i:= 1, ..., n **hacer**
        para j:= 1, ..., n **hacer**
            D[i, j]:= min (D[i, j] , D[i, k] + D[k, j])

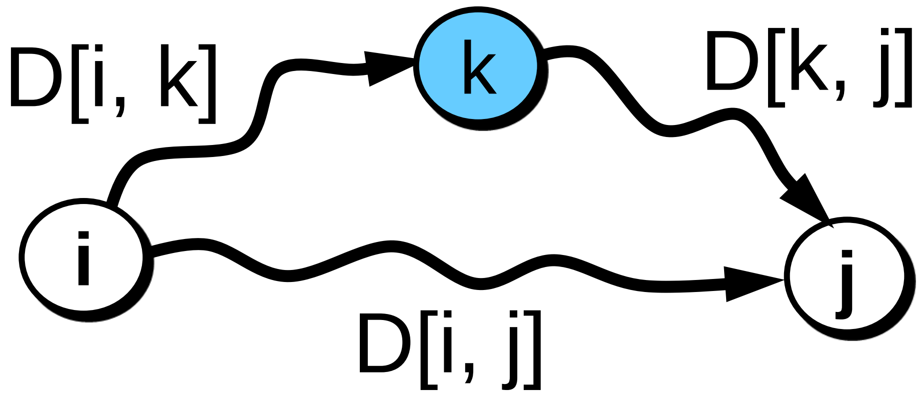
4.3.3.2. Caminos mínimos entre todos los pares

- ¿En qué se basa el algoritmo de Floyd?

- En cada paso **k**, la matriz **D** almacena los caminos mínimos entre todos los pares pudiendo pasar por los **k** primeros nodos.
- **Inicialización: D** almacena los caminos directos.
- **Paso 1:** caminos mínimos pudiendo pasar por el 1.
- ...
- **Paso n:** caminos mínimos pudiendo pasar por cualquier nodo → Lo que buscamos.

- En el paso **k**, el nodo **k** actúa de pivote.

4.3.3.2. Caminos mínimos entre todos los pares



- **Camino mínimo entre i y j, en el paso k:**
  - Sin pasar por **k**: D[i, j]
  - Pasando por **k**: D[i, k] + D[k, j]
  - Nos quedamos con el menor.
- **Ojo:** falta indicar cuáles son los caminos mínimos.
- **P: array** [1..n, 1..n] **de** entero. **P[i, j]** indica un nodo intermedio en el camino de **i** a **j**.

$$i \rightarrow \ldots \rightarrow P[i, j] \rightarrow \ldots \rightarrow j$$

4.3.3.2. Caminos mínimos entre todos los pares

Algoritmo de Floyd

D:= C

P:= 0

para k:= 1, ..., n **hacer**

    para i:= 1, ..., n **hacer**

        para j:= 1, ..., n **hacer**

            si D[i, k] + D[k, j] < D[i, j] **entonces**

                D[i, j]:= D[i, k] + D[k, j]

                P[i, j]:= k

            finsi

- ¿Cuál es el orden de complejidad del algoritmo?

4.3.3.2. Caminos mínimos entre todos los pares

- El algoritmo de Floyd se basa en una descomposición recurrente del problema:

$$D_k(i, j) := \begin{cases} C[i, j] & \text{Si } k=0 \\ \min(D_{k-1}(i, j), D_{k-1}(i, k) + D_{k-1}(k, j)) & \text{Si } k>0 \end{cases}$$

- Como la fila y columna **k** no cambian en el paso **k**, se usa una sola matriz **D**.
- ¿Cómo recuperar el camino?

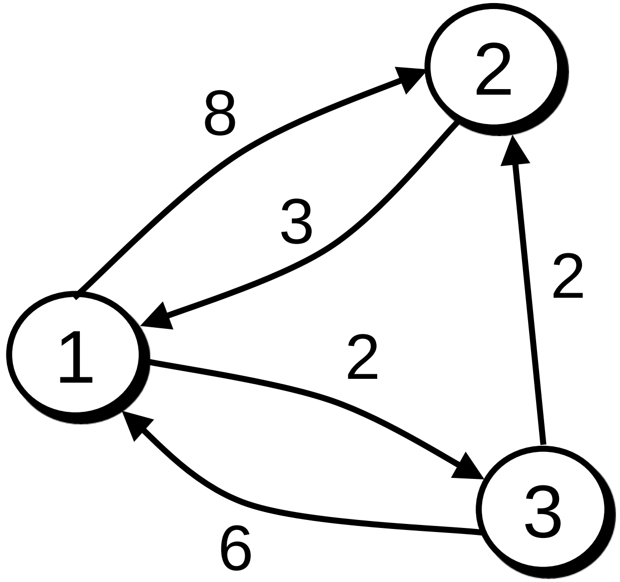
```
operación camino (i, j: entero)
        k:= P[i, j]
        si k ≠ 0 entonces
            camino (i, k)
            escribe (k)
            camino (k, j)
        finsi
```

$\leftarrow$

escribe (i)
camino (i, j)
escribe (j)

4.3.3.2. Caminos mínimos entre todos los pares

- **Ejemplo:** aplicar el algoritmo de Floyd al siguiente grafo dirigido.



D	1	2	3
1	0	8	2
2	3	0	∞
3	6	2	0

P	1	2	3
1	0	0	0
2	0	0	0
3	0	0	0

- Calcular el camino mínimo entre 1 y 2.

4.3.3.3. Cierre transitivo de un grafo

- **Problema:** dada una matriz de adyacencia **M** (de booleanos), encontrar otra matriz **A**, tal que **A[i, j]** es cierto si y sólo si existe un camino entre **i** y **j**.

Algoritmo de Warshall

- Es una simple adaptación del algoritmo de Floyd a valores booleanos.

A:= M

para k:= 1, ..., n **hacer**

  para i:= 1, ..., n **hacer**

    para j:= 1, ..., n **hacer**

      A[i, j]:= A[i, j] OR (A[i, k] AND A[k, j])

Conclusiones

- **Caminos mínimos:** problema fundamental en grafos. Diferentes problemas, con diversas aplicaciones.
- Desde un origen hasta todos los demás nodos → algoritmo de **Dijkstra**.
- **Idea:** nodos escogidos y candidatos.

- Entre todos los pares → algoritmo de **Floyd**.
- **Idea:** pivotar sobre cada nodo.

- Ambos algoritmos pueden modificarse para resolver otros problemas.

4.3.4. Algoritmos sobre grafos dirigidos

4.3.4.1. Componentes fuertemente conexas

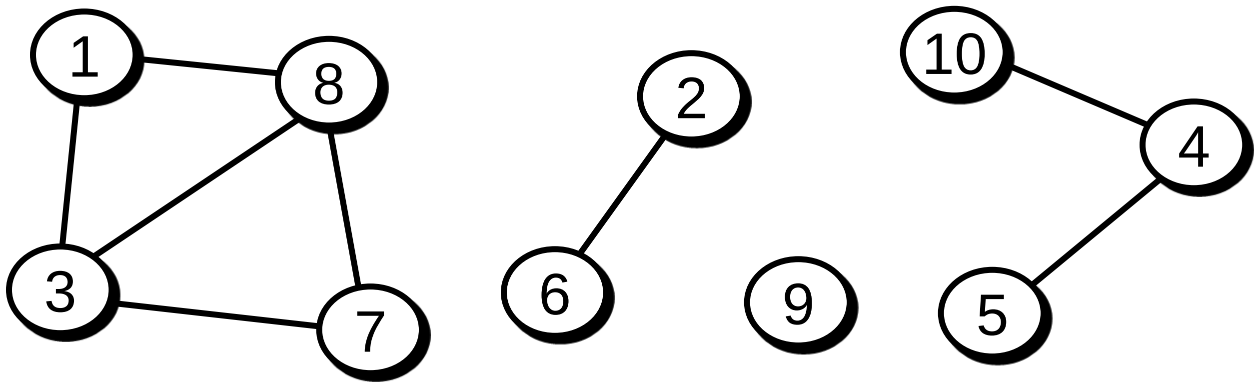
4.3.4.2. Grafos dirigidos acíclicos

Definición:

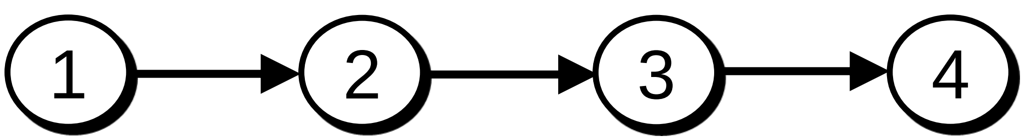
- Una **componente conexa** de un grafo **G** es un subgrafo maximal y conexo de **G**.
- En grafos dirigidos: **componente fuertemente conexa**. Existen caminos entre todos los pares de nodos y en los dos sentidos.
- **Problema:** dado un grafo, calcular sus componentes (fuertemente) conexas.

4.3.4.1. Componentes fuertemente conexas

- **Componentes conexas** en grafos no dirigidos.

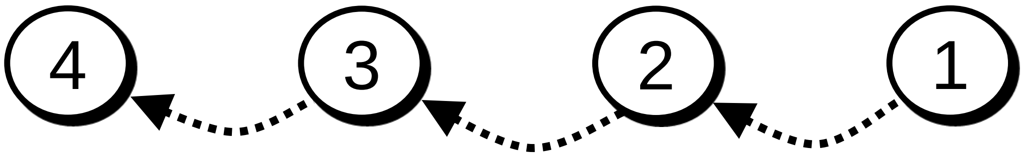


- **Solución trivial:** aplicar una bpp. Cada árbol es una componente conexa.
- **Componentes fuertemente conexas** en grafos dirigidos.
- ¿Funciona una simple bpp?



4.3.4.1. Componentes fuertemente conexas

- La bpp no funciona, pero...
- ¿Y si hubiéramos empezado la bpp de mayor a menor número...?



- **Idea:** hacer dos búsquedas en profundidad.
- En la primera se calcula un orden para la segunda.
- En la segunda se recorre el grafo (invertido), según ese orden.
- **Orden posterior de un grafo: npost[v]** = orden de terminación de la llamada recursiva de **v** en la bpp.

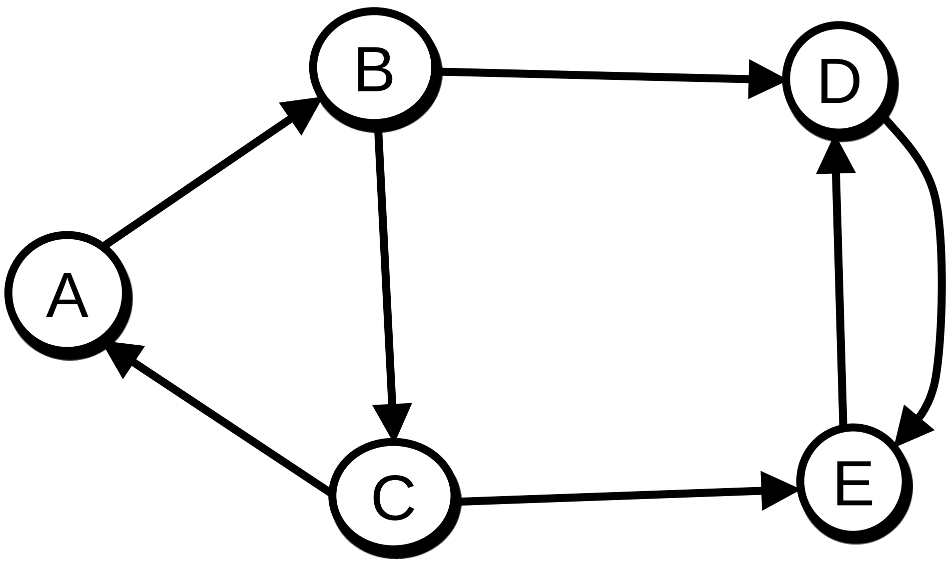
4.3.4.1. Componentes fuertemente conexas

Algoritmo para calcular las componentes fuertemente conexas de un grafo G = (V, A)

1. Realizar una bpp de **G**, numerando los vértices en orden posterior. **npost: array** [1..n] **de** entero.
2. Construir el grafo **invertido G' =** (V, A'). Para toda arista $<v, w> \in A$, tenemos $<w, v> \in A'$.
3. Realizar una bpp en **G'** empezando en el nodo con mayor **npost**. Si no se visitan todos los nodos, continuar con el nodo no visitado con mayor **npost**.
4. Cada árbol del bosque resultante del paso 3 es una **componente fuertemente conexa** de **G**.

4.3.4.1. Componentes fuertemente conexas

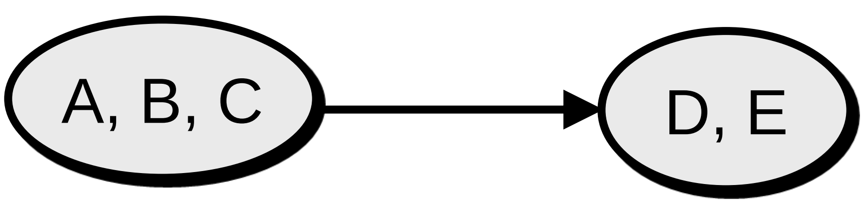
- **Ejemplo:** encontrar las componentes fuertemente conexas del siguiente grafo.



- ¿Cuál es el orden de complejidad del algoritmo?

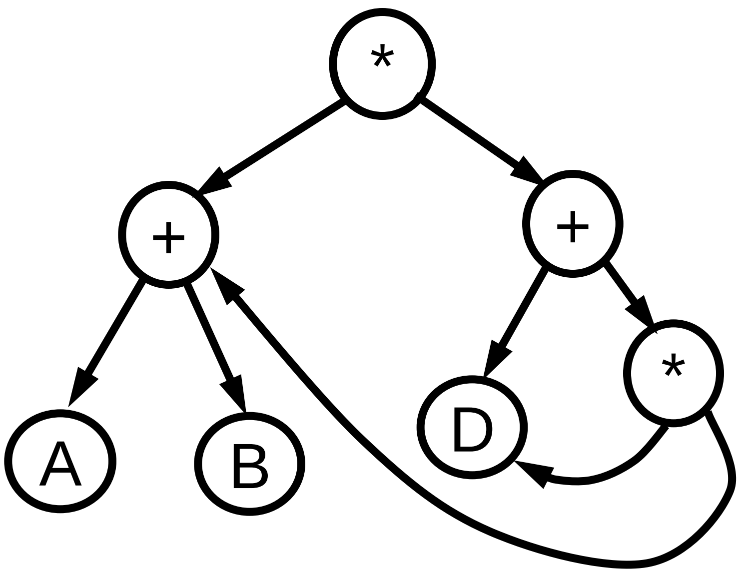
4.3.4.1. Componentes fuertemente conexas

- A partir de las componentes fuertemente conexas, podemos representar todos los caminos existentes mediante un **grafo reducido**.
- **Grafo reducido de un grafo dirigido G: G_R.**
  - Cada nodo de G_R representa un componente fuertemente conexo de **G**.
  - Existe una arista entre dos nodos de G_R si existe una arista entre algunos de los nodos de las componentes conexas de **G** correspondientes.

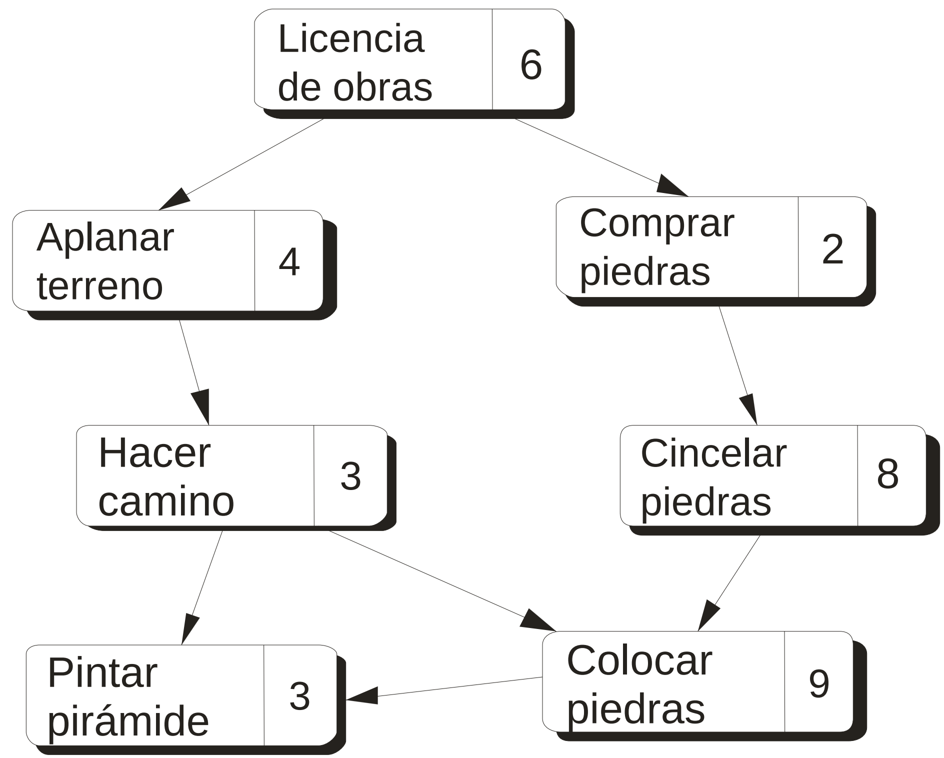


4.3.4.2. Grafos dirigidos acíclicos

- **Definición**: un **grafo dirigido acíclico** (**GDA**) es un grafo dirigido sin ciclos.
- **Ejemplos:** grafo de planificación de tareas, expresiones aritméticas (con subexpresiones comunes), grafo de prerrequisitos, etc.

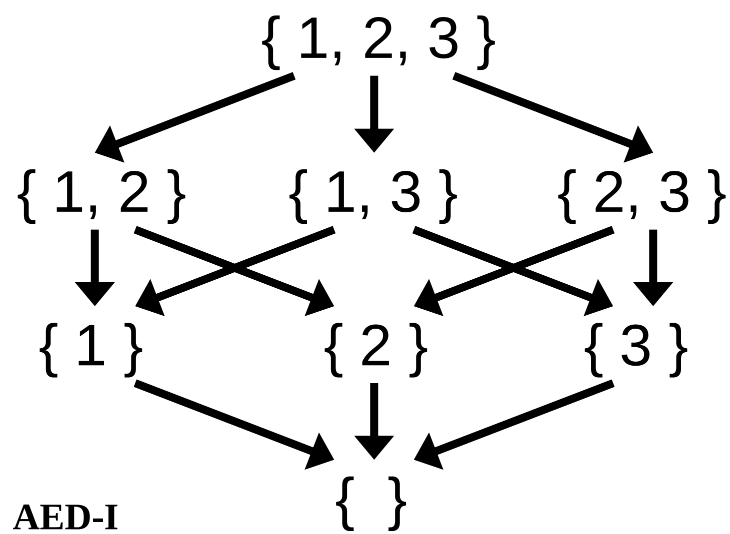


(A+B)*(D+D*(A+B))



4.3.4.2. Grafos dirigidos acíclicos

- Las propias características de la aplicación implican que no pueden existir ciclos.
- **Concepto matemático subyacente:** representación de órdenes parciales.
- **Definición:** un **orden parcial** en un conjunto **C** es una relación binaria que cumple:
  - Para cualquier elemento $a \in C$, $(a\ R\ a)$ es falso
  - Para cualquier $a, b, c \in C$, $(a\ R\ b)$ Y $(b\ R\ c) \rightarrow (a\ R\ c)$
- **Ejemplo:** La relación de inclusión propia entre conjuntos, $\subset$.

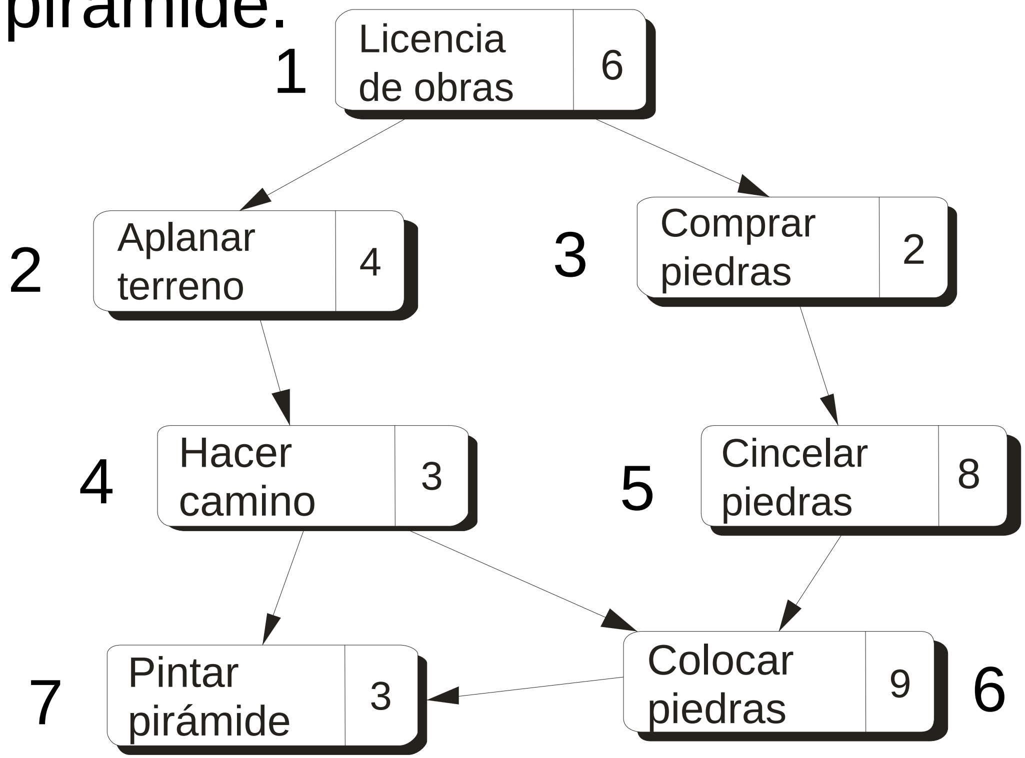


4.3.4.2. Grafos dirigidos acíclicos

- **Recorrido en orden topológico:** es un tipo de recorrido aplicable solamente a gda.
- **Idea:** un vértice sólo se visita después de haber sido visitados **todos** sus predecesores en el grafo.
- **Numeración en orden topológico: ntop[v].** Si existe una arista <v, w> entonces **ntop**[v] < **ntop**[w].
- Puede existir más de un orden válido.
- ¿Cuál es el significado del orden topológico?
- Grafo de tareas: es un posible orden de ejecución de las tareas, respetando las precedencias.
- Expresión aritmética: orden para evaluar el resultado total de la expresión (de mayor a menor **ntop**).

4.3.4.2. Grafos dirigidos acíclicos

- **Ejemplo:** ordenación topológica de las tareas para construir una pirámide.



- Existen otras ordenaciones topológicas válidas.
- Diseñar un algoritmo para calcular una ordenación topológica.

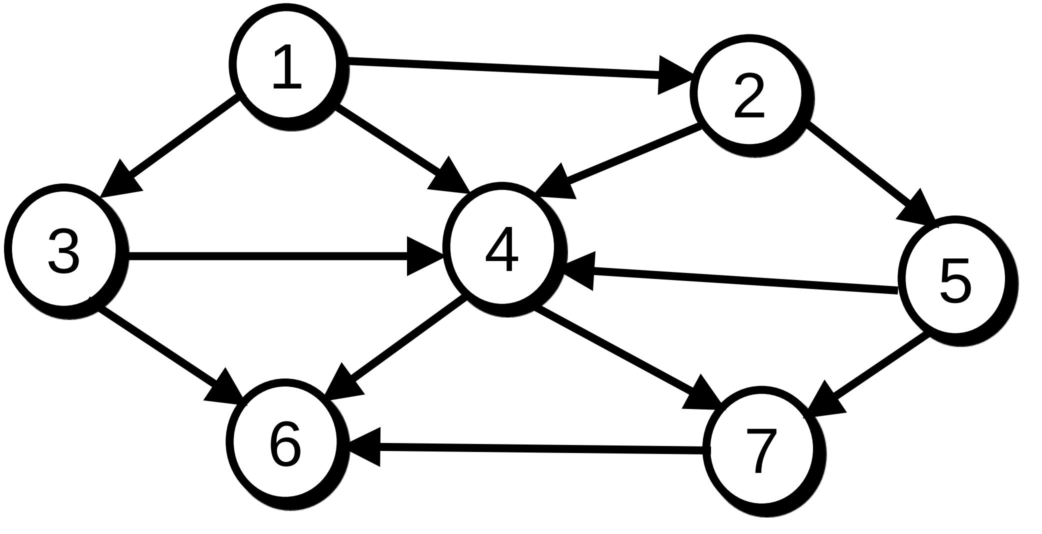
4.3.4.2. Grafos dirigidos acíclicos

Algoritmo de recorrido topológico

1. Calcular los grados de entrada de todos los nodos.
2. Buscar un nodo **v** con grado de entrada 0 (es decir, sin predecesores). Numerarlo y marcarlo como visitado. Si no hay ninguno es porque existe un ciclo.
3. Para todos los nodos adyacentes a **v**, decrementar en 1 su grado de entrada.
4. Repetir los pasos 2 y 3 hasta haber visitado todos los nodos.

4.3.4.2. Grafos dirigidos acíclicos

- **Otra posibilidad:** usar la numeración en **orden posterior** (orden de terminación de las llamadas recursivas en el procedimiento bpp).
- **Proposición:** si **npost**[v] es una numeración posterior de un gda, entonces **ntop**[n]:= n-**npost**[v] es una numeración topológica válida del gda.
- ¿Por qué?
- **Ejemplo:** aplicar los dos algoritmos al siguiente grafo.



4.3.5. Algoritmos sobre grafos no dirigidos

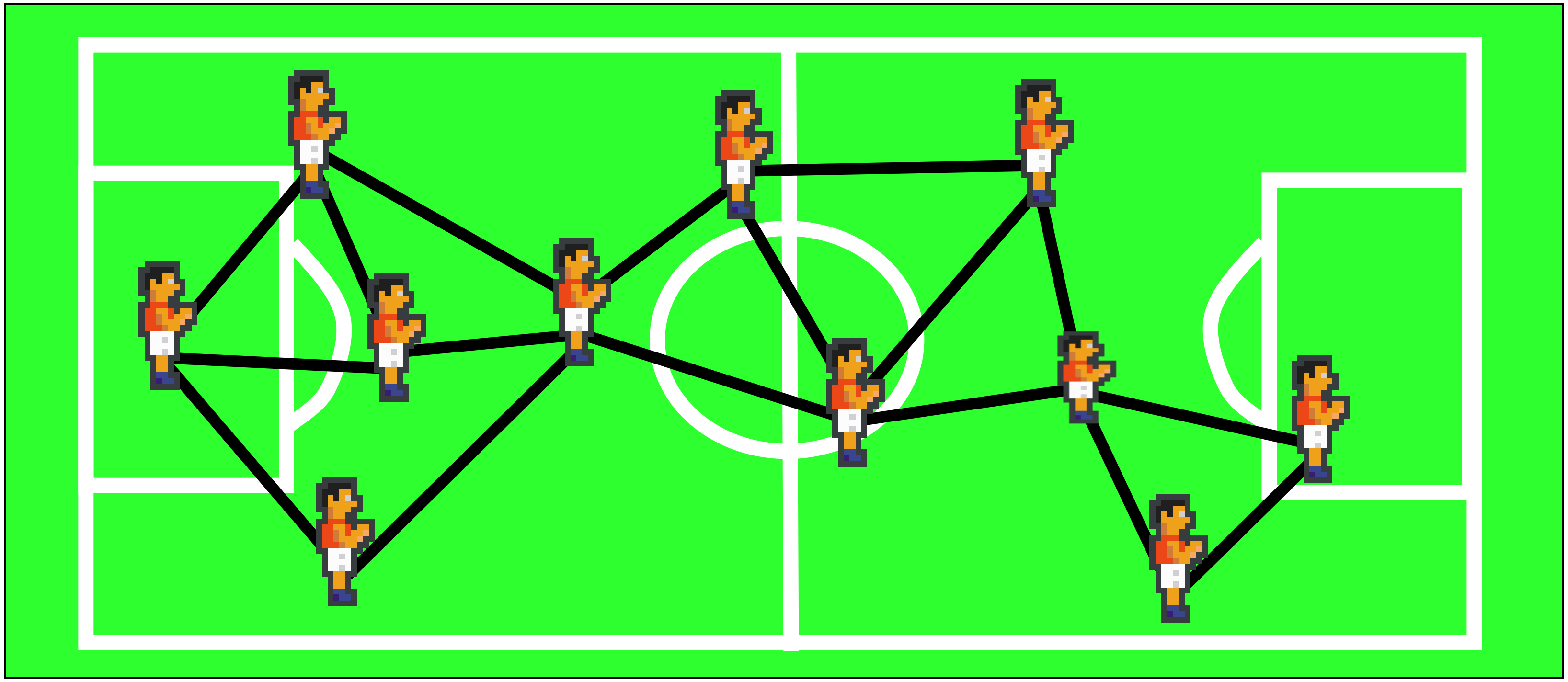
4.3.5.1. Puntos de articulación y componentes biconexas

4.3.5.2. Caminos y circuitos de Euler

- **Definición:** un **punto de articulación** de un grafo no dirigido, **G**, es un nodo **v** tal que cuando es eliminado de **G** (junto con las aristas incidentes en él) se divide una componente conexa de **G** en dos o más componentes conexas.
- **Definición:** un grafo no dirigido se dice que es **biconexo** si no tiene puntos de articulación.
- **Definición:** una **componente biconexa** de un grafo **G** es un subgrafo biconexo y maximal de **G**.

4.3.5.1. Puntos de articulación y componentes biconexas

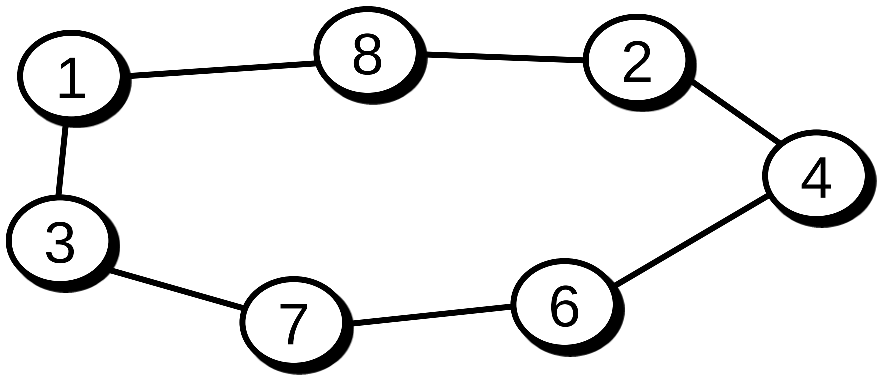
- **Ejemplo:** grafo de estrategias de pase del balón del Real Murcia.



- ¿Qué jugador, o jugadores, desconectan al equipo si los eliminamos?
- Escribir un algoritmo que lo calcule.

4.3.5.1. Puntos de articulación y componentes biconexas

- **Definición:** un grafo **G** tiene **conectividad k** si la eliminación de **k-1** nodos cualesquiera (con sus aristas) no desconecta el grafo.
- Por lo tanto, un grafo es **biconexo** si y sólo si tiene conectividad 2 ó más.



- **Posible algoritmo:** eliminar los nodos uno a uno. Para cada uno, comprobar si el grafo sigue siendo conexo.

4.3.5.1. Puntos de articulación y componentes biconexas

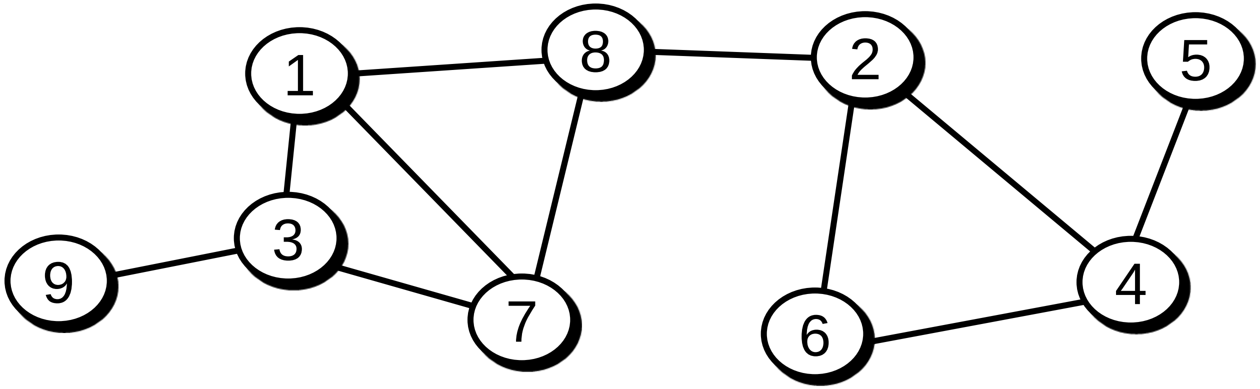
- **Otro algoritmo mejor. Idea:** calcular los caminos "alternativos" que hay para cada nodo en una bpp.

1. Realizar una bpp, numerando los nodos en el orden de recorrido en profundidad: **nbpp[1..N]**.
2. Al terminar la llamada recursiva de un nodo **v**, calcular el valor **bajo[v]** (camino alternativo), según la fórmula:

   bajo[v]:= mínimo { nbpp[v],
   nbpp[z] | siendo (v, z) un arco de retroceso,
   bajo[y] | siendo y hijo de v en el árbol }
3. La raíz es un punto de articulación si y sólo si tiene dos o más hijos en el árbol.
4. Un nodo **v** es un punto de articulación si y sólo si tiene algún hijo **w** en el árbol tal que **bajo[w]** ≥ **nbpp[v]**.

4.3.5.1. Puntos de articulación y componentes biconexas

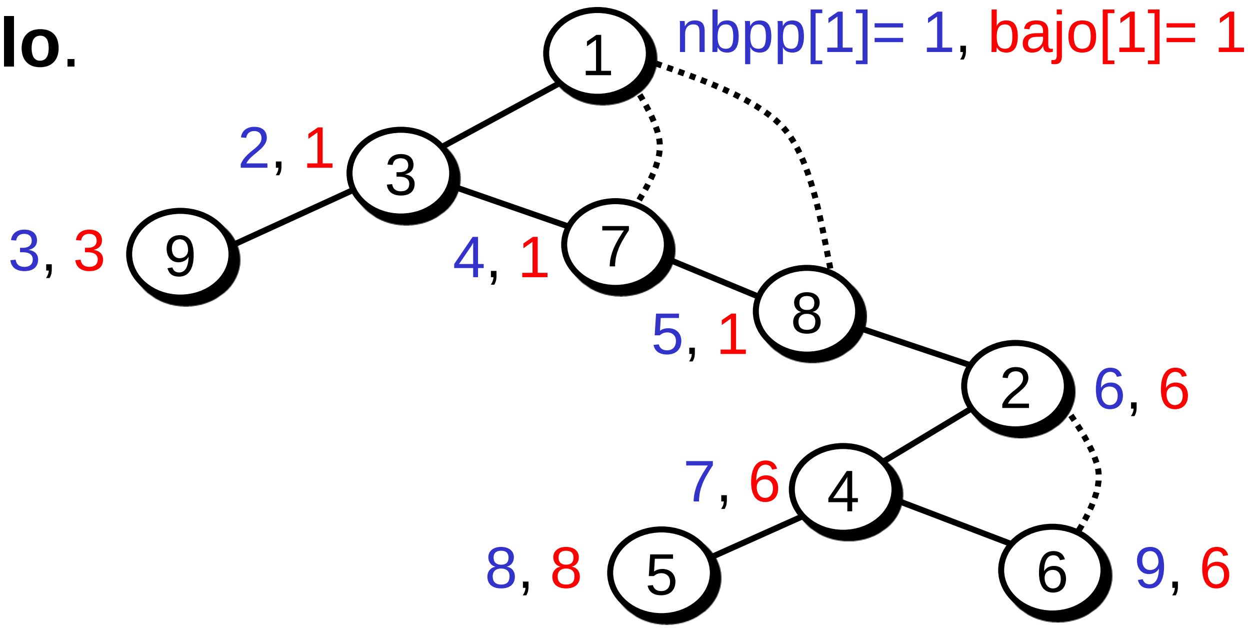
- **Ejemplo:** calcular los puntos de articulación del siguiente grafo.



- ¿Cuáles son los puntos de articulación?
- ¿Y las componentes biconexas?

4.3.5.1. Puntos de articulación y componentes biconexas

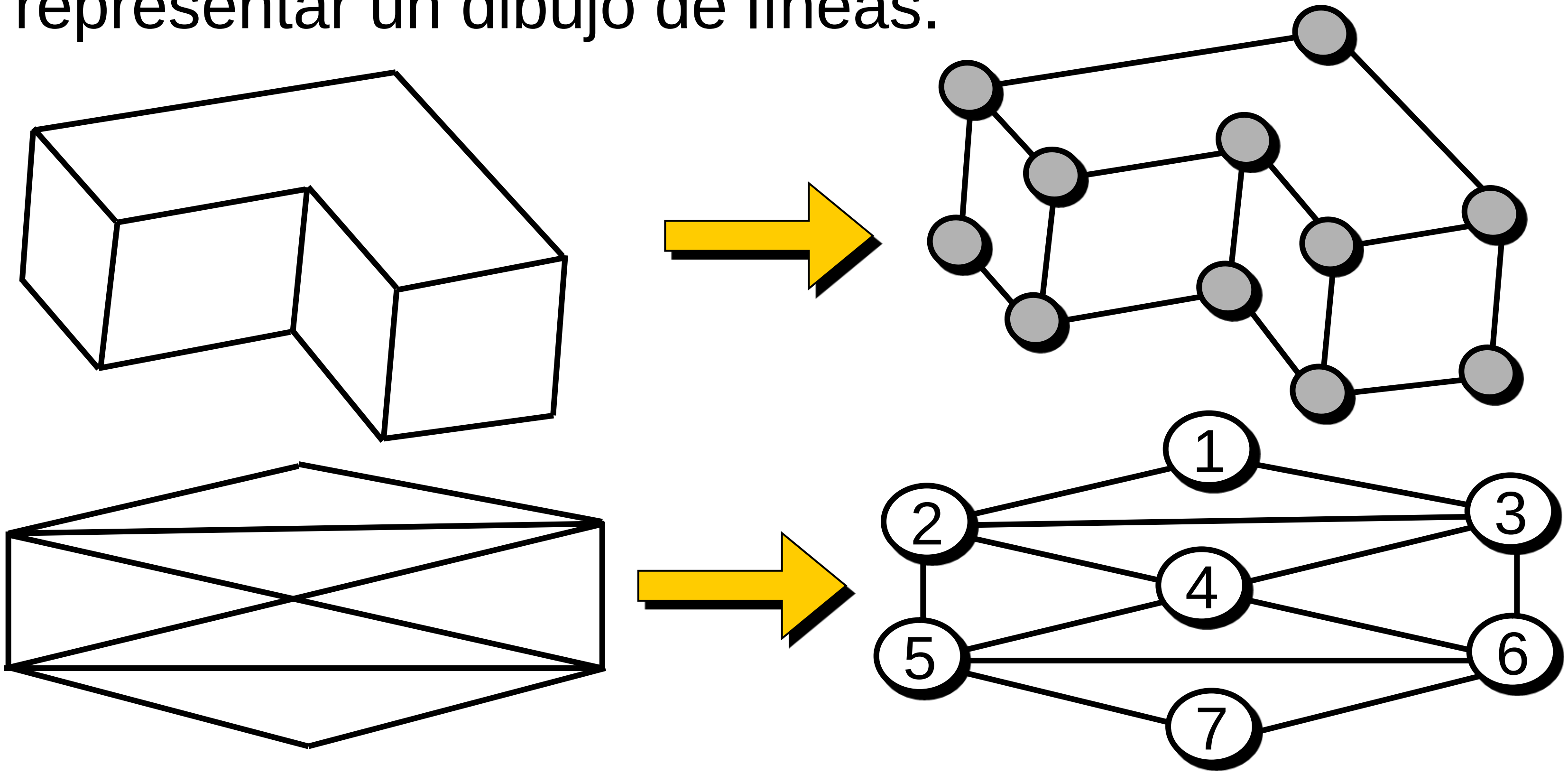
- **Ejemplo**.



- **Fundamento del algoritmo:**
  – **bajo[v]** indica el menor valor de **nbpp** alcanzable desde **v** hasta un descendiente y luego hacia arriba a través de un arco de retroceso.
  – Si se cumple la condición de 4 (bajo[w] ≥ nbpp[v]), al eliminar **v** entonces **w** y sus descendientes no pueden alcanzar los nodos antecesores de **v**.

4.3.5.2. Caminos y circuitos de Euler

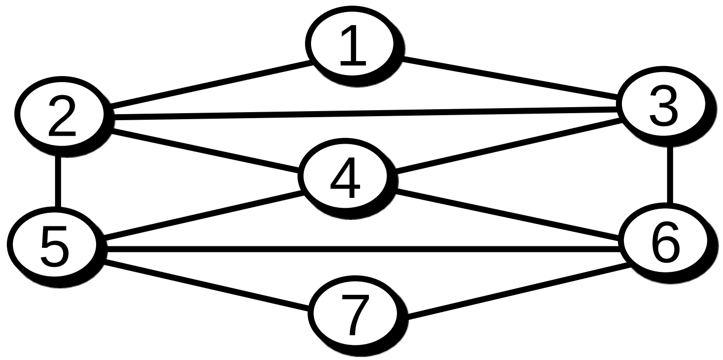
- **Aplicación:** un grafo no dirigido se utiliza para representar un dibujo de líneas.



- **Pregunta:** ¿es posible dibujar estas figuras con un bolígrafo, pintando cada línea una sola vez, sin levantar el bolígrafo y acabando donde se empezó?

4.3.5.2. Caminos y circuitos de Euler

- El problema se **transforma** en un problema de grafos.
- **Circuito de Euler:** es un ciclo (no necesariamente simple) que visita todas las aristas exactamente una vez.
- Si puede empezar y acabar en nodos distintos: **camino de Euler**.



- **Condiciones necesarias y suficientes para que exista un circuito de Euler:**
  - El grafo debe ser conexo.
  - Todos los nodos deben tener grado par, ya que el camino entra y sale de los nodos.
- ¿Y para los caminos de Euler?

4.3.5.2. Caminos y circuitos de Euler

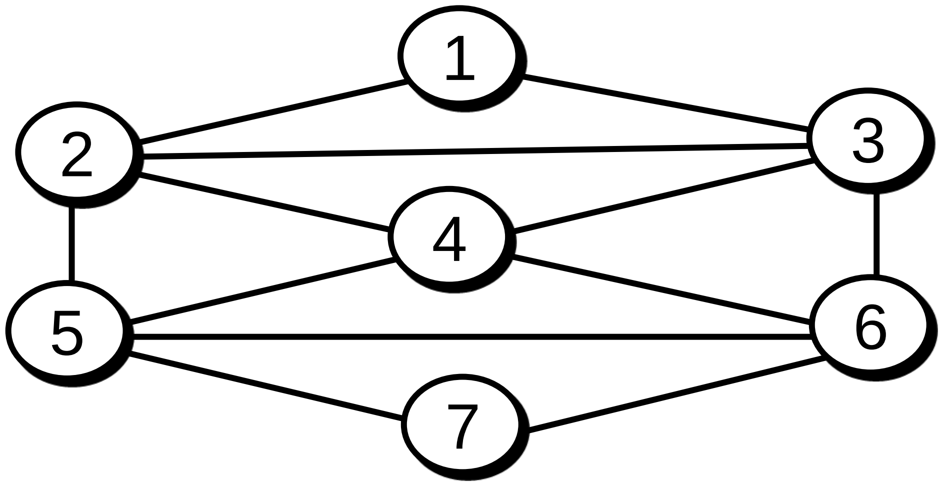
- Si existe un circuito de Euler, ¿cómo calcularlo?

- **Algoritmo para encontrar un circuito de Euler en un grafo G, partiendo de un nodo v.**
  1. Buscar un ciclo cualquiera en **G** empezando por **v**.
  2. Si quedan aristas por visitar, seleccionar el primer nodo, **w**, del ciclo que tenga una arista sin visitar. Buscar otro ciclo partiendo de **w** que pase por aristas no visitadas.
  3. Unir el ciclo del paso 1 con el obtenido en el paso 2.
  4. Repetir sucesivamente los pasos 2 y 3 hasta que no queden aristas por visitar.

- ¿Cómo encontrar un ciclo en el grafo, que pase por aristas no visitadas (pasos 1 y 2)?

4.3.5.2. Caminos y circuitos de Euler

- **Ejemplo:** encontrar un circuito de Euler para el siguiente grafo.



- ¿Cómo modificar el algoritmo para el caso del camino de Euler?

4.3.4. y 4.3.5. Algoritmos sobre grafos dirigidos y no dirigidos

Conclusiones

- Podemos utilizar grafos para **modelar problemas** de la “vida real”.

Problema de interés → Problema con grafos → Algoritmo genérico con grafos → Algoritmo para el problema de interés

- Importancia del estudio de **problemas genéricos** sobre grafos.
- La **búsqueda primero en profundidad** es una herramienta básica, subyacente en muchos de los algoritmos estudiados.

4.3.6. Otros problemas con grafos

Problemas genéricos y clásicos sobre grafos:

- **Problemas de flujo en redes:** los grafos representan canales de flujo de información, de líquidos, mercancías, coches, etc.
- **Problema del viajante:** optimización de rutas en mapas de carreteras.
- **Coloración de grafos:** los grafos representan relaciones de incompatibilidad.
- **Comparación, isomorfismo y subisomorfismo:** representación de información “semántica”, búsqueda de patrones, inteligencia artificial.

4.3.6. Otros problemas con grafos
Problemas de flujo en redes

- Supongamos un grafo dirigido **G**= (V, A) con pesos.
  - Los nodos representan puntos de una red.
  - Las aristas representan canales de comunicación existentes entre dos puntos.
  - Los pesos de cada arista **C(v, w)** representan el número máximo de unidades que pueden “fluir” desde el nodo **v** al **w**.

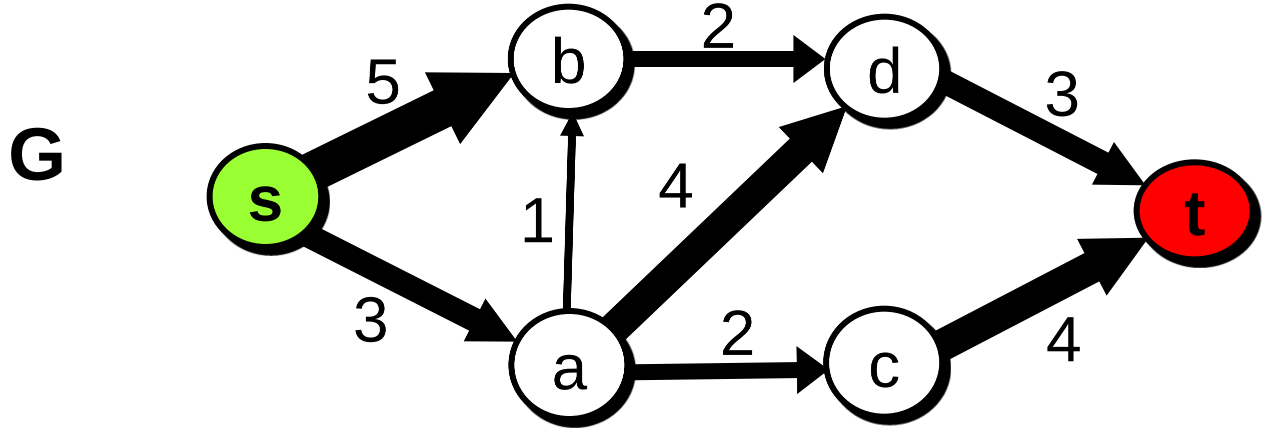
- **Problema:** encontrar el máximo volumen que se puede enviar entre dos puntos.

4.3.6. Otros problemas con grafos

- **Problema del flujo máximo:**

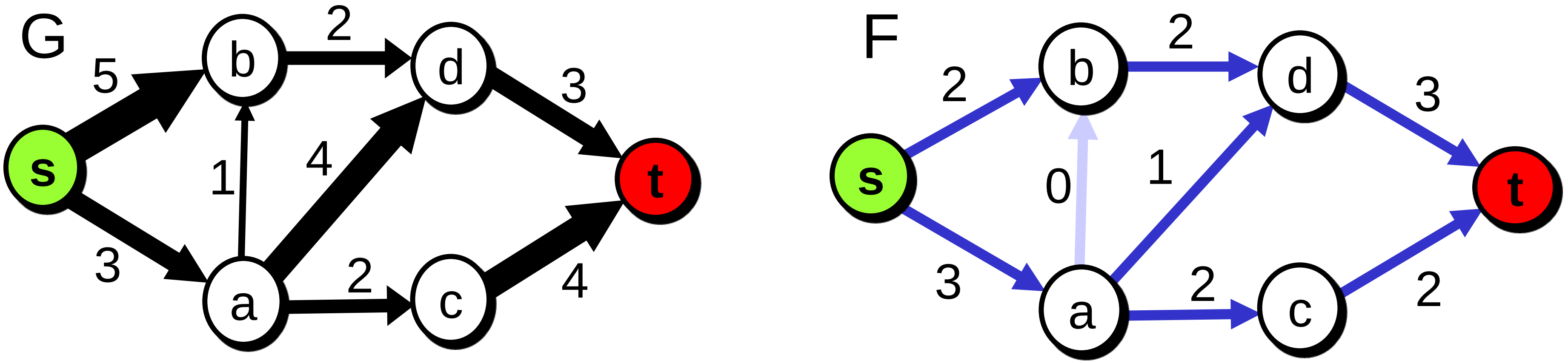
  Dado un nodo origen **s** y un nodo destino **t** en un grafo dirigido con pesos, **G**, encontrar la cantidad máxima de flujo que puede pasar de **s** a **t**.

- **Restricciones:**
  - La suma de las entradas de cada nodo interior debe ser igual a la suma de sus salidas.
  - Los valores de flujo en cada arista no pueden superar los valores máximos.



4.3.6. Otros problemas con grafos

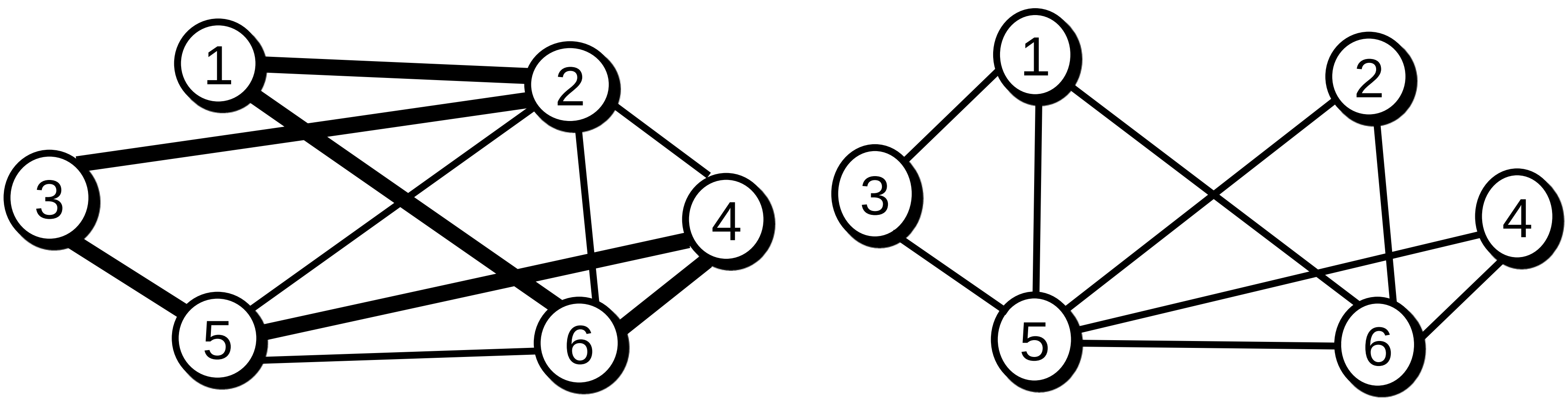
- **Solución. G:** grafo del problema. **F:** grafo resultante.



- El problema se puede resolver de forma eficiente.
- **Posible algoritmo:**
  - Encontrar un camino cualquiera desde **s** hasta **t**.
  - El máximo flujo que puede ir por ese camino es el mínimo coste de las aristas que lo forman, **m**.
  - Sumar **m** en el camino en **F**, y restarlo de **G**.
- **Ojo:** este algoritmo no garantiza solución óptima.

4.3.6. Otros problemas con grafos
Problema del ciclo hamiltoniano

- **Definición:** dado un grafo no dirigido **G**, un **ciclo de Hamilton (o hamiltoniano)** es un ciclo simple que visita todos los vértices. Es decir, pasa por todos los vértices exactamente una vez.
- **Problema del ciclo hamiltoniano.**
  Determinar si un grafo no dirigido dado tiene un ciclo hamiltoniano o no.

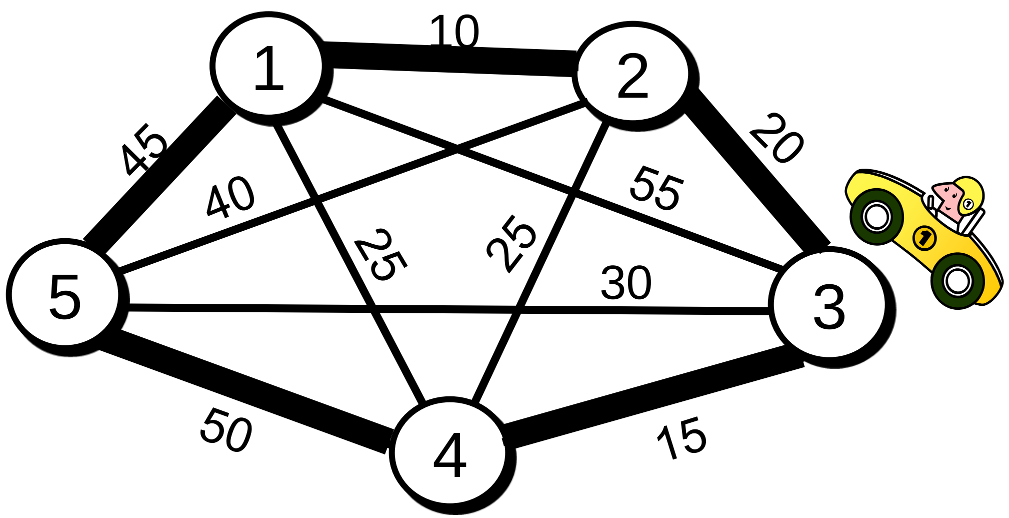


4.3.6. Otros problemas con grafos

- Aunque el problema es muy parecido al del circuito de Euler, no se conoce ningún algoritmo eficiente para resolverlo, en tiempo polinomial.
- El problema del ciclo hamiltoniano pertenece a un conjunto de problemas de difícil solución, llamados **problemas NP-completos**.
- Las soluciones conocidas requieren básicamente "evaluar todas las posibilidades", dando lugar a órdenes de complejidad exponenciales o factoriales.
- Otra alternativa es usar métodos **heurísticos**: soluciones aproximadas que pueden funcionar en algunos casos y en otros no.

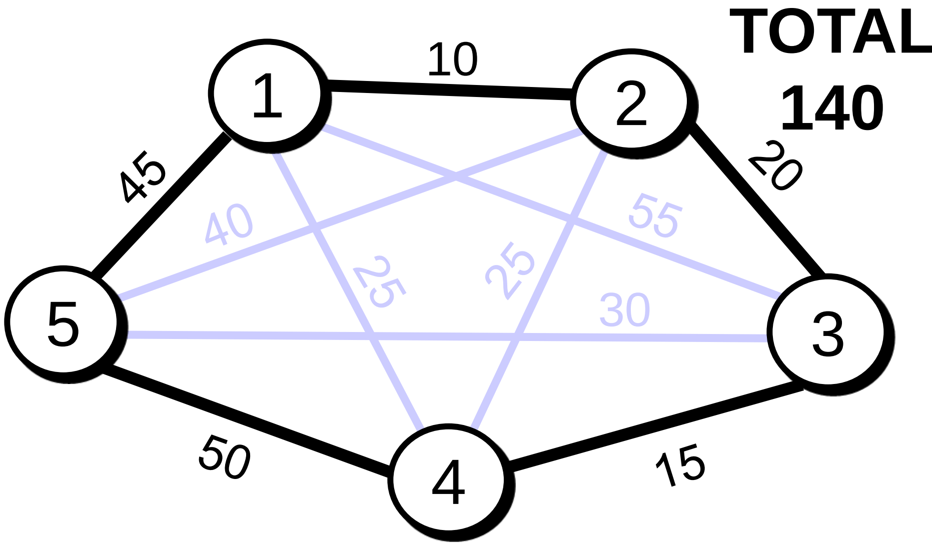
4.3.6. Otros problemas con grafos
Problema del viajante (o del agente viajero)

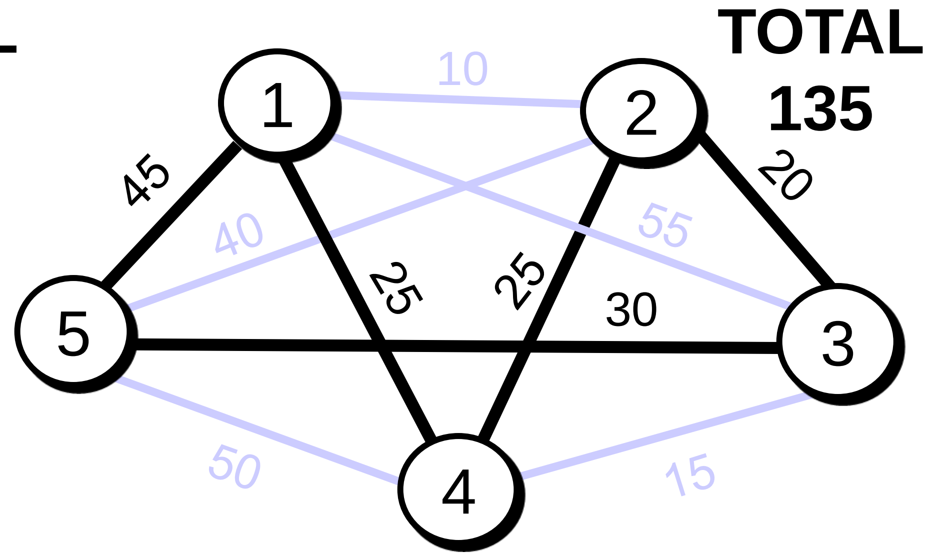
- Dado un grafo no dirigido, completo y con pesos, **G**, encontrar un ciclo simple de costo mínimo.



- **Ejemplo:** un cartero tiene que repartir cartas por todo el pueblo. ¿Qué ruta debe seguir para que el coste de desplazamiento sea mínimo?

4.3.6. Otros problemas con grafos





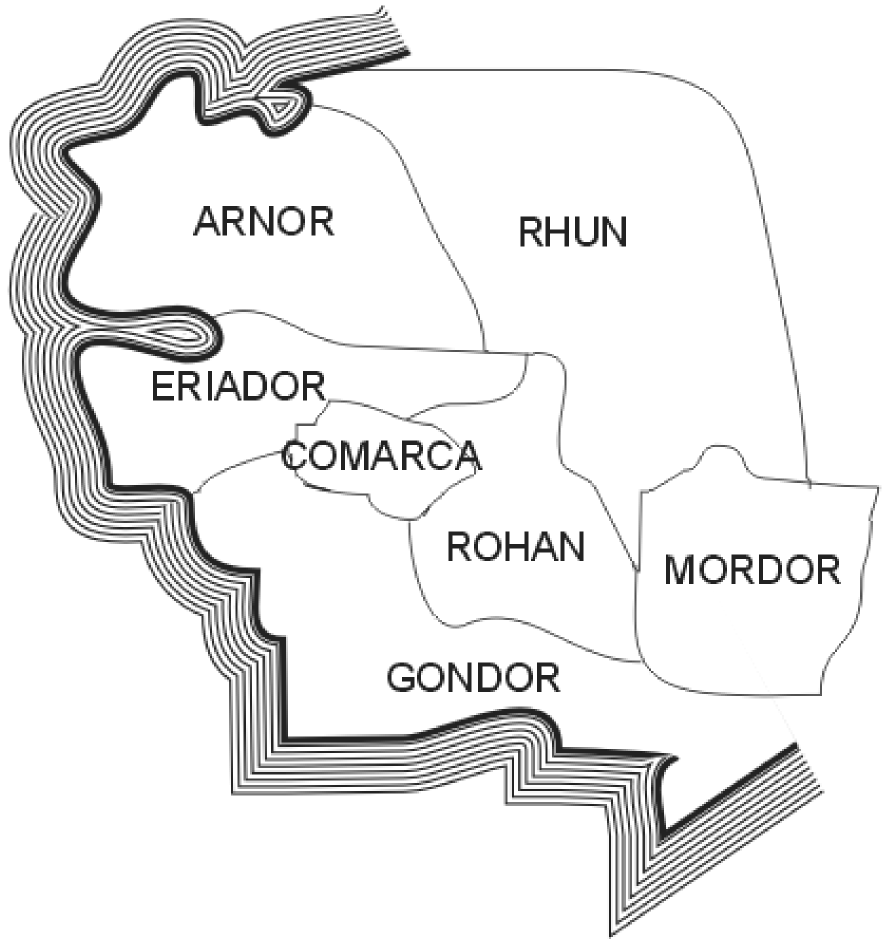
- El problema del viajante es un problema **NP-completo**, equivalente (reducible) al problema del ciclo hamiltoniano.
- No se conoce una solución con tiempo polinómico. Las soluciones conocidas tienen complejidad exponencial.
- Podemos aplicar heurísticas, técnicas probabilistas, algoritmos genéticos, computación con ADN, etc., obteniendo aproximaciones.

4.3.6. Otros problemas con grafos
Coloración de grafos

- Un grafo no dirigido **G** representa ciertos elementos.
- Una arista (**v**, **w**) representa una incompatibilidad entre los elementos **v** y **w**.
- La **coloración de un grafo** consiste en asignar un color (o etiqueta) a cada nodo, de forma que dos nodos incompatibles no tengan el mismo color.
- **Problema de coloración de grafos:** Realizar una coloración del grafo utilizando un número mínimo de colores.

4.3.6. Otros problemas con grafos

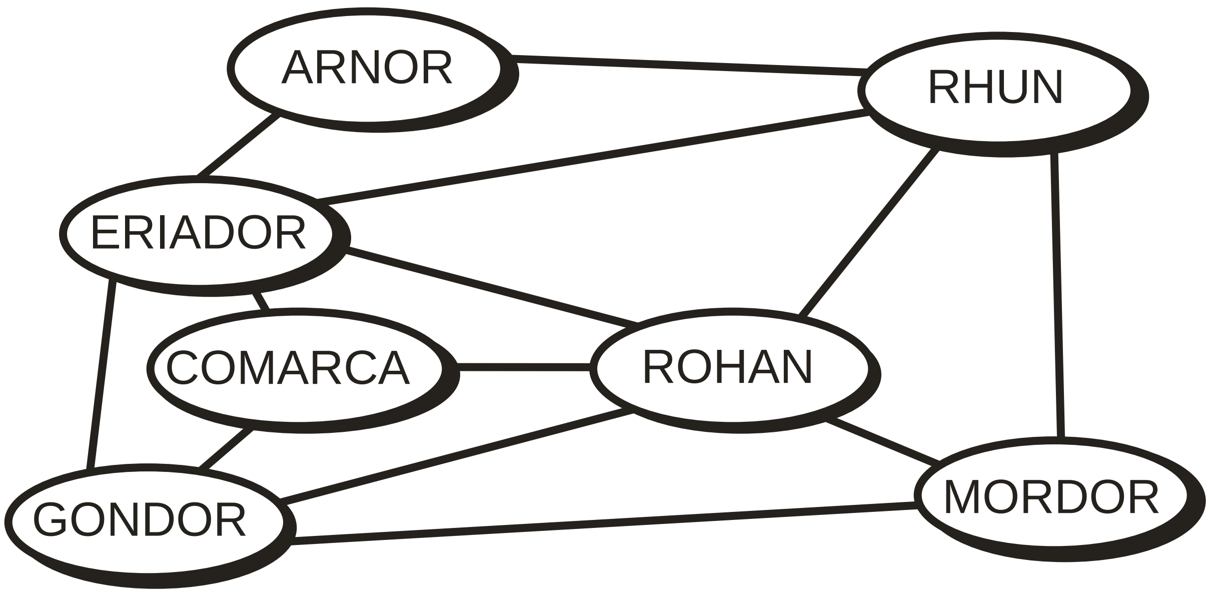
- **Ejemplo:** ¿con cuántos colores, como mínimo, se puede pintar un mapa? Dos regiones adyacentes no pueden tener el mismo color.



- Modelamos el problema con una representación de grafos.

4.3.6. Otros problemas con grafos

- **Modelado del problema:**
  - **Nodos** del grafo: regiones del mapa.
  - **Aristas** del grafo: hay una arista **(v, w)** si las regiones **v** y **w** tienen una frontera común.
  - **Solución:** encontrar la coloración mínima del grafo.



- La coloración de grafos es un problema **NP-completo**.

4.3.6. Otros problemas con grafos

Comparación e Isomorfismo de grafos

Igualdad

- **Definición:** dados dos grafos **G**= (V_G, A_G) y **F**= (V_F, A_F), se dicen que son **iguales** si $V_G = V_F$ y $A_G = A_F$.

Isomorfismo

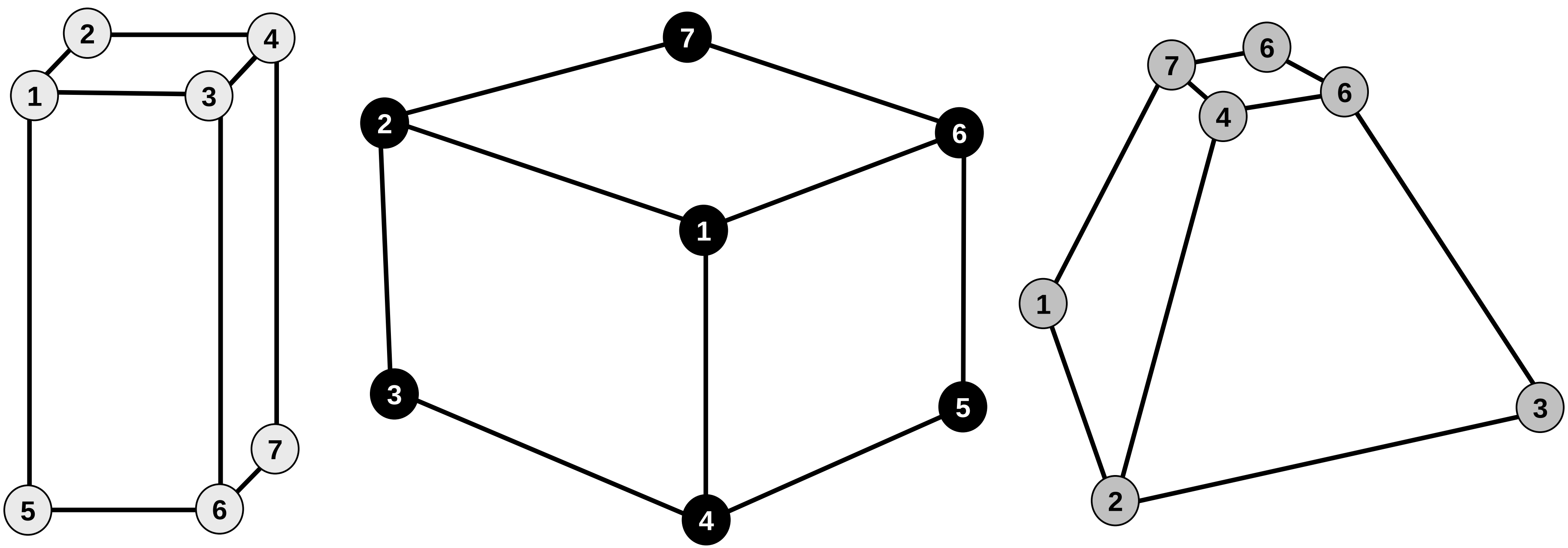
- **Definición:** dos grafos **G**= (V_G, A_G) y **F**= (V_F, A_F) se dice que son **isomorfos** si existe una asignación de los nodos de V_G con los nodos de V_F tal que se respetan las aristas.
- **Isomorfismo entre grafos.** El isomorfismo es una función:

$$\mathbf{a} : V_G \rightarrow V_F, \text{ biyectiva tal que}$$

$$(v, w) \in A_G \Leftrightarrow (a(v), a(w)) \in A_F$$

4.3.6. Otros problemas con grafos

- **Ejemplo: reconocimiento de patrones.** Identificar las figuras isomorfas y los puntos "análogos" en ambas.



- El isomorfismo de grafos es también un problema **NP-completo**.
- La solución consistiría, básicamente, en comprobar todas las posibles asignaciones.

Conclusiones

- Los grafos son una herramienta fundamental en resolución de problemas.
- **Representación:**
  - Tamaño reducido: matrices de adyacencia.
  - Tamaño grande y grafo “escaso”: listas de adyacencia.
- Existen muchos algoritmos “clásicos” para resolver diferentes problemas sobre grafos.
- **Nuestro trabajo:** saber modelar los problemas de interés usando grafos y encontrar el algoritmo adecuado para la aplicación que se requiera.
- **Problemas NP-completos sobre grafos:** diseñar un algoritmo óptimo con alto coste, o un algoritmo heurístico, aproximado pero rápido.

Continuará...