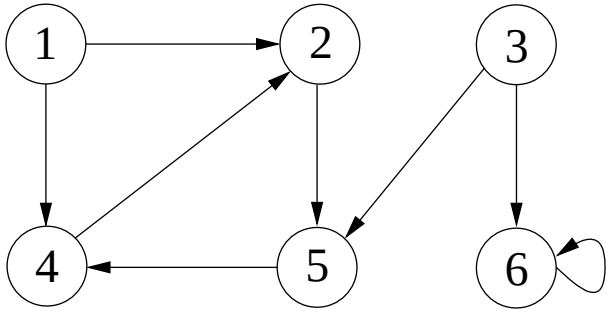
Estructuras de Datos y Algoritmos

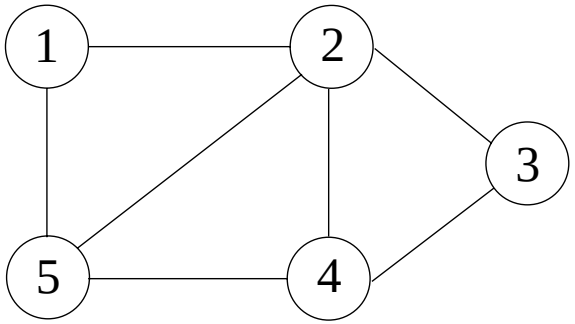
Grafos

Definiciones

- Grafo$\rightarrow$ modelo para representar relaciones entre elementos de un conjunto.
- **Grafo**: (V,E), V es un conjunto de *vértices* o *nodos*, con una relación entre ellos; E es un conjunto de pares (u,v), $u,v \in V$, llamados *aristas* o *arcos*.
- **Grafo dirigido**: la relación sobre V no es simétrica. Arista $\equiv$ par ordenado (u,v).
- **Grafo no dirigido** la relación sobre V sí es simétrica. Arista $\equiv$ par no ordenado $\{u,v\}$, $u,v \in V$ y $u \neq v$



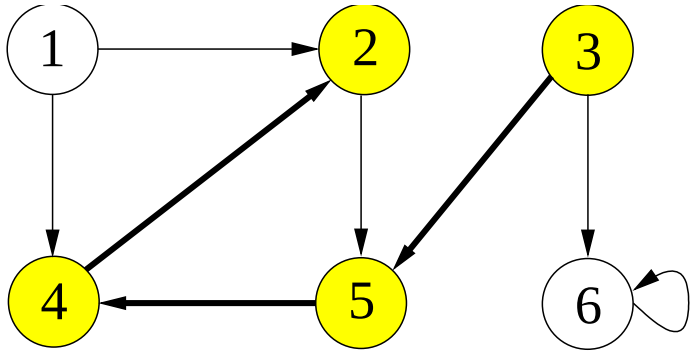
Grafo dirigido $G(V, E)$.
$V = \{1,2,3,4,5,6\}$
$E = \{(1,2),(1,4),(2,5),(3,5),(3,6),(4,2),$
$(5,4),(6,6)\}$



Grafo no dirigido $G(V, E)$.
$V = \{1,2,3,4,5\}$
$E = \{\{1,2\},\{1,5\},\{2,3\},\{2,4\},\{2,5\},$
$\{3,4\},\{4,5\}\}$

Definiciones (II)

- **Camino** desde $u \in V$ a $v \in V$: secuencia v_1, v_2, ... , v_k tal que $u = v_1$, $v = v_k$, y $(v_{i-1},v_i) \in E$, para i =2, ... ,k.
  Ej: camino desde 3 a 2 $\rightarrow$<3,5,4,2>.

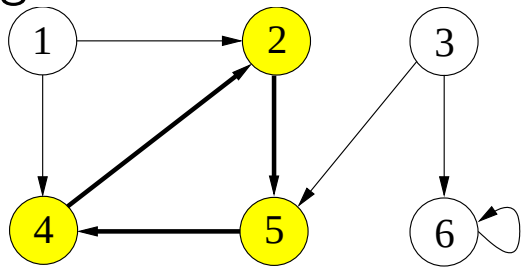


- **Longitud de un camino**: número de arcos del camino.

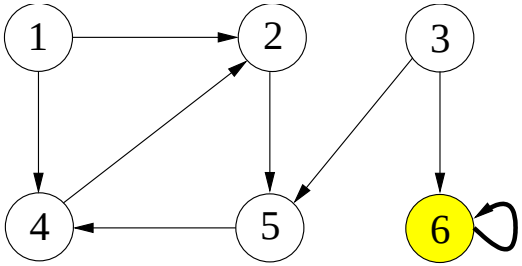
- **Camino simple**: camino en el que todos sus vértices, excepto, tal vez, el primero y el último, son distintos.

Definiciones (III)

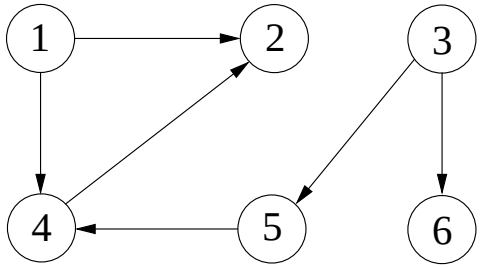
- **Ciclo**: camino simple v_1, v_2, ... , v_k tal que $v_1 = v_k$.
  Ej: <2,5,4,2> es un ciclo de longitud 3.



- **Bucle**: ciclo de longitud 1.



- **Grafo acíclico**: grafo sin ciclos.



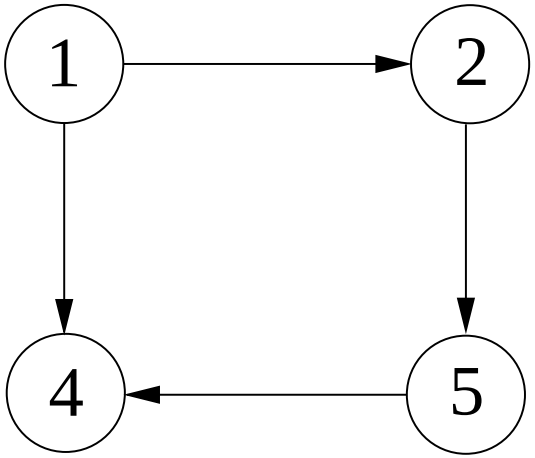
Definiciones (IV)

- v es **adyacente** a u si existe una arista $(u,v) \in E$.
- En un grafo no dirigido, $(u,v) \in E$ **incide** en los nodos u, v.
- En un grafo dirigido, $(u,v) \in E$ **incide** en v, y **parte** de u.
- **Grado** de un nodo: número de arcos que inciden en él.
- En grafos dirigidos existen el **grado de salida** y el **grado de entrada**.
  El grado del vértice será la suma de los grados de entrada y de salida.
- **Grado de un grafo**: máximo grado de sus vértices.

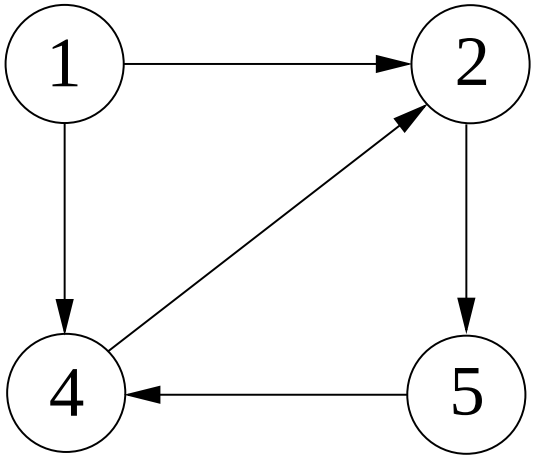
Definiciones (V)

- $G' = (V', E')$ es un **subgrafo** de $G = (V, E)$ si $V' \subseteq V$ y $E' \subseteq E$.

- **Subgrafo inducido** por $V' \subseteq V$: $G' = (V', E')$ tal que $E' = \{(u,v) \in E \mid u,v \in V'\}$.
  Ejemplos de subgrafos del grafo de la transparencia 1:



$V' = \{1,2,4,5\}$
$E' = \{(1,2),(1,4),(2,5),(5,4)\}$



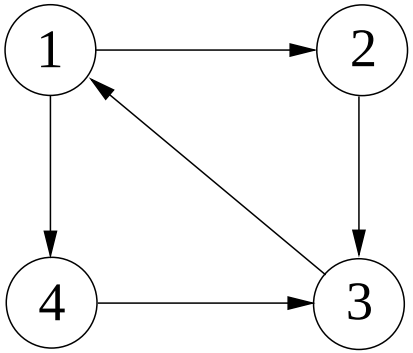
Subgrafo inducido por
$V' = \{1,2,4,5\}$

Definiciones (VI)

- v es **alcanzable desde** u, si existe un camino de u a v.

- Un grafo no dirigido es **conexo** si existe un camino desde cualquier vértice a cualquier otro.

- Un grafo dirigido con esta propiedad se denomina **fuertemente conexo**:



- Si un grafo dirigido no es fuertemente conexo, pero el grafo subyacente (sin sentido en los arcos) es conexo, el grafo es **débilmente conexo**.

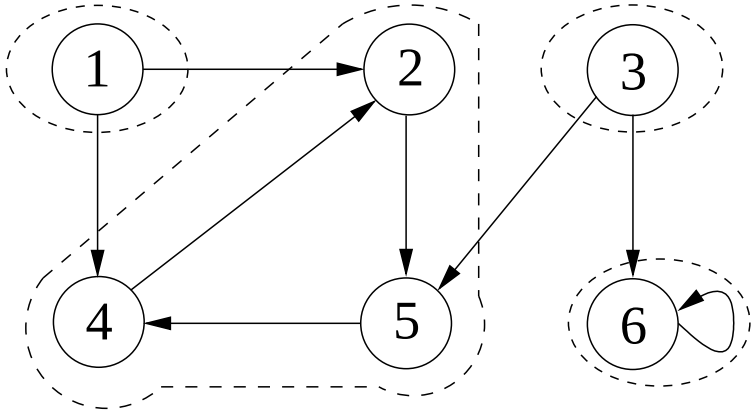
Definiciones (VII)

- En un grafo no dirigido, las **componentes conexas** son las clases de equivalencia según la relación "ser alcanzable desde".

- Un grafo no dirigido es **no conexo** si está formado por varias componentes conexas.

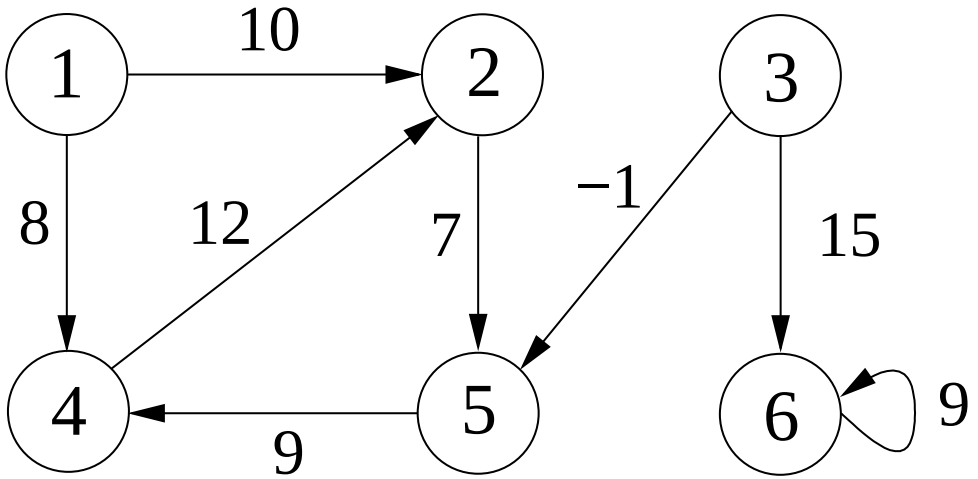
- En un grafo dirigido, las **componentes fuertemente conexas**, son las clases de equivalencia según la relación "ser mutuamente alcanzable".

- Un grafo dirigido es **no fuertemente conexo** si está formado por varias componentes fuertemente conexas.



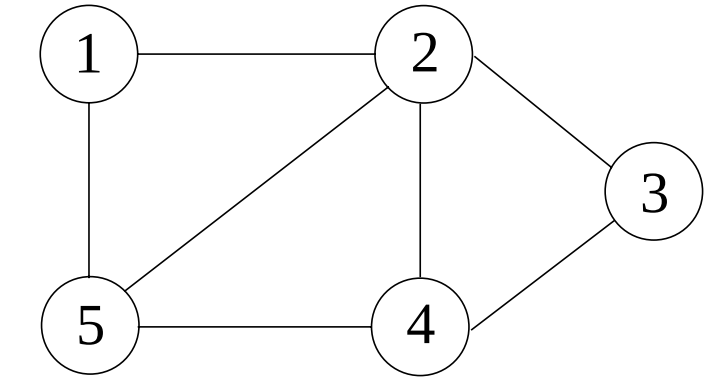
Definiciones (VIII)

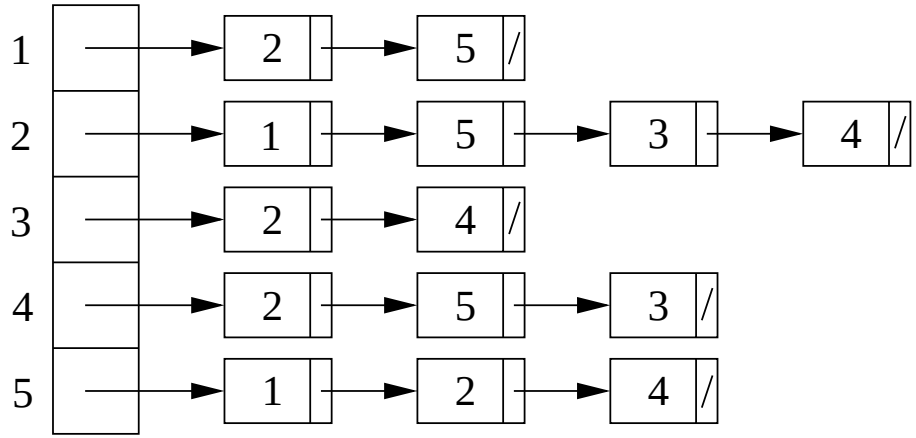
- **Grafo ponderado o etiquetado**: cada arco, o cada vértice, o los dos, tienen asociada una etiqueta.

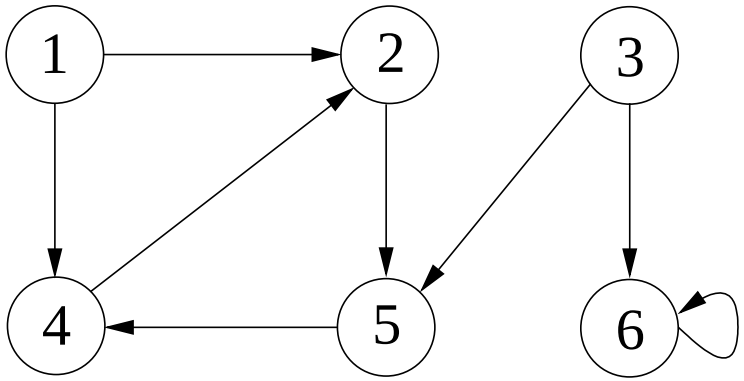


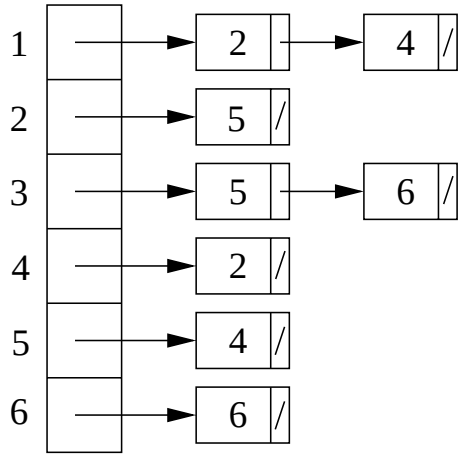
Representación de grafos: Listas de adyacencia

- $G = (V,E)$: vector de tamaño $|V|$.
- Posición $i \rightarrow$ puntero a una lista enlazada de elementos (*lista de adyacencia*). Los elementos de la lista son los vértices adyacentes a i







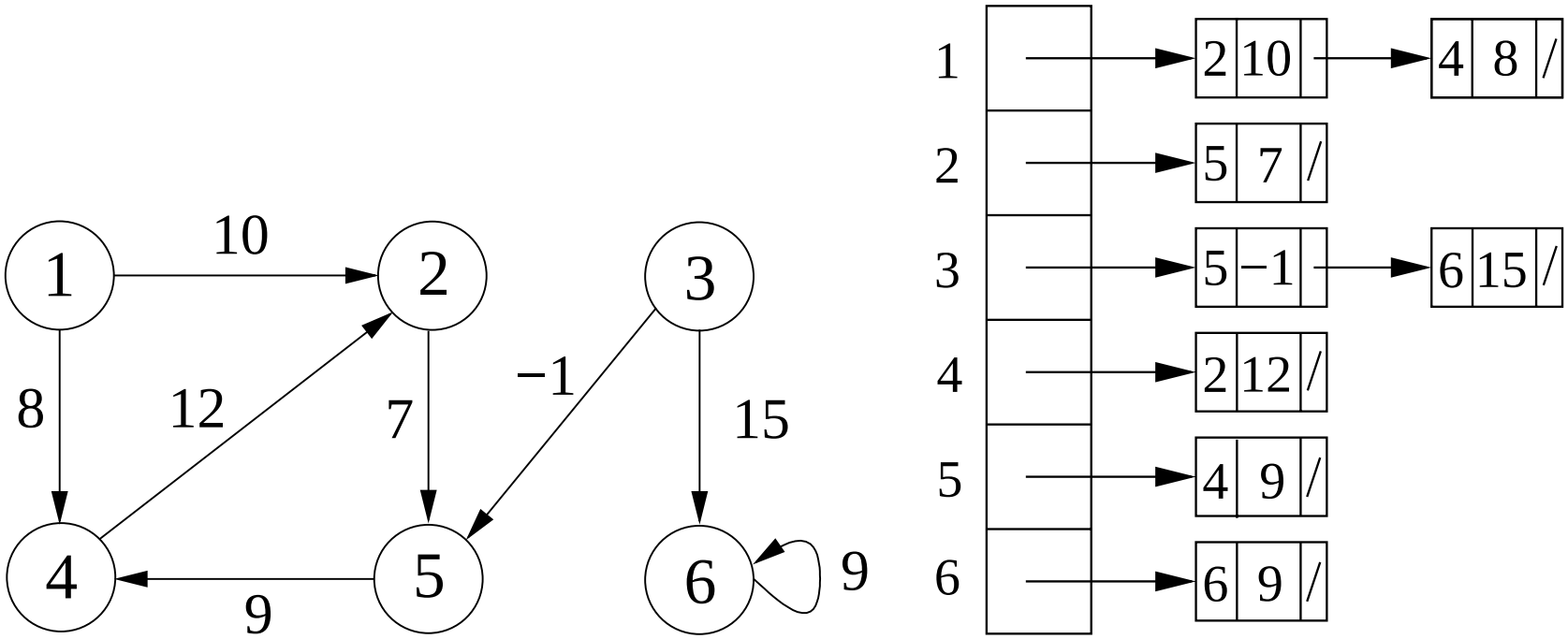


Listas de adyacencia (II)

- Si G es dirigido, la suma de las longitudes de las listas de adyacencia será $|E|$.
- Si G es no dirigido, la suma de las longitudes de las listas de adyacencia será $2|E|$.
- Coste espacial, sea dirigido o no: $O(|V| + |E|)$.
- Representación apropiada para grafos con $|E|$ menor que $|V|^2$.
- **Desventaja**: si se quiere comprobar si una arista (u,v) pertenece a $E \Rightarrow$ buscar v en la lista de adyacencia de u.
  Coste $O(Grado(G)) \subseteq O(|V|)$.

Listas de adyacencia (III)

- Representación extensible a grafos ponderados.
  El peso de (u,v) se almacena en el nodo de v de la lista de adyacencia de u.



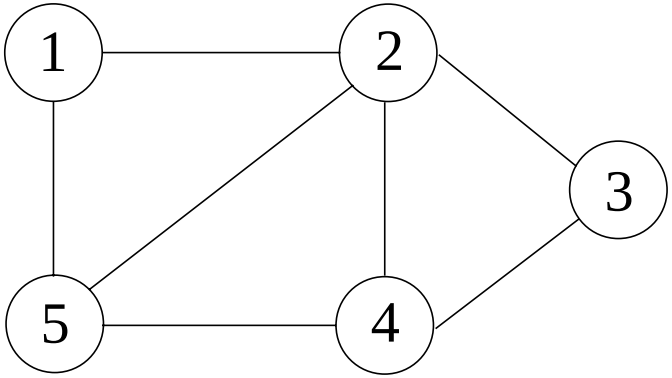
Listas de adyacencia (IV)

- Definición de tipos en C (grafos ponderados):

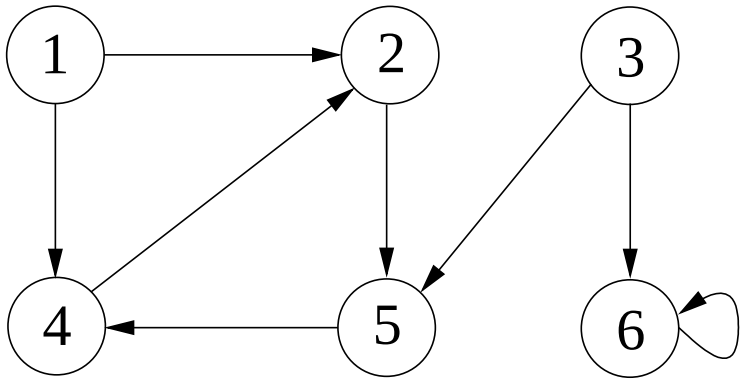
```
#define MAXVERT ...
typedef struct vertice{
  int nodo, peso;
  struct vertice *sig;
}vert_ady;
typedef struct{
  int talla;
  vert_ady *ady[MAXVERT];
}grafo;
```

Representación de grafos: Matriz de adyacencia

- $G = (V,E)$: matriz A de dimensión $|V| \times |V|$.
- Valor a_{ij} de la matriz: $a_{ij} = \begin{cases} 1 & \text{si } (i,j) \in E \\ 0 & \text{en cualquier otro caso} \end{cases}$



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



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

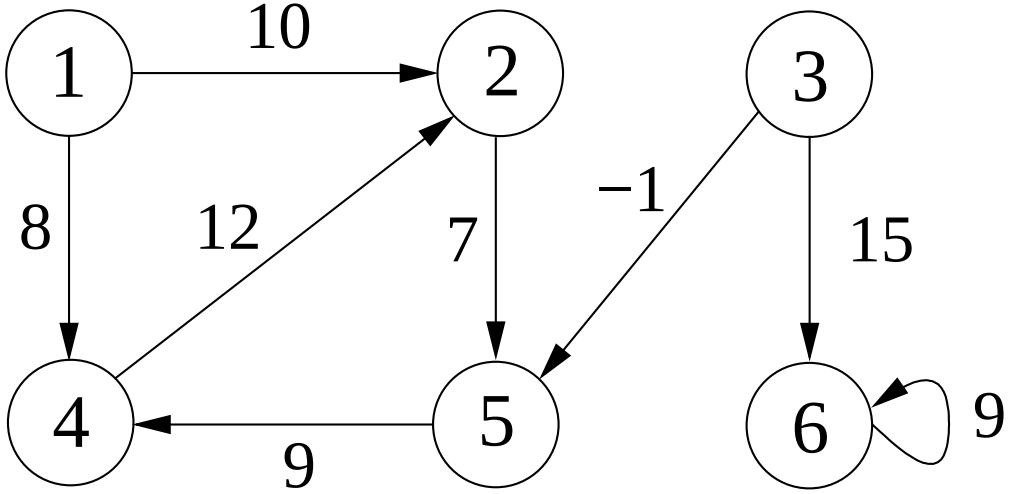
Matriz de adyacencia (II)

- Coste espacial: $O(|V|^2)$.
- Representación es útil para grafos con número de vértices pequeño, o grafos densos ($|E| \approx |V| \times |V|$).
- Comprobar si una arista (u,v) pertenece a $E \rightarrow$ consultar posición $A[u][v]$.
  Coste $O(1)$.

Matriz de adyacencia (III)

- Representación grafos ponderados:
  El peso de (i,j) se almacena en $A[i,j]$.

$$a_{ij} = \begin{cases} w(i,j) & \text{si } (i,j) \in E \\ 0 \ o \ \infty & \text{en cualquier otro caso} \end{cases}$$



	1	2	3	4	5	6
1	0	10	0	8	0	0
2	0	0	0	0	7	0
3	0	0	0	0	−1	15
4	0	12	0	0	0	0
5	0	0	0	9	0	0
6	0	0	0	0	0	9

Matriz de adyacencia (IV)

- Definición de tipos en C:

```
#define MAXVERT ...
typedef struct{
  int talla;
  int A[MAXVERT][MAXVERT];
}grafo;
```

Recorrido de grafos: recorrido primero en profundidad

$\rightarrow$ Generalización del orden previo (preorden) de un árbol.

Estrategia:

- Partir de un vértice determinado v.
- Cuando se visita un nuevo vértice, explorar cada camino que salga de él.
  Hasta que no se ha finalizado de explorar uno de los caminos no se comienza con el siguiente.
- Un camino deja de explorarse cuando lleva a un vértice ya visitado.
- Si existían vértices no alcanzables desde v el recorrido queda incompleto:
  seleccionar alguno como nuevo vértice de partida, y repetir el proceso.

Recorrido primero en profundidad (II)

Esquema recursivo: dado $G = (V, E)$

1. Marcar todos los vértices como *no visitados*.

2. Escoger vértice u como punto de partida.

3. Marcar u como visitado.

4. $\forall v$ adyacente a u, $(u,v) \in E$, si v no ha sido visitado, repetir recursivamente (3) y (4) para v.

   - Finalizar cuando se hayan visitado todos los nodos alcanzables desde u.
   - Si desde u no fueran alcanzables todos los nodos del grafo: volver a (2), escoger un nuevo vértice de partida v no visitado, y repetir el proceso hasta que se hayan recorrido todos los vértices.

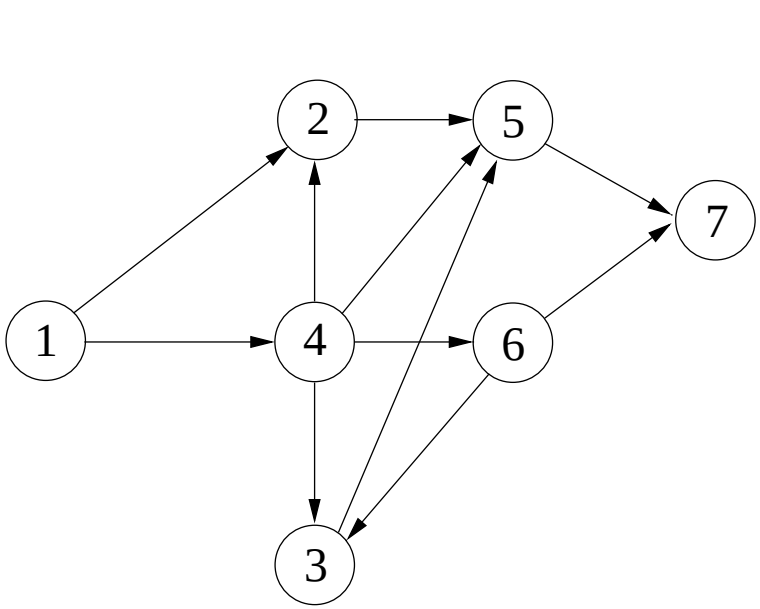
Recorrido primero en profundidad (III)

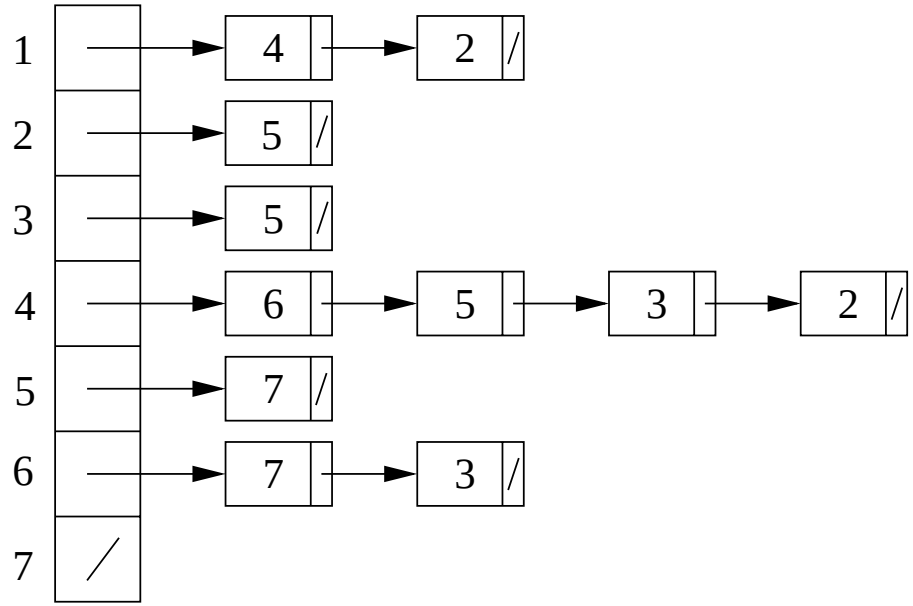
$\rightarrow$ usar un vector *color* (talla $|V|$) para indicar si u ha sido visitado (color[u]=AMARILLO) o no (color[u]=BLANCO):

Algoritmo Recorrido_en_profundidad(G){
 para cada vértice $u \in V$
  color[u] = BLANCO
 fin_para
 para cada vértice $u \in V$
  si (color[u] = BLANCO) Visita_nodo(u)
 fin_para
}

Algoritmo Visita_nodo(u){
 color[u] = AMARILLO
 para cada vértice $v \in V$ adyacente a u
  si (color[v] = BLANCO) Visita_nodo(v)
 fin_para
}

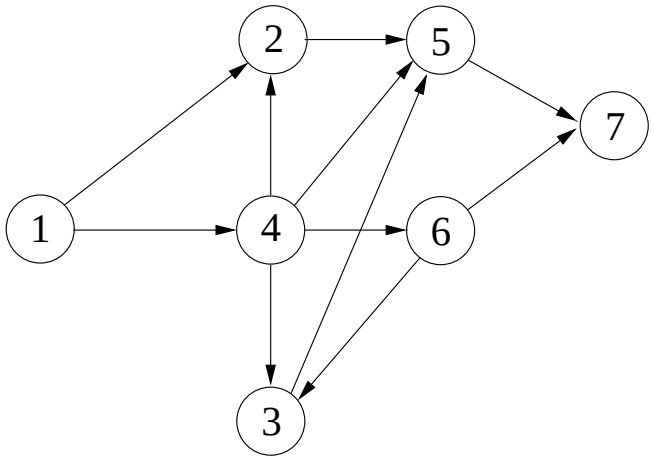
Recorrido primero en profundidad: Ejemplo



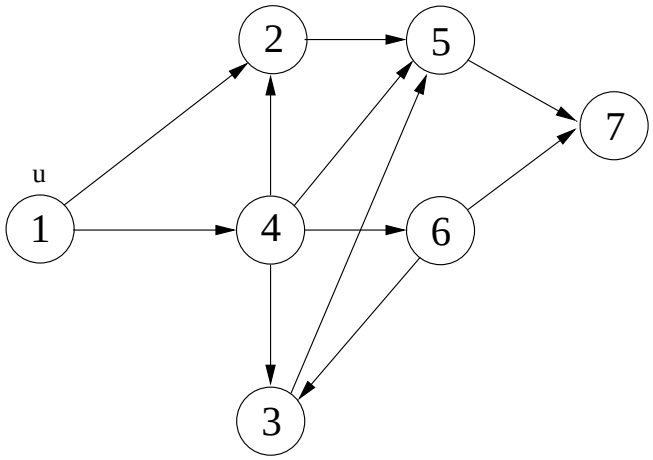


¡OJO!: el recorrido depende del orden en que aparecen los vértices en las listas de adyacencia.

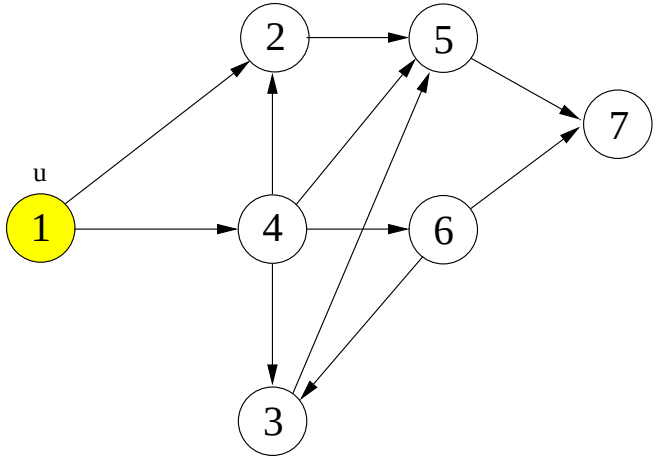
Recorrido primero en profundidad: Ejemplo (II)



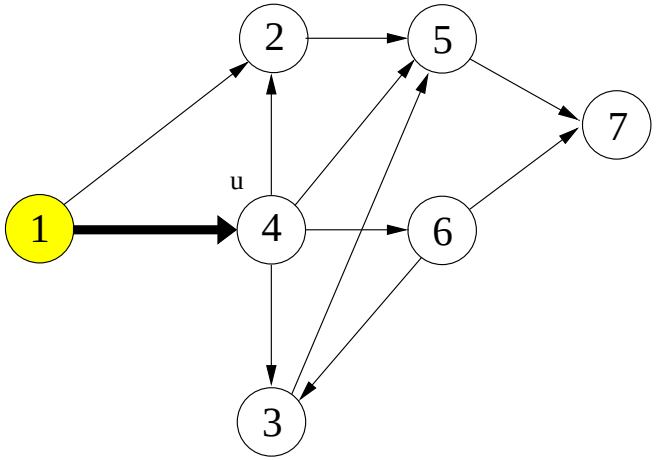
Rec_en_profund(G)



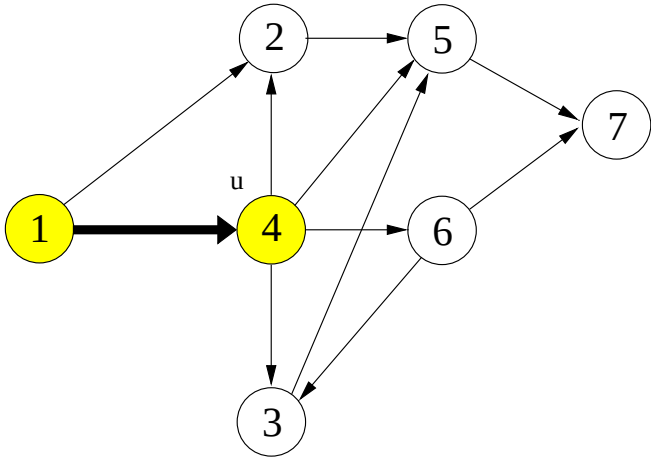
Visita_nodo(1)



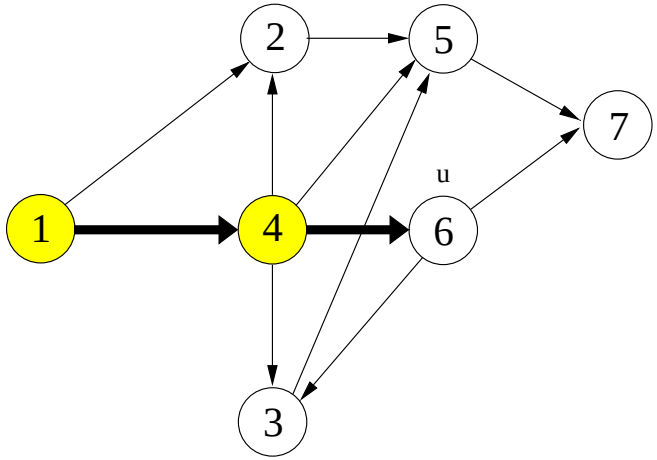
color[1]=AMARILLO



Visita_nodo(4)

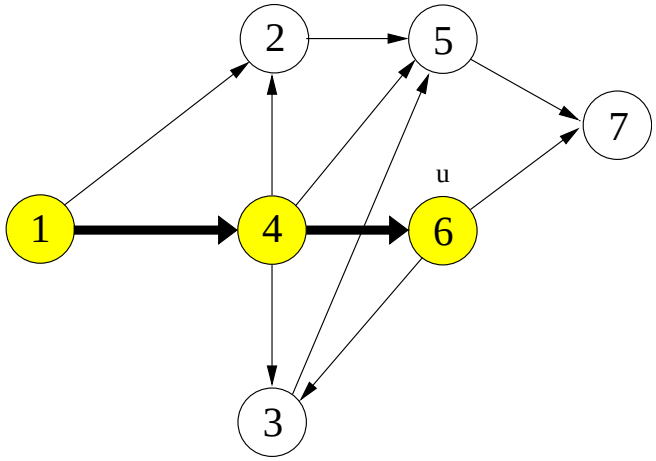


color[4]=AMARILLO

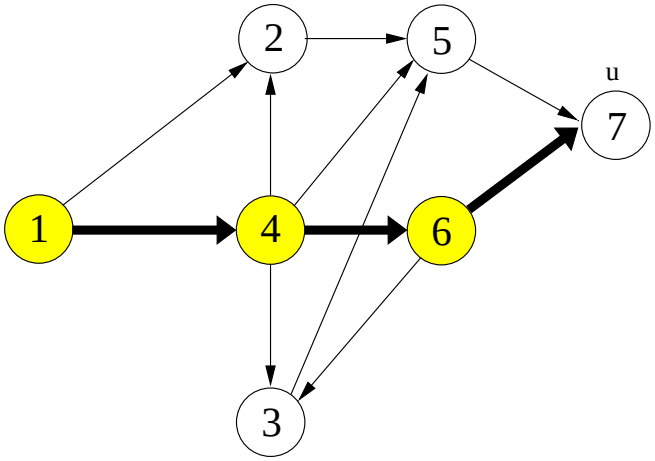


Visita_nodo(6)

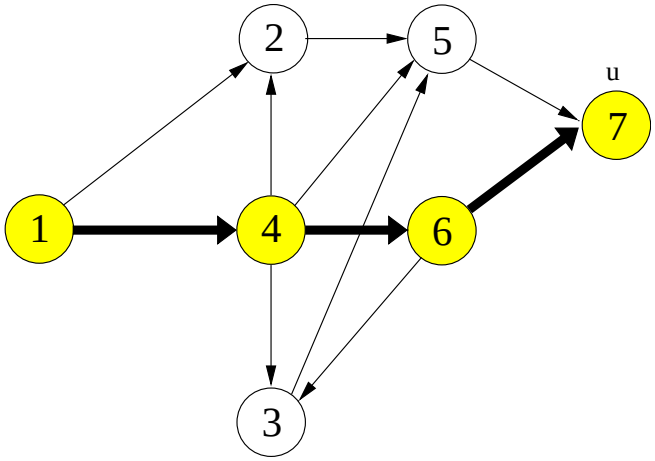
Recorrido primero en profundidad: Ejemplo (III)



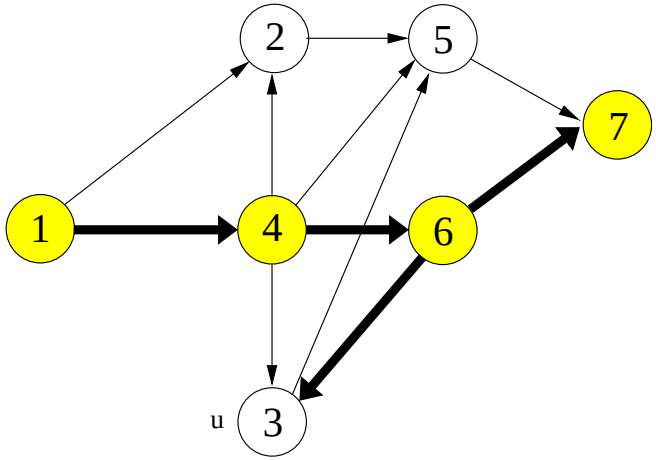
color[6]=AMARILLO



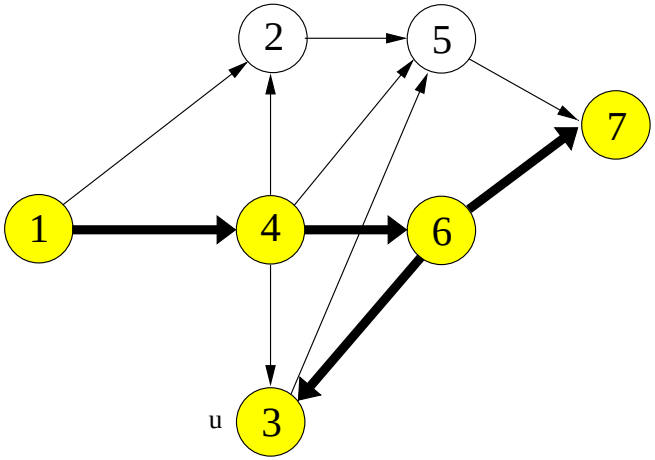
Visita_nodo(7)



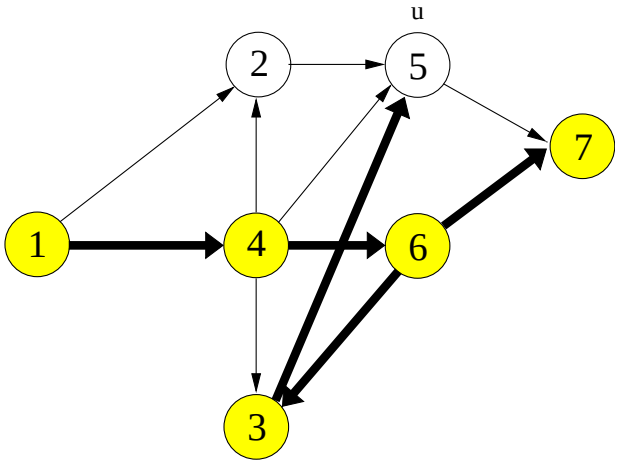
color[7]=AMARILLO



Visita_nodo(3)

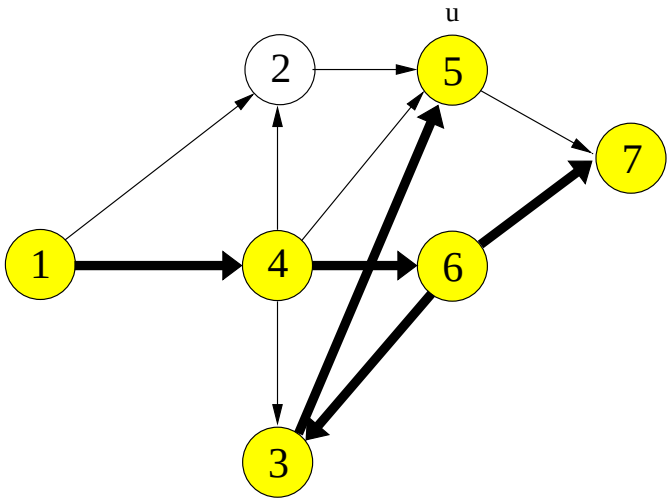


color[3]=AMARILLO

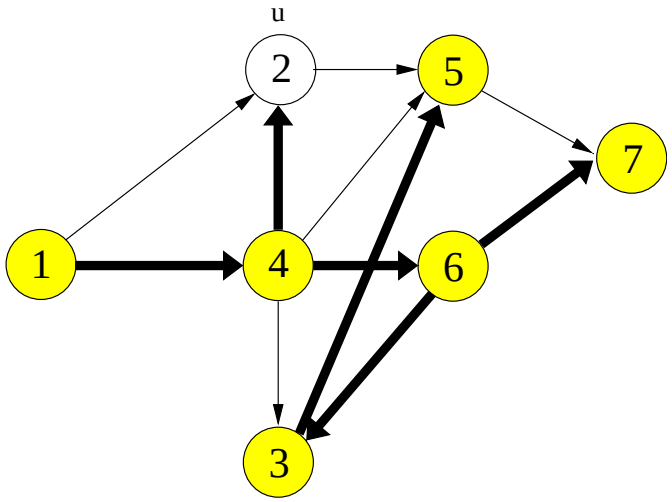


Visita_nodo(5)

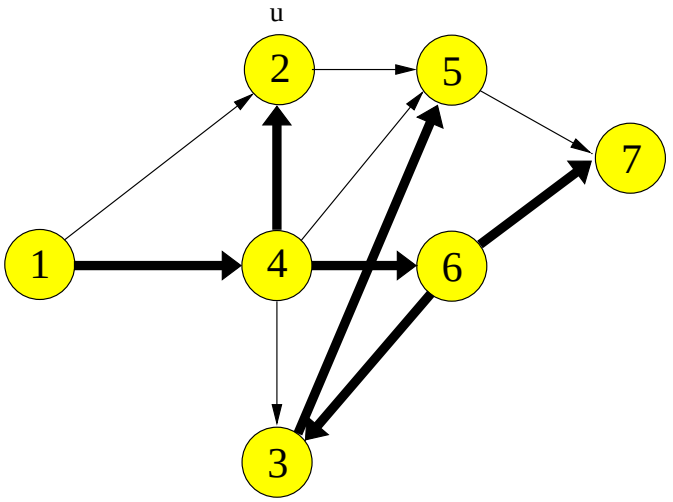
Recorrido primero en profundidad: Ejemplo (IV)



color[5]=AMARILLO



Visita_nodo(2)



color[2]=AMARILLO

Recorrido primero en profundidad: coste temporal

- $G = (V, E)$ se representa mediante listas de adyacencia.
- *Visita_nodo* se aplica únicamente sobre vértices no visitados $\rightarrow$ sólo una vez sobre cada vértice.
- *Visita_nodo* depende del número de vértices adyacentes que tenga u (longitud de la lista de adyacencia).
- coste de todas las llamadas a *Visita_nodo*:

$$\sum_{v \in V} |\text{ady}(v)| = \Theta(|E|)$$

- Añadir coste asociado a los bucles de *Recorrido_en_profundidad*: $O(|V|)$.

$\Rightarrow$ coste del recorrido en profundidad es $O(|V| + |E|)$.

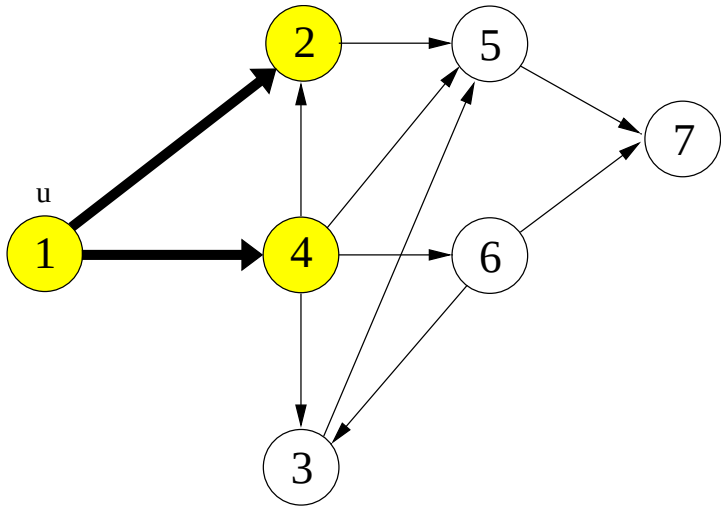
Recorrido de grafos: recorrido primero en anchura

$\longrightarrow$ Generalización del recorrido por niveles de un árbol.

Estrategia:

- Partir de algún vértice u, visitar u y, después, visitar cada uno de los vértices adyacentes a u.
- Repetir el proceso para cada nodo adyacente a u, siguiendo el orden en que fueron visitados.

Coste: $O(|V| + |E|)$.



Ordenación topológica

- Aplicación inmediata del recorrido en profundidad.
- $G = (V, E)$ dirigido y acíclico:
  la **ordenación topológica** es una permutación $v_1, v_2, v_3, \ldots, v_{|V|}$ de los vértices, tal que si $(v_i,v_j) \in E$, $v_i \neq v_j$, entonces v_i precede a v_j en la permutación.
- Ordenación no posible si G es cíclico.
- La ordenación topológica no es única.
- Una ordenación topológica es como una ordenación de los vértices a lo largo de una línea horizontal, con los arcos de izquierda a derecha.
- Algoritmo como el recorrido primero en profundidad. Utiliza una pila P en la que se almacena el orden topológico de los vértices.

Ordenación topológica (II)

Algoritmo Ordenación_topológica(G){

para cada vértice $u \in V$
color[u] = BLANCO
fin_para

$P = \emptyset$

para cada vértice $u \in V$
si (color[u] = BLANCO) Visita_nodo(u)
fin_para

devolver(P)

}

Algoritmo Visita_nodo(u){

color[u] = AMARILLO

para cada vértice $v \in V$ adyacente a u
si (color[v] = BLANCO) Visita_nodo(v)
fin_para

apilar(P,u)

}

Ordenación topológica (III)

Finaliza Visita_nodo(u)

↓

Se han visitado los vértices alcanzables desde u
(los vértices que suceden a u en el orden topológico).

↓

Antes de acabar *Visita_nodo(u)* apilamos u

↓

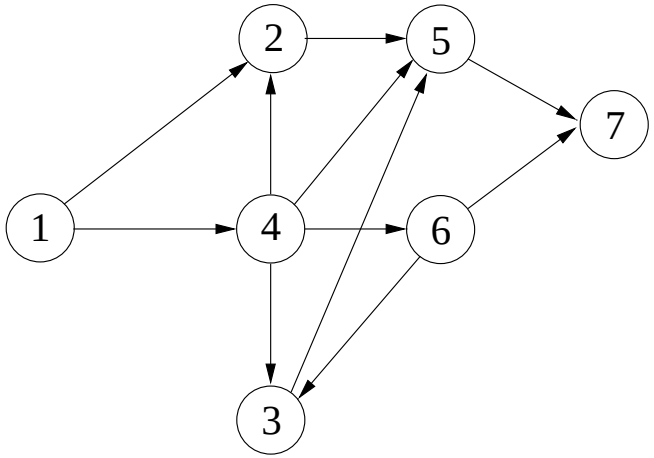
El vértice se apila después de apilar todos los alcanzables desde él.

↓

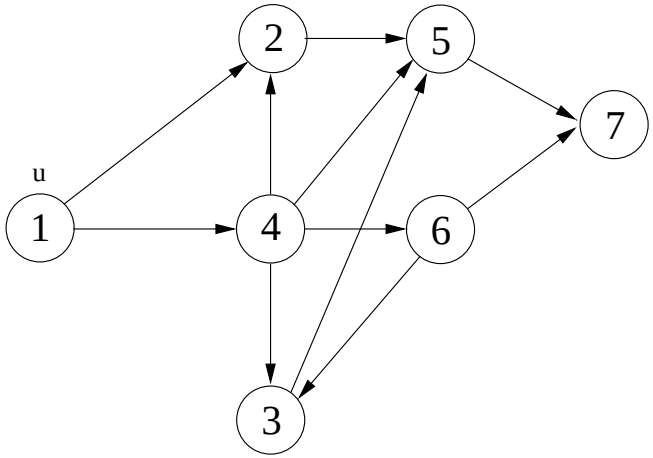
Al final en P estarán los vértices ordenados topológicamente.

Coste: equivalente a recorrido primero en profundidad, $O(|V| + |E|)$.

Ordenación topológica: Ejemplo

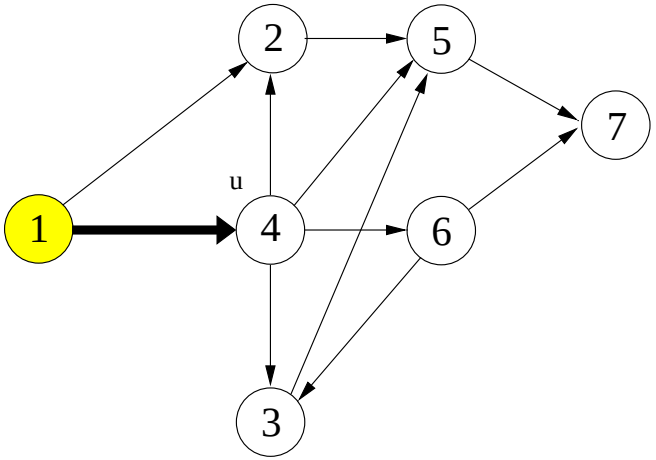


Orden_topologico(G)



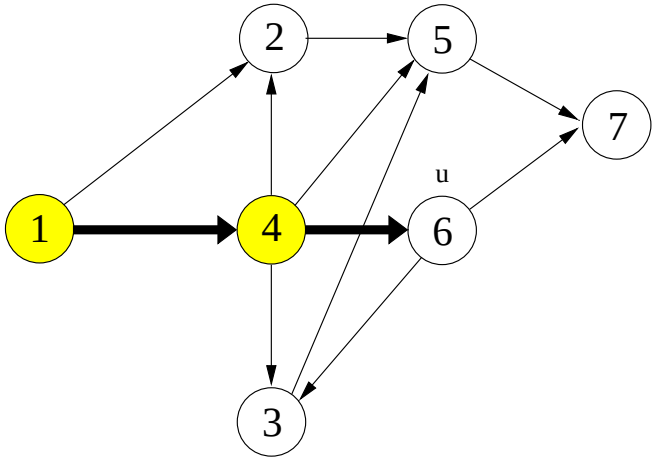
Visita_nodo(1)

$P = \{\}$



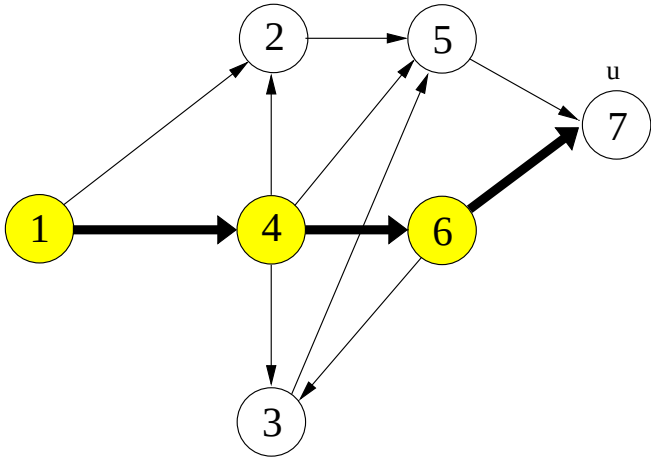
Visita_nodo(4)

$P = \{\}$



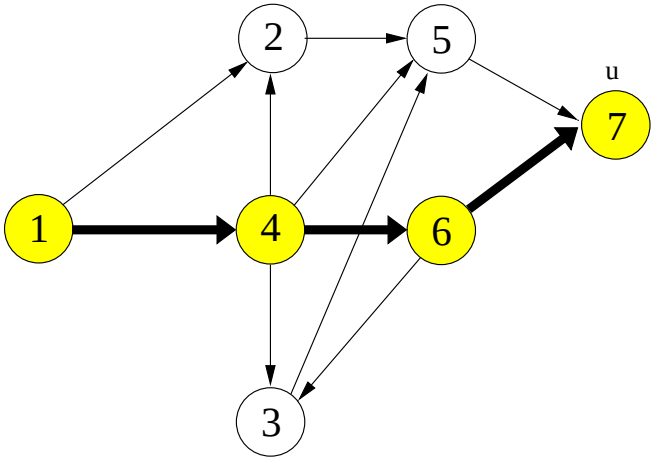
Visita_nodo(6)

$P = \{\}$



Visita_nodo(7)

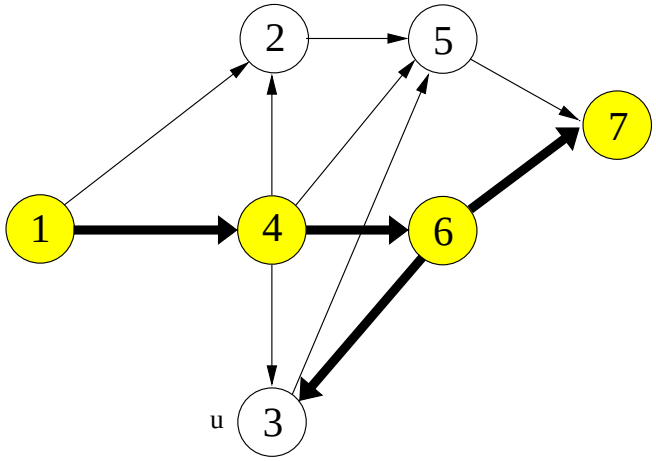
$P = \{\}$



apilar(P,7)

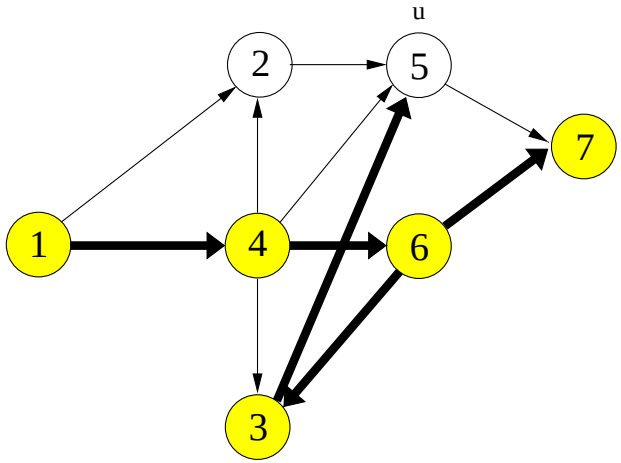
$P = \{7\}$

Ordenación topológica: Ejemplo (II)



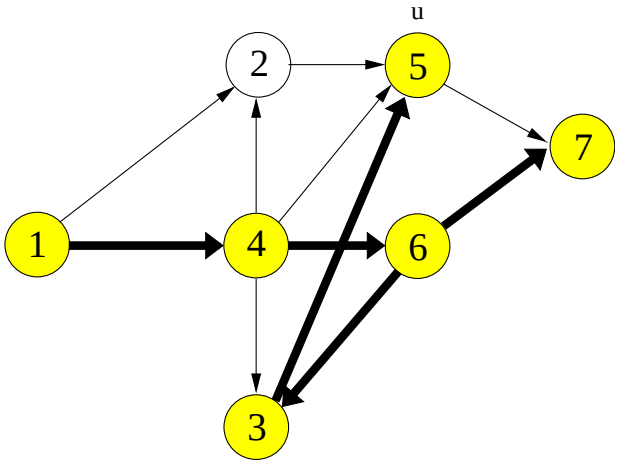
Visita_nodo(3)

$P = \{7\}$



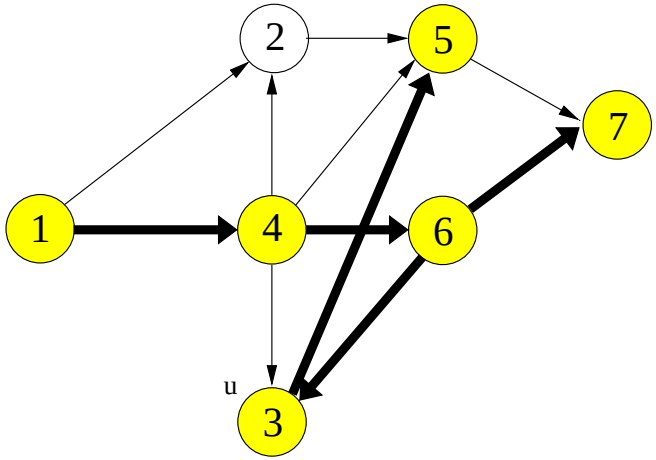
Visita_nodo(5)

$P = \{7\}$



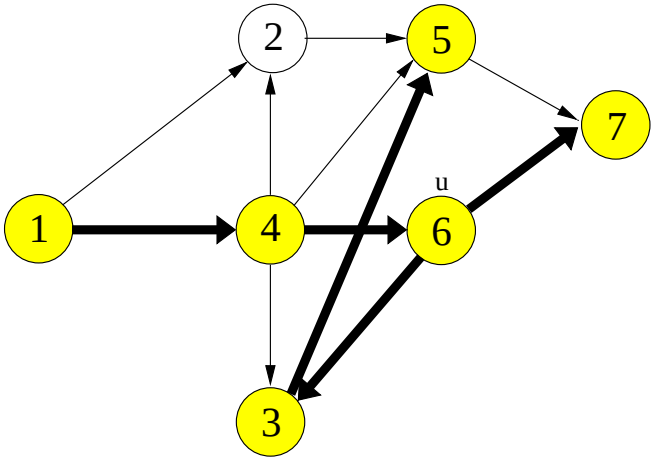
apilar(P,5)

$P = \{5,7\}$



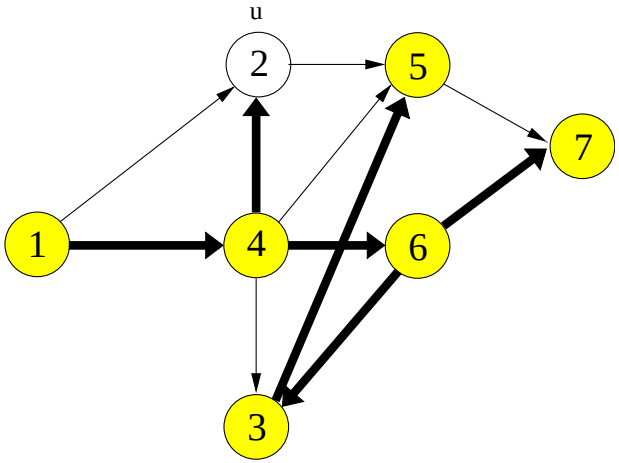
apilar(P,3)

$P = \{3,5,7\}$



apilar(P,6)

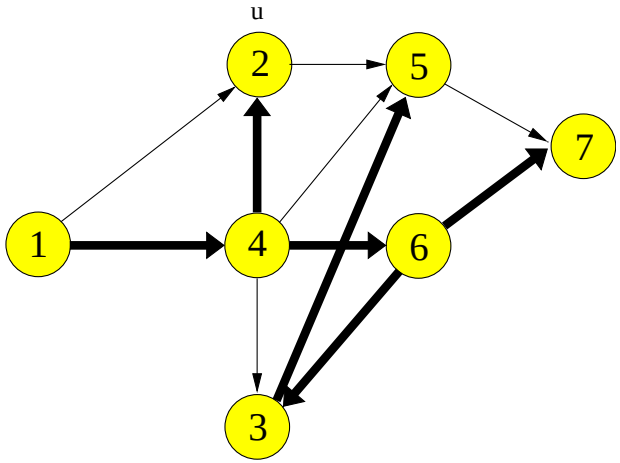
$P = \{6,3,5,7\}$



Visita_nodo(2)

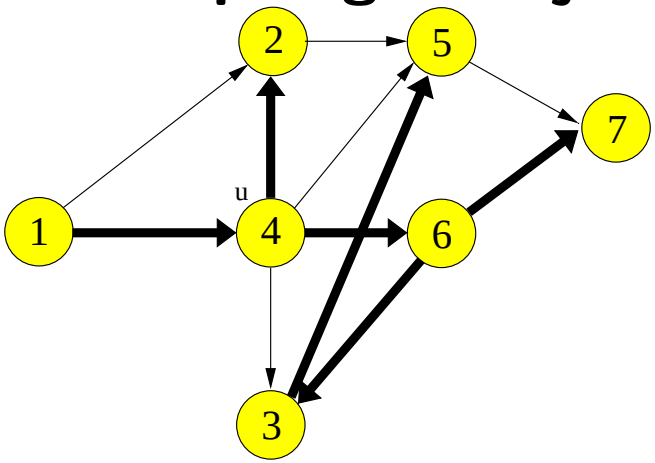
$P = \{6,3,5,7\}$

Ordenación topológica: Ejemplo (III)



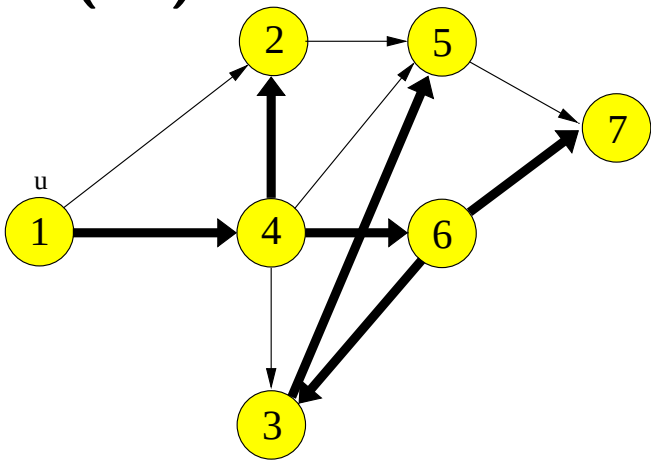
apilar(P,2)

$P = \{2,6,3,5,7\}$



apilar(P,4)

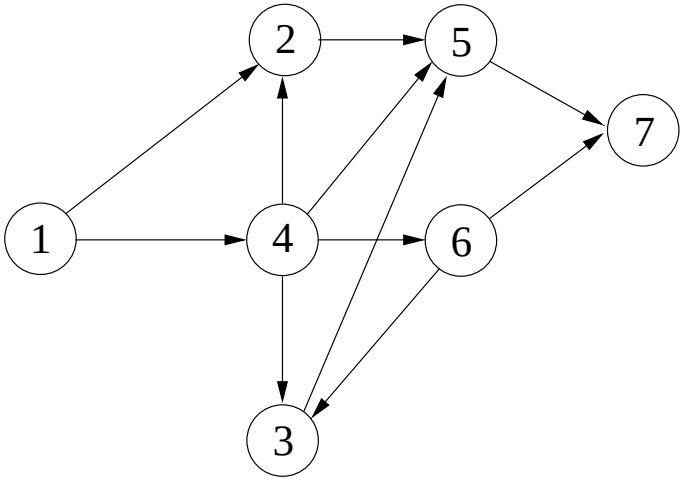
$P = \{4,2,6,3,5,7\}$

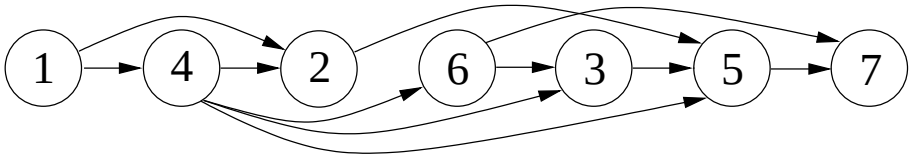


apilar(P,1)

$P = \{1,4,2,6,3,5,7\}$

Ordenación de los vértices en una línea horizontal, con todos los arcos de izquierda a derecha:





Caminos de mínimo peso

$G = (V, E)$ dirigido y ponderado con el peso de cada arista $w(u,v)$.

- **Peso de un camino** $p =< v_0, v_1, \ldots, v_k >$: la suma de los pesos de las aristas que lo forman:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

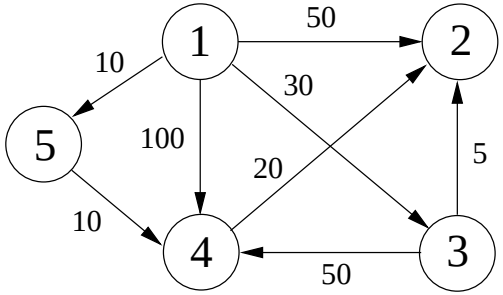
- **Camino de mínimo peso** desde u a v: camino que tiene un peso menor entre todos los caminos de u a v, o ∞ si no existe camino de u a v.

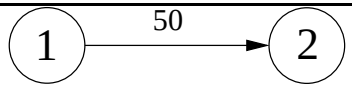
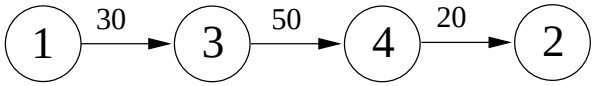
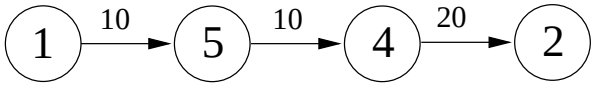
- **Longitud** de un camino de u a v: peso de un camino de u a v.

- **Camino más corto** de u a v: camino de mínimo peso de u a v.

Caminos de mínimo peso (II)

Caminos desde 1 a 2:



Camino	Longitud (peso o coste)
	50
	35
	100
	120
	40

Caminos de mínimo peso (III)

- Usamos un grafo dirigido y ponderado para representar las comunicaciones entre ciudades:
  - vértice $=$ ciudad.
  - arista (u,v) $=$ carretera de u a v;
    el peso asociado a la arista es la distancia.

- Camino más corto $=$ ruta más rápida.

- Variantes del cálculo de caminos de menor longitud:
  - Obtención de los caminos más cortos desde un vértice origen a todos los demás.
  - Obtención de los caminos más cortos desde todos los vértices a uno destino.
  - Obtención del camino más corto de un vértice u a un vértice v.
  - Obtención de los caminos más cortos entre todos los pares de vértices.

Algoritmo de Dijkstra

Problema: $G = (V, E)$ grafo dirigido y ponderado con pesos no negativos; dado un vértice origen s, obtener los caminos más cortos al resto de vértices de V.

- Si existen aristas con pesos negativos, la solución podría ser errónea.
- Otros algoritmos permiten pesos negativos, siempre que no existan ciclos de peso negativo.
- Idea: explotar la propiedad de que el camino más corto entre dos vértices contiene caminos más cortos entre los vértices que forman el camino.

Algoritmo de Dijkstra (II)

El algoritmo de Dijkstra mantiene estos conjuntos:

- Un conjunto de vértices S que contiene los vértices para los que la distancia más corta desde el origen ya es conocida. Inicialmente $S = \emptyset$.

- Un conjunto de vértices $Q = V - S$ que mantiene, para cada vértice, la distancia más corta desde el origen pasando a través de vértices que pertenecen a S (Distancia provisional). Para guardar las distancias provisionales usaremos un vector $D[1..|V|]$, donde $D[i]$ indicará la distancia provisional desde el origen s al vértice i.
  Inicialmente, $D[u] = \infty \; \forall u \in V - \{s\}$ y $D[s] = 0$.

Además:

- Un vector $P[1..|V|]$ para recuperar los caminos mínimos calculados.
  $P[i]$ almacena el índice del vértice que precede al vértice i en el camino más corto desde s hasta i.

Algoritmo de Dijkstra (III)

Estrategia:

1. Extraer de Q el vértice u cuya distancia provisional $D[u]$ sea menor.
   $\rightarrow$ esta distancia es la menor posible entre el vértice origen s y u.
   La distancia provisional se correspondía con el camino más corto utilizando vértices de S.
   $\Rightarrow$ ya no es posible encontrar un camino más corto desde s hasta u utilizando algún otro vértice del grafo

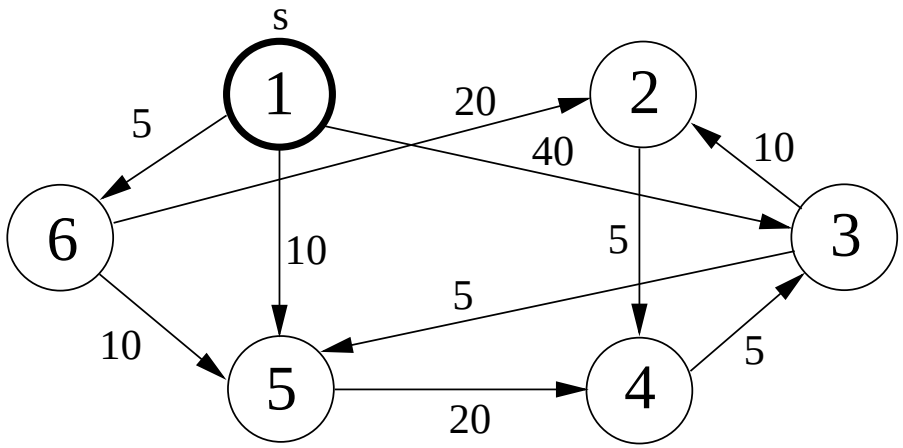
2. Insertar u, para el que se ha calculado el camino más corto desde s, en S ($S = S \cup \{u\}$).
   Actualizar las distancias provisionales de los vértices de Q adyacentes a u que mejoren usando el nuevo camino.

3. Repetir 1 y 2 hasta que Q quede vacío
   $\Rightarrow$ en D se tendrá, para cada vértice, la distancia más corta desde el origen.

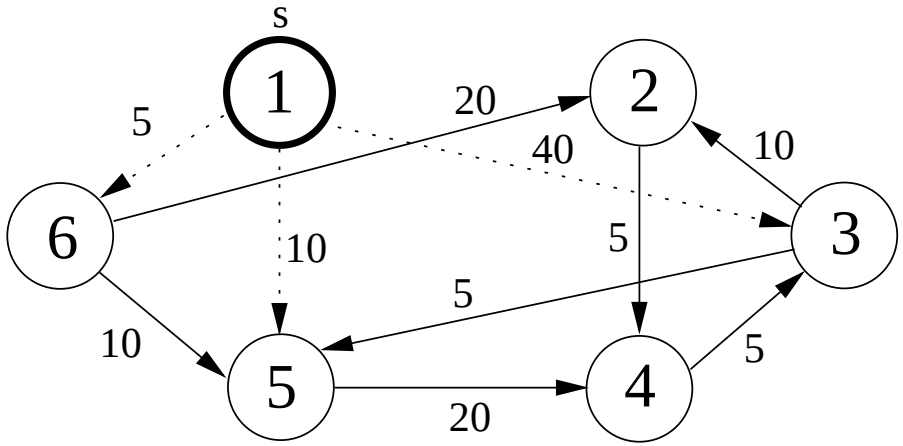
Algoritmo Dijkstra(G, w, s)
{
 para cada vértice $v \in V$ **hacer**
  $D[v] = \infty$;
  $P[v] = NULO$;
 fin_para
 $D[s] = 0$; $S = \emptyset$; $Q = V$;
 mientras $Q \neq \emptyset$ **hacer**
  $u =$ extract_min(Q); /* según D */
  $S = S \cup \{u\}$;
  para cada vértice $v \in V$ adyacente a u **hacer**
   si $D[v] > D[u] + w(u,v)$ **entonces**
    $D[v] = D[u] + w(u,v)$;
    $P[v] = u$;
   fin_si
  fin_para
 fin_mientras
}

Algoritmo de Dijkstra: Ejemplo

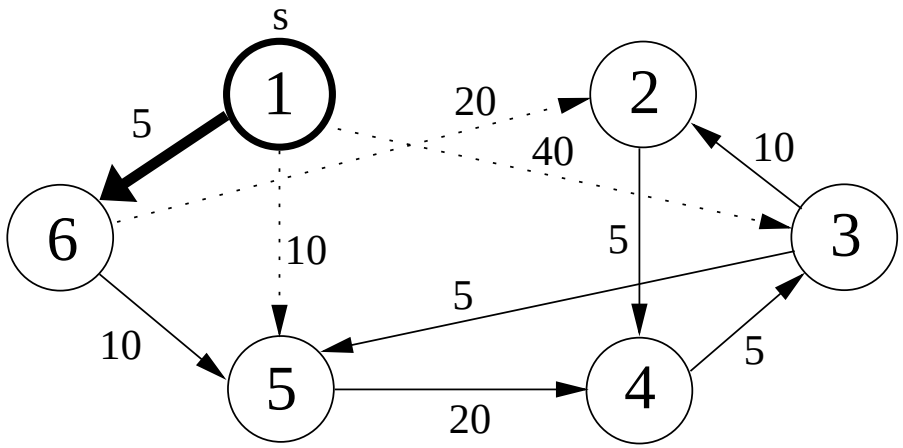
- Vértice origen 1.
- Aristas con trazo discontínuo = caminos provisionales desde el origen a los vértices.
- Aristas con trazo grueso = caminos mínimos ya calculados.
- Resto de aristas con trazo fino.



S	Q	u		1	2	3	4	5	6
$\{\}$	$\{1,2,3,4,5,6\}$	$-$	D	0	∞	∞	∞	∞	∞
			P	$NULO$	$NULO$	$NULO$	$NULO$	$NULO$	$NULO$

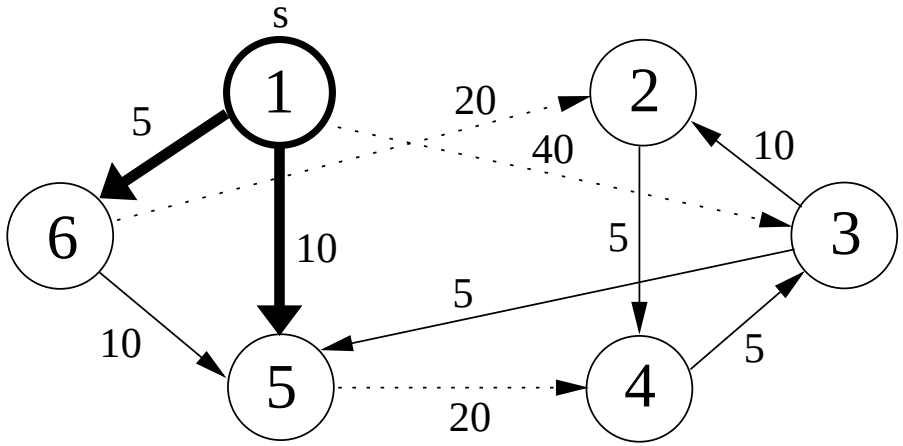


S	Q	u		1	2	3	4	5	6
$\{1\}$	$\{2,3,4,5,6\}$	1	D	0	∞	40	∞	10	5
			P	$NULO$	$NULO$	1	$NULO$	1	1



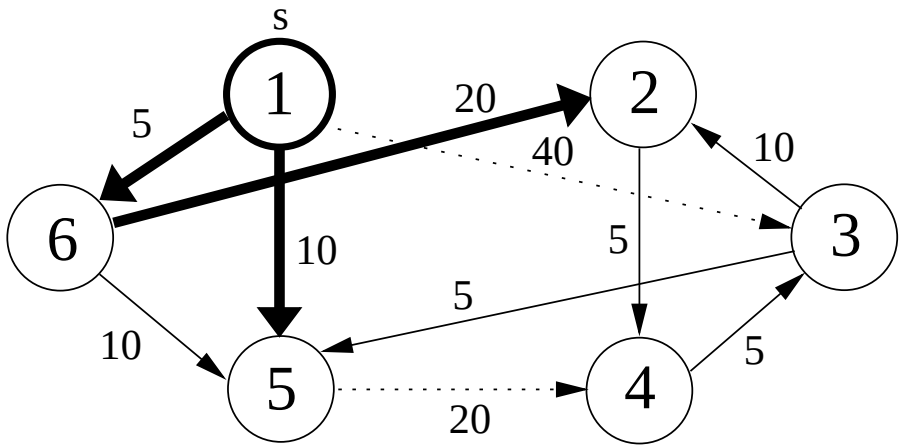
S	Q	u
{1,6}	{2,3,4,5}	6

	1	2	3	4	5	6
D	0	25	40	∞	10	5
P	$NULO$	6	1	$NULO$	1	1



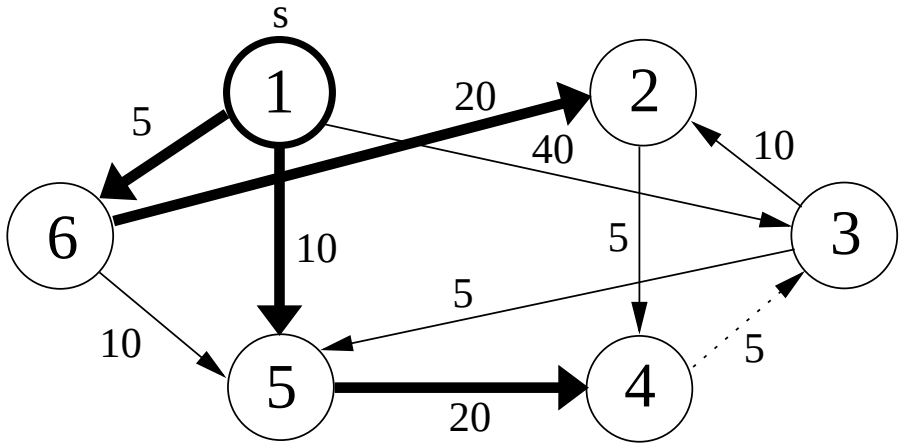
S	Q	u
{1,6,5}	{2,3,4}	5

	1	2	3	4	5	6
D	0	25	40	30	10	5
P	$NULO$	6	1	5	1	1



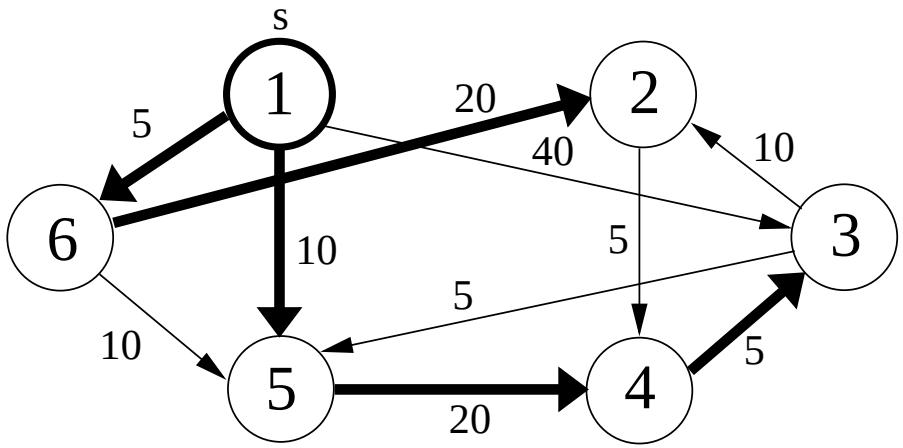
S	Q	u
{1,6,5,2}	{3,4}	2

	1	2	3	4	5	6
D	0	25	40	30	10	5
P	$NULO$	6	1	5	1	1



S	Q	u
{1,6,5,2,4}	{3}	4

	1	2	3	4	5	6
D	0	25	35	30	10	5
P	$NULO$	6	4	5	1	1



S	Q	u
{1,6,5,2,4,3}	{}	3

	1	2	3	4	5	6
D	0	25	35	30	10	5
P	$NULO$	6	4	5	1	1

Coste temporal del algoritmo

- Consideración: representación mediante listas de adyacencia.

- El bucle *mientras* se repite hasta que el conjunto Q queda vacío.
  Inicialmente Q tiene todos los vértices y en cada iteración se extrae uno de Q
  $\rightarrow |V|$ iteraciones.

- El bucle *para* interno al bucle *mientras* se repite tanto como vértices adyacentes tenga el vértice extraído de Q.
  El número total de iteraciones del bucle *para* será el número de vértices adyacentes de los vértices del grafo.
  $\rightarrow$ El número de arcos del grafo: $|E|$.

- La operación *extract_min* $\in O(|V|)$ (conjunto Q = vector).
  extract_min se realiza $|V|$ veces.
  Coste total de *extract_min* $\in O(|V|^2)$.

Sumando costes del bucle *para* y *extract_min*
$\Rightarrow$ Coste Dijkstra $\in O(|V|^2 + |E|) = O(|V|^2)$

Coste temporal del algoritmo (II)

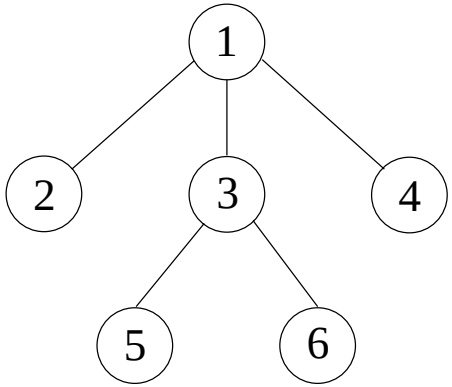
Optimización: *extract_min* sería más eficiente si Q fuese un montículo:

- Construir el montículo $\in O(|V|)$.
- La operación *extract_min* $\in O(\log |V|)$
  extract_min se realiza $|V|$ veces.
  Coste total de *extract_min* $\in O(|V| \log |V|)$.
- El bucle *para* supone modificar la prioridad (distancia provisional) del vértice en el montículo y reorganizar el montículo $\in O(\log |V|)$.
  $\rightarrow$ cada iteración del bucle *para* $\in O(\log |V|)$.
  El bucle *para* se realiza $|E|$ veces $\rightarrow O(|E| \log |V|)$.

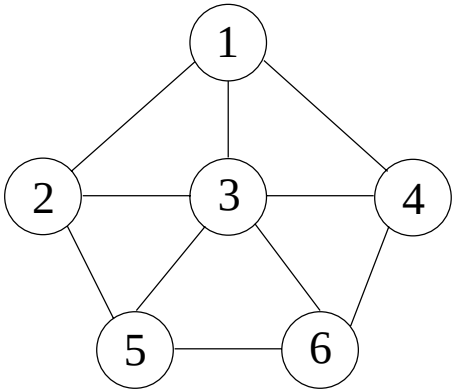
$\Rightarrow$ Coste Dijkstra $O(|V| + |V| \log |V| + |E| \log |V|) = O((|V| + |E|) \log |V|)$.

Árbol de expansión de coste mínimo

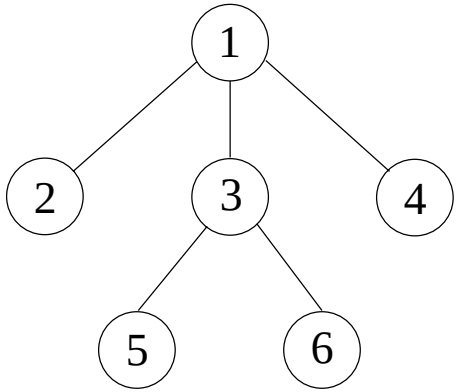
- **Árbol libre**: grafo no dirigido, acíclico y conexo. Propiedades:
  1. Un árbol libre con $n \geq 1$ nodos tiene $n - 1$ arcos.
  2. Si se añade una nueva arista, se crea un ciclo.
  3. Cualquier par de de vértices están conectados por un único camino.



- **Árbol de expansión** de $G = (V, E)$: árbol libre $T = (V', E')$, tal que $V' = V$ y $E' \subseteq E$.



$\longrightarrow$ Árbol de expansión $\longrightarrow$

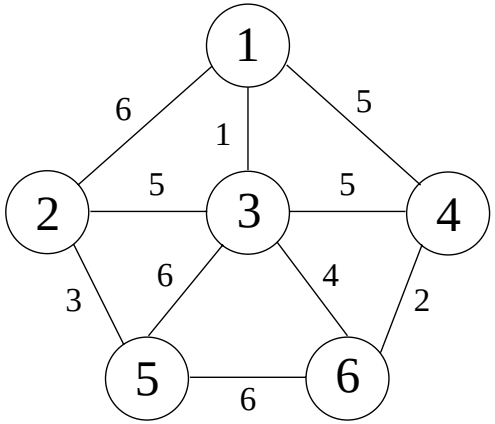


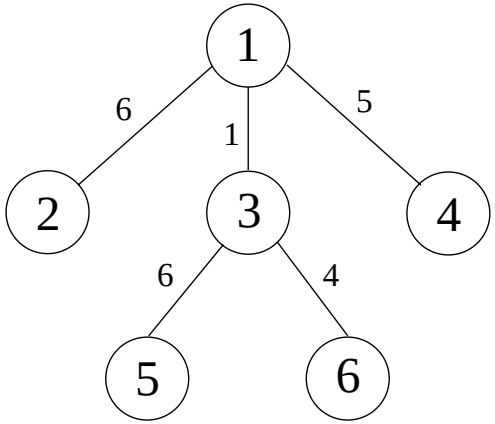
Árbol de expansión de coste mínimo (II)

- **Coste de un árbol de expansión** T: la suma de los costes de todos los arcos del árbol.

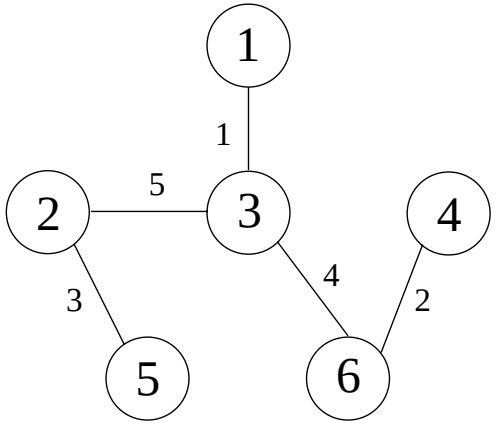
$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

- Un **árbol de expansión será de coste mínimo**, si se cumple que su coste es el menor posible respecto al coste de cada uno de los posibles árboles de expansión del grafo.





Coste 22



Coste mínimo: 15

Problema: Sea $G = (V, E)$ un grafo no dirigido, conexo y ponderado; obtener $G' = (V, E')$ donde $E' \subseteq E$ tal que G' sea un árbol de expansión de coste mínimo de G.

Algoritmo de Kruskal

El algoritmo mantiene estas estructuras:

- Un conjunto A que mantiene las aristas que ya han sido seleccionadas como pertenecientes al árbol de expansión de coste mínimo. Al final A contendrá el subconjunto de aristas que forman el árbol de expansión decoste mínimo.

- Un MF-set que se usa para saber qué vértices están unidos entre sí en el bosque (conjunto de árboles) que se va obteniendo durante el proceso de construcción del árbol de expansión de coste mínimo:
  - Conforme se añaden aristas al conjunto A, se unen vértices entre sí formando árboles.
  - Estos árboles se enlazarán entre sí hasta obtener un único árbol que será el árbol de expansión de coste mínimo.

Algoritmo de Kruskal (II)

Estrategia:

1. Inicialmente:
   - $A = \emptyset$
   - el MF-set tendrá $|V|$ subconjuntos cada uno de los cuales contendrá uno de los vértices.

2. De entre todas las aristas (u,v), seleccionar como candidata para el AECM aquélla no seleccionada todavía y con menor peso:
   - Si u y v no están conectados entre sí en el AECM (provocarán incremento de coste mínimo) $\rightarrow$ unir los vértices conectados con v con los vértices conectados con u y añadir (u,v) a A.
   - Si u y v ya están conectados entre sí, no se realiza ninguna acción.

3. Repetir (2) una vez para cada arista (u,v).

Algoritmo de Kruskal (III)

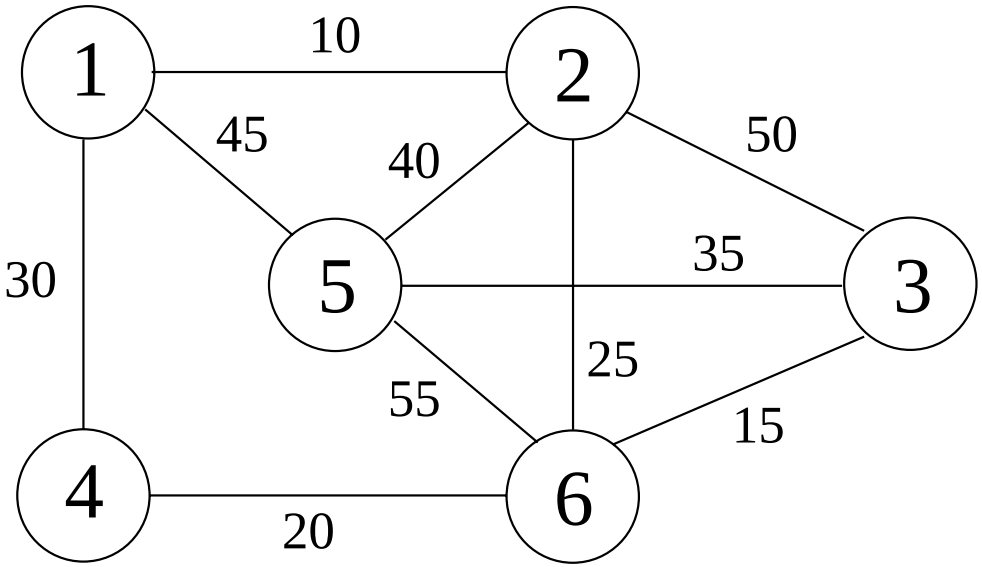
Dado $G = (V, E)$ no dirigido, conexo y ponderado:

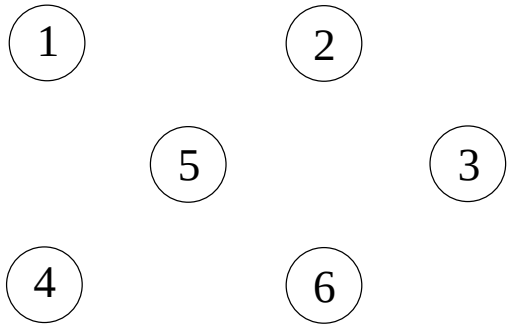
Algoritmo Kruskal(G)
{
 $A = \emptyset$
 para cada vértice $v \in V$ **hacer**
  Crear_Subconjunto(v)
 fin_para
 ordenar las aristas pertenecientes a E, según su peso, en orden no decreciente
 para cada arista $(u,v) \in E$, siguiendo el orden no decreciente **hacer**
  si Buscar(u) $\neq$ Buscar(v) **entonces**
   $A = A \cup \{(u,v)\}$
   Union(Buscar(u),Buscar(v))
  fin_si
 fin_para
 devuelve(A)
}

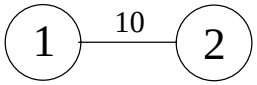
Algoritmo de Kruskal: Ejemplo

Para cada iteración se muestran:

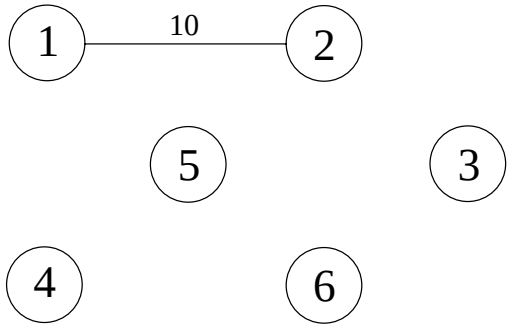
- El estado del conjunto A y del MF-set.
- El bosque que se va obteniendo durante la construcción del AECM.
- Qué arista ha sido seleccionada para formar parte del árbol y las acciones llevadas a cabo.



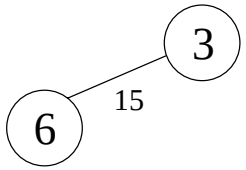
A	MF-set (componentes conexas)	Árbol de coste mínimo
$\{\}$	$\{\{1\},\{2\},\{3\},\{4\},\{5\},\{6\}\}$	



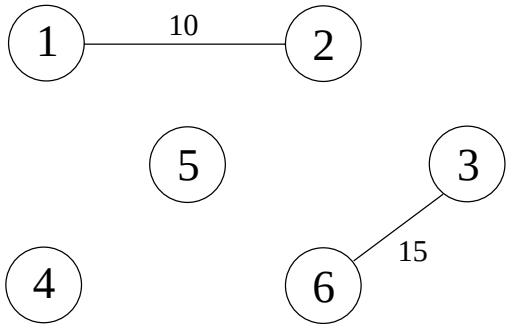
Buscar(1) $\neq$ Buscar(2) $\longrightarrow$ $A = A \cup \{(1,2)\}$; Union(Buscar(1),Buscar(2))

$\{(1,2)\}$	$\{\{1,2\},\{3\},\{4\},\{5\},\{6\}\}$	

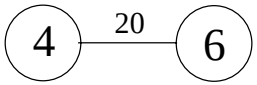
A	MF-set (componentes conexas)	Árbol de coste mínimo



Buscar$(6) \neq$ Buscar$(3) \longrightarrow A = A \cup \{(6,3)\}$; Union(Buscar$(6)$,Buscar$(3)$)

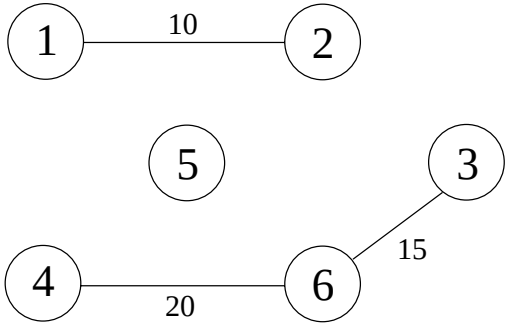
$\{(1,2),(6,3)\}$	$\{\{1,2\},\{6,3\},\{4\},\{5\}\}$	

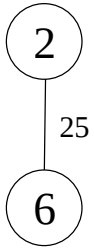
A	MF-set (componentes conexas)	Árbol de coste mínimo



Buscar$(4) \neq$ Buscar$(6) \longrightarrow A = A \cup \{(4,6)\}$; Union(Buscar$(4)$,Buscar$(6)$)

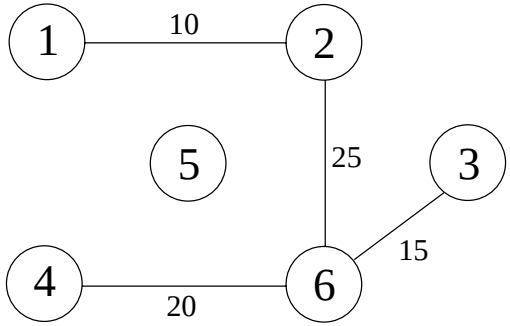
$\{(1,2),(6,3),(4,6)\}$ $\{\{1,2\},\{4,6,3\},\{5\}\}$



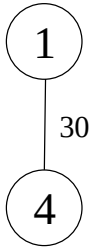


Buscar$(2) \neq$ Buscar$(6) \longrightarrow A = A \cup \{(2,6)\}$; Union(Buscar$(2)$,Buscar$(6)$)

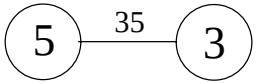
$\{(1,2),(6,3),(4,6),(2,6)\}$ $\{\{1,2,4,6,3\},\{5\}\}$



A	MF-set (componentes conexas)	Árbol de coste mínimo



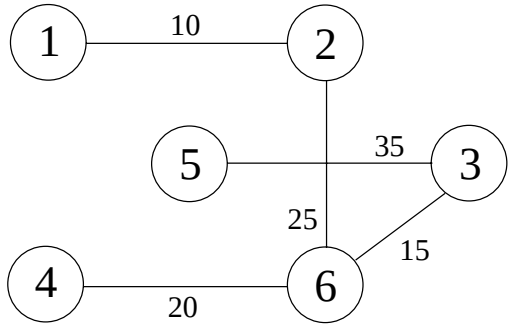
Buscar(1) = Buscar(4)



Buscar(5) $\neq$ Buscar(3) $\longrightarrow$ $A = A \cup \{(5,3)\}$; Union(Buscar(5),Buscar(3))

$\{(1,2),(6,3),(4,6),(2,6),(5,3)\}$

$\{\{5,1,2,4,6,3\}\}$



Coste temporal del algoritmo

Asunción: Las operaciones *Unión* y *Buscar* aplican *unión por rango* y *compresión de caminos*.

- Construcción de $|V|$ subconjuntos disjuntos $\in O(|V|)$.
- *Unión* y *Buscar* para cada arista del grafo $\in O(|E|)$.
- Selección de aristas en orden no decreciente.
  Organizar aristas mediante un montículo:
  - Extraer arista de mínimo peso $\in O(\log |E|)$.
  - $|E|$ operaciones de extracción $\rightarrow O(|E| \log |E|)$.
  - Construcción del montículo $\in O(|E|)$.

$\Rightarrow$ Coste Kruskal $\in O(|V| + |E| + |E| \log |E| + |E|) = O(|E| \log |E|)$.