GRAFOS

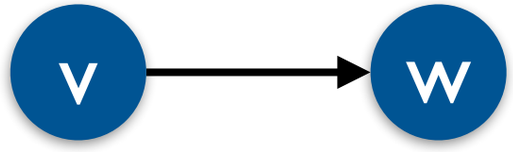
Joaquín Fernández-Valdivia
Javier Abad
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Granada

DECSAI

Grafos

- Un **grafo** es un par G = (V,E)
  - V: conjunto de vértices o nodos
  - $E \subseteq V^2 = V \times V$: conjunto de aristas o arcos
- **Grafo no dirigido**: aristas (no orientadas): el orden de los componentes de los pares de E no es relevante: (v, w) = (w, v)

- **Grafo dirigido**: arcos (con dirección): el orden de los componentes de los pares de E es relevante: (v, w) ≠ (w, v)



Grafos

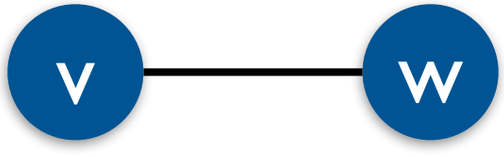
- Grafos no dirigidos
  - **Grado** de un vértice: número de aristas que lo contienen

- Grafos dirigidos
  - **Grado de salida** de un vértice, v:
    - Número de arcos cuyo vértice inicial es v
  - **Grado de entrada** de un vértice, v:
    - Número de arcos cuyo vértice final es v

Grafos

- Nodos/vértices adyacentes:

  Vértices conectados por una arista (o arco)

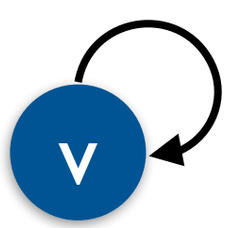


- Aristas/arcos adyacentes:

  Arcos/aristas con un vértice común

- Bucle o lazo:

  Arco/arista cuyos vértices inicial y final son el mismo



Grafos

- **Camino**: Dados u, v $\in$ V, se dice que existe un camino entre u y v si existe una secuencia de pares de E

$$(u, v_1), (v_1, v_2), ..., (v_n, v)$$

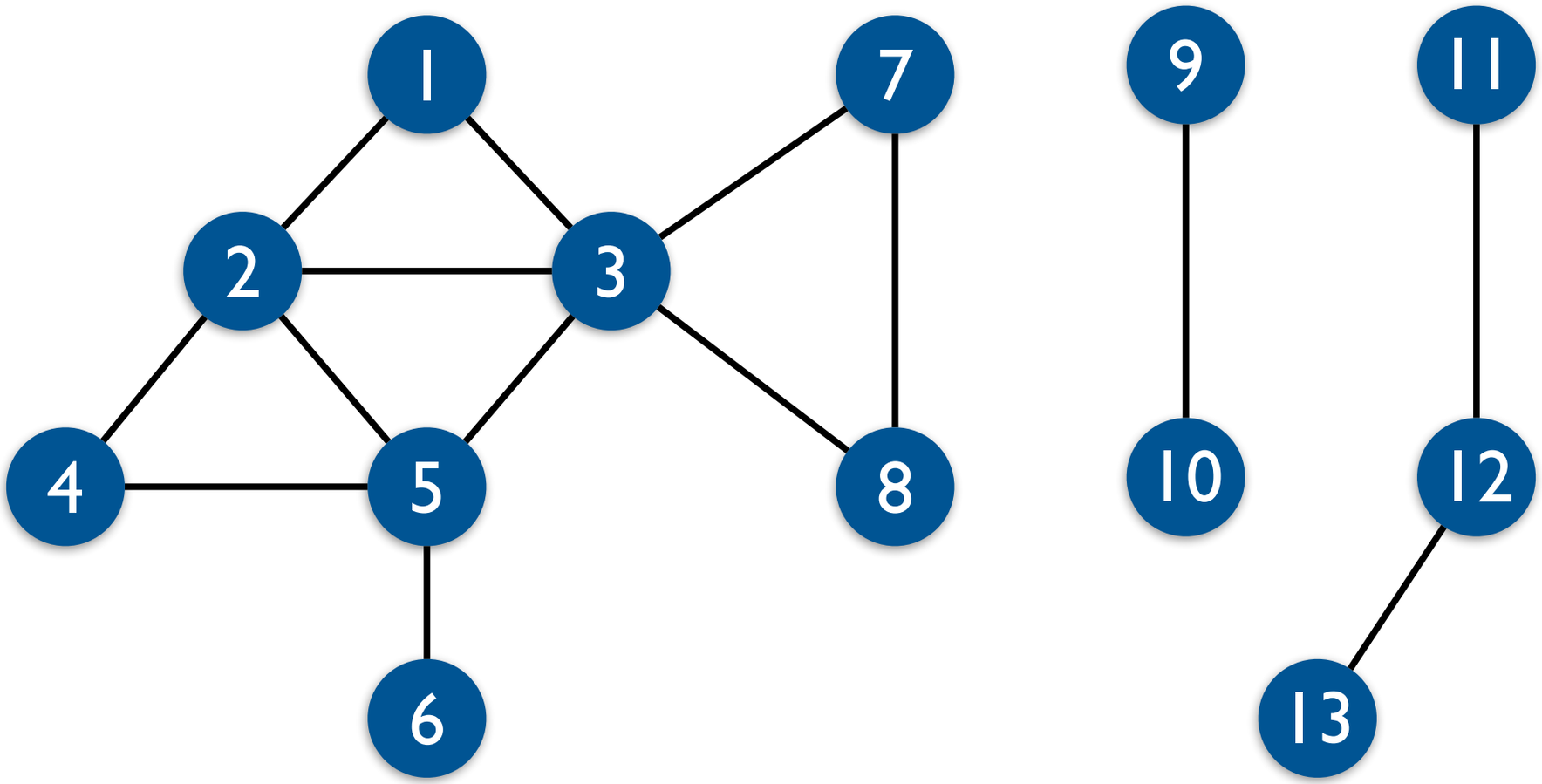
  - **Longitud del camino**: número de pares que lo componen, o número de arcos del camino

- Un camino de u a v es un **ciclo** cuando u = v

- Un camino se dice **simple** cuando en él no hay ciclos

Grafos

- Grafo conexo: un grafo no dirigido es un grafo conexo si para todo par de nodos, u y v, existe un camino de u a v
- Componentes conexas: cada uno de los conjuntos maximales conexos



Grafos

Tipos de grafos:

- **Grafo etiquetado**: cada arista y/o vértice tiene asociada una etiqueta/valor
- **Grafo ponderado**: grafo etiquetado en el que existe un valor numérico asociado a cada arista o arco
- **Multigrafo**: grafo en el que se permite que entre dos vértices exista más de una arista o arco
- **Árbol**: grafo conexo que no tiene ciclos

Grafos - Representación

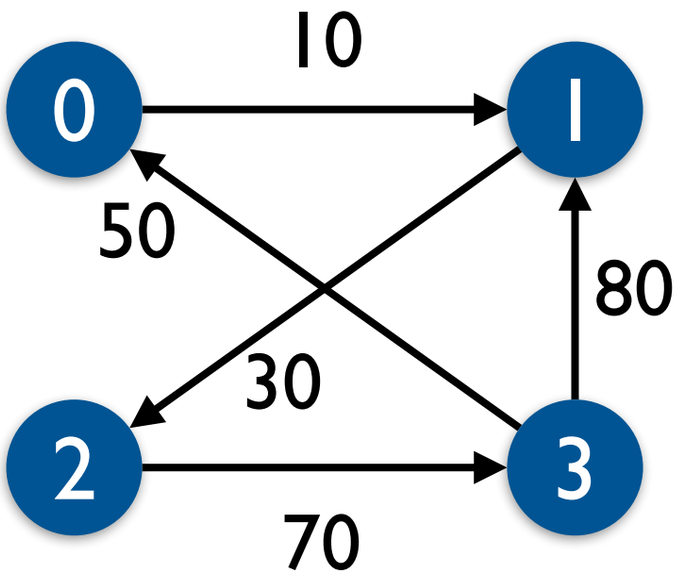
- **Matriz de adyacencia**

  Matriz cuadrada, A, de dimensión |V| × |V|, tal que

  ➡ A[i][j] = 1 indica que el lado (i,j) ∈ E

  ➡ A[i][j] = 0 indica que el lado (i,j) ∉ E

Ejemplo: Matriz de adyacencia y de costes



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

$$\begin{bmatrix} 0 & 10 & 0 & 0 \\ 0 & 0 & 30 & 0 \\ 0 & 0 & 0 & 70 \\ 50 & 80 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} - & 10 & \infty & \infty \\ \infty & - & 30 & \infty \\ \infty & \infty & - & 70 \\ 50 & 80 & \infty & - \end{bmatrix}$$

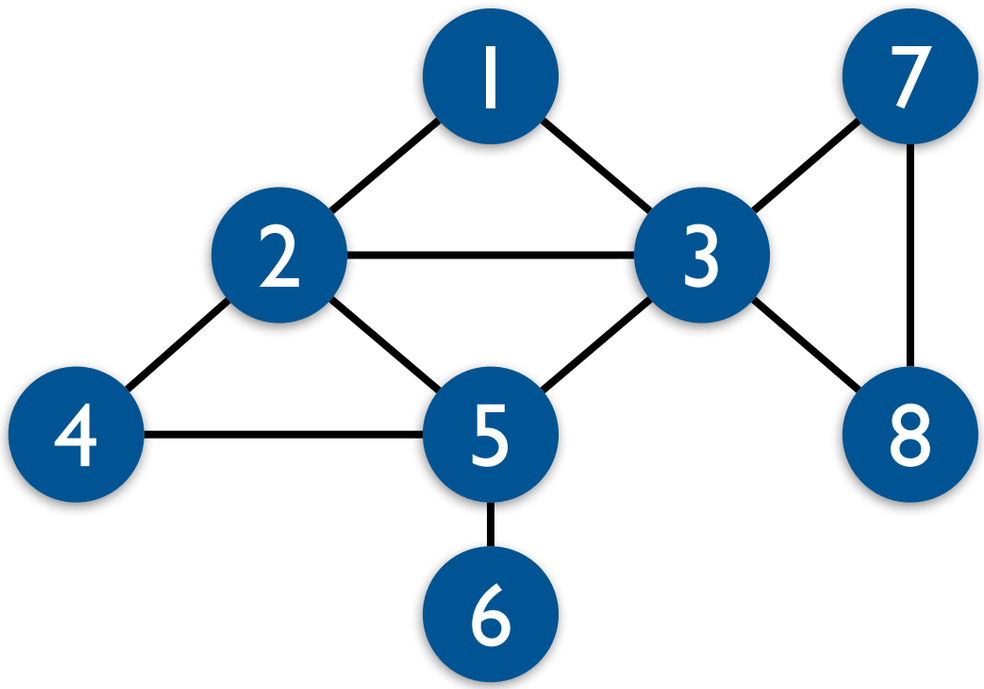
Grafos - Representación

Ventaja:

- Acceso eficiente a una arista (O(1))

Inconvenientes:

- Identificación de todas las aristas ($O(n^2)$)
- Desperdicio de memoria para grafos dispersos, consumo proporcional a n^2



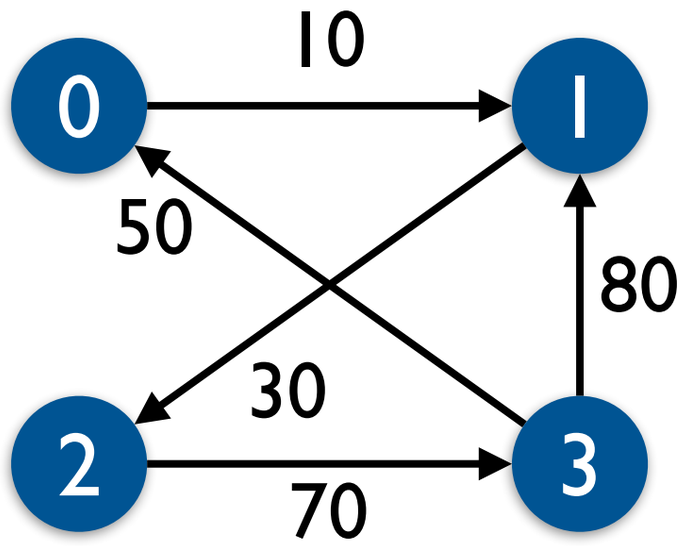
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

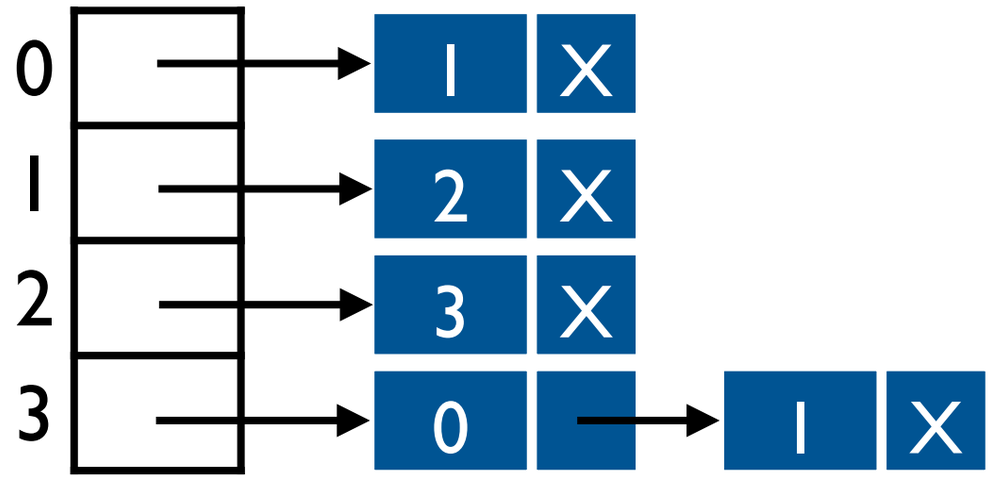
Grafos - Representación

- **Listas de adyacencias**:

  Vector de listas enlazadas de nodos adyacentes

Ejemplo:





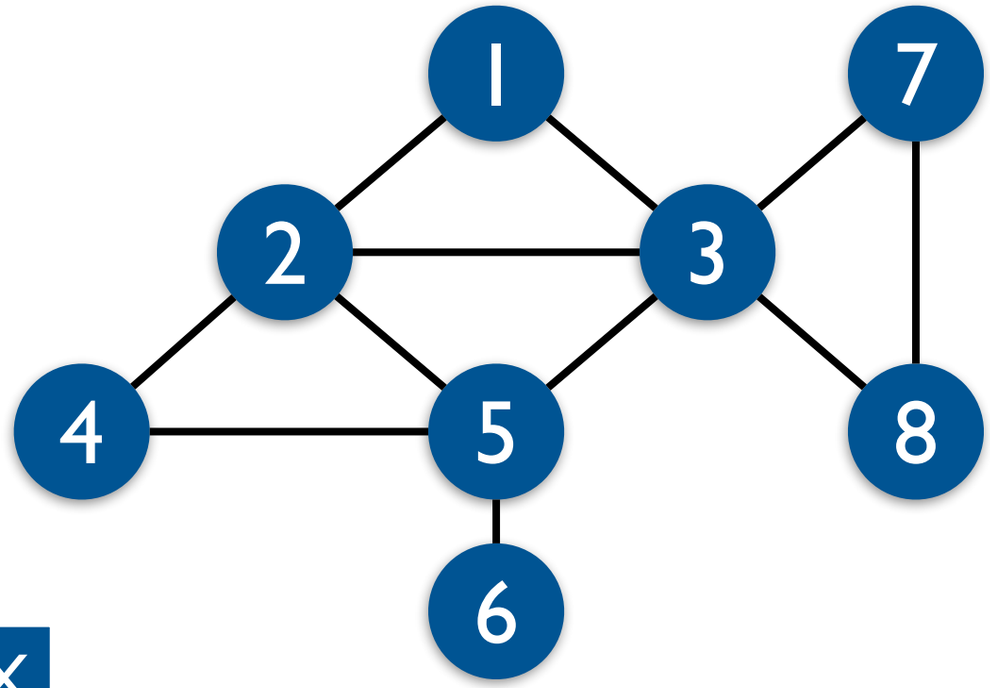
Ventajas:

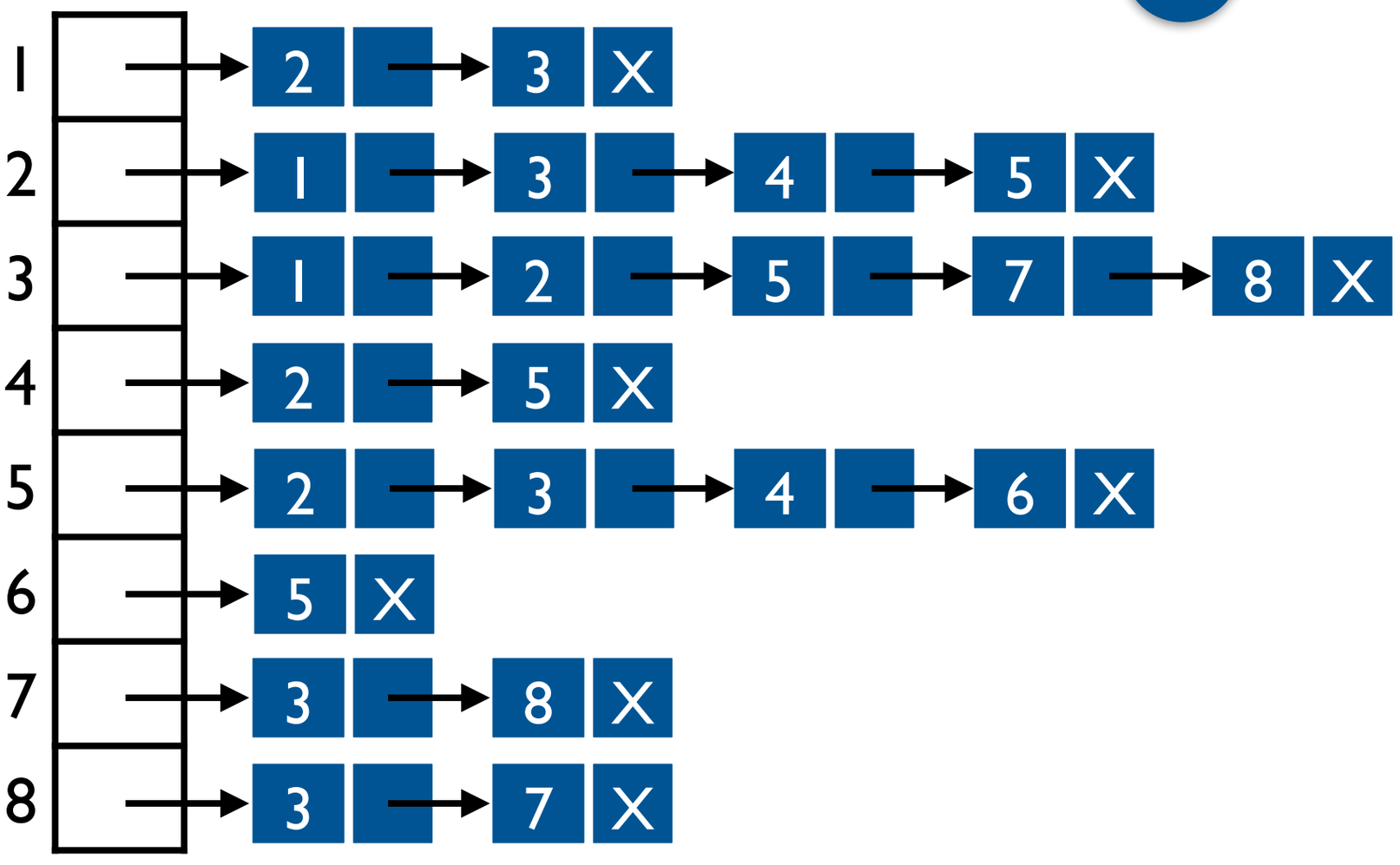
- Espacio proporcional a |V| × |E| (grafos poco densos)
- O(|V| × |E|) para identificar todas las aristas

Inconvenientes:

- O(grado(u)) para comprobar si (u,v) ∈ E
- Ineficiente para encontrar los arcos que llegan a un nodo

Grafos - Representación



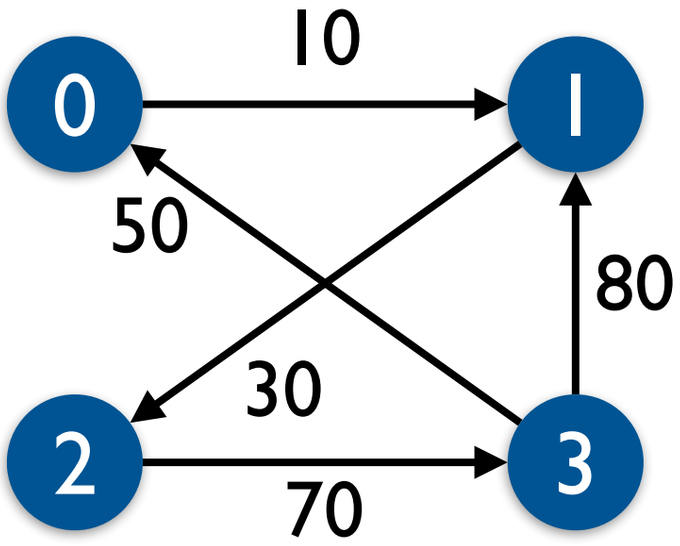


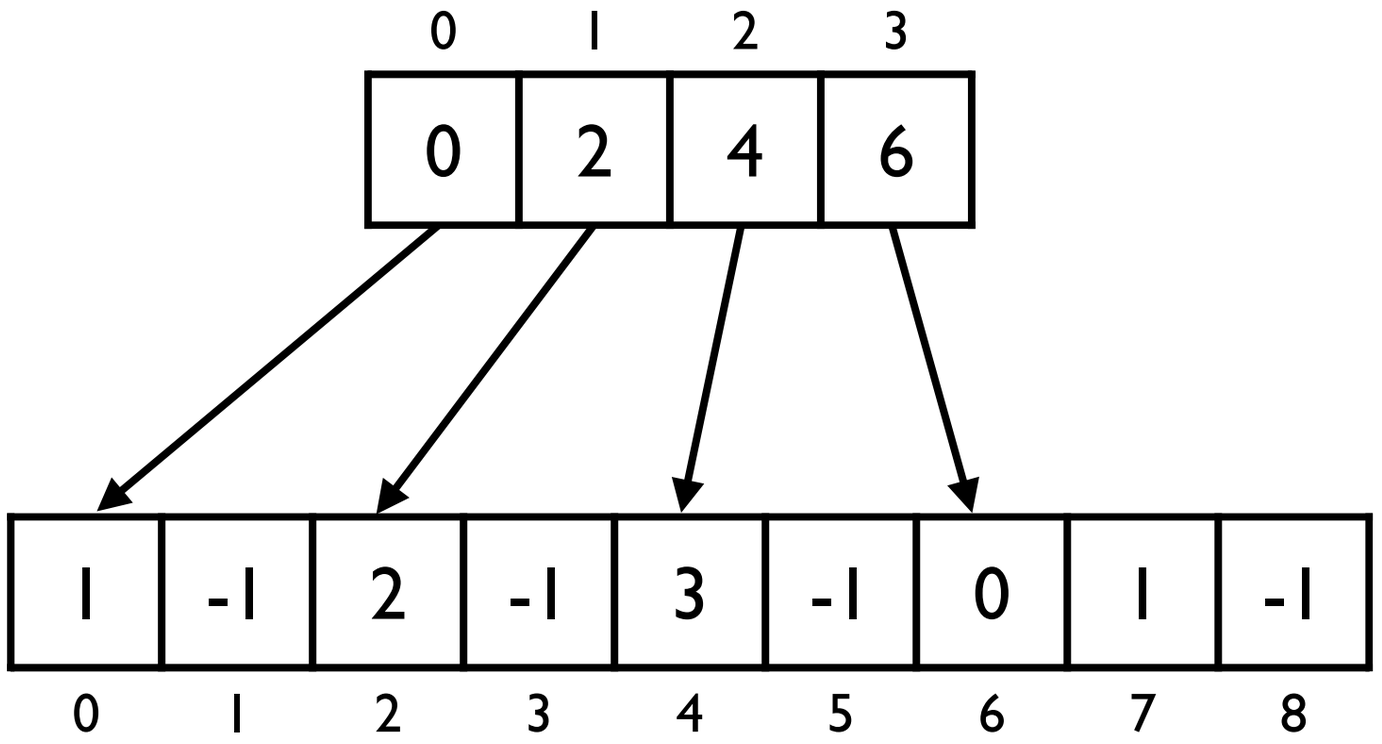
Grafos - Representación

- **Lista de cursores**

  Se usan dos vectores. En el primero se incluyen por orden todas las listas de adyacencia separadas por un valor separador (p.ej., -1). El segundo, C, con dimensión |V| tal que C[v] indica la posición del primer vector en en la que comienza la lista de adyacencia de v.

- Ejemplo:





Grafos - Representación

- **Matriz de incidencia**: matriz, I, de dimensión |V| × |E| tal que
  - ➡I[i,e] = I si e = (i,j)
  - ➡I[i,e] = -I si e = (j,i)
  - ➡I[i,e] = 0 si i ∉ e

- **Multilista**: mediante una variación de la estructura básica de la multilista podemos conseguir una implementación adecuada de la matriz de incidencia, más eficiente en grafos dispersos

- **Estructuras duales**: mezclando varias de las estructuras anteriores podemos obtener una representación de un grafo que resulta eficiente en la mayoría de algoritmos

Caminos más cortos desde un nodo

- **Algoritmo de Dijkstra**: determina el camino más corto desde un vértice dado al resto de vértices del grafo.
- Tenemos un grafo dirigido y ponderado con pesos no negativos cuyos nodos están numerados: 0, ..., n-1
- Es un algoritmo *greedy*
- En cada etapa del algoritmo obtiene la mejor solución sin considerar futuras mejoras
- La solución óptima obtenida puede refinarse/mejorarse en cada paso

Caminos más cortos desde un nodo

- V: conjunto de vértices del grafo
- C: matriz de costes, de forma que C[i,j] indica el coste de ir del nodo i al nodo j. Si no existe ese arco, el coste es ∞
- S: conjunto de vértices {v} para los que ya hemos obtenido el camino mínimo de 0 a v
- D: vector que almacena en la posición v el coste de viajar de 0 a v pasando sólo por nodos de S
- P: vector de nodos, tal que P[v] es el nodo anterior a v en el camino mínimo que hemos construido de 0 a v

Caminos más cortos desde un nodo

```
1.  S = {0}
2.  Para (i=1; i<n; i++)
3.    D[i] = C[0,i];
4.    P[i] = 0;
5.  Para (i=0; i<n-1; i++)
6.    Escoger un vértice w ∈ V-S tal que D[w] es un mínimo;
7.    Insertar(S,w);
8.    Para (cada vértice v ∈ V-S)
9.      Si (D[w] + C[w,v] < D[v])
10.       D[v] = D[w] + C[w,v];
11.       P[v] = w;
```

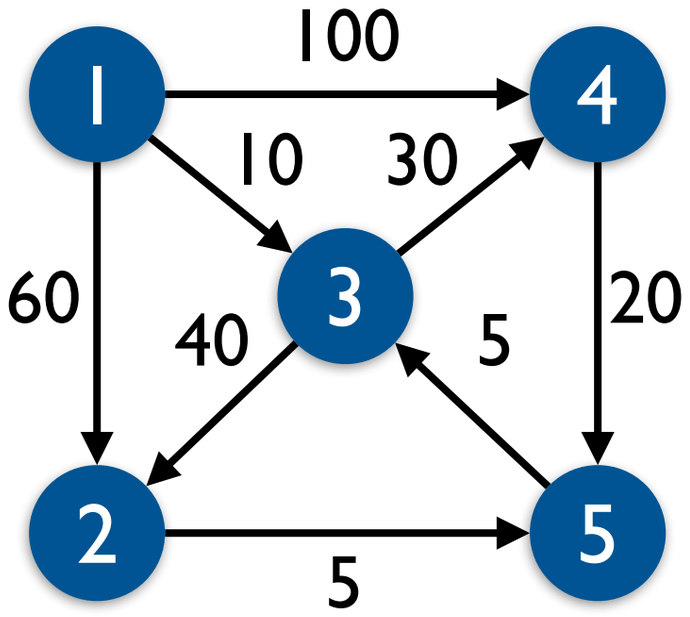
Eficiencia:

Matriz de adyacencia: $O(n^2)$

Lista de adyacencia: $O(a \log n)$

Ejemplo de aplicación: planificación de vuelos, planificación de envíos (logística), enrutamiento de paquetes en redes, reconocimiento del habla, procesamiento de imágenes…

Caminos más cortos desde un nodo. Ejemplo



Encontrar caminos mínimos del vértice 1 a los demás

Inicialmente:

S = {1}

D[2] = 60; D[3] = 10; D[4] = 100; D[5] = ∞

P[i] = 1 ∀i

Iteración 1:

V-S = {2, 3, 4, 5} w = 3 $\Longrightarrow$ S = {1, 3} $\Longrightarrow$ V-S = {2, 4, 5}

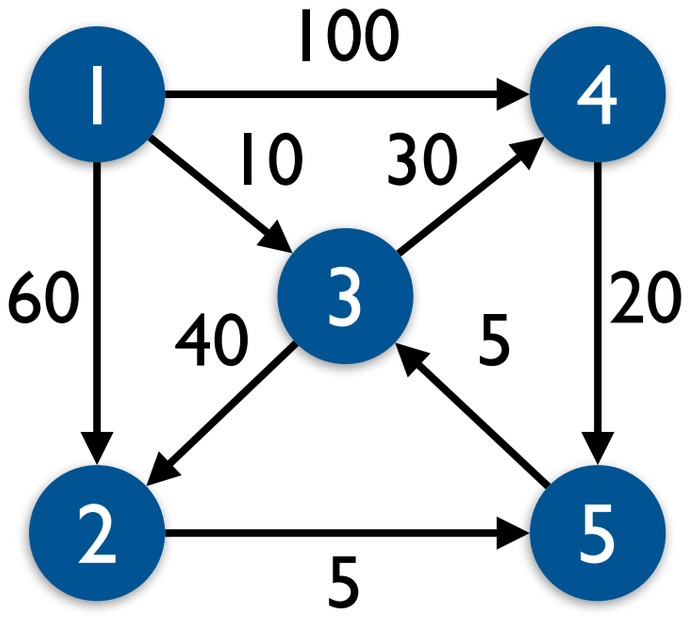
D[2] = min(D[2], D[3] + C[3,2]) = min(60, 50) = 50 $\Longrightarrow$ P[2] = 3

D[4] = min(D[4], D[3] + C[3,4]) = min(100, 40) = 40 $\Longrightarrow$ P[4] = 3

D[5] = min(D[5], D[3] + C[3,5]) = min(∞, ∞) = ∞

D[2] = 50; D[4] = 40; D[5] = ∞ P[2] = 3; P[4] = 3; P[5] = 1

Caminos más cortos desde un nodo. Ejemplo



Encontrar caminos mínimos del vértice 1 a los demás

Tras iteración 1:

D[2] = 50 P[2] = 3

D[3] = 10 P[3] = 1

D[4] = 40 P[4] = 3

D[5] = ∞ P[5] = 1

Iteración 2:

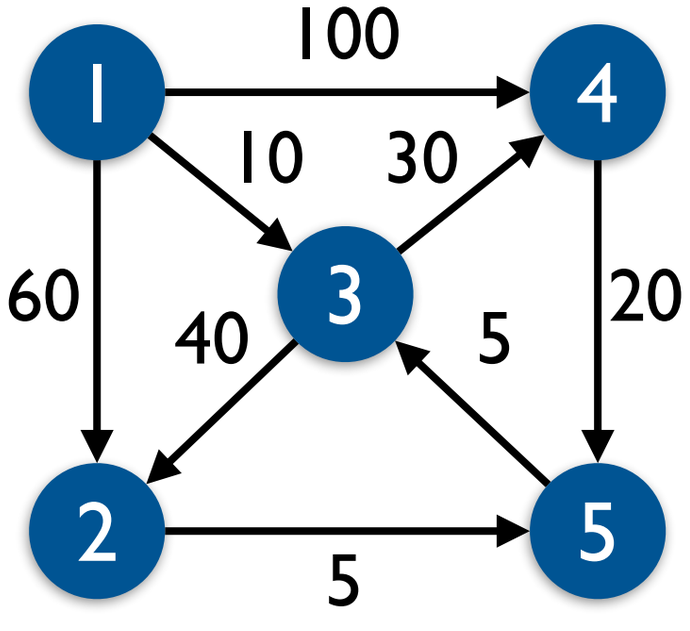
V-S = {2, 4, 5} $w = 4 \Longrightarrow S = \{1, 3, 4\} \Longrightarrow$ V-S = {2, 5}

D[2] = min(D[2], D[4] + C[4,2]) = min(50, ∞) = 50

D[5] = min(D[5], D[4] + C[4,5]) = min(∞, 60) = 60 $\Longrightarrow$ P[5] = 4

D[2] = 50; D[5] = 60 P[2] = 3; P[5] = 4

Caminos más cortos desde un nodo. Ejemplo



Encontrar caminos mínimos del vértice 1 a los demás

Tras iteración 2:

D[2] = 50 P[2] = 3

D[3] = 10 P[3] = 1

D[4] = 40 P[4] = 3

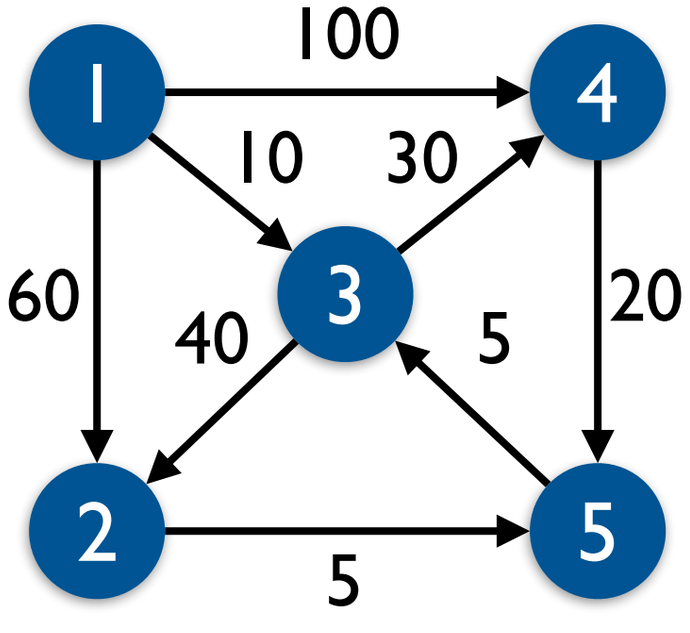
Iteración 3: D[5] = 60 P[5] = 4

V-S = {2, 5} w = 2 $\Longrightarrow$ S = {1, 3, 4, 2} $\Longrightarrow$ V-S = {5}

D[5] = min(D[5], D[2] + C[2,5]) = min(60, 55) = 55 $\Longrightarrow$ P[5] = 2

D[5] = 55 P[5] = 2

Caminos más cortos desde un nodo. Ejemplo



Encontrar caminos mínimos del vértice 1 a los demás

Tras iteración 3:

D[2] = 50 P[2] = 3

D[3] = 10 P[3] = 1

D[4] = 40 P[4] = 3

Final: D[5] = 55 P[5] = 2

w = 5 $\Longrightarrow$ S = {1, 3, 4, 2, 5} $\Longrightarrow$ FIN

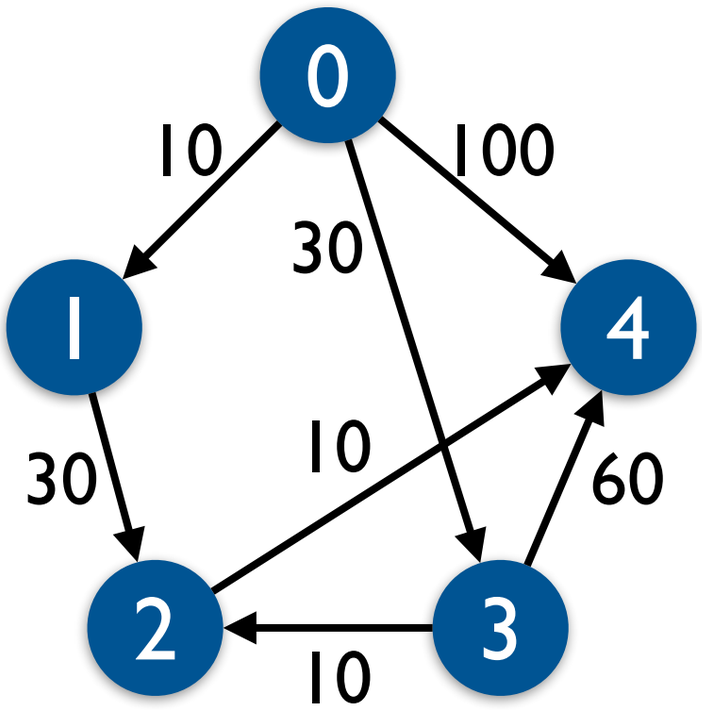
Camino de 1 a 2: 1, 3, 2 Coste 50

Camino de 1 a 3: 1, 3 Coste 10

Camino de 1 a 4: 1, 3, 4 Coste 40

Camino de 1 a 5: 1, 3, 2, 5 Coste 55

Caminos más cortos desde un nodo. Ejemplo



Iteración	S	w	D[1..4]
0	{0}		10, ∞, 30, 100
1	{0, 1}	1	10, 40, 30, 100
2	{0, 1, 3}	3	10, 40, 30, 90
3	{0, 1, 3, 2}	2	10, 40, 30, 50
4	{0, 1, 3, 2, 4}	4	10, 40, 30, 50

$$\begin{bmatrix} 0 & 10 & \infty & 30 & 100 \\ \infty & 0 & 30 & \infty & \infty \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 60 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

Iteración	S	P[0]	P[1]	P[2]	P[3]	P[4]
0	{0}	0	0	0	0	0
1	{0, 1}	0	0	1	0	0
2	{0, 1, 3}	0	0	1	0	3
3	{0, 1, 3, 2}	0	0	1	0	2
4	{0, 1, 3, 2, 4}	0	0	1	0	2

Recorridos en grafos dirigidos

- La resolución de muchos problemas relacionados con grafos dirigidos requiere realizar la visita sistemática de los vértices y/o arcos del grafo

- Principales tipos de recorrido:
  - ➡ **Búsqueda en profundidad**: generalización del recorrido en preorden de un árbol
  - ➡ **Búsqueda en anchura**: similar al recorrido por niveles de un árbol. Visita un nodo; luego intenta visitar un vecino de éste; después, un vecino del vecino, y así sucesivamente

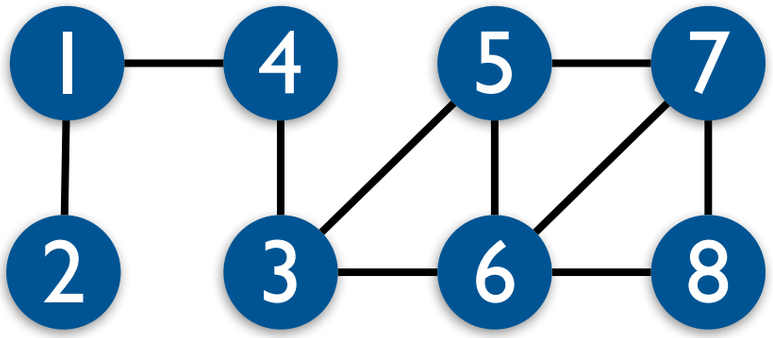
Búsqueda primero en profundidad

[DFS: *Depth-First Search*]

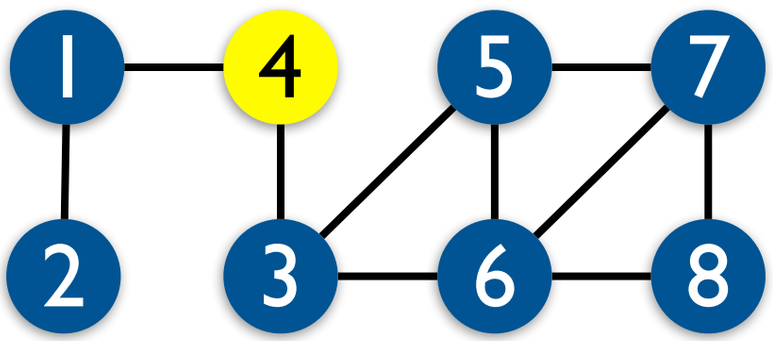
```
BusquedaProfundidad (Grafo G(V,E)) {
  for (i=0; i<V.longitud(); i++)
    visitado[i] = false;
  for (i=0; i<V.longitud(); i++)
    if (!visitado[i])
      DFS(G,i);
}


DFS(Grafo G(V,E), int i){
  visitado[i] = true;
  foreach (v[j] adyacente a v[i])
    if (!visitado[j])
      DFS(G,j);
}
```

O(|V| + |E|)
si usamos
lista de adyacencia

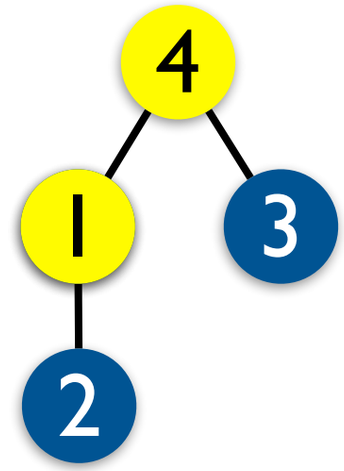
Búsqueda primero en profundidad

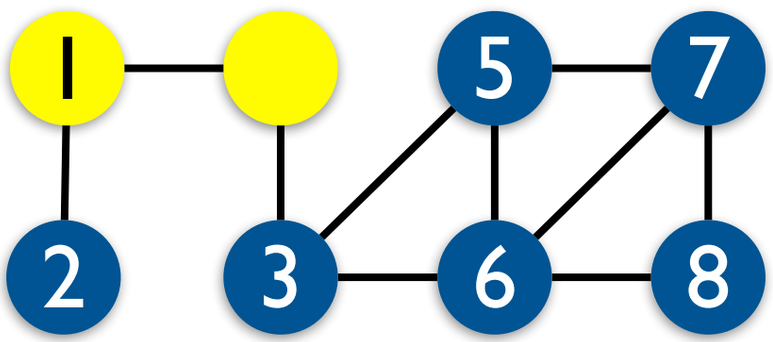


Pila S={}



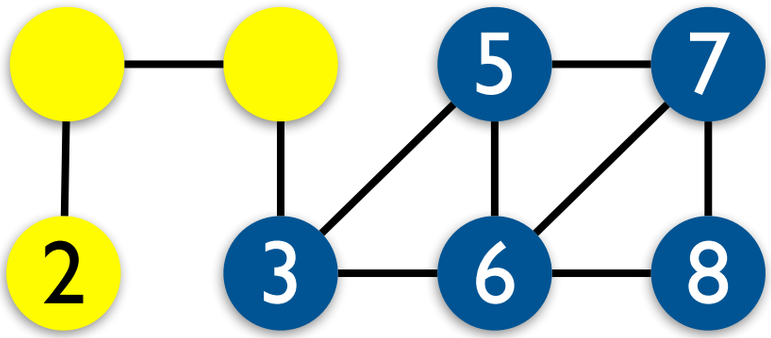
Pila S={4}

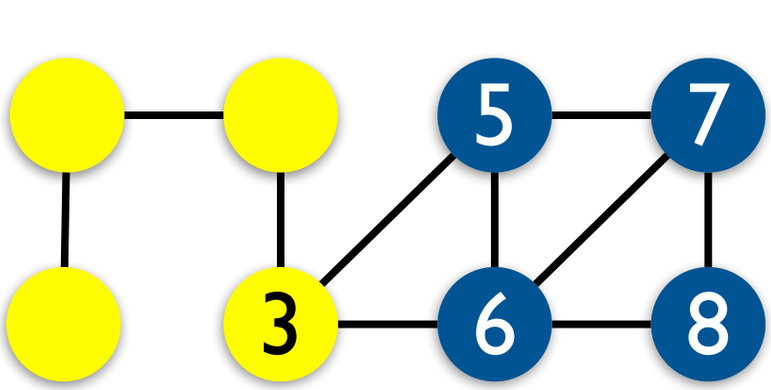


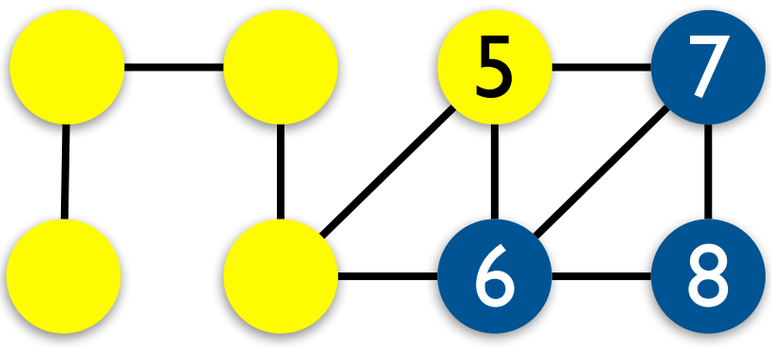


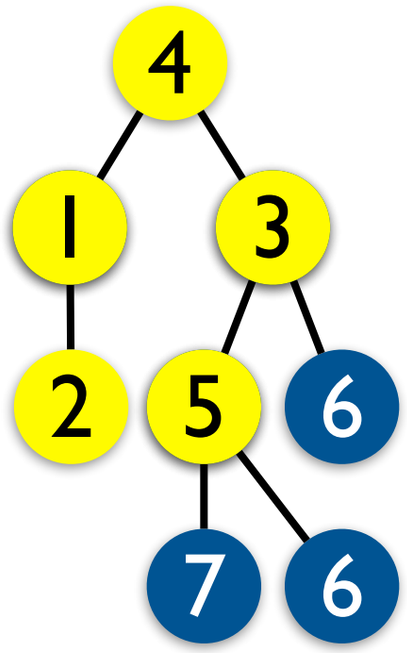
Pila S={1,4}

Búsqueda primero en profundidad

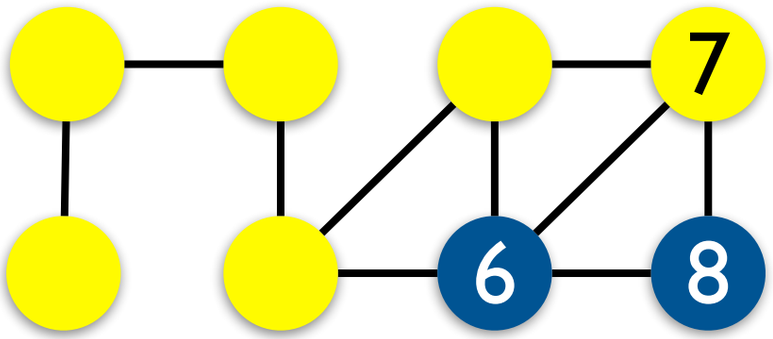


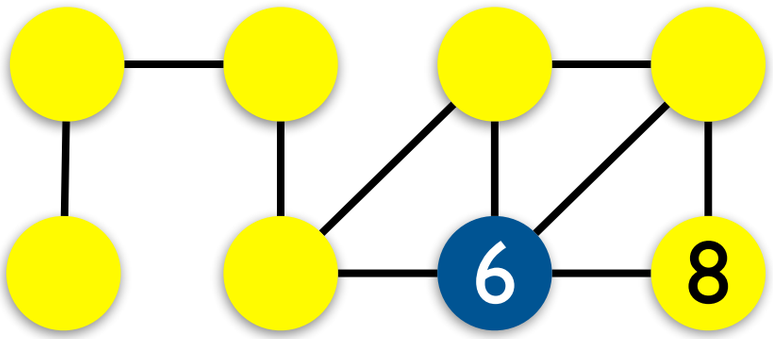


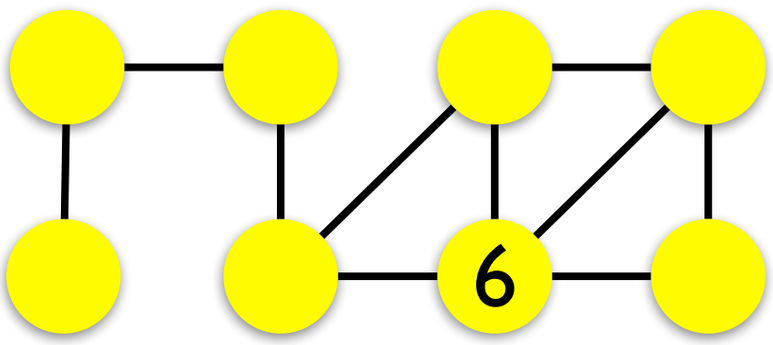


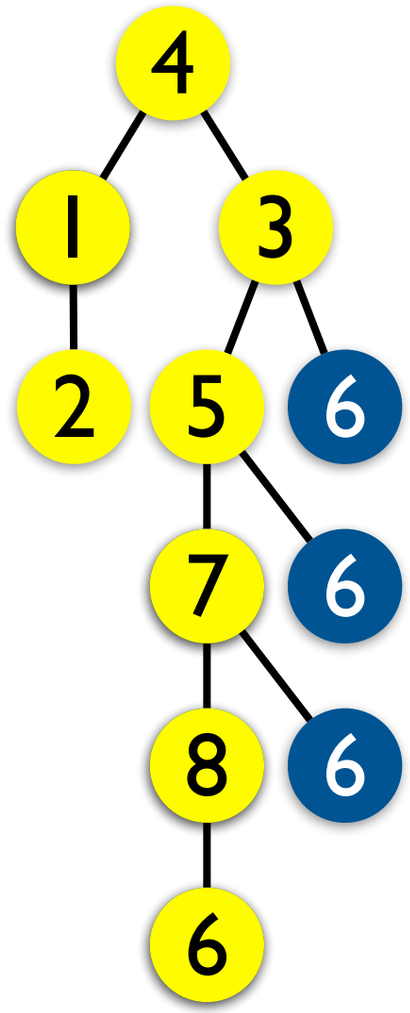


Búsqueda primero en profundidad









Búsqueda primero en profundidad

- Es necesario llevar cuenta de nodos visitados y no visitados

- **El recorrido no es único**: depende del vértice inicial y del orden de visita de los nodos adyacentes

- El orden de visita del grafo puede interpretarse como un árbol: **árbol de expansión en profundidad** asociado al grafo

Búsqueda primero en anchura

[BFS: Bread*th-First Search*]

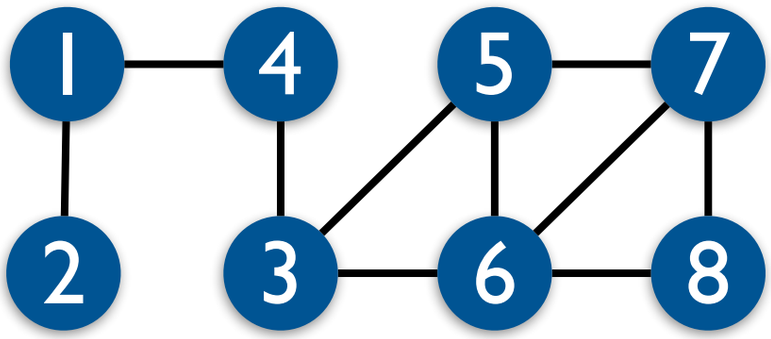
- Comenzando en un nodo, v:
  - Primero, visitamos v
  - Después se visitan todos los vértices adyacentes a v
  - A continuación, los adyacentes a cada uno de éstos... y así sucesivamente
- El algoritmo utiliza una **cola de vértices**

Búsqueda primero en anchura

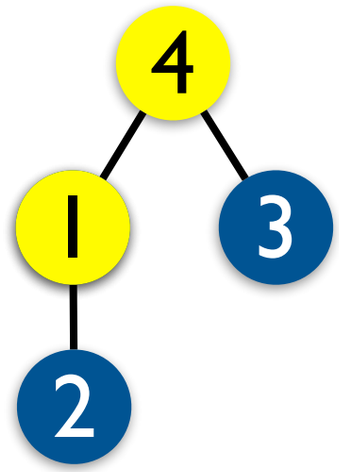
```
BusquedaAnchura (Grafo G(V,E)) {
  for (i=0; i<V.longitud(); i++)
    visitado[i] = false;
  for (i=0; i<V.longitud(); i++)
    if (!visitado[i])
      BFS(G,i);
}
BFS(Grafo G(V,E), int i){
  Cola Q;
  visitado[i] = true;
  Q.insertar(i);
  while (!Q.vacia()){
    x = Q.frente(); Q.pop();
    foreach(v[j] adyacente a v[x])
    if (!visitado[j]){
      visitado[j] = true;
      Q.insertar(j);
    }
  }
}
```

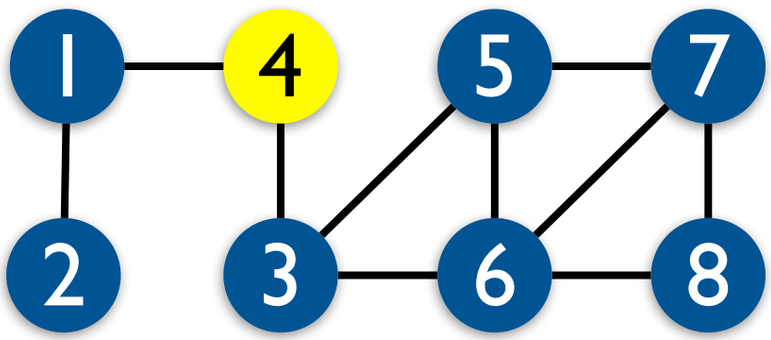
O(|V| + |E|)
si usamos
lista de adyacencia

Búsqueda primero en anchura

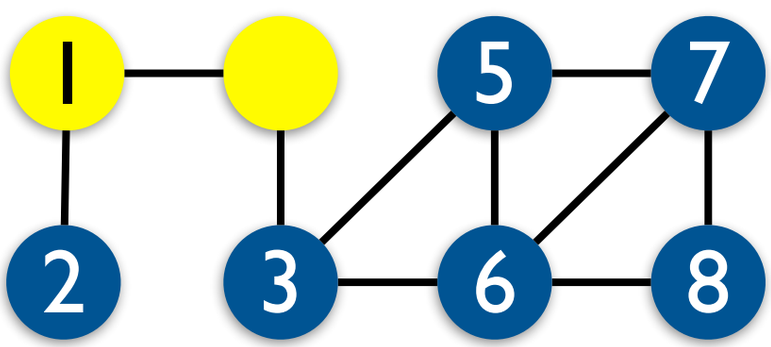


Cola Q={4}



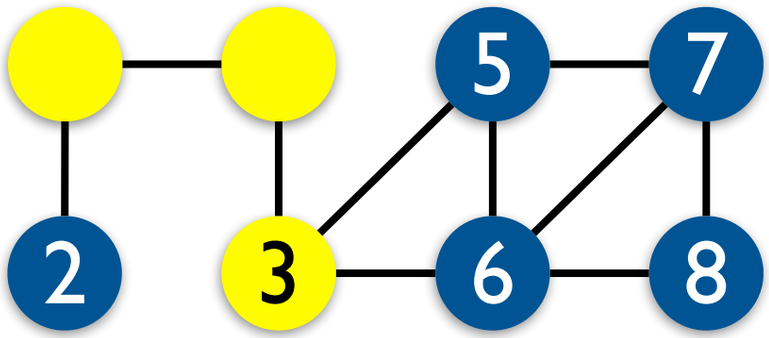


Cola Q={1,3}

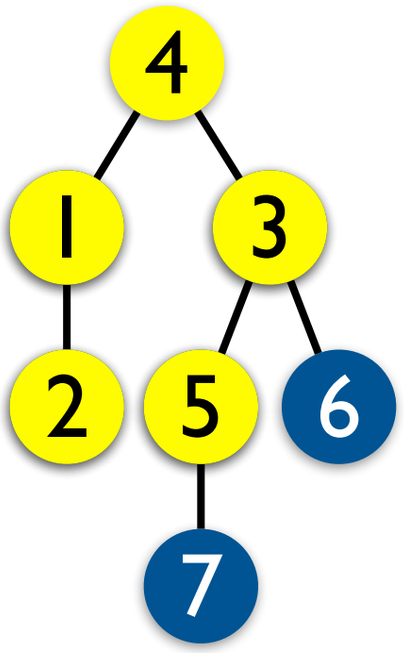


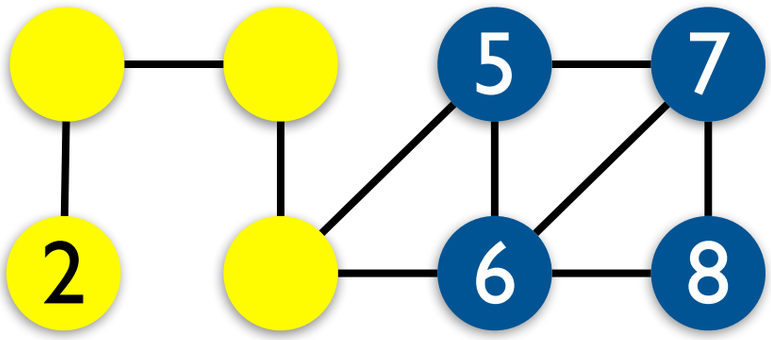
Cola Q={3,2}

Búsqueda primero en anchura

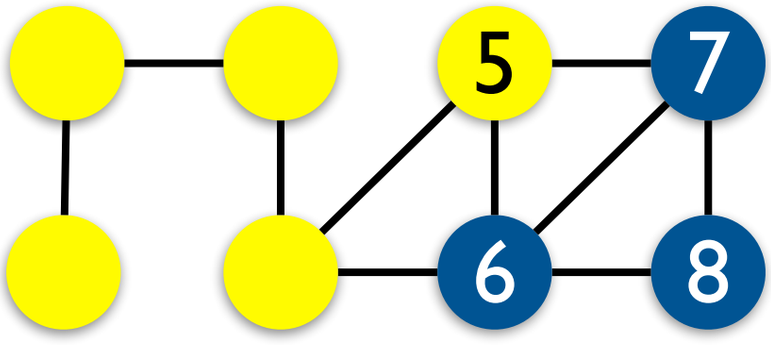


Cola Q={2,5,6}



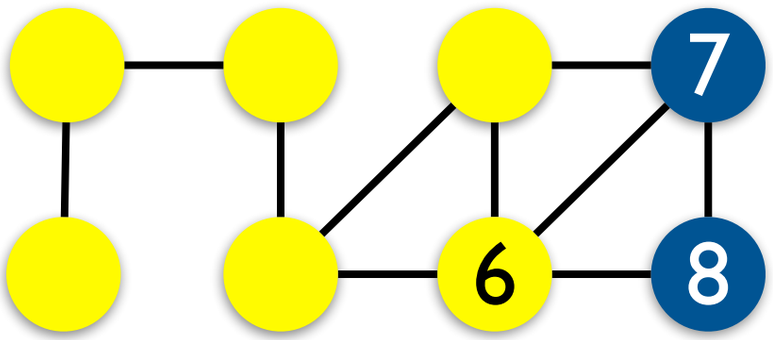


Cola Q={5,6}

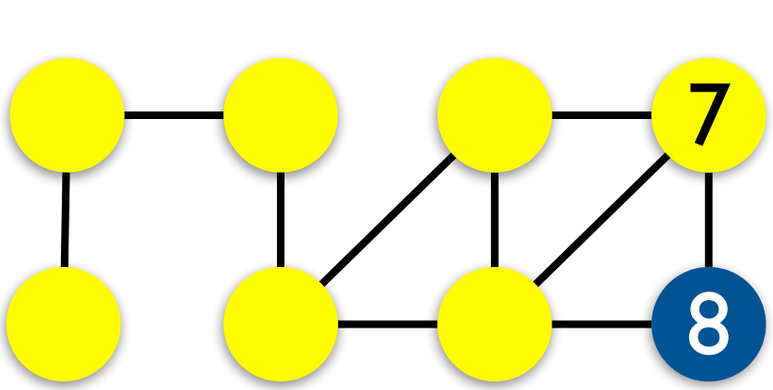


Cola Q={6,7}

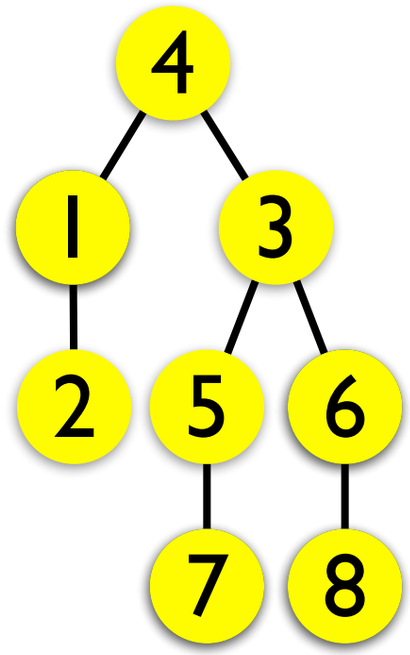
Búsqueda primero en anchura

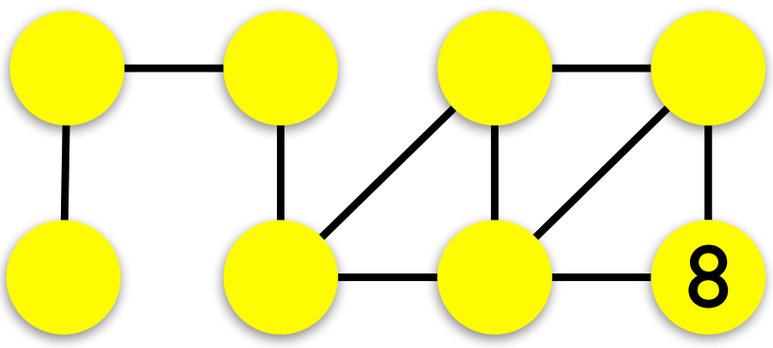


Cola Q={7,8}



Cola Q={8}





Cola Q={}

Determinación de ciclos en un grafo

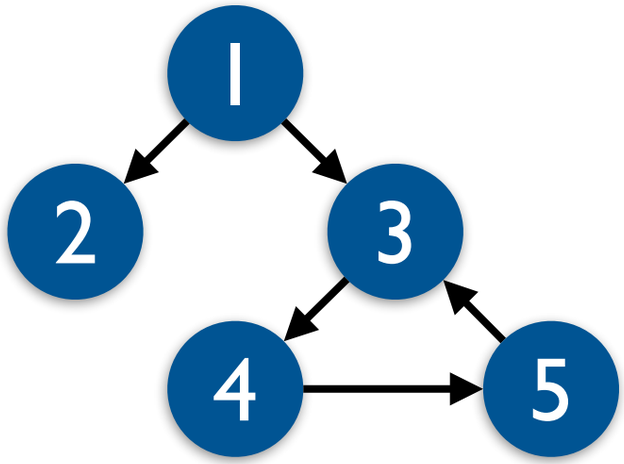
- Idea básica:

  Hacer sobre el grafo una búsqueda primero en profundidad marcando en cada llamada recursiva el vértice visitado hasta que en una llamada encontremos uno previamente visitado, en cuyo caso paramos, porque habremos encontrado un ciclo

Determinación de ciclos en un grafo

```
1. fin = false; ciclo = false;
2. ciclico(vertice v, vector visitados, int fin,
           int ciclo, vertice ultimo){
3.   visitados[v] = true;
4.   i=1;
5.   while (i≤n && !fin)
6.     if(M(v,i) == 1)
7.       if(visitados[i] == false)
8.         ciclico(i, visitados, fin, ciclo, ultimo);
9.       else{
10.        fin = true;
11.       ciclo = true;
12.       ultimo = i;
13.      }
14.    else i++;
15.  if(ciclo){
16.    imprimir(v);
17.    if (ultimo == v) ciclo = false;
18.  }
19.}
```

Determinación de ciclos en un grafo



CICLO

3, 4, 5, 3

ciclico(1)
- visitado[1] = true;
- ciclico(2)
  - visitado[2] = true;
  - X
- ciclico(3)
  - visitado[3] = true;
  - ciclico(4)
    - visitado[4] = true;
    - ciclico(5)
      - visitado[5] = true;
      - ciclico(3) $\Longrightarrow$ FIN