

Algoritmos y Estructura de Datos III – Sección TQ – 1er semestre

Grafos

Representación y algoritmos resaltantes

Prof. Cristian Cappelletti
abril/2018

Contenido

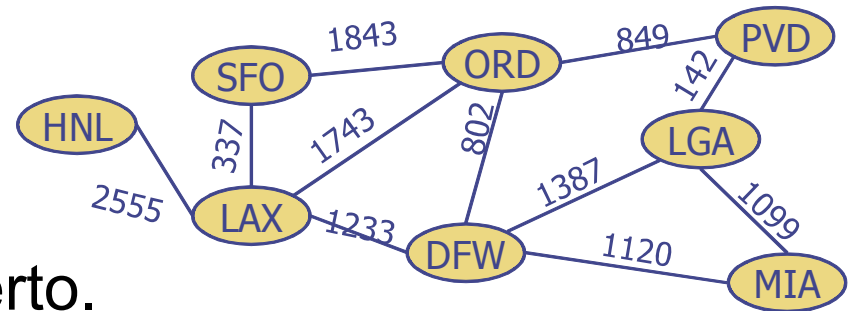
- Introducción
 - Definición
 - Aplicaciones
 - Terminología
- Estructura de datos
- Algoritmos

Definición

- Un grafo es un par (V,E) , donde:
 - **V** es un conjunto de nodos, llamados *vértices*
 - **E** es una colección de par de vértices, llamadas *aristas*.
 - $G=(V,E)$
 - Vértices y aristas son posiciones y guardan datos.

- Ejemplos:

- Un *vértice* representa un aeropuerto y guarda las tres letras del código del aeropuerto.
- Una *arista* representa una ruta de vuelo entre dos aeropuertos y guarda la cantidad de kilómetros de la ruta

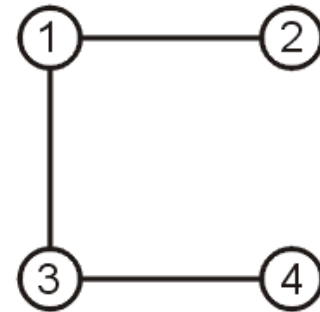


Definición (cont.)

- El número de vértices es $|V|$
- El número de aristas es $|E|$
- Ejemplos:

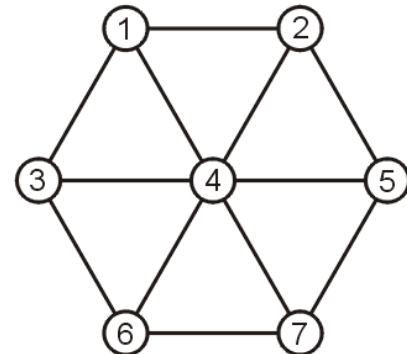
$$V = \{v_1, v_2, v_3, v_4\} \quad |V| = 4$$

$$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_3, v_4\}\} \quad |E| = 3$$



$$V = \{1, 2, 3, 4, 5, 6, 7\} \quad |V| = 7$$

$$E = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{2, 5\}, \{3, 4\}, \\ \{3, 6\}, \{4, 5\}, \{4, 6\}, \{4, 7\}, \{5, 7\}, \{6, 7\}\} \quad |E| = 12$$



Aplicaciones

- Impresión de circuitos electrónicos.
- Circuitos integrados.
- Redes de transporte.
- Redes de computadoras (Internet, web, etc).
- Redes en general
- Compiladores y análisis automático de programas
- Administración de proyectos
- Análisis de uso/interacción en redes sociales
- Seguridad computacional
- Etc, etc

Conceptos

- **Tipo de aristas**

- Dirigidas

- Par de vértices ordenados (u,v)
 - Primer vértice u es el origen
 - Segundo vértice v es el destino
 - Ejemplo: un vuelo



- No dirigidas

- Par de vértices no ordenados (u,v)
 - Ejemplo: una ruta de vuelo



- **Grafos dirigido (*digrafos*)**

- Todas las aristas son dirigidas (ejemplo: red de vuelos)

- **Grafo no dirigido**

- Todas las aristas son no dirigidas (ejemplo: red de rutas).

Conceptos

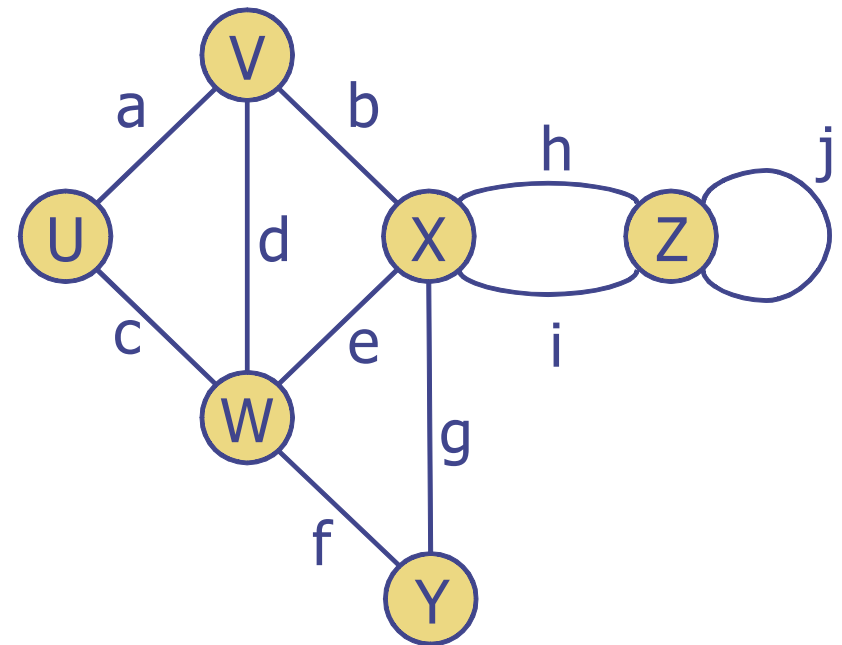
Preguntas:

- Dado un grafo no dirigido $G=(V,E)$, ¿Cuál es el número máximo de aristas ($|E|$) que se puede tener sabiendo $|V|$?
- ¿Cómo puede acotar $|E|$?

$$|E| = \frac{|V|. (|V|-1)}{2} = O(|V|^2)$$

Terminología

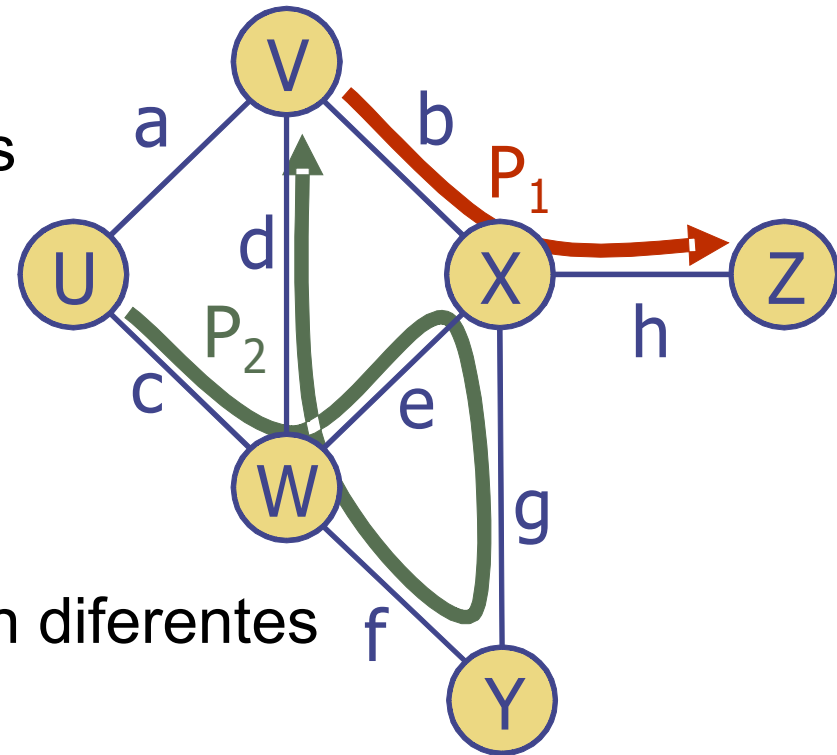
- Vértices finales de una arista
 - U y V son “*endpoints*” de a
- Aristas incidentes en un vértice
 - a, d y b son incidentes en V
- Vértices adyacentes
 - U y V son adyacentes
- Grado de un vértice
(cantidad de aristas incidentes en él)
 - X tiene grado 5
- Aristas paralelas
 - h, i son aristas paralelas
- Auto-ciclo
 - j tiene un auto-ciclo



Terminología

- Camino

- Secuencia de vértices y aristas alternados
- Inicia en un vértice y termina en un vértice



- Camino simple

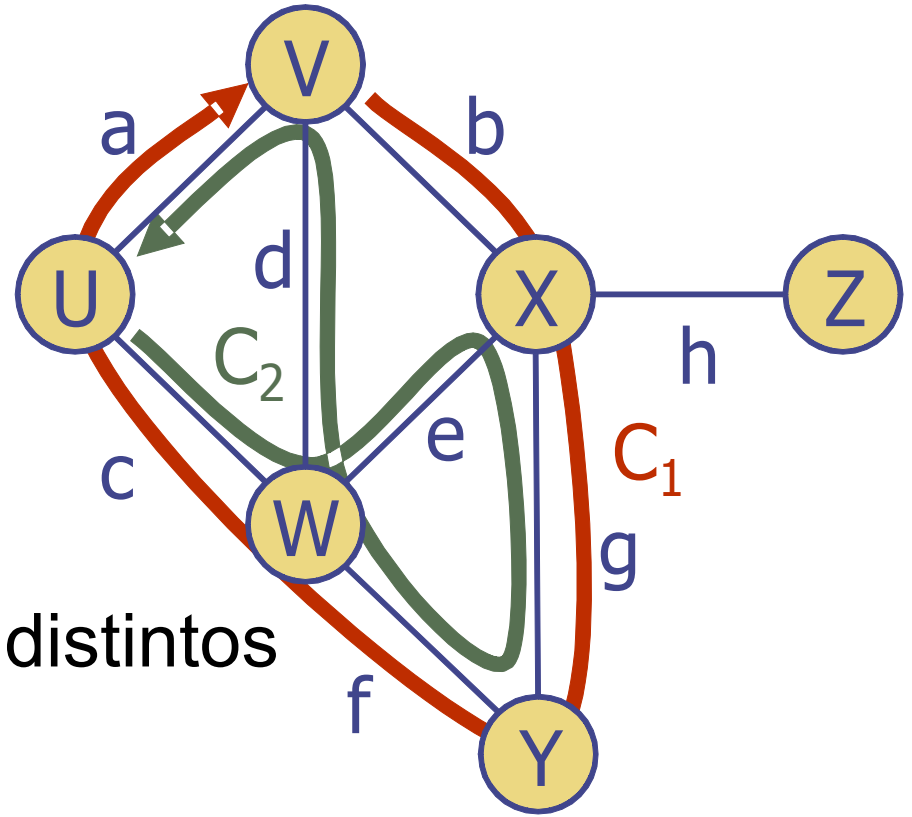
- Todos los vértices y aristas son diferentes

- Ejemplos

- **P₁ (V,b,X,h,Z)** es un camino simple
- **P₂ (U,c,W,e,X,g,Y,f,W,d,V)** no es un camino simple

Terminología

- Ciclo
 - Secuencia circular alternados de vértices y aristas
- Ciclo simple
 - Los vértices y aristas son distintos
- Ejemplos
 - $C_1 = (V, b, X, g, Y, f, W, c, U, a)$ es un ciclo simple
 - $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a)$ no es un ciclo simple



Tipo de grafos según densidad

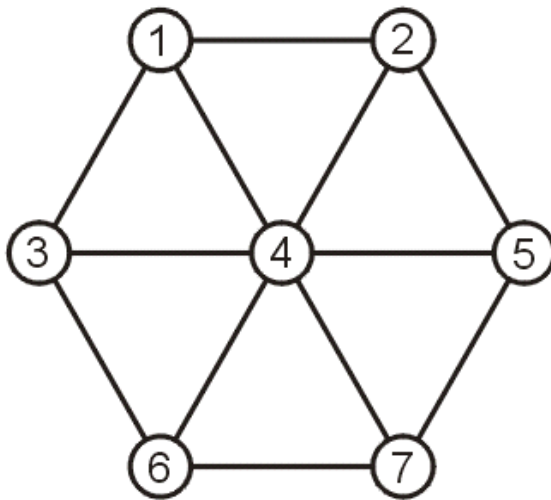
- En general se puede pensar (a lo sumo):

$$0 \leq |E| \leq |V|^2$$

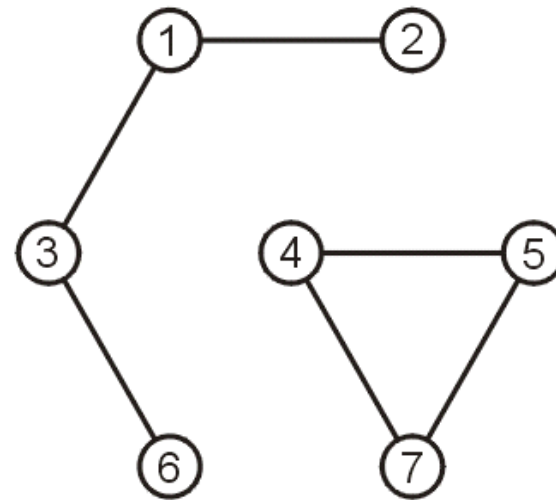
- Cuando la mayoría de los vértices están presentes, tenemos $|E| = \Theta(|V|^2)$ y el grafo es **denso**
- Es un grafo **disperso** cuando $|E| = \Theta(|V|)$ o algo más (es lo usual).

Conectividad

- Un grafo es conexo si existe un camino entre cualquier par de vértices



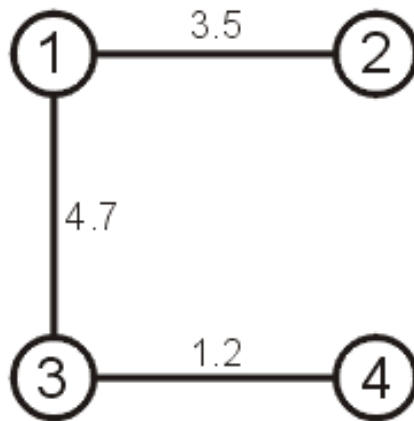
CONEXO



No CONEXO

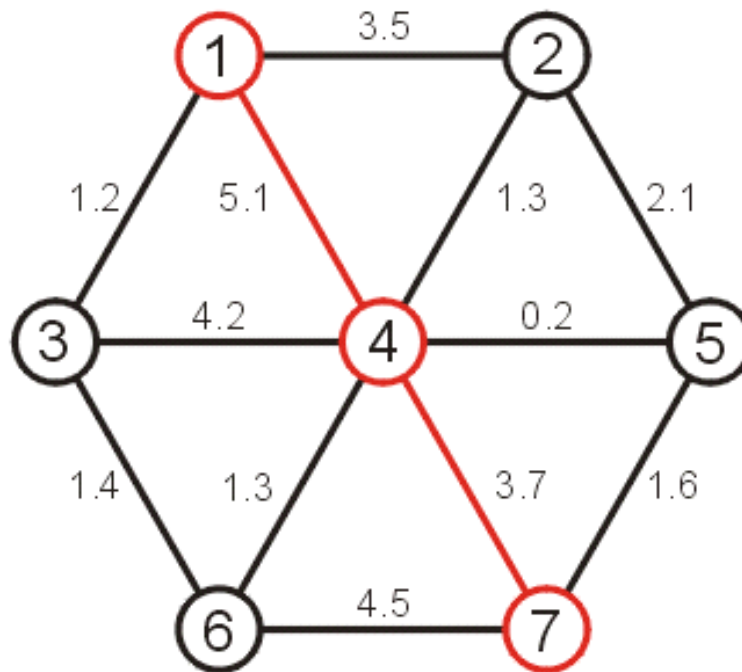
Pesos

- Un grafo se denomina “con peso” si cada arista tiene asociada un peso.



- La longitud de un camino es la suma de los pesos de todas las aristas que la conforman.
- Un grafo sin peso tienen sus aristas con pesos iguales (ej : 1)

- La longitud del camino (1,4,7) en el siguiente grafo es $5.1 + 3.7 = 8.8$



Grafos dirigidos

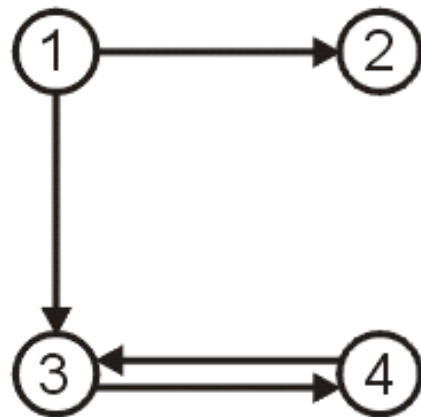
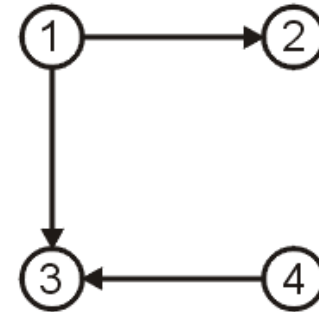
- Las aristas en un grafo se asocian con una dirección
- Todas las aristas son pares ordenados (v_i, v_j) donde esto denota una conexión de v_i a v_j .
- No sugiere que haya una conexión entre v_j a v_i .

Grafos dirigidos

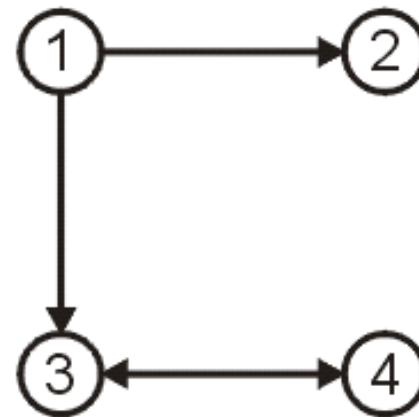
Ejemplos:

$$G = \{1, 2, 3, 4\}$$

$$E = \{(1, 2), (1, 3), (4, 3)\}$$



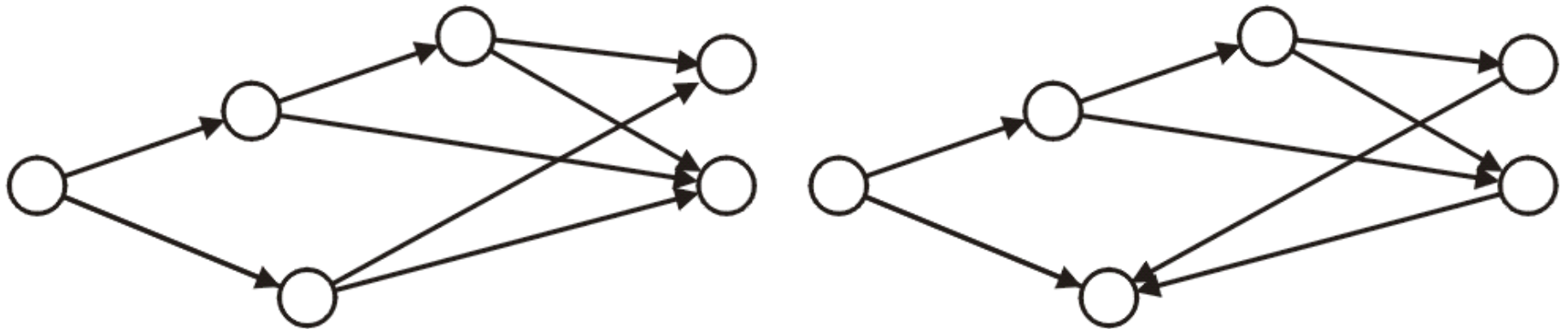
or



$$E = \{(1, 2), (1, 3), (3, 4), (4, 3)\}$$

Grafos dirigidos acíclicos - GDA

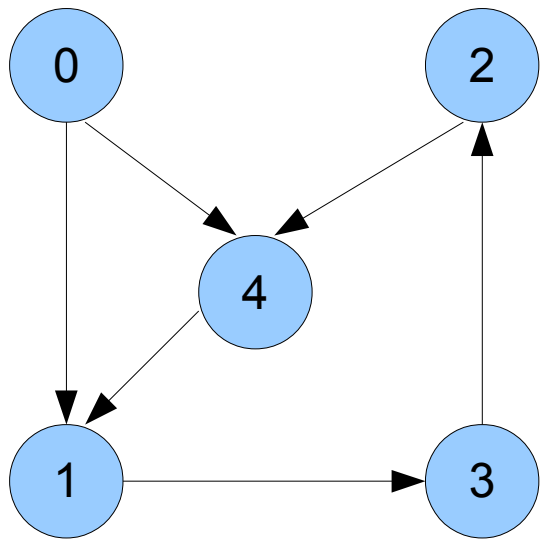
- Es un grafo dirigido sin ciclos.
- Ejemplos:



- Aplicación: Orden de tareas, plan curricular, etc.

Representación

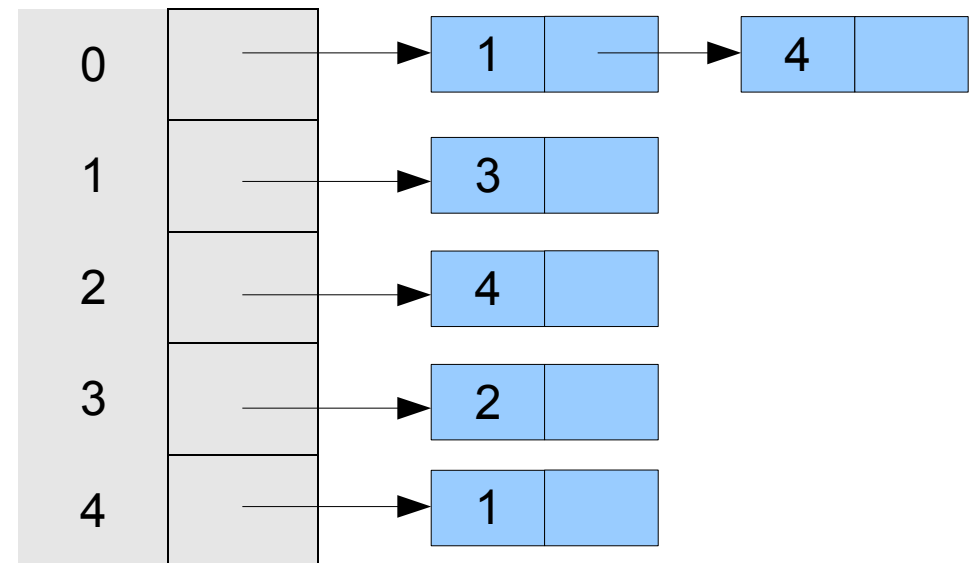
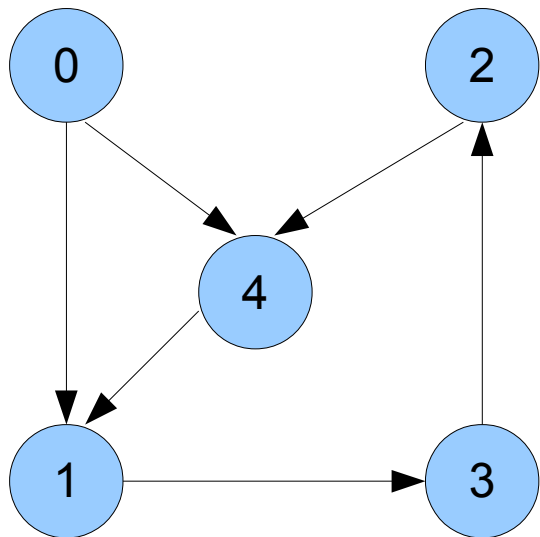
- Matriz de adyacencia



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

Representación

- Lista de adyacencia



Representación

- Rendimiento

	Matriz de adyacencia	Lista de Adyacencia
Espacio	V^2	$V+E$
AristasIncidentes(v)	V	grado(v)
sonAdyacentes(v,w)	1	$\min(\text{grado}(v), \text{grado}(w))$
insertarVertice(o)	V^2	1
insertarArista(v,w,o)	1	1
removerVertice(v)	V^2	grado(v)
removerArista(e)	1	$\text{grado}(e.v1) + \text{grado}(e.v2)$

Barrido de grafos

- Búsqueda en profundidad (DFS)
- Búsqueda en amplitud (BFS)
- Orden topológico

```

class LList implements List { .. Implementa Lista enlazada }

class GraphList extends LList {
    public Link currLink() { return curr; }
    public void setCurr(Link who) { curr = who; }
}

class Link {                // A singly linked list node
    private Object element;    // Object for this node
    private Link next;        // Pointer to next node in list
    Link(Object it, Link nextval)    // Constructor 1
        { element = it; next = nextval; }    // Given Object
    Link(Link nextval) { next = nextval; } // Constructor 2
    Link next() { return next; }
    Link setNext(Link nextval) { return next = nextval; }
    Object element() { return element; }
    Object setElement(Object it) { return element = it; }
} // class Link

class Edgel implements Edge {
    private int vert1, vert2;    // Indices of v1, v2
    private Link itself; // Pointer to node in the adjacency list
    public Edgel(int vt1, int vt2, Link it) // Constructor
        { vert1 = vt1; vert2 = vt2; itself = it; }

    public int v1() { return vert1; }
    public int v2() { return vert2; }
    Link theLink() { return itself; } // Access into adjacency list
}

```

Estructura de datos para los ejemplos

```
class Graphl implements Graph {           // Graph: Adjacency list

    private GraphList[] vertex;           // The vertex list
    private int numEdge;                   // Number of edges
    public int[] Mark;                     // The mark array

    public Graphl(int n) {                 // Constructor
        Mark = new int[n];
        vertex = new GraphList[n];
        for (int i=0; i<n; i++)
            vertex[i] = new GraphList();
        numEdge = 0;
    }

    public int n() { return Mark.length; } // Number of vertices
    public int e() { return numEdge; }     // Number of edges

    public int first(int v) { // Get the first edge for a vertex
        vertex[v].setFirst();
        if (vertex[v].currValue() == null) return null;
        return new Edgel(v, ((int[])vertex[v].currValue())[0],
                           vertex[v].currLink());
    }
}
```

```

public boolean isEdge(Edge e) { // True if this is an edge
    if (e == null) return false;
    vertex[e.v1()].setCurr(((Edge)e).theLink());
    if (!vertex[e.v1()].isInList()) return false;
    return ((int[])vertex[e.v1()].currValue())[0] == e.v2();
}

public int v1(Edge e) { return e.v1(); } // Where edge comes from
public int v2(Edge e) { return e.v2(); } // Where edge goes to

public boolean isEdge(int i, int j) { // True if this is an edge
    GraphList temp = vertex[i];
    for (temp.setFirst(); ((temp.currValue() != null) &&
        (((int[])temp.currValue())[0] < j)); temp.next());
    return (temp.currValue() != null) &&
        (((int[])temp.currValue())[0] == j);
}

public Edge next(Edge e) { // Get next edge for a vertex
    vertex[e.v1()].setCurr(((Edge)e).theLink());
    vertex[e.v1()].next();
    if (vertex[e.v1()].currValue() == null) return null;
    return new Edge(e.v1(), ((int[])vertex[e.v1()].currValue())[0],
        vertex[e.v1()].currLink());
}

```



```

public void setEdge(int i, int j, int weight) { // Set edge weight
    Assert.notFalse(weight!=0, "Cannot set weight to 0");
    int[] currEdge = { j, weight };
    if (isEdge(i, j)) // Edge already exists in graph
        vertex[i].setValue(currEdge);
    else { // Add new edge to graph
        vertex[i].insert(currEdge);
        numEdge++;
    }
}

public void setEdge(Edge w, int weight) // Set edge weight
    { if (w != null) setEdge(w.v1(), w.v2(), weight); }

public void delEdge(int i, int j) // Delete edge
    { if (isEdge(i, j)) { vertex[i].remove(); numEdge--; } }

public void delEdge(Edge w) // Delete edge
    { if (w != null) delEdge(w.v1(), w.v2()); }

public int weight(int i, int j) { // Return weight of edge
    if (isEdge(i, j)) return ((int[])vertex[i].currValue())[1];
    else return Integer.MAX_VALUE;
}

public void setMark(int v, int val) { Mark[v] = val; } // Set Mark

public int getMark(int v) { return Mark[v]; } // Get Mark
}

```

Búsqueda en profundidad (DFS)

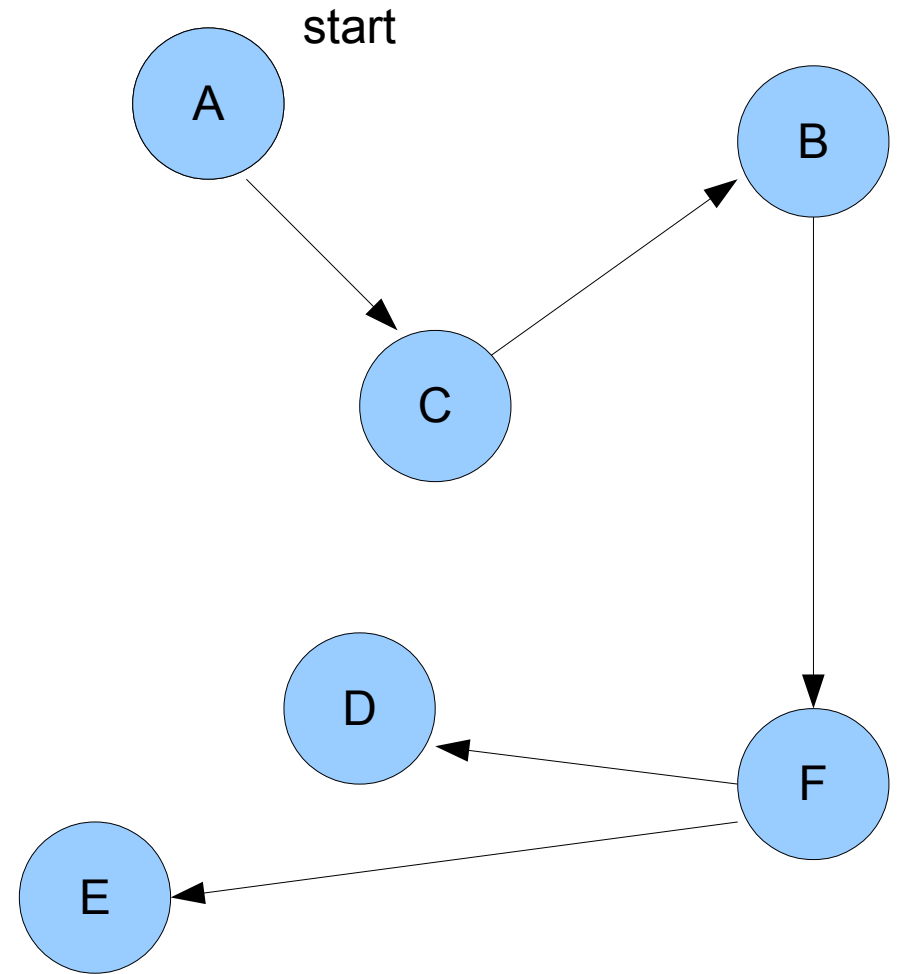
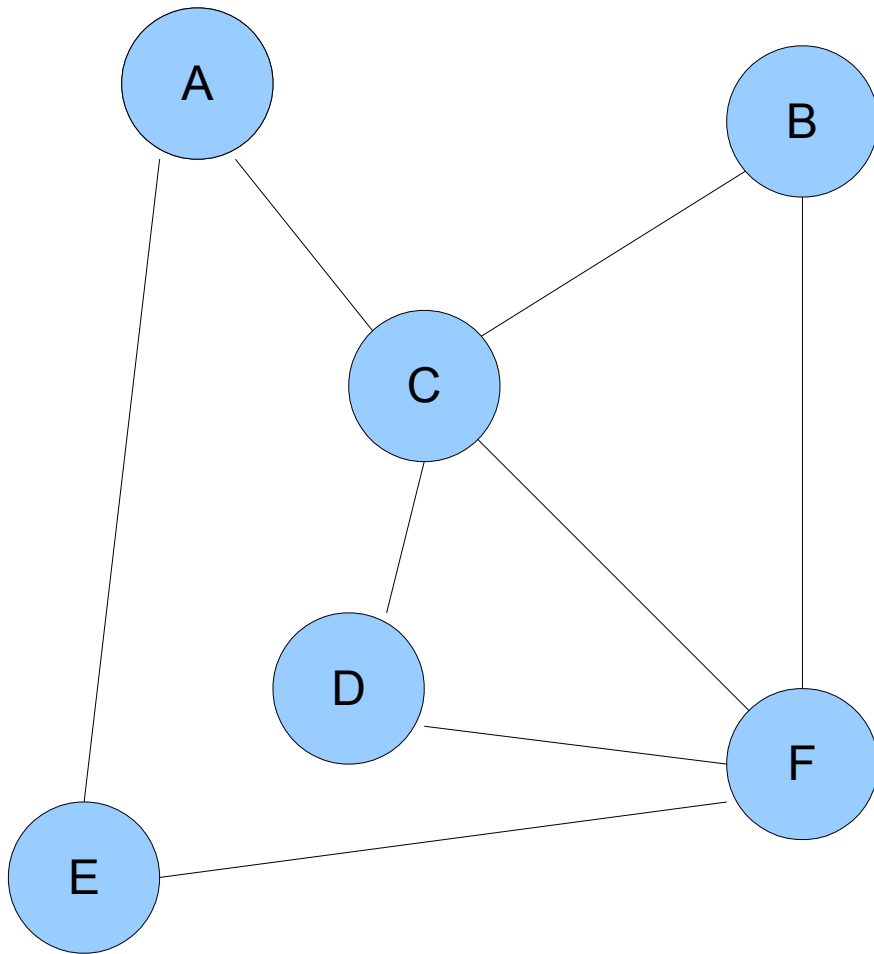
- Cuando se visita un vértice v , recursivamente se visita todos los vértices vecinos no visitados.

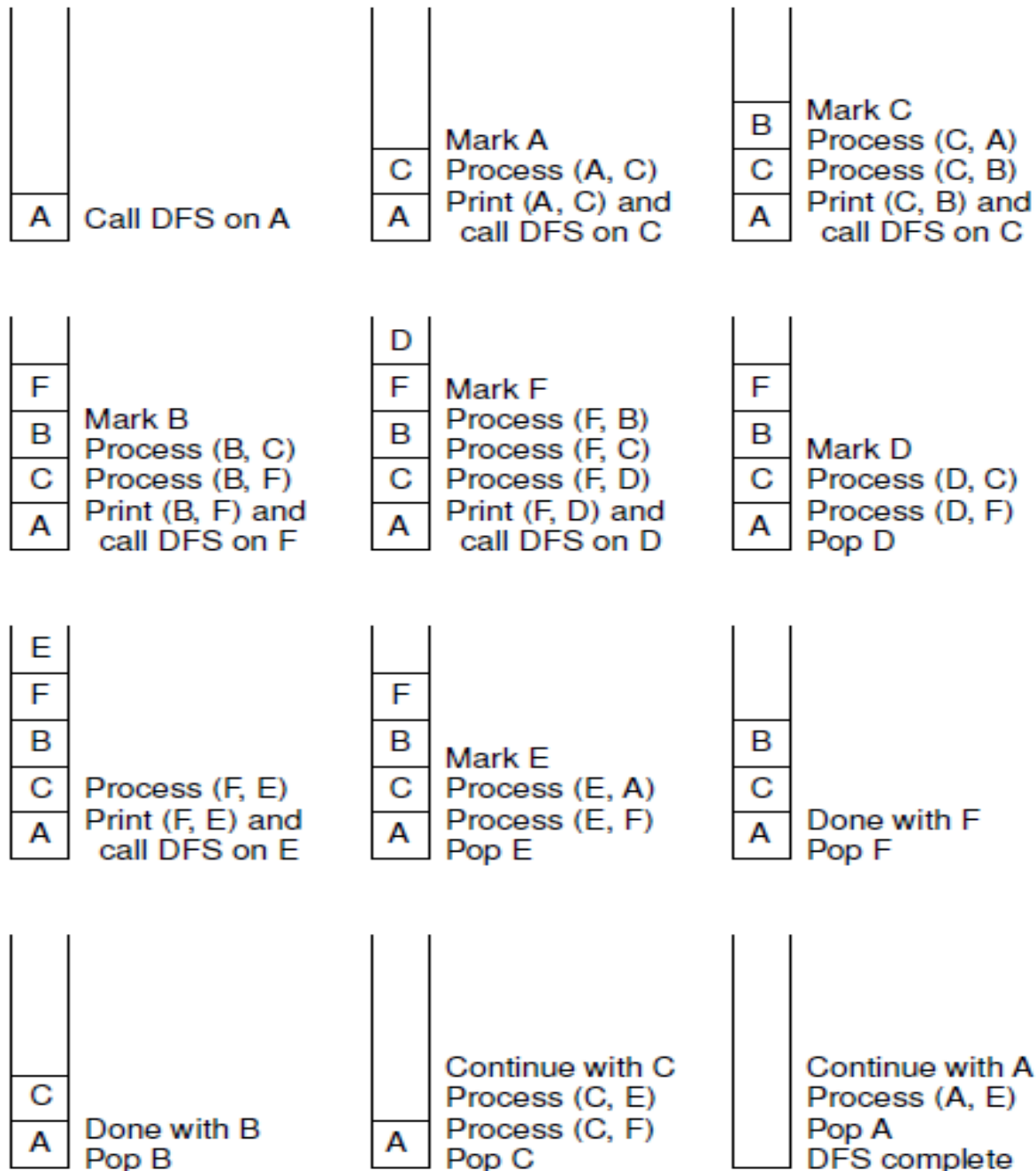
```
static void DFS ( Graph G, int v ) { //Codigo de [Shaffer1998]
    PreVisita(G,v)
    G.setMark(v, VISITADO)

    for ( Edge w = G.first(v); G.isEdge(w); w=G.next(w) ){
        if ( G.getMark(G.v2(w)) == NOVISITADO )
            DFS (G.v2(w))

    PostVisita(G,v)
}
```

Búsqueda en profundidad





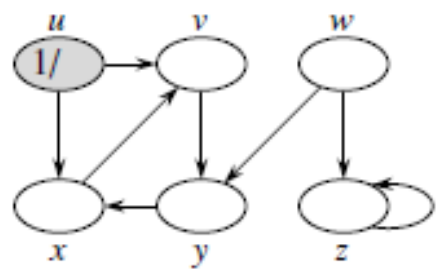
Algoritmo DFS (CLRS)

DFS(G)

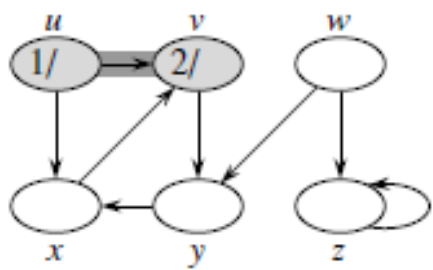
```
1  for each vertex  $u \in V[G]$ 
2      do  $color[u] \leftarrow \text{WHITE}$ 
3          $\pi[u] \leftarrow \text{NIL}$ 
4   $time \leftarrow 0$ 
5  for each vertex  $u \in V[G]$ 
6      do if  $color[u] = \text{WHITE}$ 
7          then DFS-VISIT( $u$ )
```

DFS-VISIT(u)

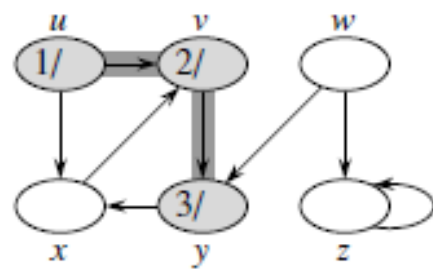
```
1   $color[u] \leftarrow \text{GRAY}$        $\triangleright$  White vertex  $u$  has just been discovered.
2   $time \leftarrow time + 1$ 
3   $d[u] \leftarrow time$ 
4  for each  $v \in Adj[u]$        $\triangleright$  Explore edge  $(u, v)$ .
5      do if  $color[v] = \text{WHITE}$ 
6          then  $\pi[v] \leftarrow u$ 
7              DFS-VISIT( $v$ )
8   $color[u] \leftarrow \text{BLACK}$        $\triangleright$  Blacken  $u$ ; it is finished.
9   $f[u] \leftarrow time \leftarrow time + 1$ 
```



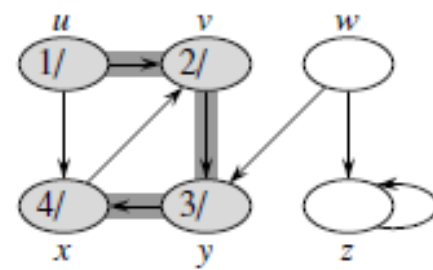
(a)



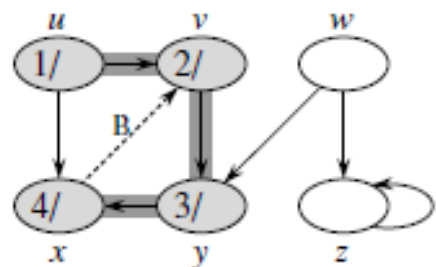
(b)



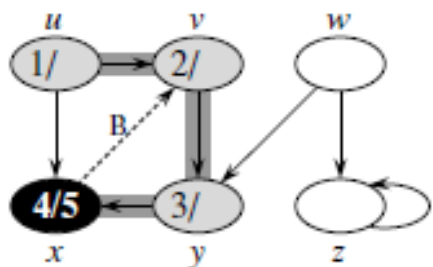
(c)



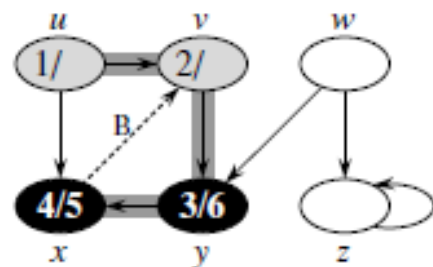
(d)



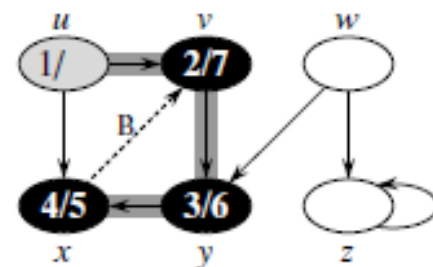
(e)



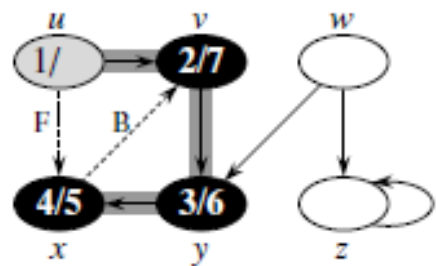
(f)



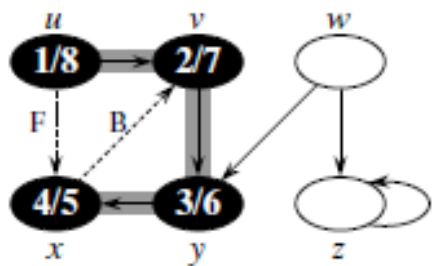
(g)



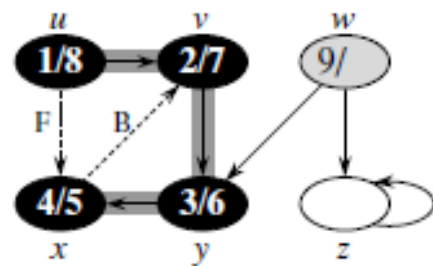
(h)



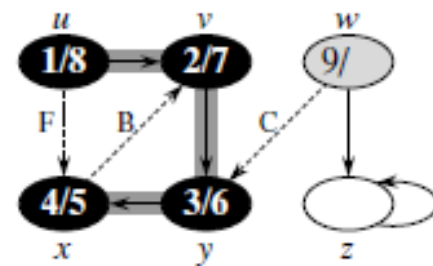
(i)



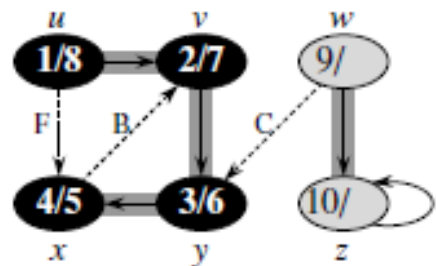
(j)



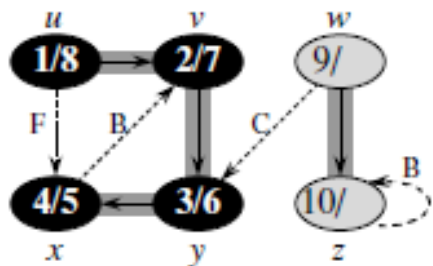
(k)



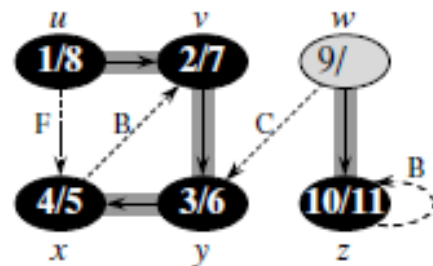
(l)



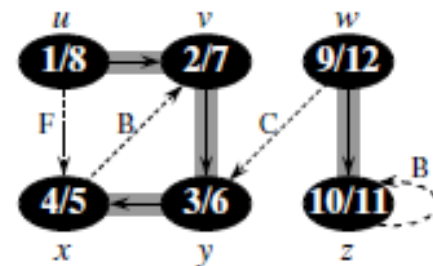
(m)



(n)



(o)



(p)

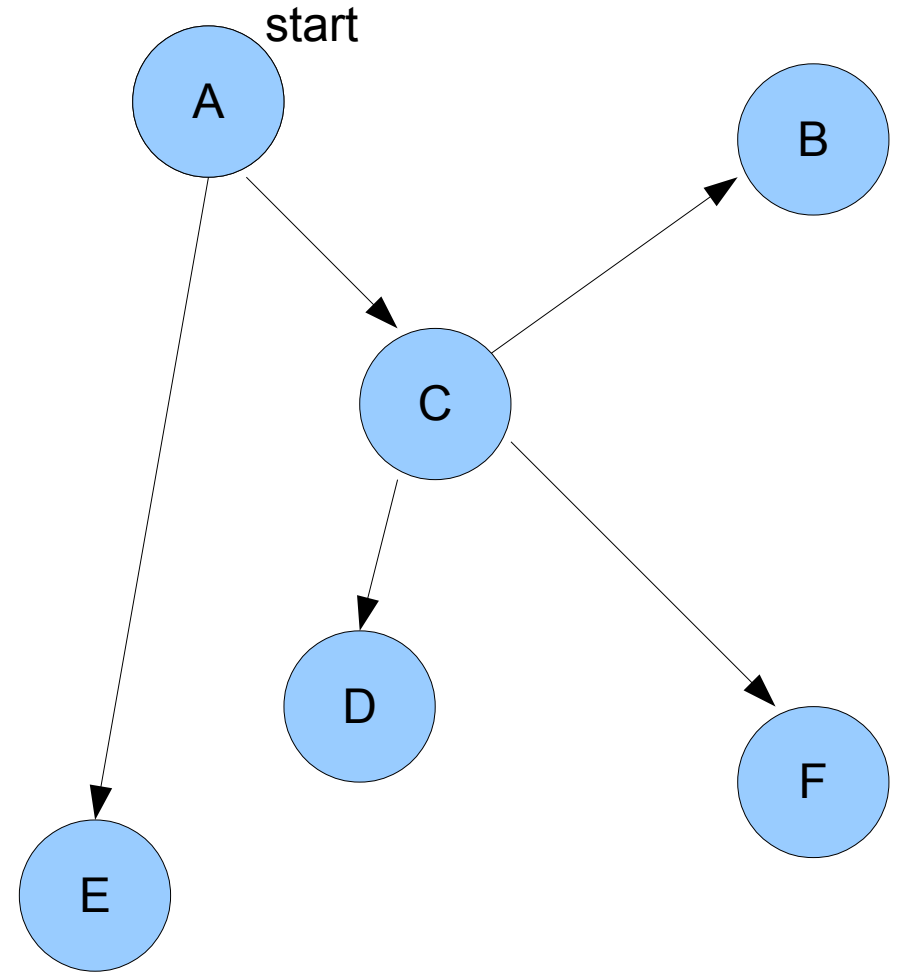
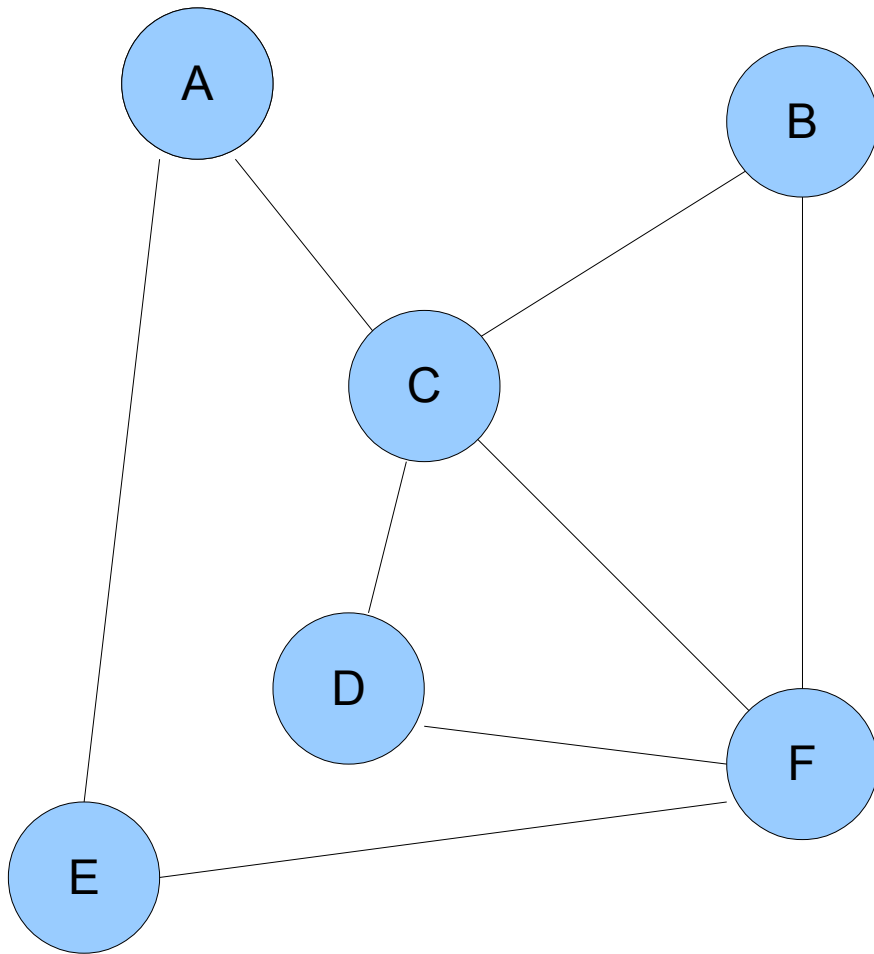
Búsqueda en amplitud (BFS)

- Examina todos los vértices conectados al vértice inicial antes de visitar los más lejanos

```
static void BFS ( Graph G, int start ) { // Codigo de [Shaffer1998]
    Cola Q = new Cola(G.n()) //
    Q.encolar(start)
    G.marcar(start, VISITADO)

    while ( !Q.vacio() ) {
        v = Q.decolar()
        PreVisita(G,v)
        for ( Arista w = G.first(v); G.isEdge(w); w=G.next(w) ){
            if ( G.getMark(G.v2(w)) == NOVISITADO ) {
                G.setMark(G.v2(w), VISITADO)
                Q.encolar(G.v2(w))
            }
        }
        PostVisita(G,v)
    }
}
```

Búsqueda en amplitud



A	
---	--

Initial call to BFS on A.
Mark A and put on the queue.

C	E	
---	---	--

Dequeue A.
Process (A, C).
Mark and enqueue C. Print (A, C).
Process (A, E).
Mark and enqueue E. Print(A, E).

E	B	D	F	
---	---	---	---	--

Dequeue C.
Process (C, A). Ignore.
Process (C, B).
Mark and enqueue B. Print (C, B).
Process (C, D).
Mark and enqueue D. Print (C, D).
Process (C, F).
Mark and enqueue F. Print (C, F).

B	D	F	
---	---	---	--

Dequeue E.
Process (E, A). Ignore.
Process (E, F). Ignore.

D	F	
---	---	--

Dequeue B.
Process (B, C). Ignore.
Process (B, F). Ignore.

F	
---	--

Dequeue D.
Process (D, C). Ignore.
Process (D, F). Ignore.

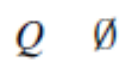
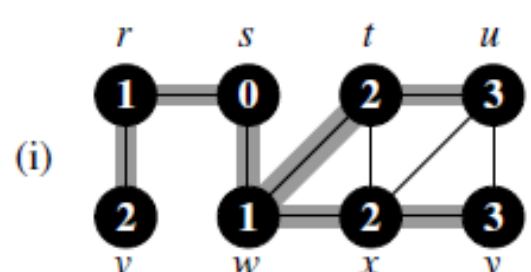
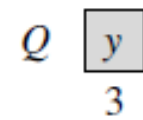
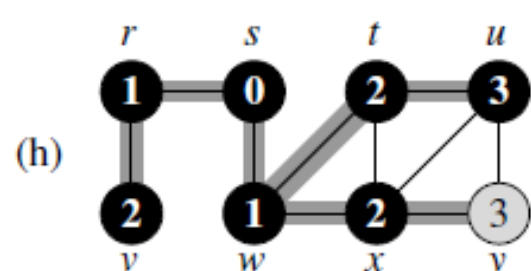
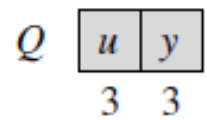
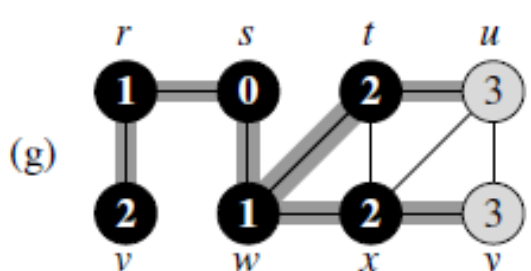
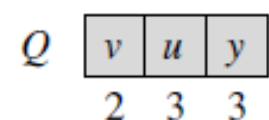
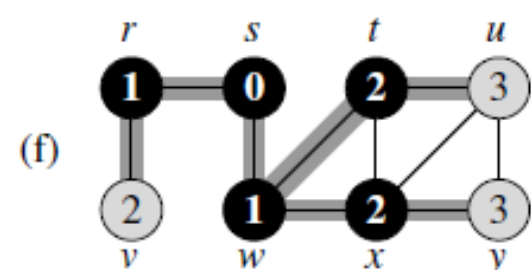
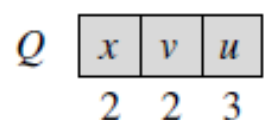
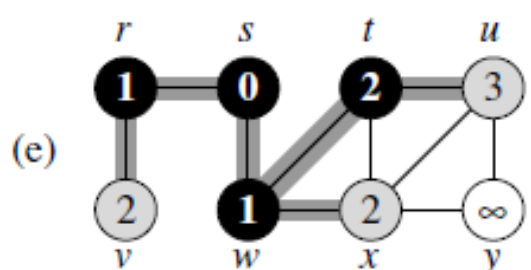
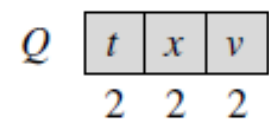
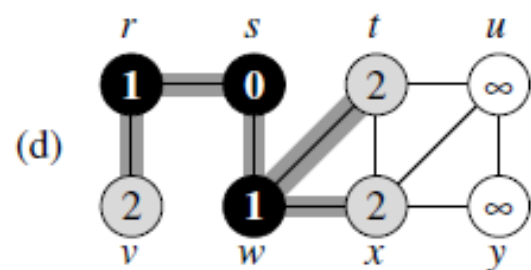
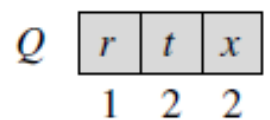
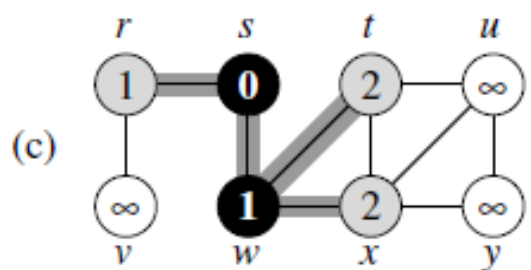
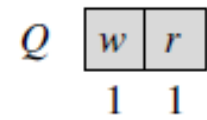
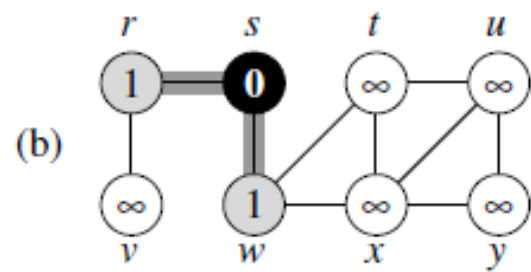
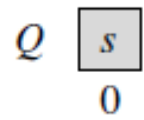
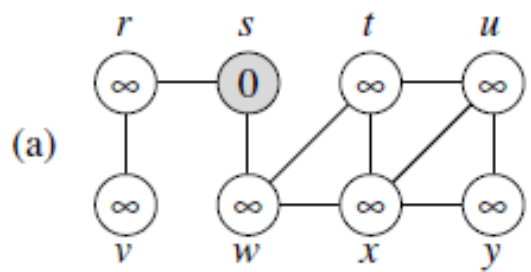
--	--	--	--	--

Dequeue F.
Process (F, B). Ignore.
Process (F, C). Ignore.
Process (F, D). Ignore.
BFS is complete.

Algoritmo BFS (CLRS)

BFS(G, s)

```
1  for each vertex  $u \in V[G] - \{s\}$ 
2      do  $color[u] \leftarrow \text{WHITE}$ 
3           $d[u] \leftarrow \infty$ 
4           $\pi[u] \leftarrow \text{NIL}$ 
5   $color[s] \leftarrow \text{GRAY}$ 
6   $d[s] \leftarrow 0$ 
7   $\pi[s] \leftarrow \text{NIL}$ 
8   $Q \leftarrow \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11     do  $u \leftarrow \text{DEQUEUE}(Q)$ 
12         for each  $v \in Adj[u]$ 
13             do if  $color[v] = \text{WHITE}$ 
14                 then  $color[v] \leftarrow \text{GRAY}$ 
15                      $d[v] \leftarrow d[u] + 1$ 
16                      $\pi[v] \leftarrow u$ 
17                     ENQUEUE( $Q, v$ )
18      $color[u] \leftarrow \text{BLACK}$ 
```



Orden topológico

- Recorrido en un grafo dirigido acíclico “por niveles”. *Es un recorrido en amplitud*
- Dado dos vértices v_i y v_j en un GDA solo puede existir caminos de v_i a v_j ó solo de v_j a v_i .
- Si existe un camino de v_i hasta v_j , entonces v_j aparece después de v_i en el ordenamiento topológico.

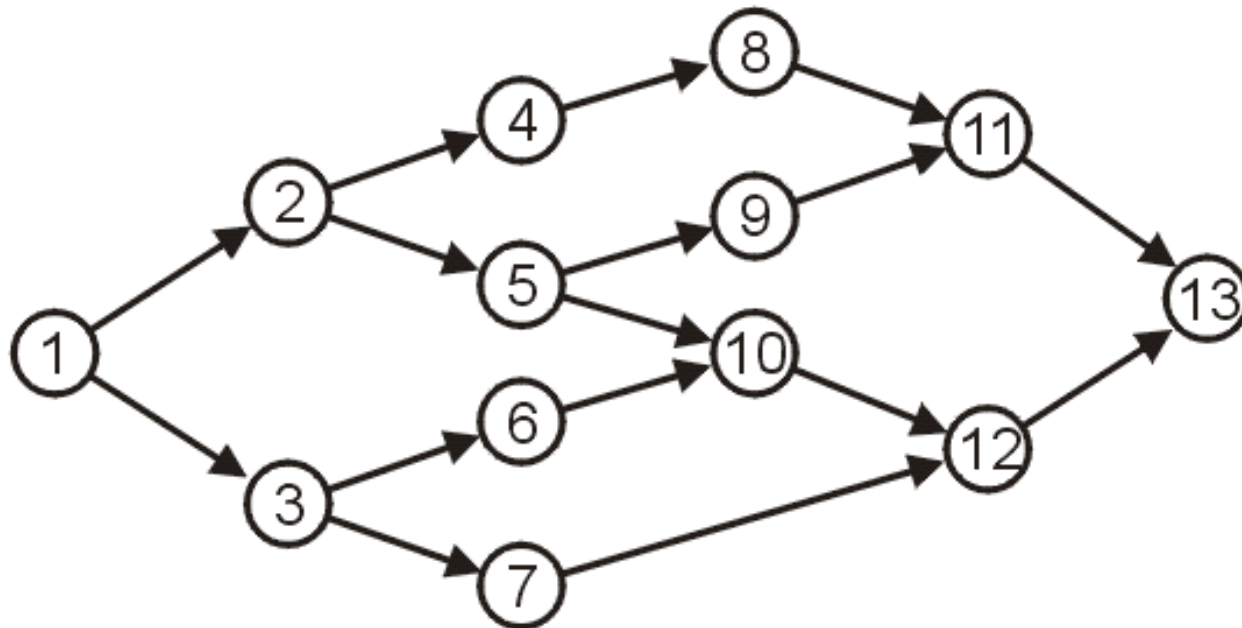
Orden topológico

Ejemplos de uso:

- Listar las materias de un plan curricular por orden de prerrequisito.
- Listar las tareas de una actividad en orden de ejecución.
 - Construcción de librerías o generación de programas donde existan dependencia entre ellos. (Compilación y enlazado).
 - Instalación de paquetes en Linux (apt-get utiliza este algoritmo).
 - Construcción de una casa o edificio

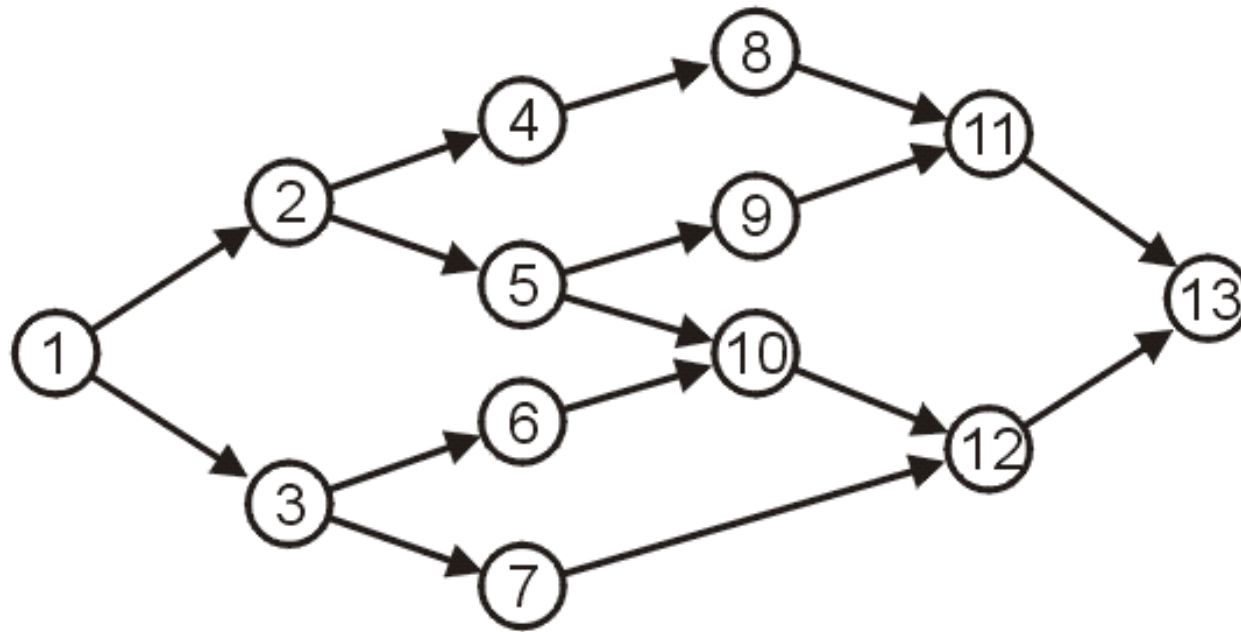
Orden topológico

- Por ejemplo en este GDA existe una ordenación topológica:
1,2,3,4,5,6,7,8,9,10,11,12,13



Orden topológico

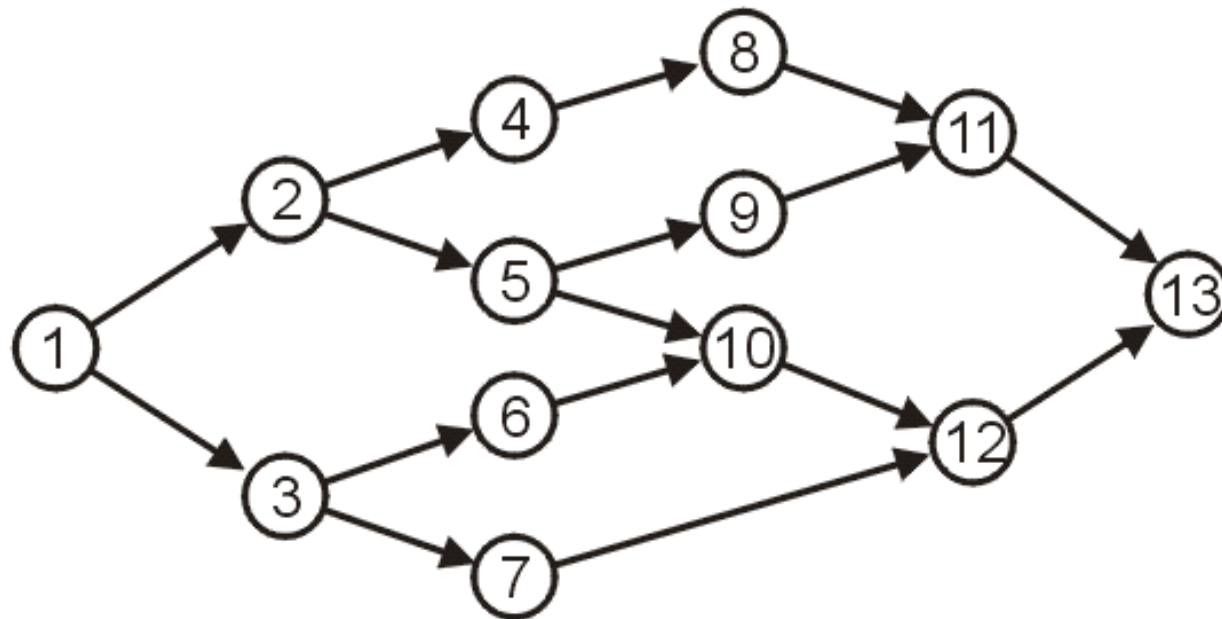
- Puede no ser única
 - 1, 3, 2, 7, 6, 5, 4, 10, 9, 8, 12, 11, 13
 - 1, 2, 4, 8, 5, 9, 11, 3, 6, 10, 7, 12, 13



Orden topológico - algoritmo

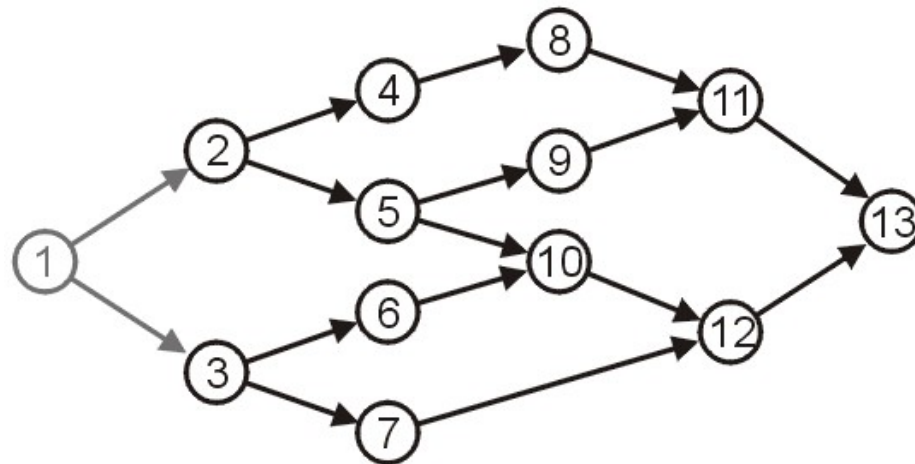
- Comenzamos con el nodo con grado de entrada (*indegree*) igual a cero

1



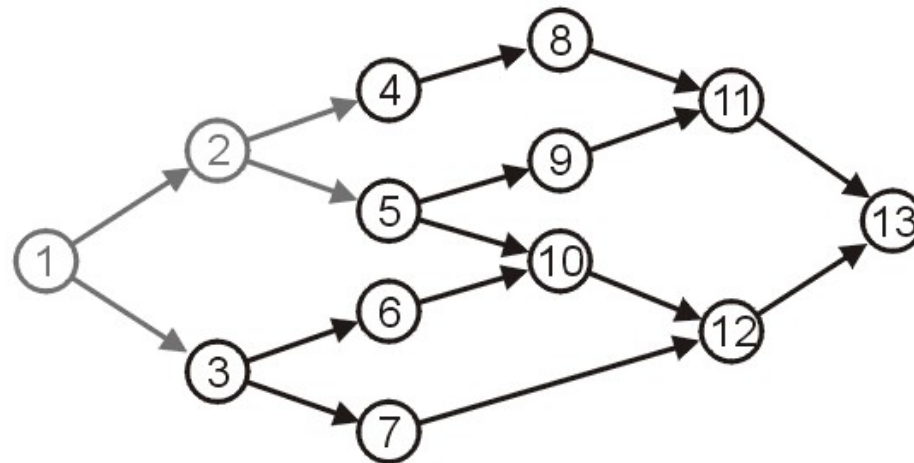
- Ignoramos las aristas que conectan el vértice 1 con otros vértices y tomamos el que tiene grado de entrada cero

1, 2

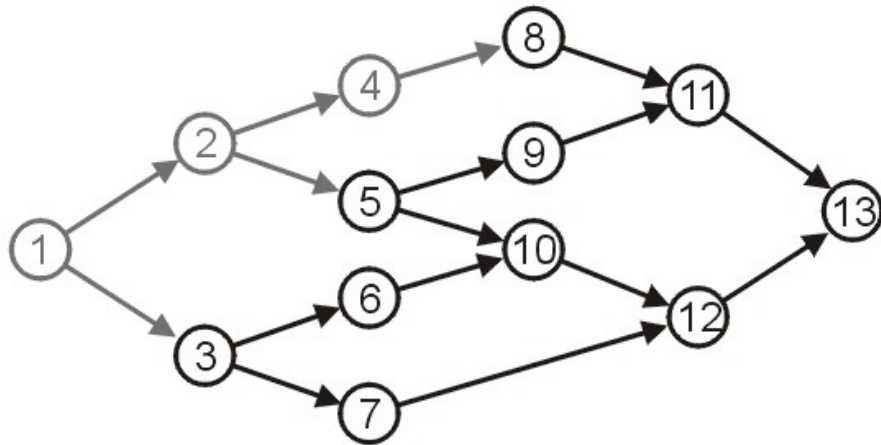


- Ignoramos todos los vértices que conectan 1 y 2. Podemos elegir 4, 5 o 3

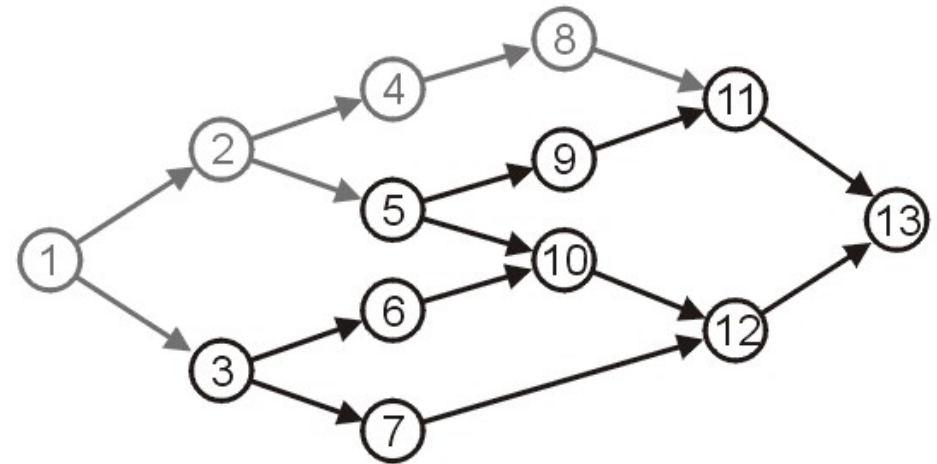
1,2,4



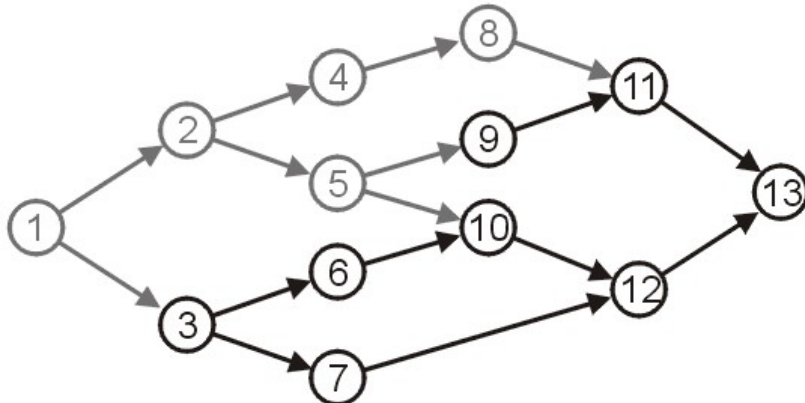
1,2,4,8



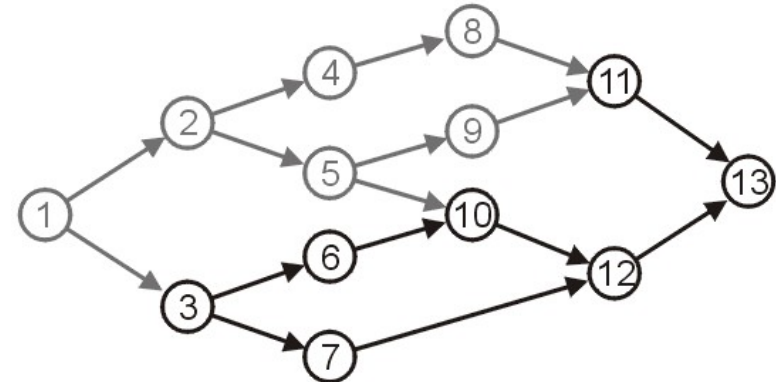
1,2,4,8,5



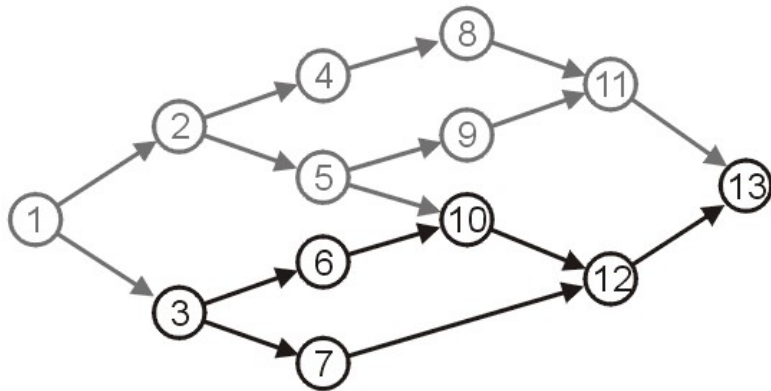
1,2,4,8,5,9



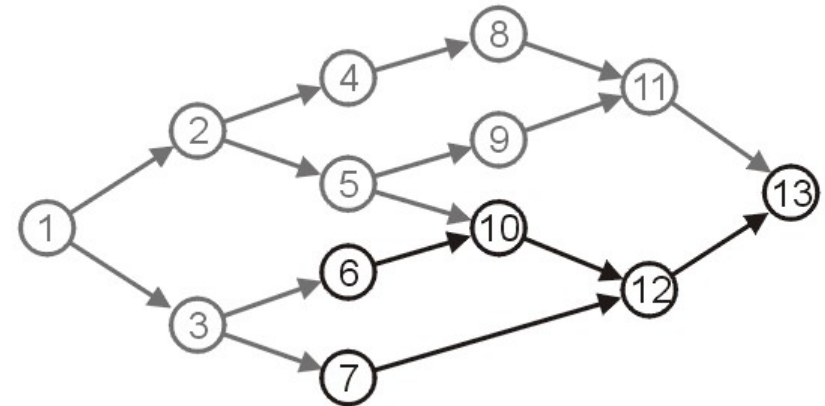
1,2,4,8,5,9,11



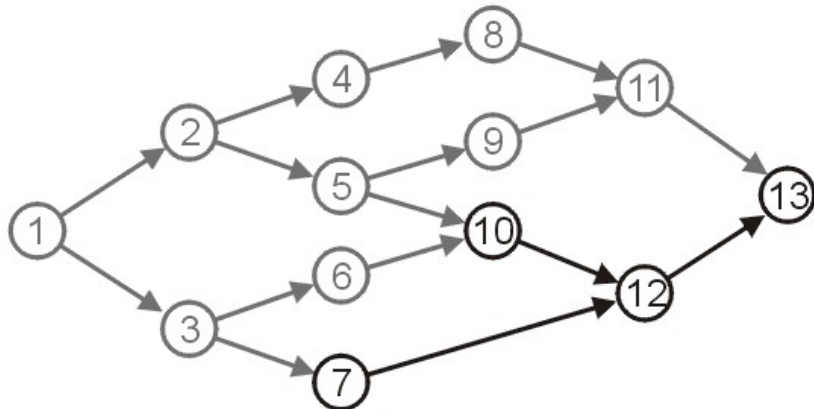
1,2,4,8,5,9,11,3



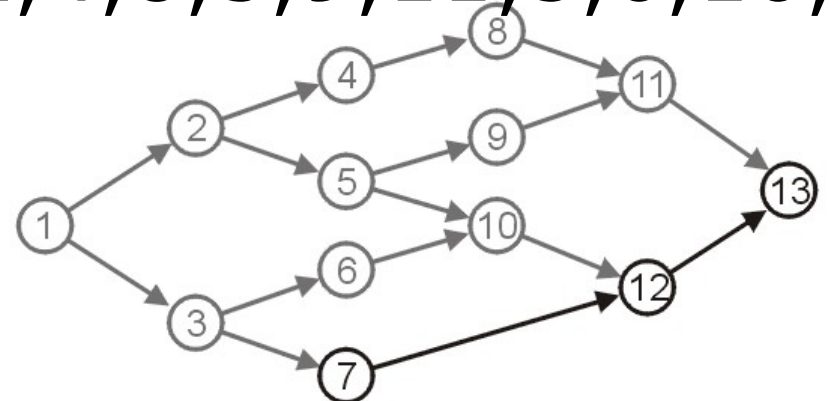
1,2,4,8,5,9,11,3,6



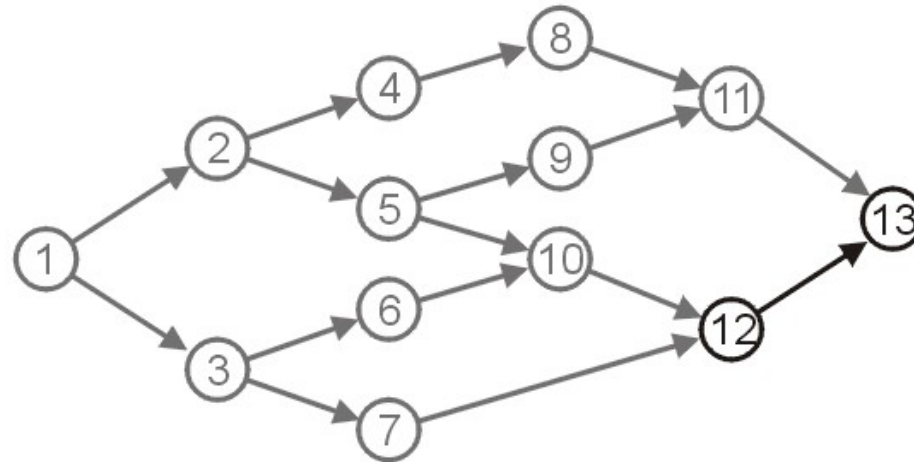
1,2,4,8,5,9,11,3,6,10



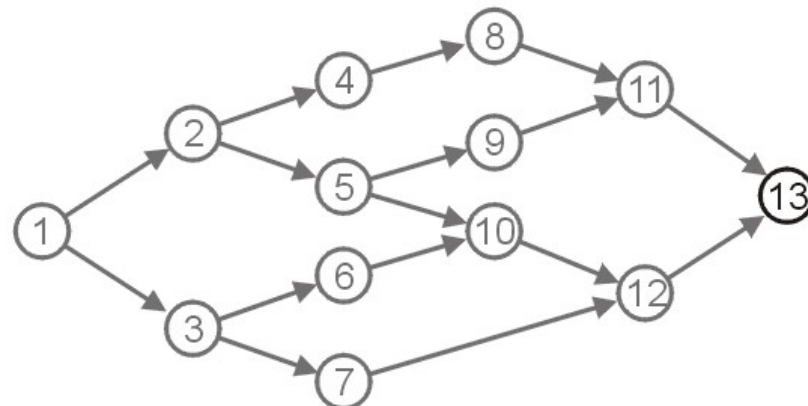
1,2,4,8,5,9,11,3,6,10,7



1,2,4,8,5,9,11,3,6,10,7,12



1,2,4,8,5,9,11,3,6,10,7,12,13



Orden topológico: usando una cola

```
static void OrdenTopologico ( Graph G ) {  
    Cola<Integer> Q = new Cola <Integer> (G.n());  
    int [] Count    = new int[G.n()];  
  
    for ( int v = 0; v < G.n(); v++ ) Count[v] = 0;  
  
    for ( int v = 0; v < G.n(); i++ )  
        for ( Edge w = G.first(v); G.isEdge(w); w=G.next(w) )  
            Count[G.v2(w)]++;  
  
    for ( int v = 0; v < G.n(); i++ )  
        if ( Count[v] == 0 )  
            Q.encolar(v);  
  
    while ( Q.length() > 0 ) ) {  
        v = Q.decolar().intValue();  
        imprimir(v);  
        for ( Edge w = G.first(v); G.isEdge(w); w=G.next(w) ) {  
            Count[G.v2(w)]--;  
            if ( Count[G.v2(w)] == 0 )  
                Q.encolar(G.v2(w));  
        }  
    }  
    imprimir(v)  
}
```

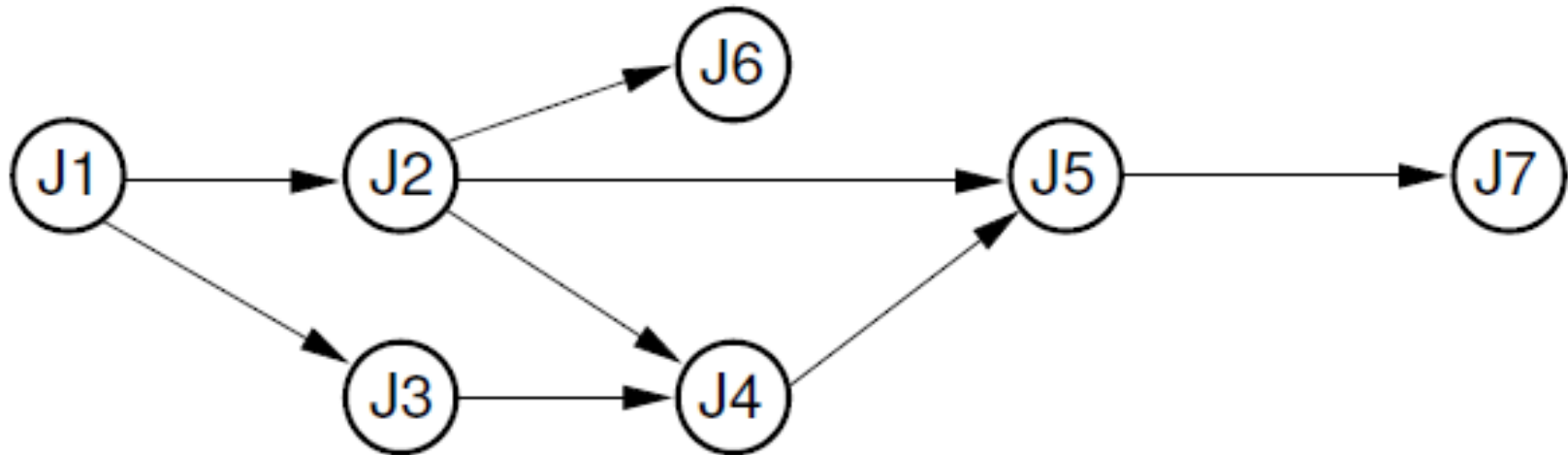
Orden topológico

usando DFS (procesando al final)

```
static void OrdenTopologico ( Graph G ) {  
    for ( int n = 0; i < G.n(); i++ )  
        G.setMark(i,NOVISITADO)  
  
    for ( int n = 0; i < G.n(); i++ )  
        if ( G.getMarca(i) == NOVISITADO ) {  
            procesarTopologico(G,i)  
        }  
}  
  
static void procesarTopologico(Graph G, int v) {  
    G.marcar(v, VISITADO)  
  
    for ( Edge w = G.first(v); G.isEdge(w); w=G.next(w) ){  
        if ( G.getMark(G.v2(w)) == NOVISITADO ) {  
            procesarTopologico(G, G.v2(w))  
        }  
  
        imprimir(v)  
    }  
}
```

Orden topológico

- Probemos el algoritmo anterior con el siguiente grafo, ¿qué se imprime?



Ejercicio 1

- Utilizando alguno de los algoritmos mencionados hasta aquí ajustarlos para que ahora pueda detectar un ciclo en un grafo. La idea es que dado un Grafo, el algoritmo retorne si tiene o no ciclo.

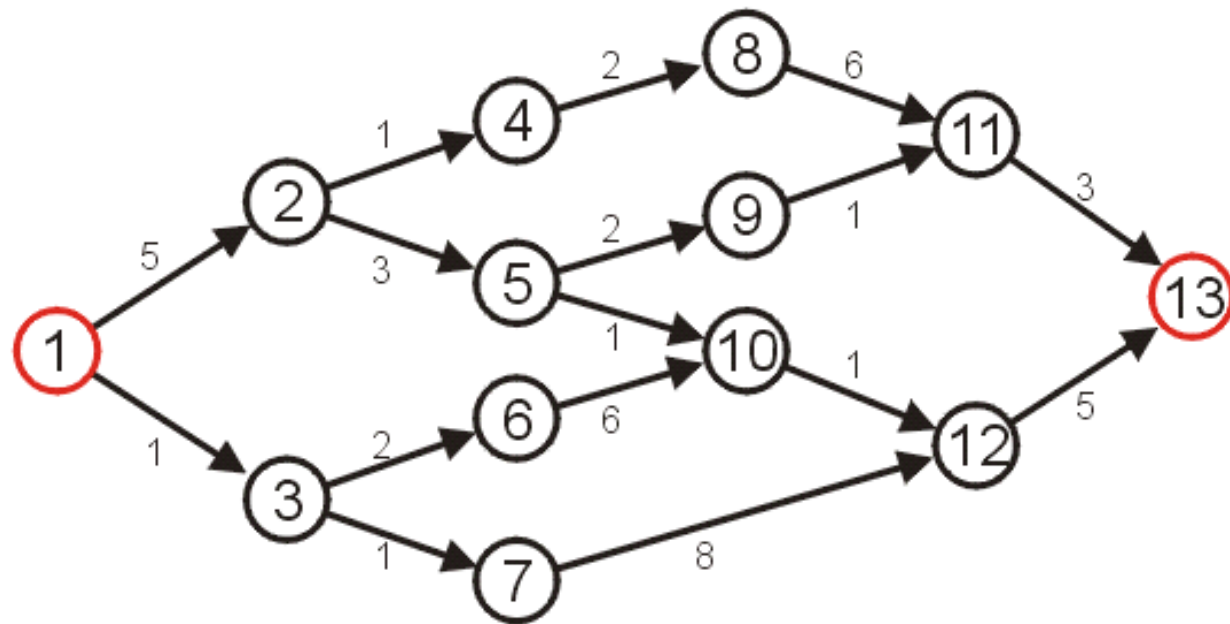
Considere para los casos de que sea un grafo dirigido y un grafo no-dirigido.

Camino más corto

- Para grafos sin pesos, un recorrido en amplitud (BFS) puede solucionar el problema del camino más corto entre un vértice y el resto.
- Dado un grafo con peso dirigido, BFS ya no dará la solución correcta. ***¿Porqué?***
- Recordar que en un grafo con peso, la longitud del camino es la suma de los pesos de cada arista en el camino.

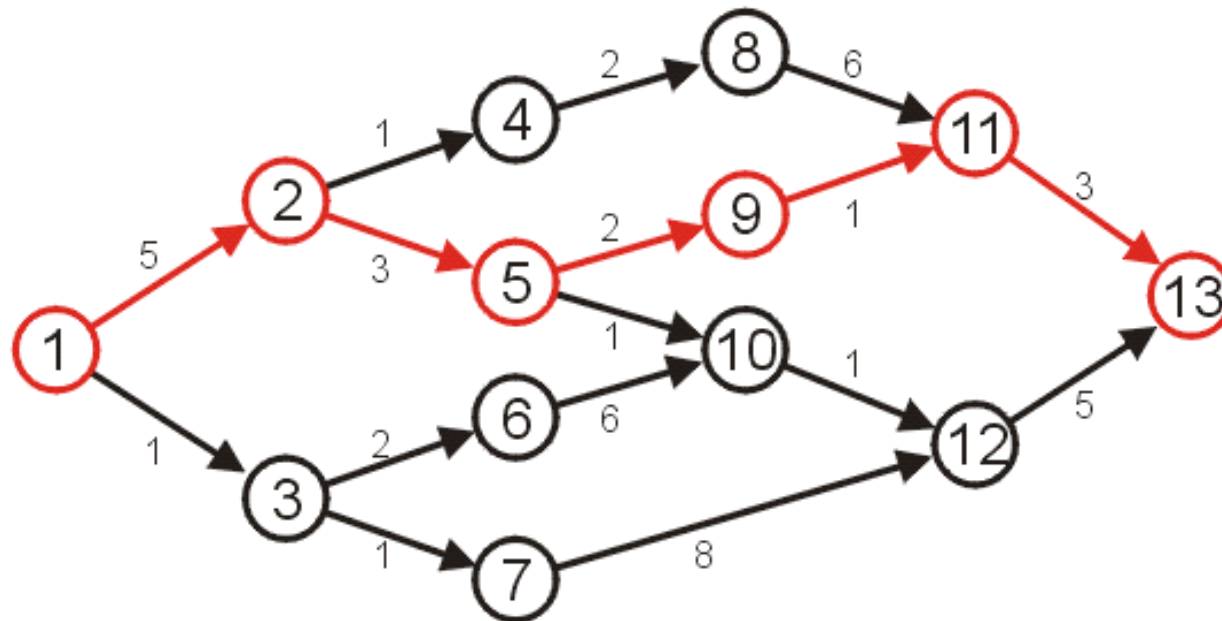
Camino más corto

- Suponga que desea encontrar el camino más corto en el grafo del ejemplo anterior, entre el vértice 1 y el 13



Camino más corto

- Después de revisar el grafo podemos determinar el camino más corto

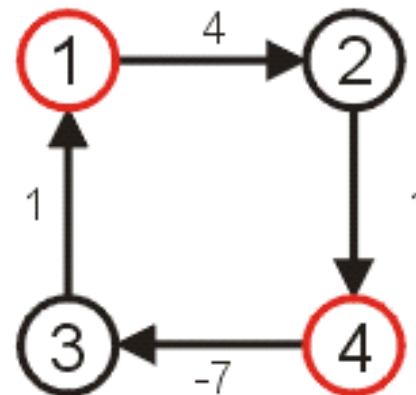


Camino más corto

- El objetivo de este algoritmo es encontrar el camino más corto y su longitud.
- Asumimos que los pesos de las aristas son números positivos

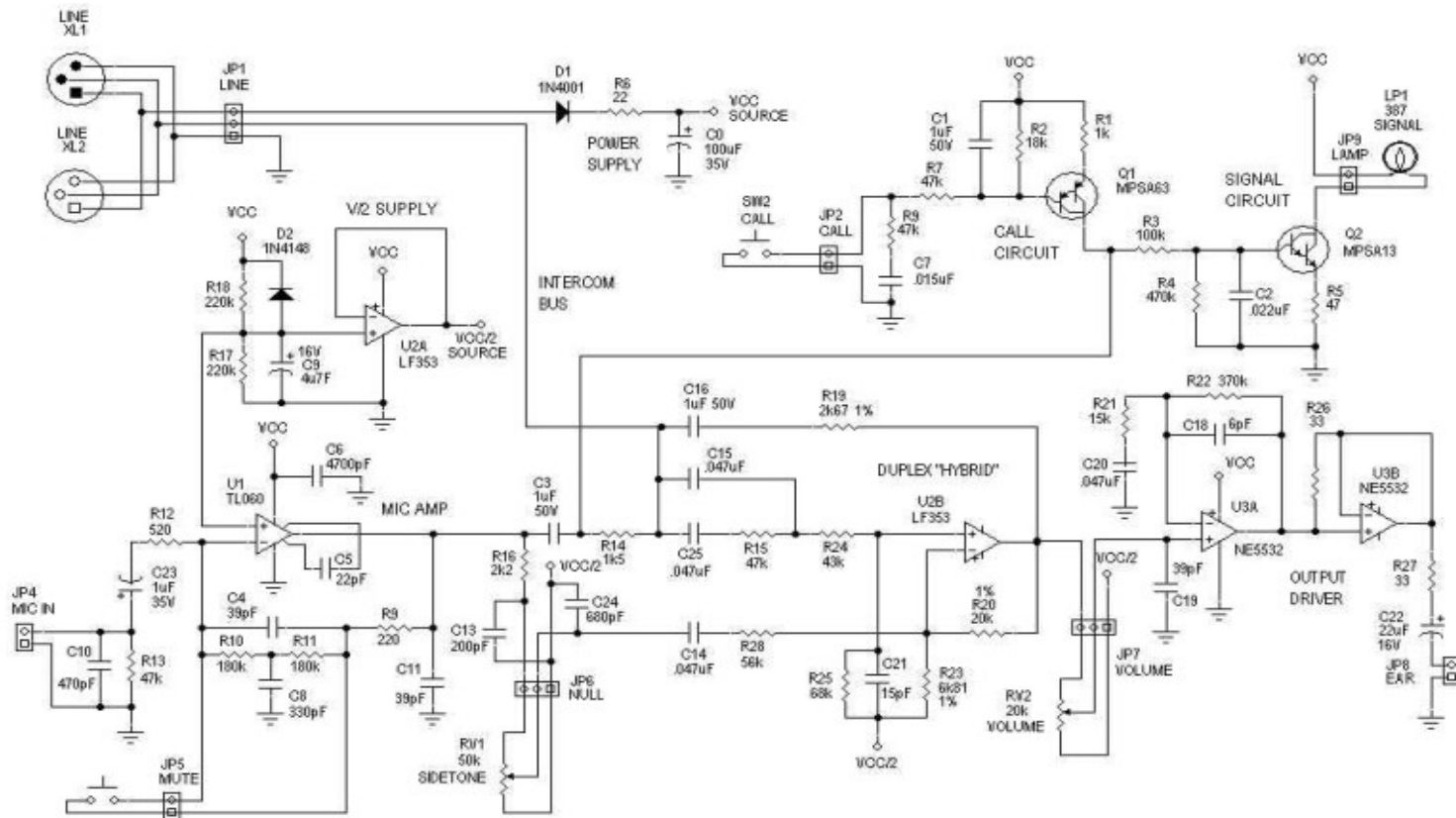
Camino más corto

- Ciclos negativos
 - Si hay aristas con costos negativos entonces es posible que en cada paso a través del ciclo disminuya el largo total del camino
 - Por tanto, el camino más corto estaría indefinido para estos grafos
 - Considere el camino más corto del vértice 1 al 4 en el siguiente grafo



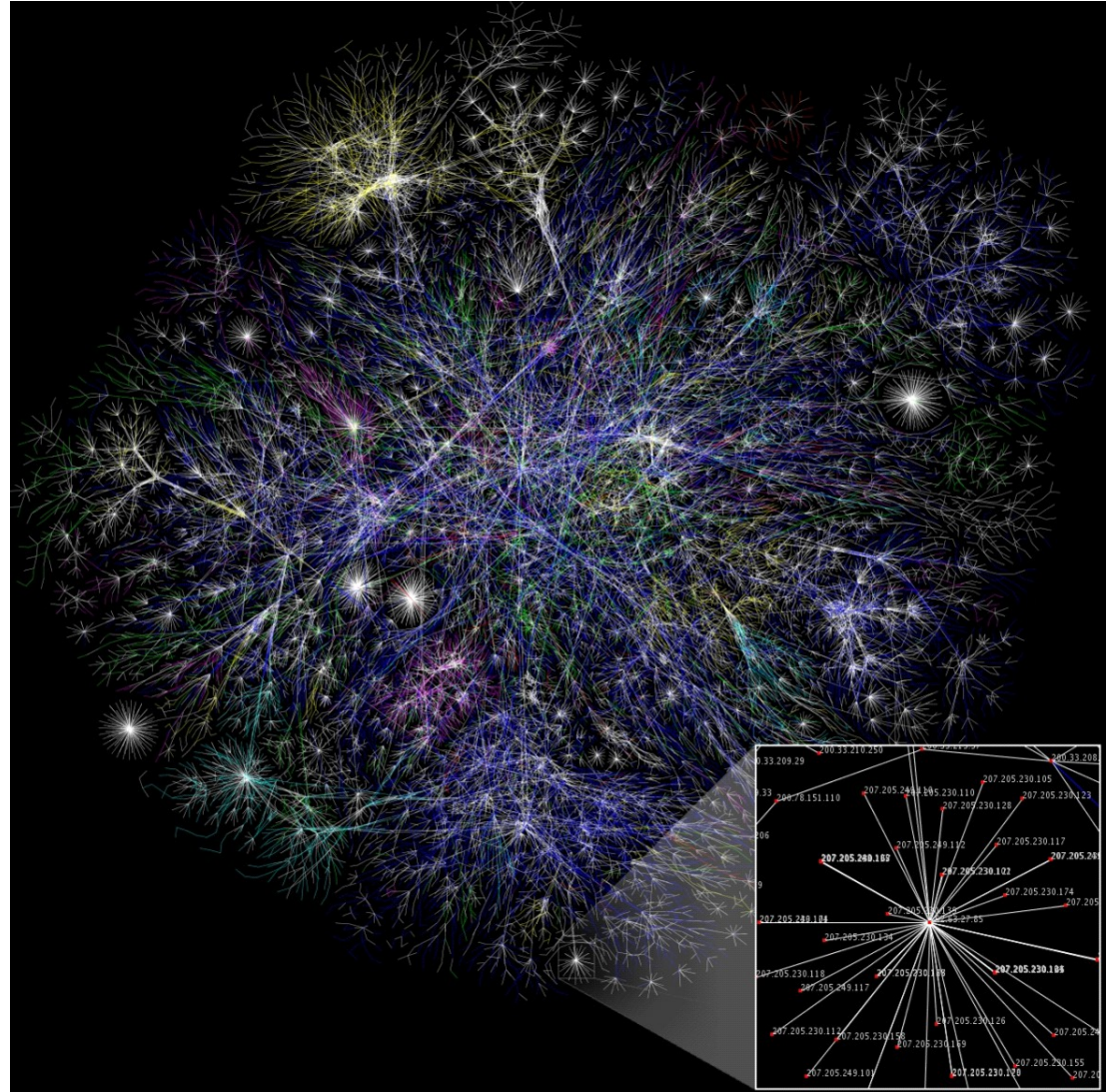
Aplicaciones del camino más corto

- Diseño de circuitos



Aplicaciones del camino más corto

- Internet y enrutamiento
- A la derecha un grafo de los enrutadores y sus conexiones en una porción de Internet



Aplicaciones del camino más corto


Google maps [Mostrar opciones de búsqueda](#)

Busca empresas, direcciones y lugares de interés.

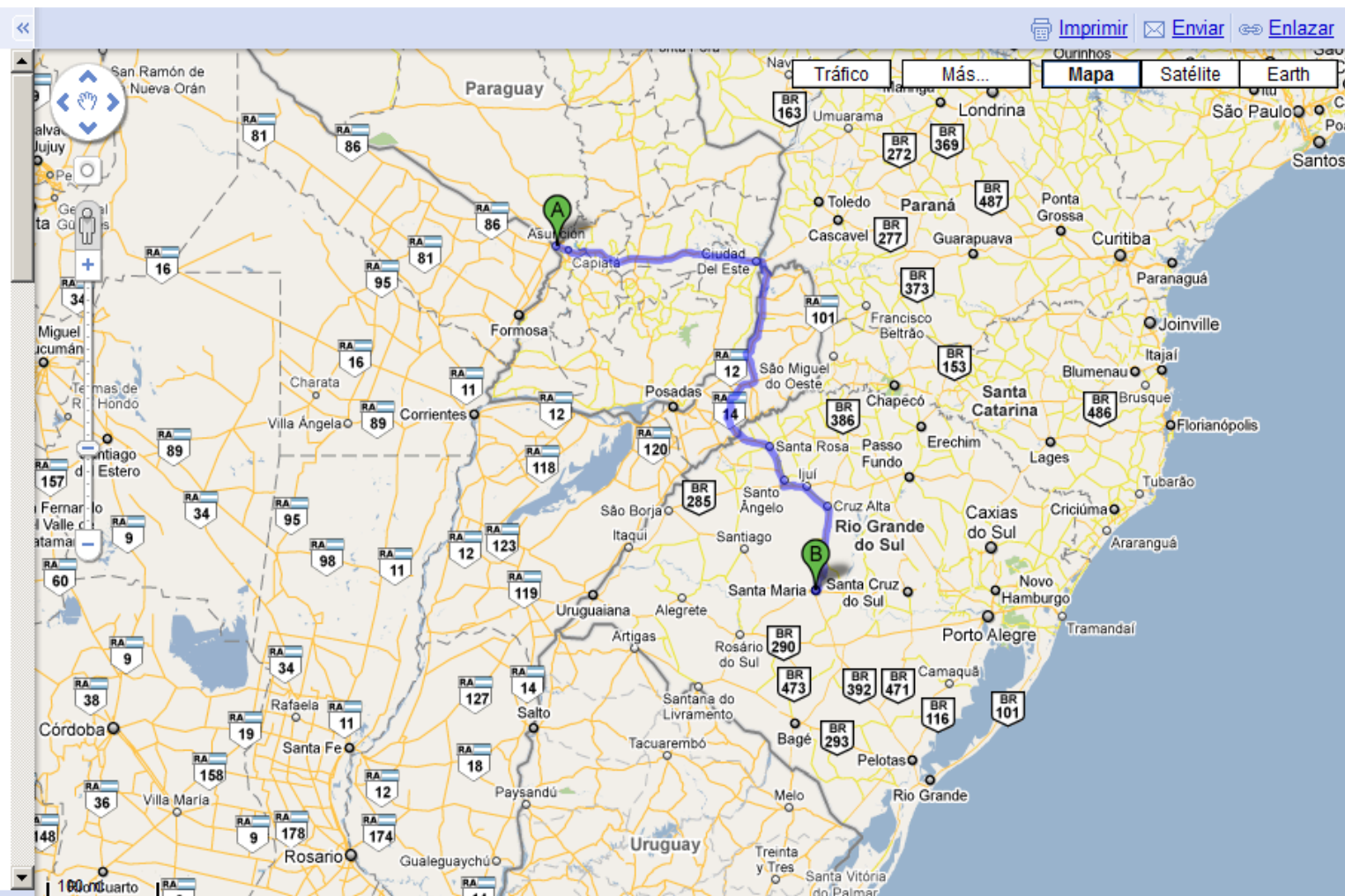
Cómo llegar [Mis mapas](#)

[Añadir destino](#) - [Mostrar opciones](#)

Indicaciones de ruta en coche para Santa María - RS, Brasil
970 km – aprox. 13h 0 min

 Asunción
Paraguay

1. Dirígete hacia el **nordeste** hacia **Sisa** 140 m
2. Toma la **2ª a la derecha** hasta **Sisa** 280 m
3. Gira a la **izquierda** en **Brasil** 1,2 km
4. Gira a la **derecha** en **Artigas** 13,4 km
5. Gira a la **izquierda** para continuar por **Artigas** 12,0 km
6. Continúa por **Carretera 2**. 107 km
7. Gira a la **derecha** hacia **Carretera 7** 86 m
8. Toma la **2ª a la derecha** hasta **Carretera 7** 165 km
9. Gira a la **derecha** hacia **Carretera 7** 110 m
10. Toma la **2ª a la derecha** hasta **Carretera 7** 18,7 km
11. Continúa recto por **Carretera 7** 550 m
12. Continúa recto por **Carretera 7** 9,4 km

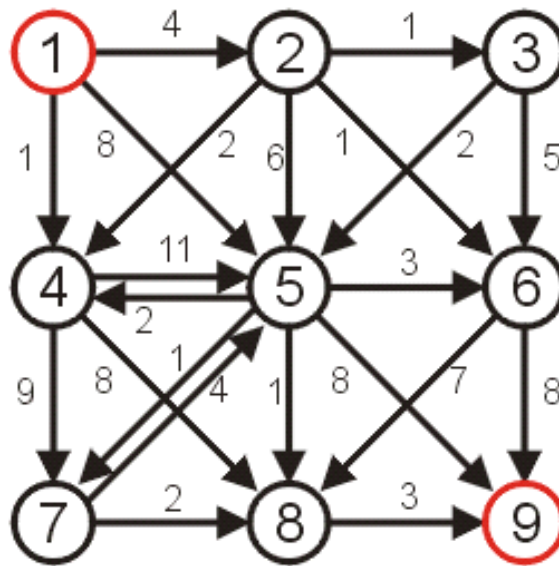


Camino más corto



Edsger W. Dijkstra
Turing award 1972

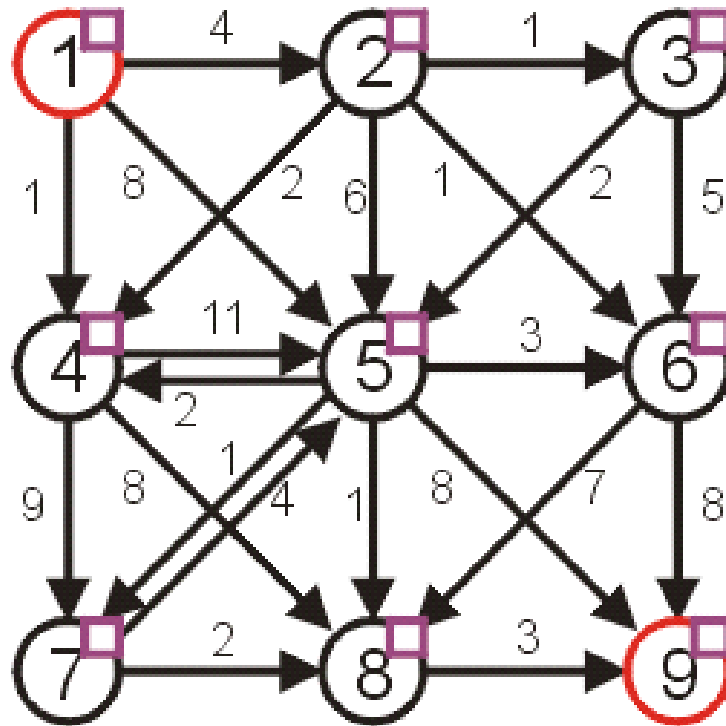
- Algoritmo de Dijkstra
 - Visitar todos los vértices al menos una vez
 - Necesitamos un arreglo de valores lógicos, inicialmente en Falso



1	F
2	F
3	F
4	F
5	F
6	F
7	F
8	F
9	F

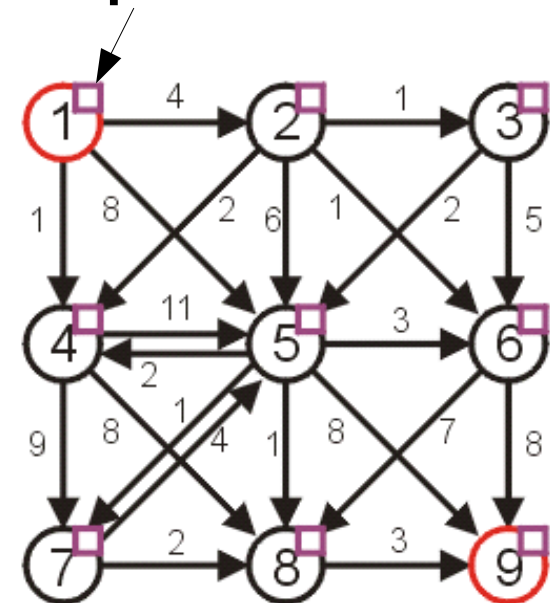
Dijkstra

- Gráficamente sería lo siguiente:



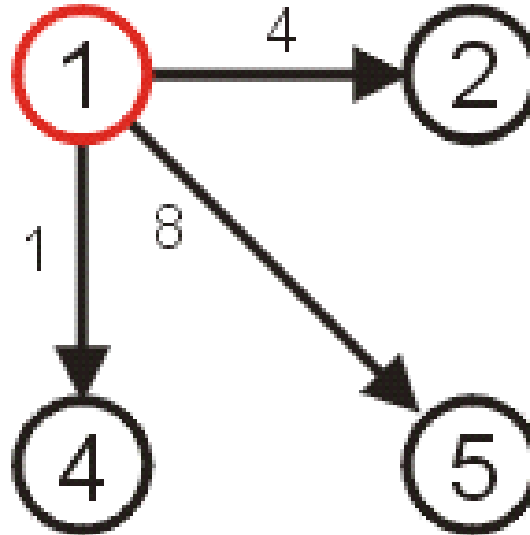
Dijkstra

- Comenzamos en el camino más corto posible y tratar de ir extendiendo el camino.
- Inicialmente, comenzamos por el camino de longitud 0 (desde donde queremos saber el camino)
 - Es 0, porque es el camino a si mismo

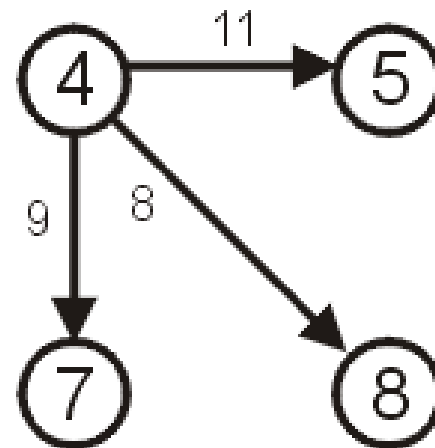


Dijkstra

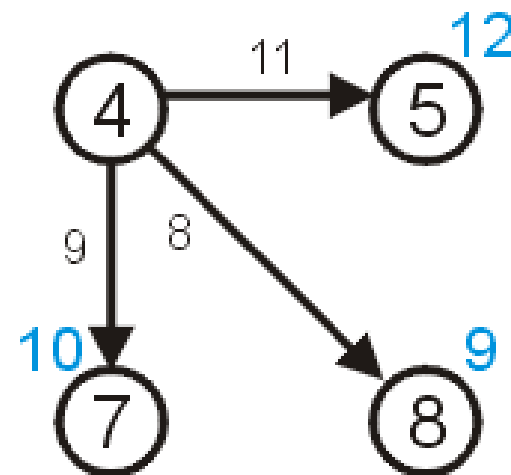
- Si tratamos de extender el camino, consideramos los siguientes caminos:
 - (1,2) long 4
 - (1,4) long 1
 - (1,5) long 8



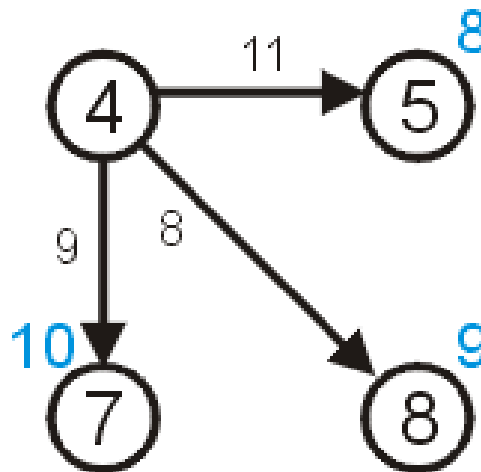
- El camino más corto es (1,4) de long. 1
- Por tanto, si ahora examinamos el vértice 4, deducimos los siguientes caminos
 - (1,4,5) long. 12
 - (1,4,7) long. 10
 - (1,4,8) long. 9



- Fijarse que no tenemos que conocer realmente que el camino a 4 fue (1,4), lo que necesitamos es saber que la longitud menor a 4 es 1.
- Por tanto, guardamos esta longitud
 - 5 de long. 12
 - 7 de long. 10
 - 8 de long. 9



- Nosotros conocemos que el camino es de longitud 8 al vértice 5 con camino (1,5).
- Por tanto, solo debemos guardar 8 en vez de 12.



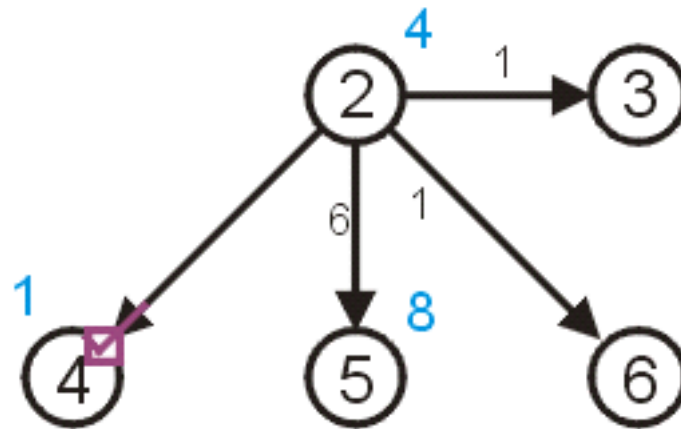
- Esto es, conocemos hasta aquí que existe el camino de 1 a los siguientes vértices

Vertex	Length
1	0
2	4
4	1
5	8
7	10
8	9

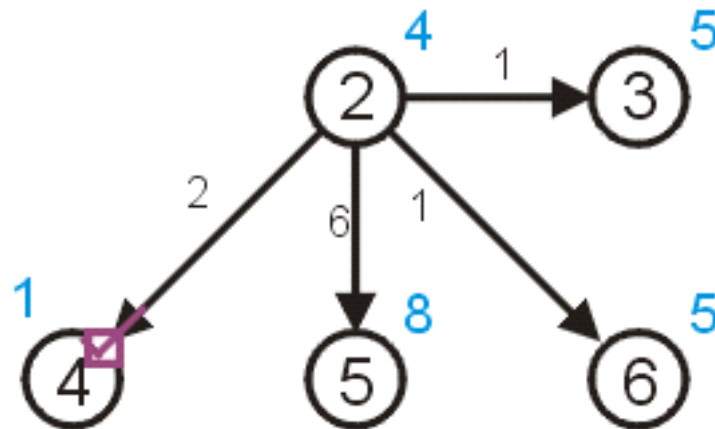
- Ver que no existe un camino más corto a los vértices 1 o 4.
- Consideramos que ya fueron visitados

Vertex	Length
1	0
2	4
4	1
5	8
7	10
8	9

- Siguiendo la estrategia, tomamos el siguiente vértice que tiene el camino más corto.
- Es el vértice 2



- Tratamos de actualizar el camino más corto del vértice 3 y 6 (ambos de long. 5) sin embargo:
 - Existe un camino de long $8 < 10$ al vértice 5 ($10=4+6$)
 - Y conocemos que el camino más corto a 4 es 1



- Para saber los vértices que todavía no alcanzamos, asignamos un valor de distancia inicial:
 - Infinito ∞
 - Un número grande
 - Un valor negativo

Usamos ∞

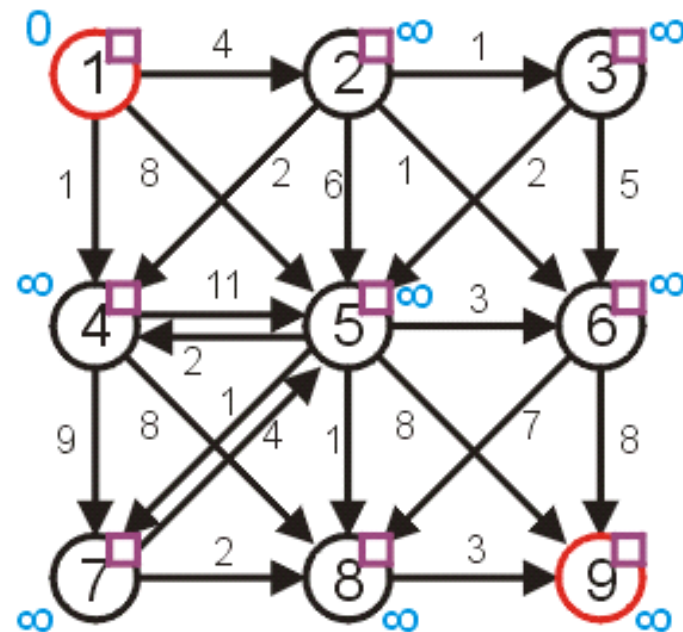
- Si la distancia es ∞ entonces no existe un vértice precedente
- De otra forma, existe un vértice precedente.

- Por tanto, inicializamos:
 - Colocar 0 al vértice inicial
 - ∞ al resto
 - Todos se marcan como no visitados

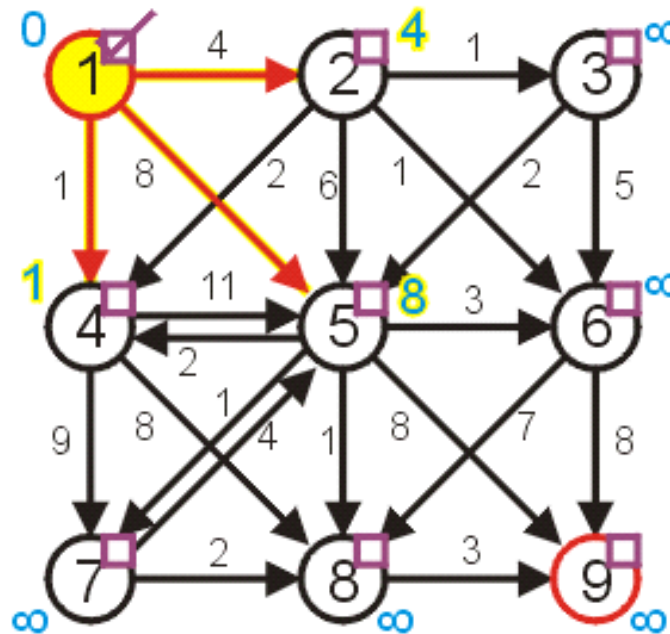
- Entonces, iteramos:
 - Encontrar el vértice no visitado
 - Marcar como visitado
 - Por cada no visitado del vértice adyacente al vértice actual:
 - Agregar la distancia al vértice actual
 - Si esta es menor a la distancia actual, actualizar y colocar como vértice padre del vértice adyacente al vértice actual.

- Condición de parada:
 - Nosotros paramos cuando el vértice que hemos visitado es el último vértice.
 - Si en algún punto, todos los no visitados vértices tienen distancia ∞ , entonces no existe camino desde del vértice inicial al vértice final.

- Ejemplo completo

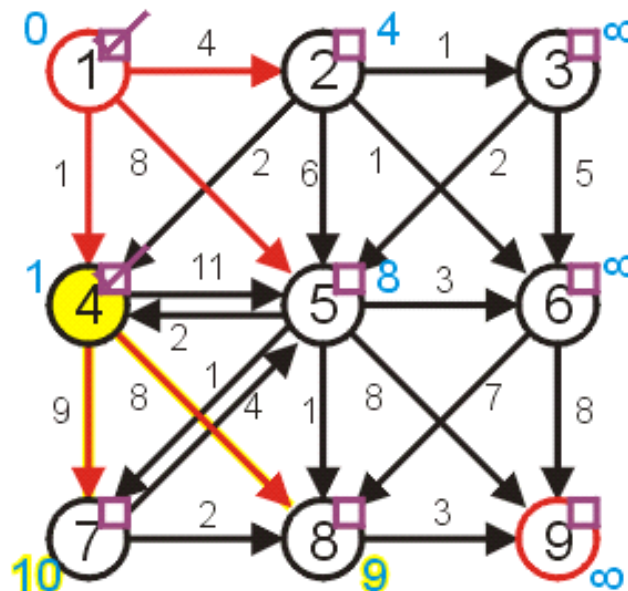


- Visitamos el vértice 1 y actualizamos los vecinos, marcamos como visitado
 - El camino más corto a 2, 4 y 5 son actualizados



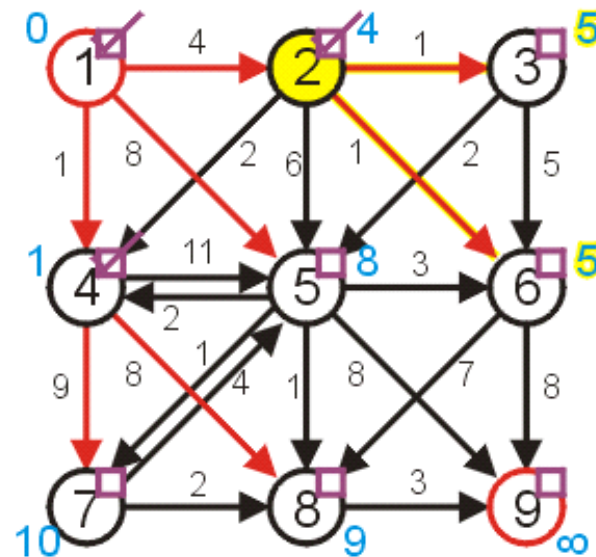
- El siguiente vértice visitado es 4

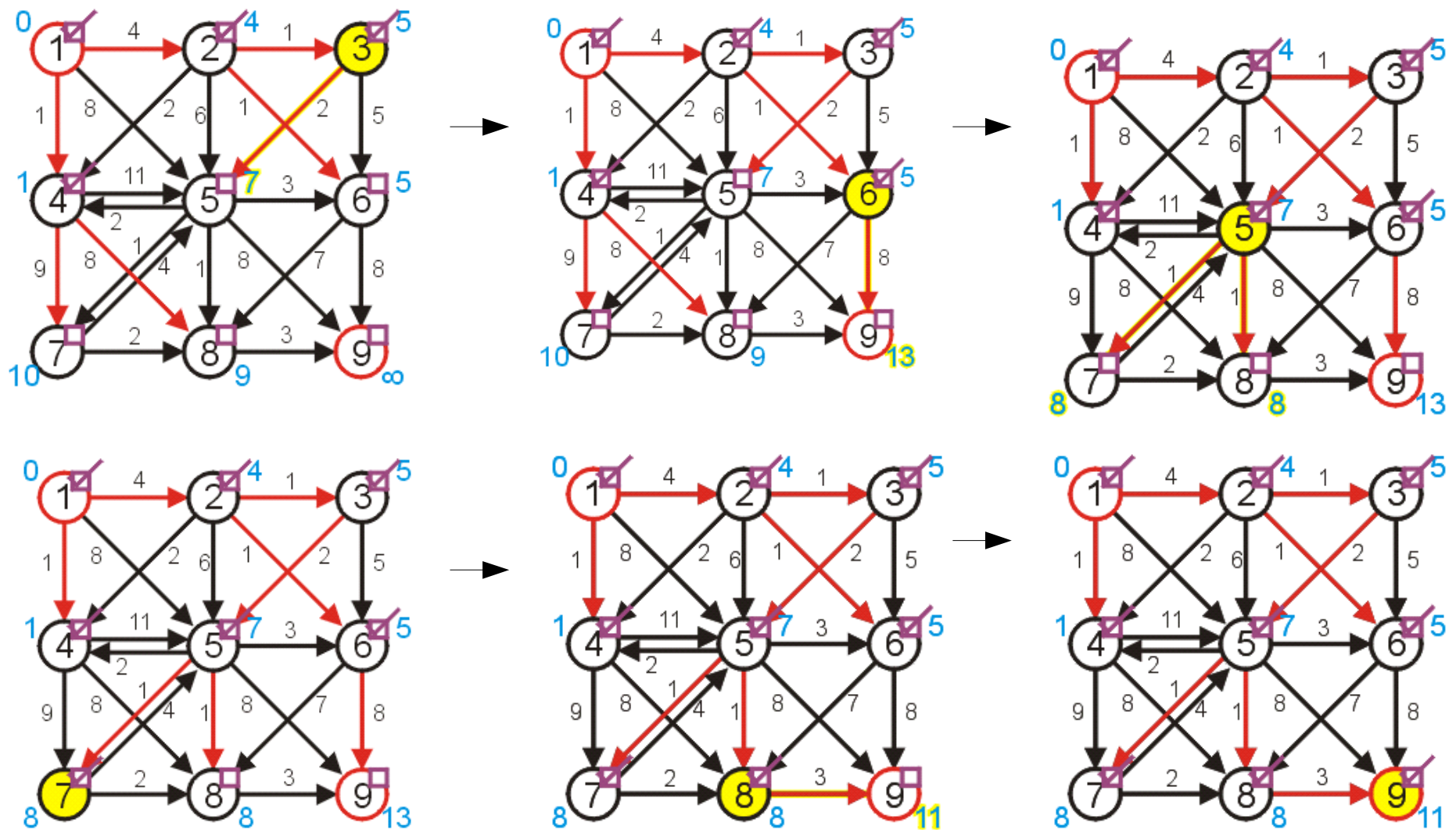
- Vertice 5 $1 + 11 \geq 8$ no actualizamos
- Vértice 7 $1 + 9 < \infty$ actualizamos
- Vértice 8 $1 + 8 < \infty$ actualizamos



- Ahora, visitamos el vértice 2

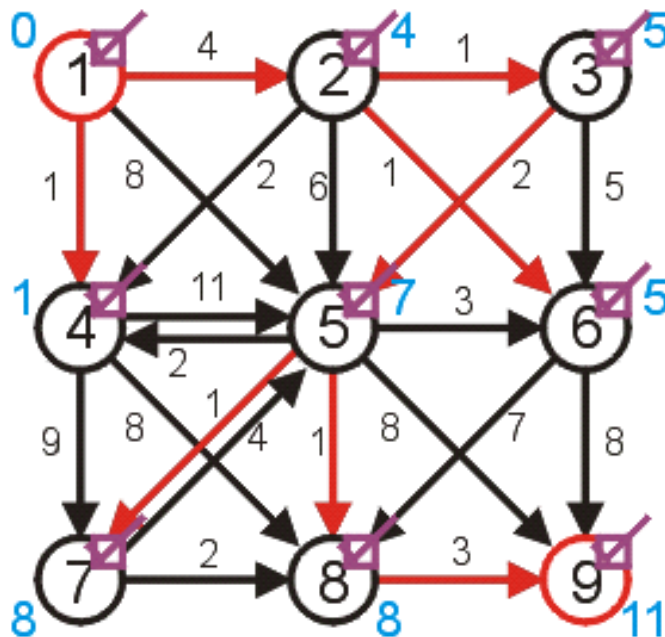
- Vértice 3 $4 + 1 < \infty$ actualizamos
- Vértice 4 ya visitado
- Vértice 5 $4 + 6 \geq 8$ no actualizamos
- Vértice 6 $4 + 1 < \infty$ actualizamos





Finalmente el camino más corto es

- 9,8,5,3,2,1



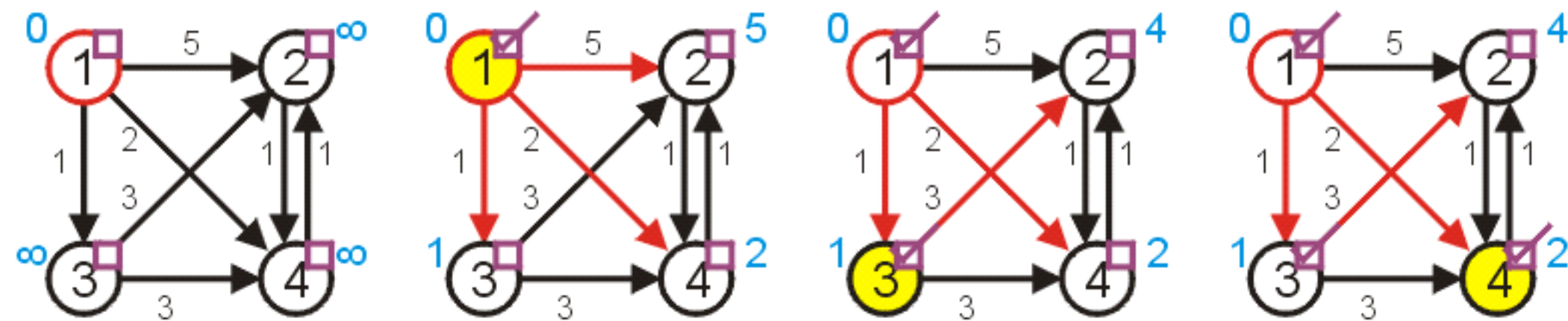
¿Qué información se ve en esta columna?



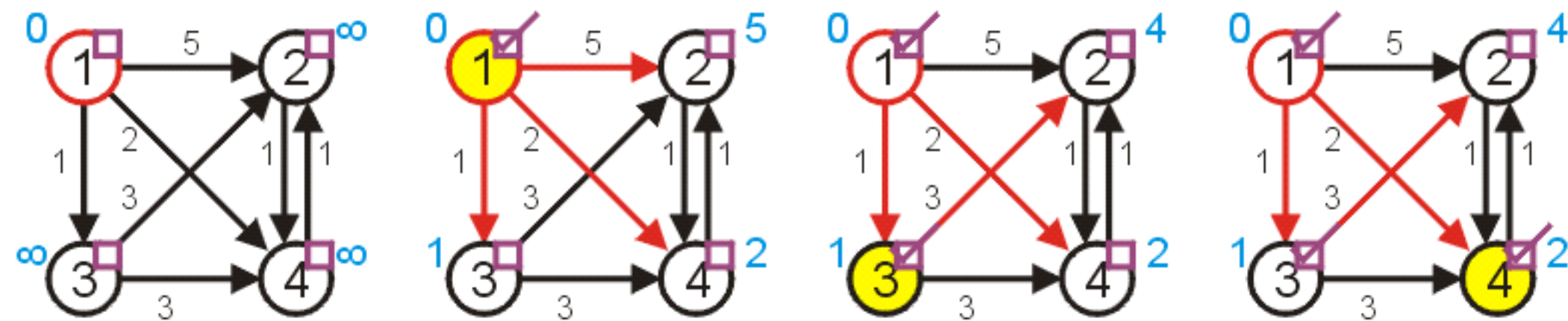
Vértice	Long.	Origen
1	0	-
2	4	1
3	5	2
4	1	1
5	7	3
6	5	2
7	8	5
8	8	5
9	11	8

- En el ejemplo nosotros visitamos todos los vértices
- No siempre ocurre esto, considere el siguiente ejemplo

- Encontrar el camino entre 1 y 4
 - El camino más corto se encuentra luego que solo 3 vértices se hayan visitado
 - El algoritmo termina cuando alcanzamos el vértice 4
 - Solo tenemos información en los vértices 1,3 y 4
 - El vértice 2 no tiene información de camino



- Encontrar el camino entre 1 y 4
 - El camino más corto se encuentra luego que solo 3 vértices se hayan visitado
 - El algoritmo termina cuando alcanzamos el vértice 4
 - Solo tenemos información en los vértices 1,3 y 4
 - El vértice 2 no tiene información de camino



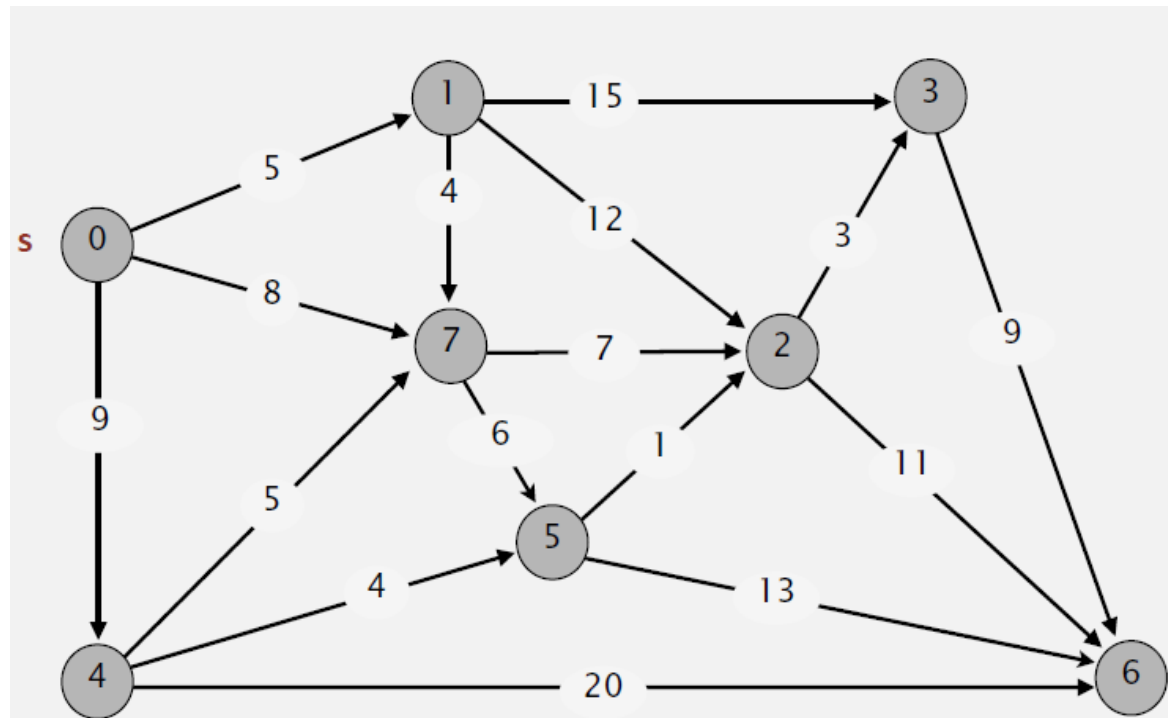
Algoritmo de Dijkstra

```
static void Dijkstra(Graph G, int s, int[] D) {  
    for (int i=0; i<G.n(); i++) // Inicializar  
        D[i] = Integer.MAX VALUE;  
    D[s] = 0;  
    for (int i=0; i<G.n(); i++) { // Procesar los vertices  
        int v = minVertex(G, D); // Encontrar el vertice más cercano  
        G.setMark(v, VISITED);  
        if (D[v] == Integer.MAX VALUE) return; // Inalcanzable  
        for (int w = G.first(v); w < G.n(); w = G.next(v, w))  
            if (D[w] > (D[v] + G.weight(v, w)))  
                D[w] = D[v] + G.weight(v, w);  
    }  
}  
  
static int minVertex(Graph G, int[] D) {  
    int v = 0; // Inicializar v con cualquier vertice no visitado  
    for (int i=0; i<G.n(); i++)  
        if (G.getMark(i) == UNVISITED) { v = i; break; }  
  
    for (int i=0; i<G.n(); i++) // Encontrar el menor valor  
        if ((G.getMark(i) == UNVISITED) && (D[i] < D[v]))  
            v = i;  
    return v;  
}
```

¿Cuál es el costo de este algoritmo?

Considere un DAG:

- El algoritmo de orden topológico puede ser usado para calcular el camino más corto.
- Encuentre el camino más corto para el DAG de abajo.



```

public class AcyclicSP
{
    private DirectedEdge[] edgeTo;
    private double[] distTo;

    public AcyclicSP(EdgeWeightedDigraph G, int s)
    {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];

        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;

        Topological topological = new Topological(G);
        for (int v : topological.order())
            for (DirectedEdge e : G.adj(v))
                relax(e);
    }
}

```

```

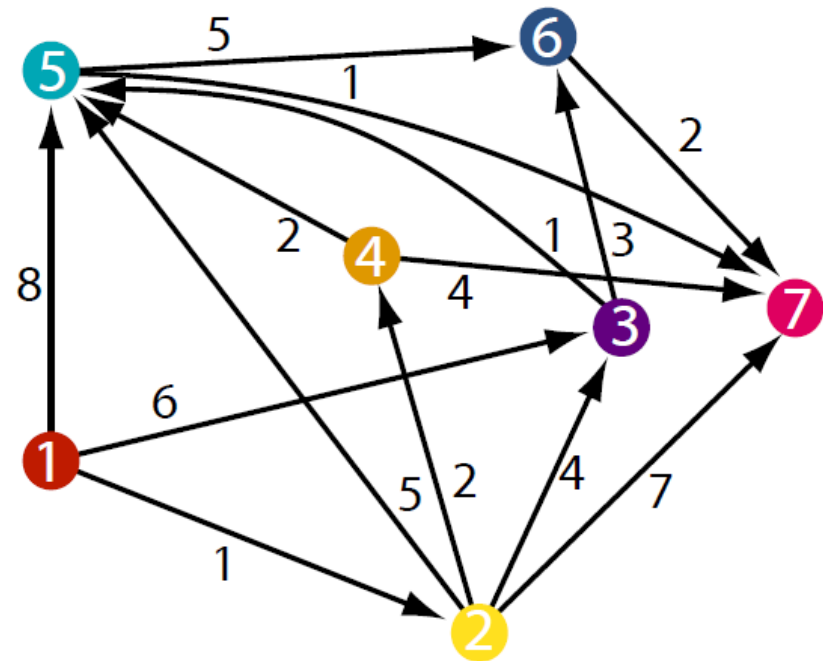
private void relax(DirectedEdge e)
{
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight())
    {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
    }
}

```

Orden topológico

Ejercicio 2

- Proponga el pseudocódigo para el algoritmo de Dijkstra.
- Utilizando su algoritmo resuelva para el siguiente grafo

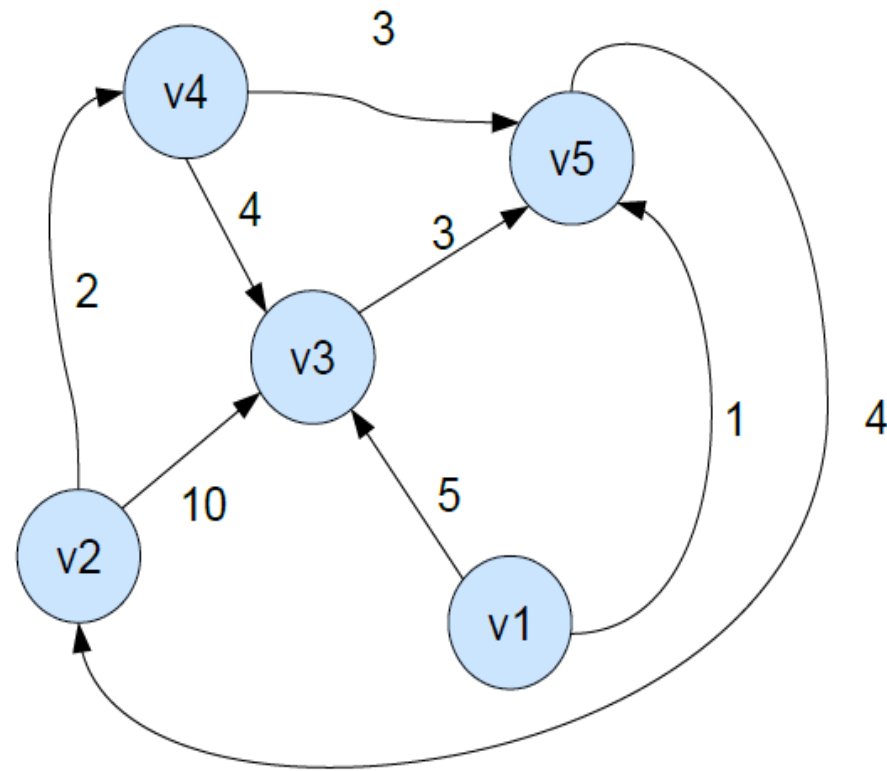


Ejercicio 3

Dado un grafo dirigido G ponderado, diseñe un algoritmo que calcule el diámetro de G . Este corresponde a la máxima excentricidad de cualquier vértice. La excentricidad de un vértice es la longitud del camino más corto desde ese vértice al más alejado de él.

Su algoritmo debe indicar si el grafo esta conectado o no. En caso de que no este conectado el diámetro es infinito.

Ejemplo para ejercicio 3



Excentricidad para v1	
<u>Distancia Mínima</u>	<u>Valor</u>
(v1, v2)	5
(v1, v3)	5
(v1, v4)	7
(v1, v5)	1

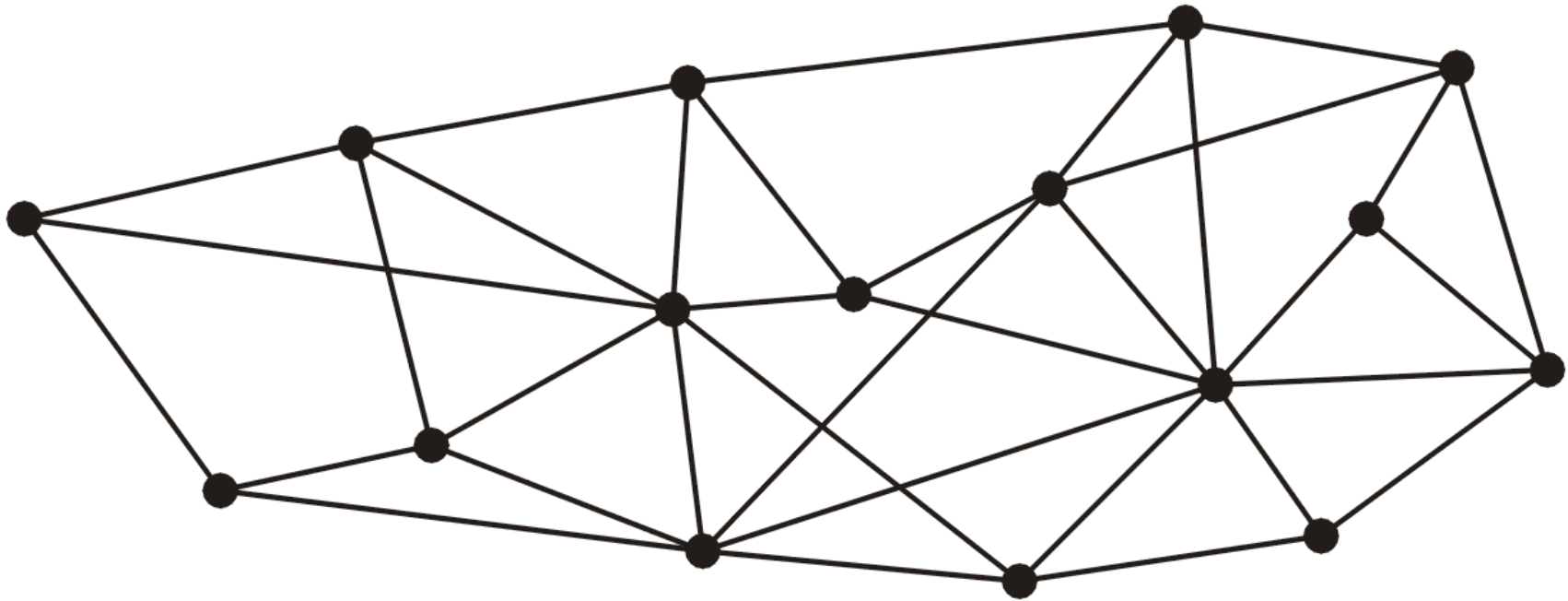
En este ejemplo, la excentricidad de v1 es 7 y el vértice más alejado v4.

Árbol de recubrimiento mínimo (Minimum Spanning Tree)

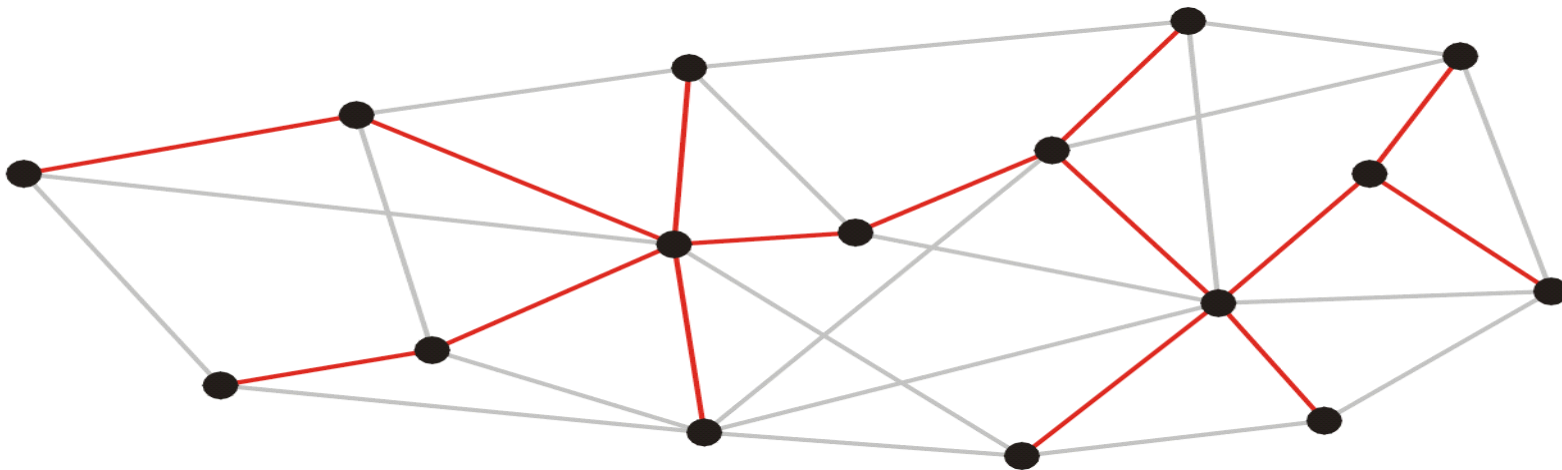
- Dado un grafo conexo con $|V|$ vértices, un *árbol de recubrimiento* es definido como una colección de $n-1$ aristas que conectan todos los vértices.
- No es necesariamente único

MST

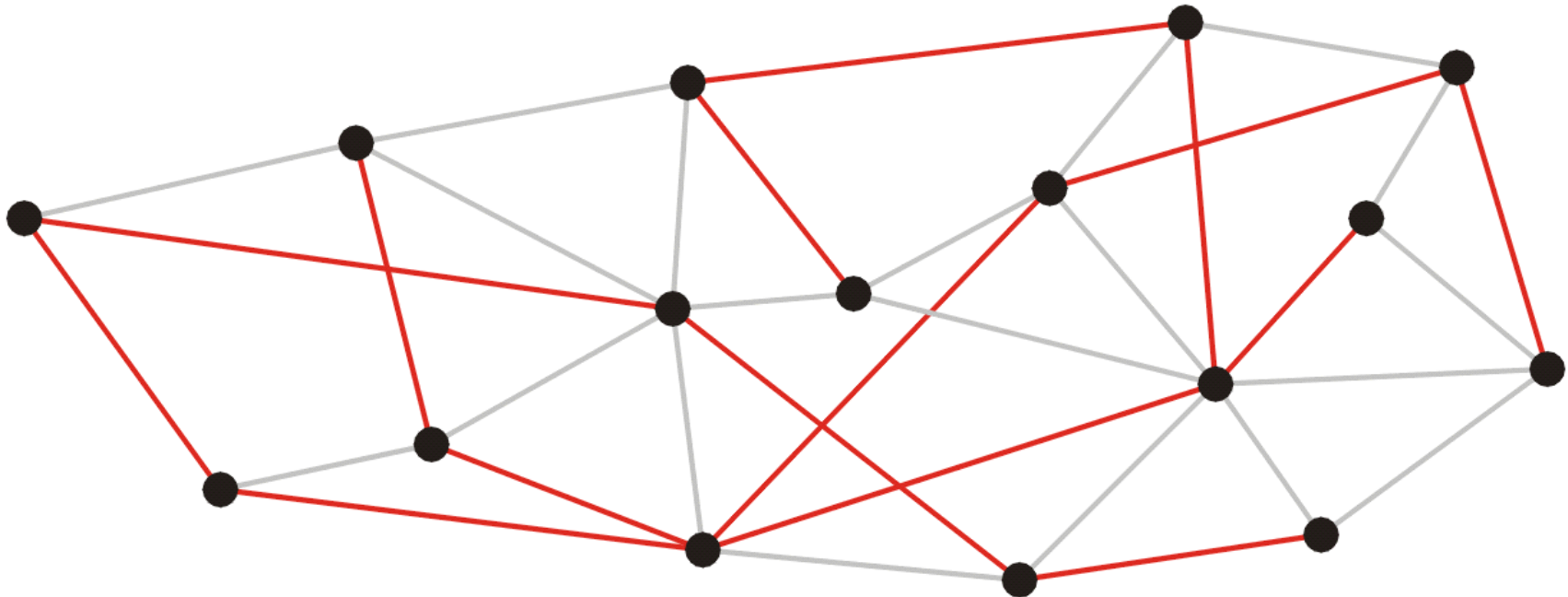
- Este grafo tiene 16 vértices y 35 aristas



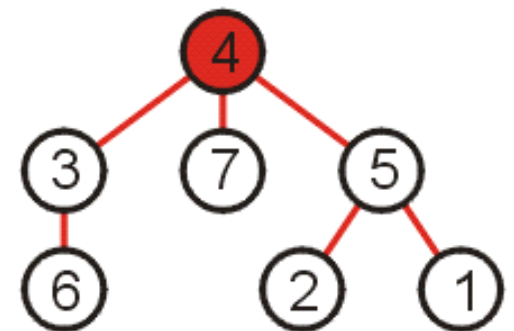
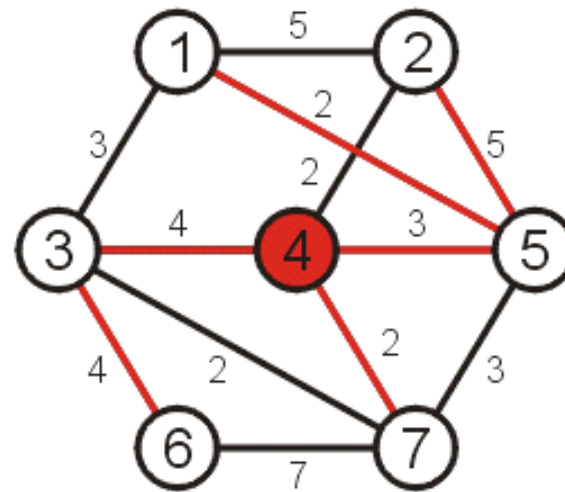
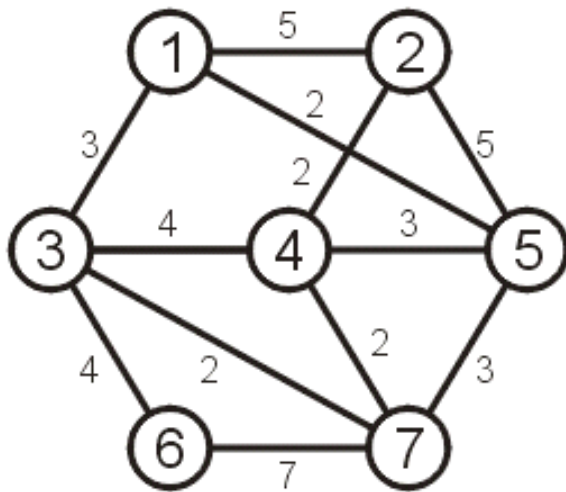
- Estas 15 aristas forman un árbol de recubrimiento



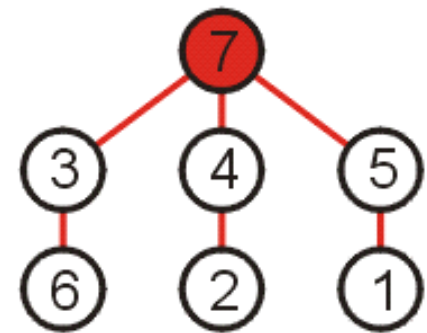
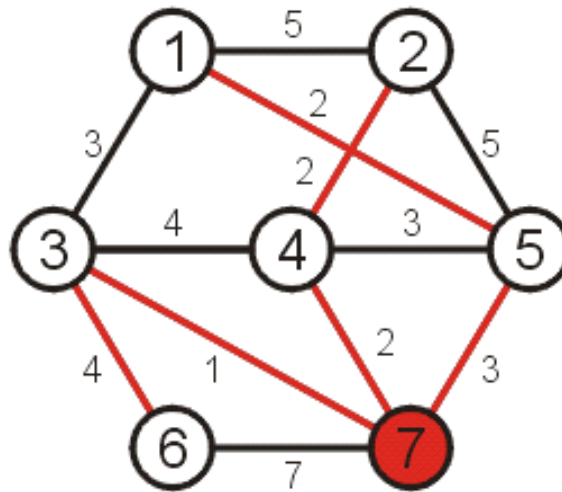
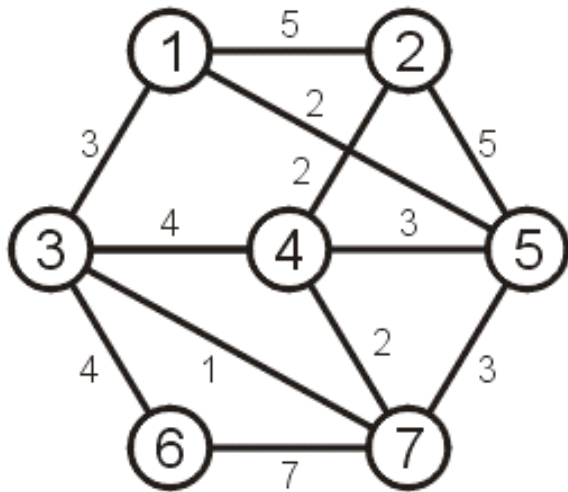
- Estas también



- El peso de un árbol de recubrimiento es la suma de los pesos de todas las aristas que lo conforman.
- Ejemplo: la suma es 20.

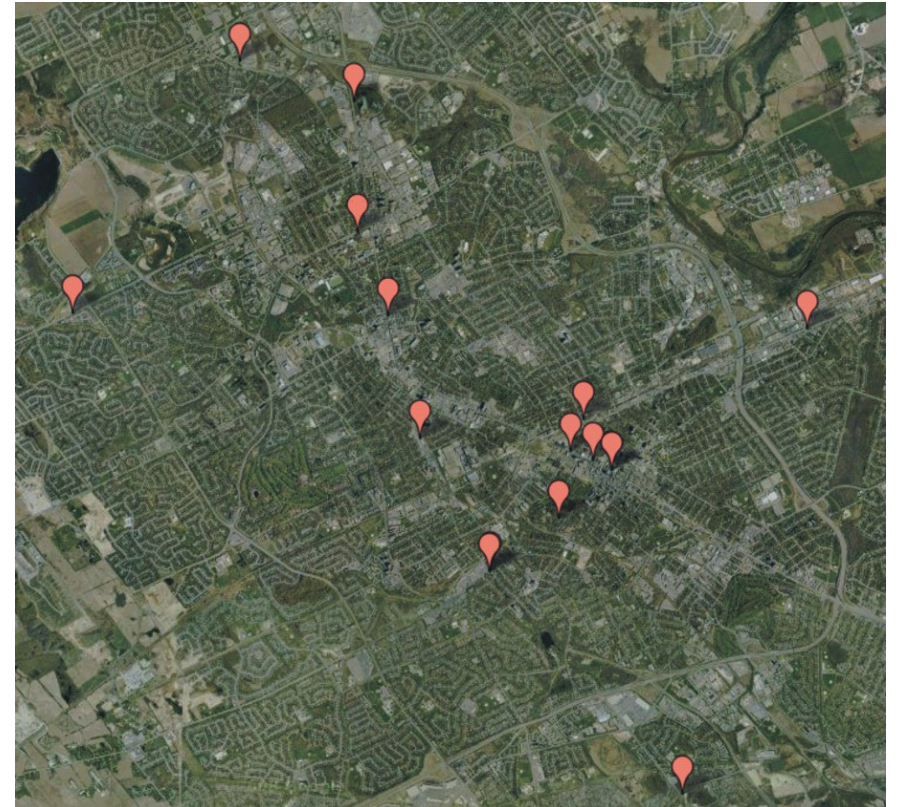


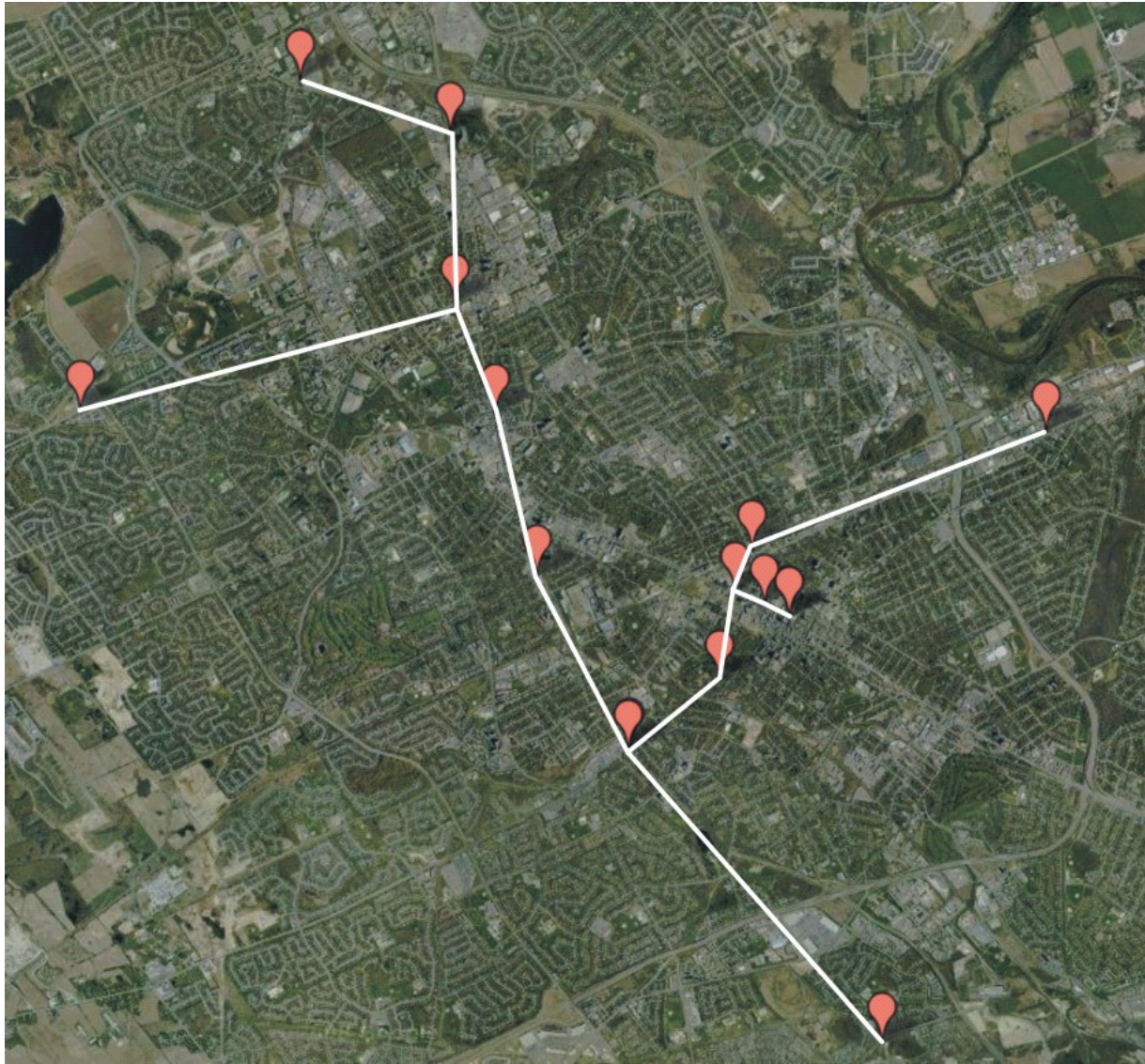
- ¿Que árbol de recubrimiento minimiza el peso?
 - Este es el MST (árbol de recubrimiento mínimo)
- El peso es 14 para este ejemplo



Aplicación de MST

- Considere como encontrar la mejor forma de conectar un número de redes LAN
 - Minimizando el número de puentes
 - Los costos no dependen exactamente solo de la distancia
 - El MST provee la solución





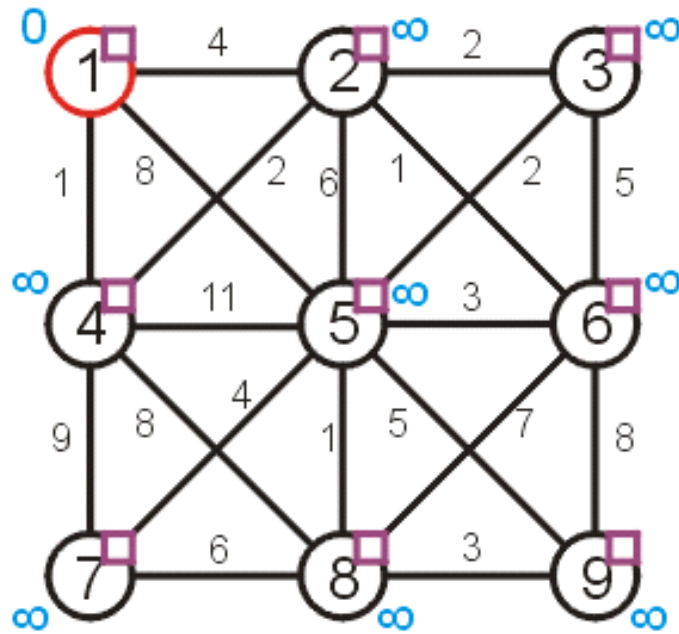
Algoritmos

- Prim
- Kruskal
- Borůvka (no veremos)

Algoritmo de Prim

- Iniciar en un vértice cualquiera
- Expandir agregando los vértices más “baratos” que todavía no están en el MST.

Algoritmo de Prim



		Distance	Parent
1	F	0	0
2	F	∞	0
3	F	∞	0
4	F	∞	0
5	F	∞	0
6	F	∞	0
7	F	∞	0
8	F	∞	0
9	F	∞	0

Visitamos el vértice **1**, y actualizamos el **2, 4 y 5**.

El siguiente vértice visitado es **4** (porque tiene menor costo o distancia).

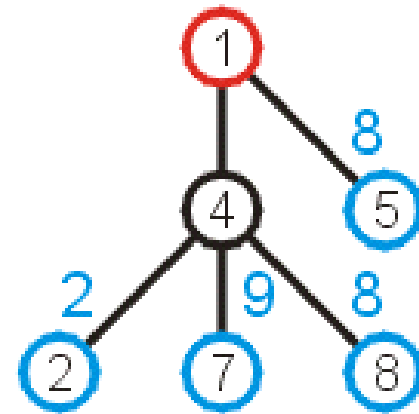
Actualizamos el 2,7,8

No actualizamos el 5

- Ahora que hemos actualizado todos los vértices adyacentes al vértice 4, podemos extender el árbol agregando una de las siguientes aristas

(1,5) (4,2) (4,7) o (4,8)

Agregamos el de menor costo (4,2)



El siguiente visitado es el 2
Actualizamos el 3, 5 y 6

Luego, visitamos el 6
Actualizamos el 5, 8 y 9

Ahora visitamos el 3 y actualizamos el
5

Visitamos el 5 y actualizamos el
7, 8 y 9.

Visitamos el 8 y solo actualizamos el
9

Al visitar el 9 no hay vértices que actualizar

Al visitar el 7 tampoco hay vértices a actualizar

- En este punto no hay más vértices visitados y por tanto al algoritmo termina.
- Si existe vértices con distancia ∞ entonces indica que el grafo no está conectado
 - En ese caso el MST solo cubre un subgrafo conectado.

- Usando el puntero a los padres, construimos el MST

MST - Prim

- Algoritmo voraz (greedy)
- Tiempo de ejecución
 $O(|E| \ln |V|)$


```

static void Prim(Graph G, int s, int[] D) {
    int[] V = new int[G.n()]; // V[i] guarda el vertice mas cercano a i
    for (int i=0; i<G.n(); i++) // Inicializar
        D[i] = Integer.MAX_VALUE;
    D[s] = 0;
    for (int i=0; i<G.n(); i++) { // Procesar los vertices
        int v = minVertex(G, D);
        G.setMark(v, VISITED);
        if (v != s) AddEdgetoMST(V[v], v); // Agrego al árbol
        if (D[v] == Integer.MAX_VALUE) return; // Vertice inalcanzable
        for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
            if (D[G.v2(w)] > G.weight(w)) {
                D[G.v2(w)] = G.weight(w);
                V[G.v2(w)] = v;
            }
    }
}

```

```

static int minVertex(Graph G, int[] D) {
    int v = 0; // Inicializar v con cualquier vertice no visitado;
    for (int i=0; i<G.n(); i++)
        if (G.getMark(i) == UNVISITED) { v = i; break; }
    for (int i=0; i<G.n(); i++) // Encontrar el menor valor
        if ((G.getMark(i) == UNVISITED) && (D[i] < D[v]))
            v = i;
    return v;
}

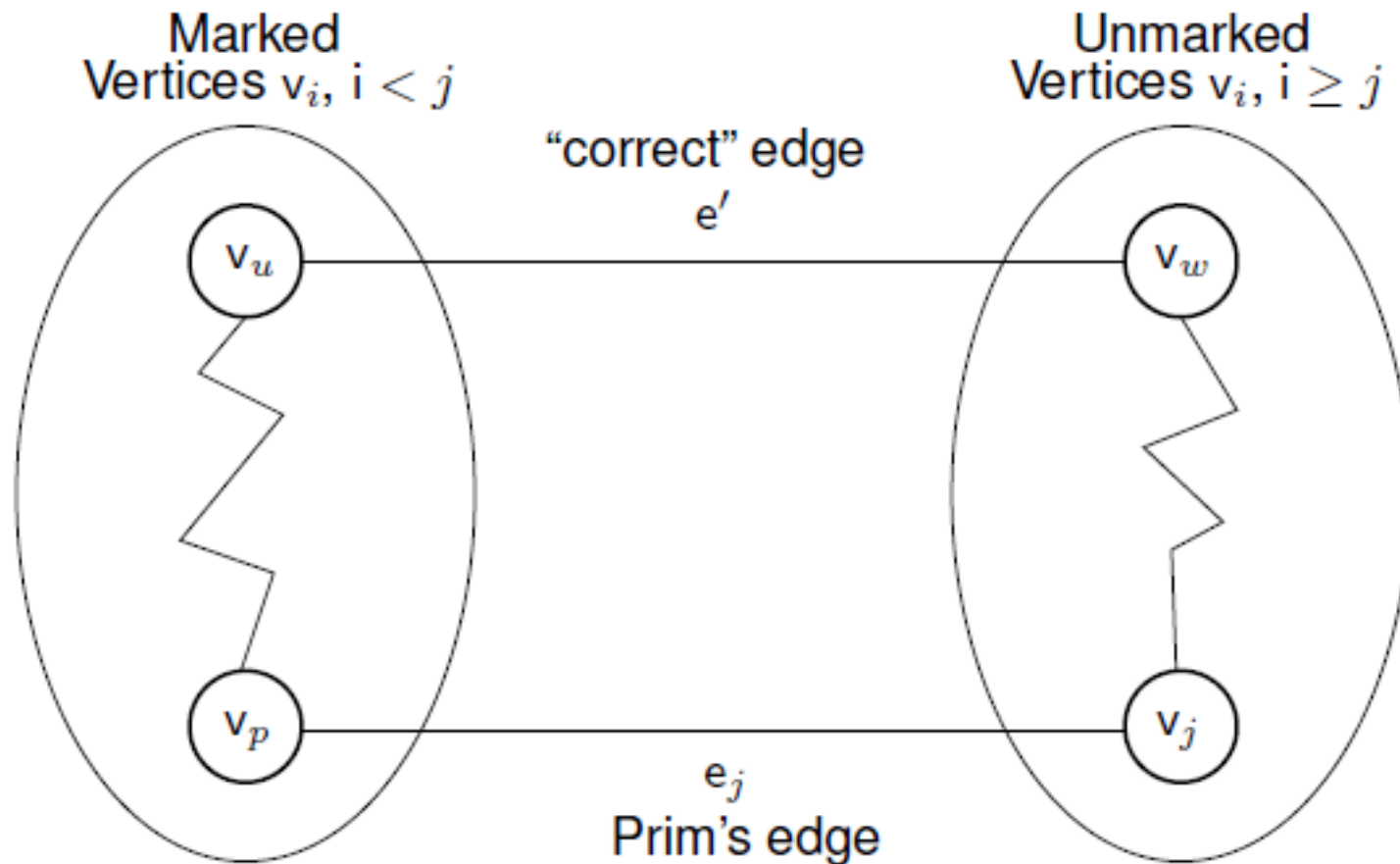
```

Breve demostración de la optimalidad del Algoritmo de Prim (tomada del libro de Shaffer)

Dado $G=(V,E)$ ser un grafo para el cual el algoritmo de Prim (AP) **no** genera un MST. Definimos un orden de vértices de acuerdo al orden en el que fueron agregados por el AP al MST: v_0, v_1, \dots, v_{n-1} . Dada la arista e_i que conecta (v_x, v_i) para algún $x < i$ y $i \geq 1$. Dado e_j ser la primera arista agregada por el AP tal que el conjunto de aristas seleccionada NO puede ser extendida para formar un MST para G . Es decir, es la primera arista donde el AP "falla". Dado \mathbf{T} ser el MST verdadero. Llamamos v_p ($p < j$) el vértice conectado por la arista e_j , esto es, $e_j = (v_p, v_j)$.

Ya que \mathbf{T} es un árbol, existe algún camino en \mathbf{T} que conecta v_p y v_j . Debe por tanto existir alguna arista e' en el camino conectando v_u y v_w , con $u < j$ y $w \geq j$. Ya que e_j no es parte de \mathbf{T} , agregar la arista e_j a \mathbf{T} forma un ciclo. La arista e' debe ser de menor costo que e_j ya que el AP no genera el MST. Sin embargo, AP debe seleccionar la arista de menor costo posible. Pero debió haber seleccionado e' , no e_j . Esto es, UNA CONTRADICCIÓN que el AP pudo haber seleccionado una arista errónea, por tanto el AP debe ser correcto.

- Prueba del algoritmo de PRIM (Shaffer)



MST - Algoritmo de Kruskal

- Es también de tipo Greedy (voraz).
- Idea:
 - Partimos el conjunto de vértices en $|V|$ clases equivalentes, cada una consistente de 1 vértice
 - Procesamos las aristas en orden de peso.
 - Una arista es agregada al MST y dos clases equivalentes son combinadas, si la arista conecta dos vértices en diferentes clases.
 - Este proceso es repetido hasta que se tenga una sola clase equivalente.

Kruskal

- Dado el siguiente grafo

Algoritmo Kruskal (CLRS)

MST-KRUSKAL(G, w)

```
1   $A \leftarrow \emptyset$ 
2  for each vertex  $v \in V[G]$ 
3      do MAKE-SET( $v$ )
4  sort the edges of  $E$  into nondecreasing order by weight  $w$ 
5  for each edge  $(u, v) \in E$ , taken in nondecreasing order by weight
6      do if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
7          then  $A \leftarrow A \cup \{(u, v)\}$ 
8              UNION( $u, v$ )
9  return  $A$ 
```

Basado en la Estructura de Datos Conjuntos disjuntos (disjoint-set). Esta estructura mantiene una colección $S = \{S_1, S_2, \dots, S_k\}$ de conjuntos disjuntos dinámicos.

MAKE-SET(x) crea un solo nuevo conjunto con x como miembro

UNION(x, y) une dos conjuntos dinámicos que contiene x e y .

FIND-SET(x) retorna un puntero al conjunto que le contiene a x

Bibliografía

- Mark A. Weiss. *Estructura de datos en Java. Compatible con Java 2*. Pearson. AW. 2000.
- Clifford A. Shaffer. *Data Structures & Algorithm Analysis in Java*. Ed. 3.2 2012 (cap. 11)
- T. Cormen, C. Leiserson, R. Rivest y C. Stein. *Introduction to algorithms*. Second edition. MIT Press. 2001. (cap: 22-23-24)
- Clase Algorithms and Data Structures (ECE 250 – Universidad de Waterloo, Canadá)
<http://starlinux.uwaterloo.ca/~ltahvild/courses/ECE250/>