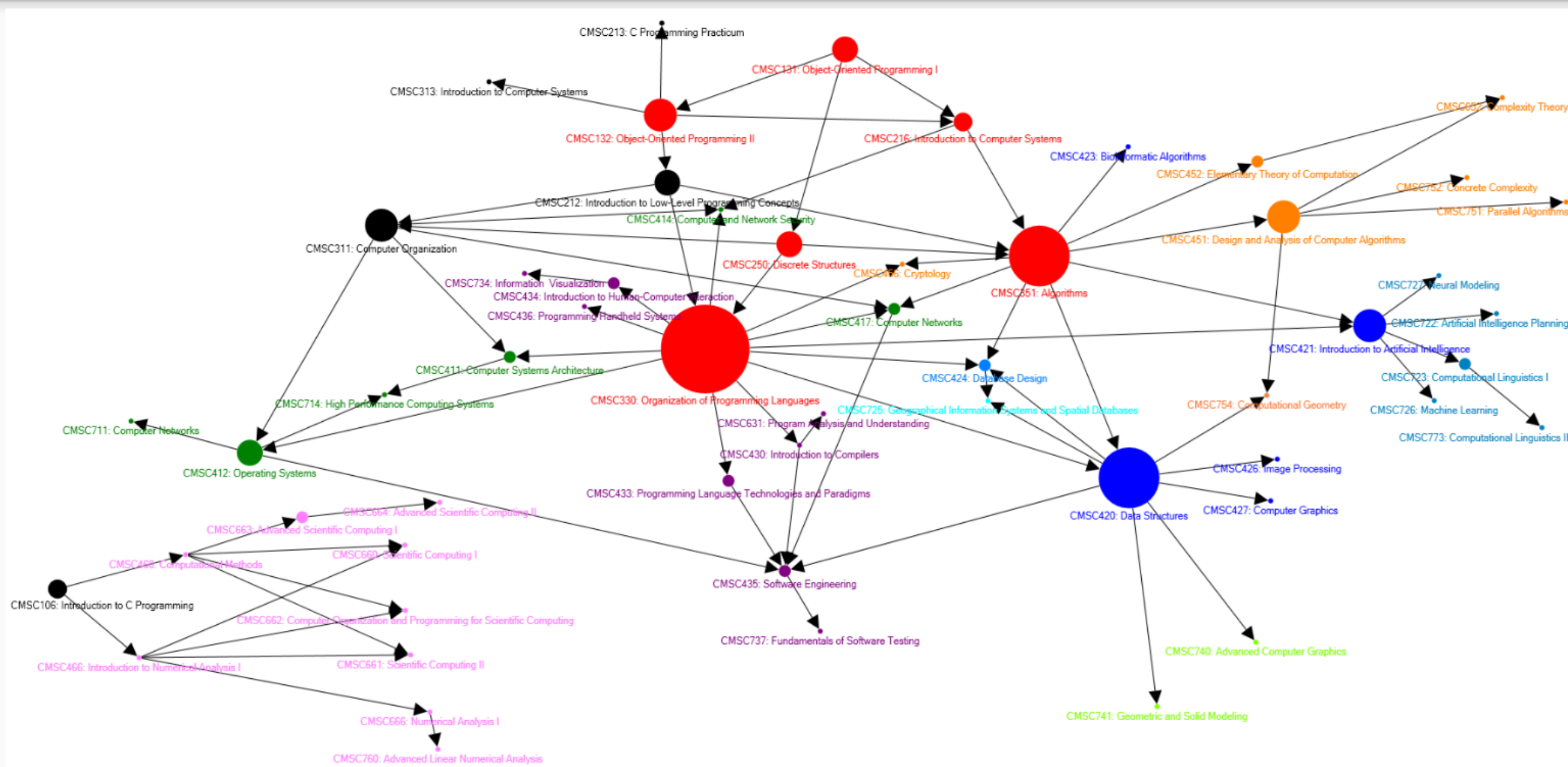


Welcome to Algorithms and Data Structures! - CS2100

Qué es un grafo?

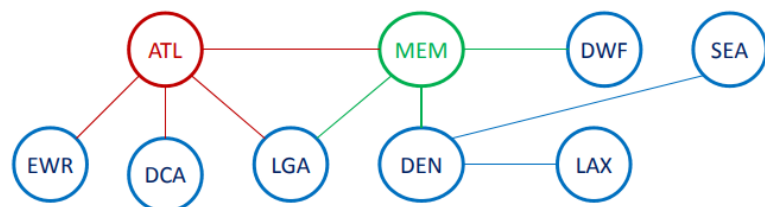
Quizás?



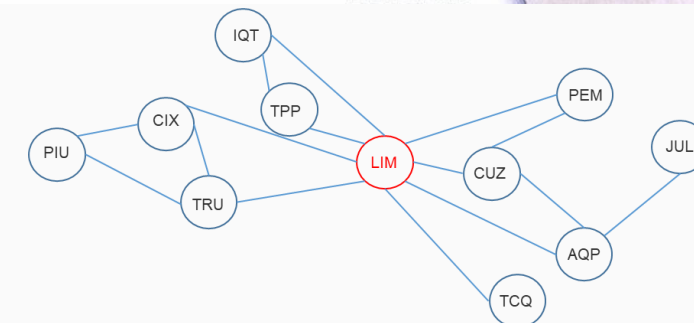
Redes Sociales?

Mapas? Rutas?

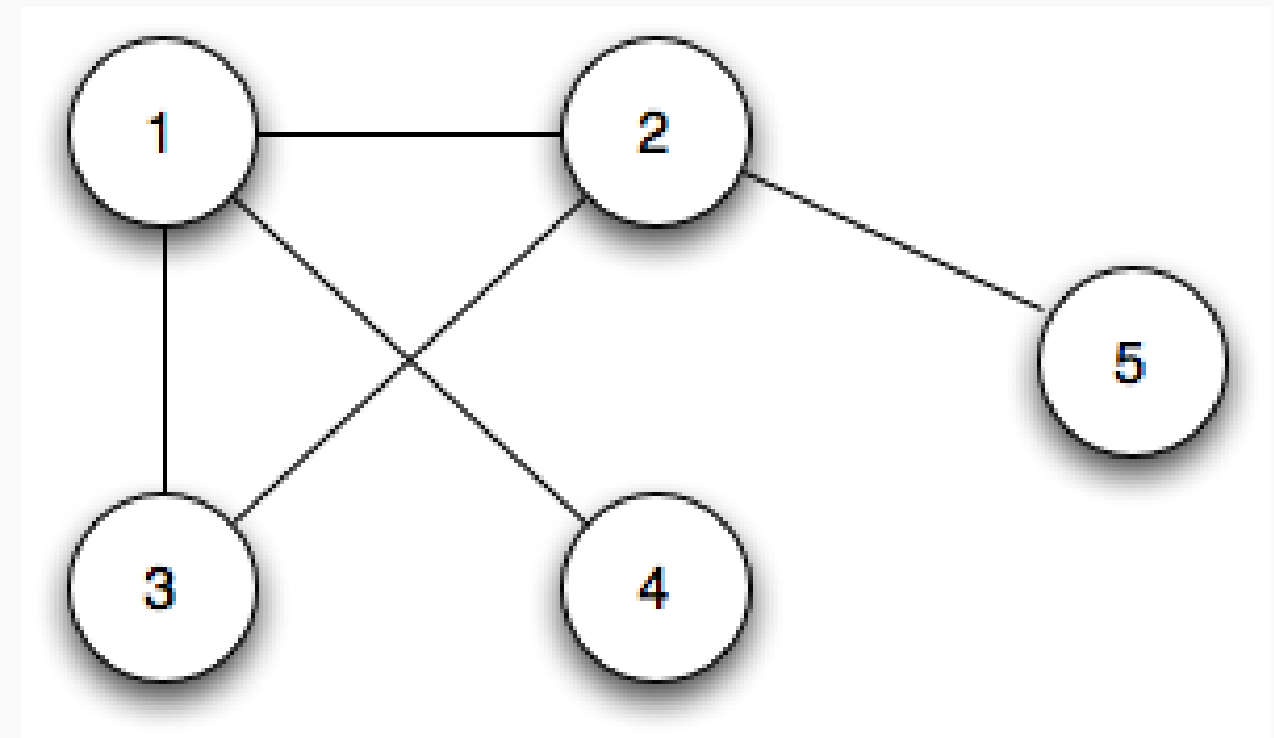
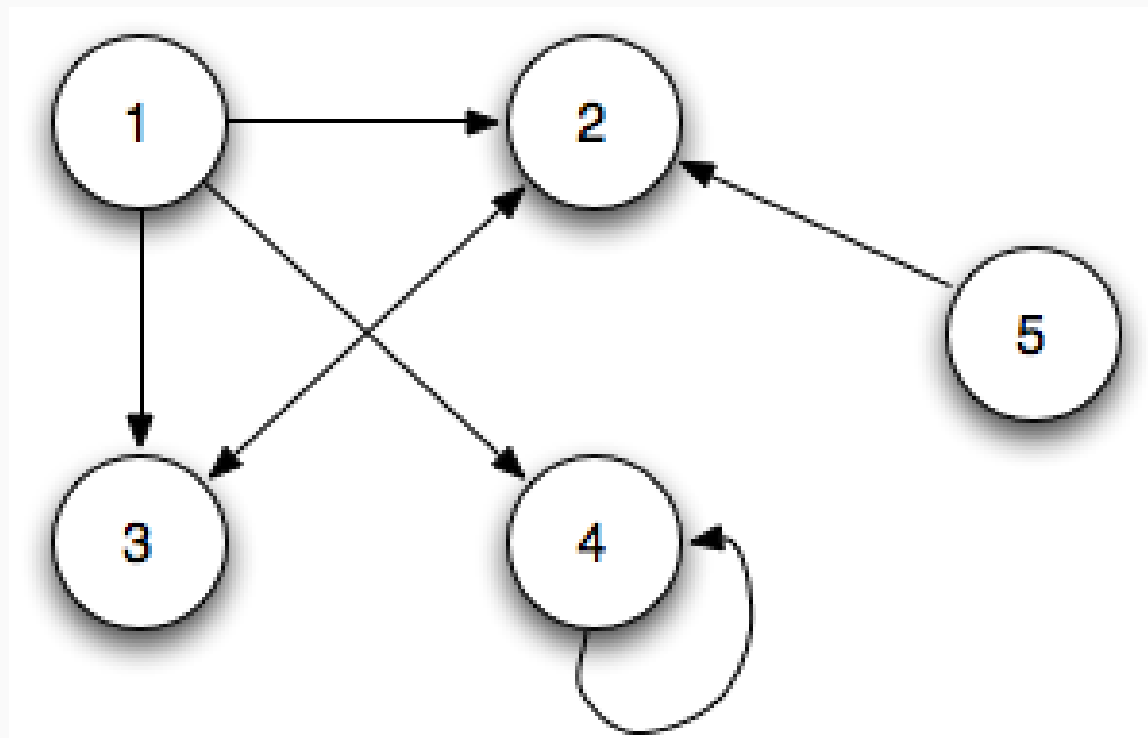
Delta Airlines



LAN Airlines

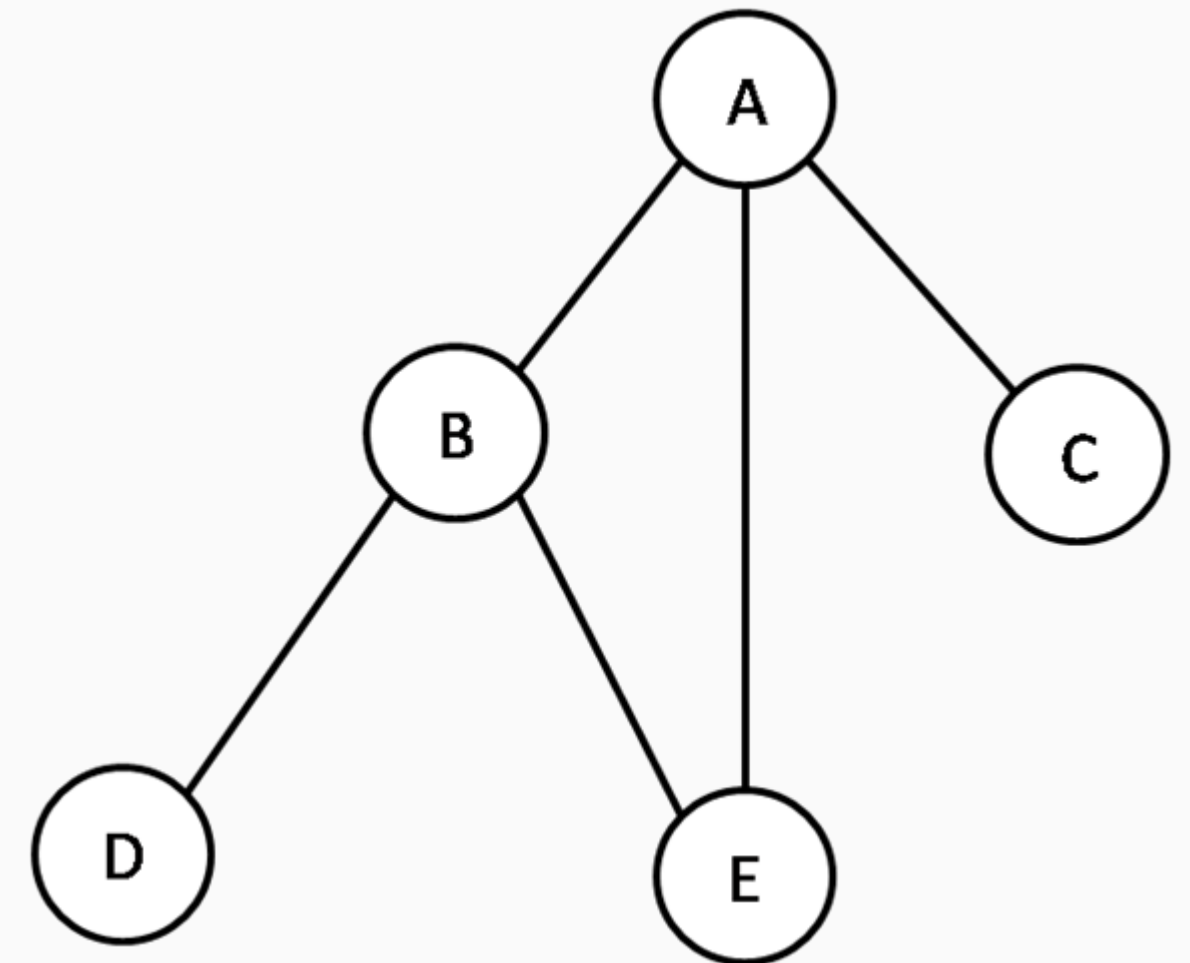


Qué tal esto?



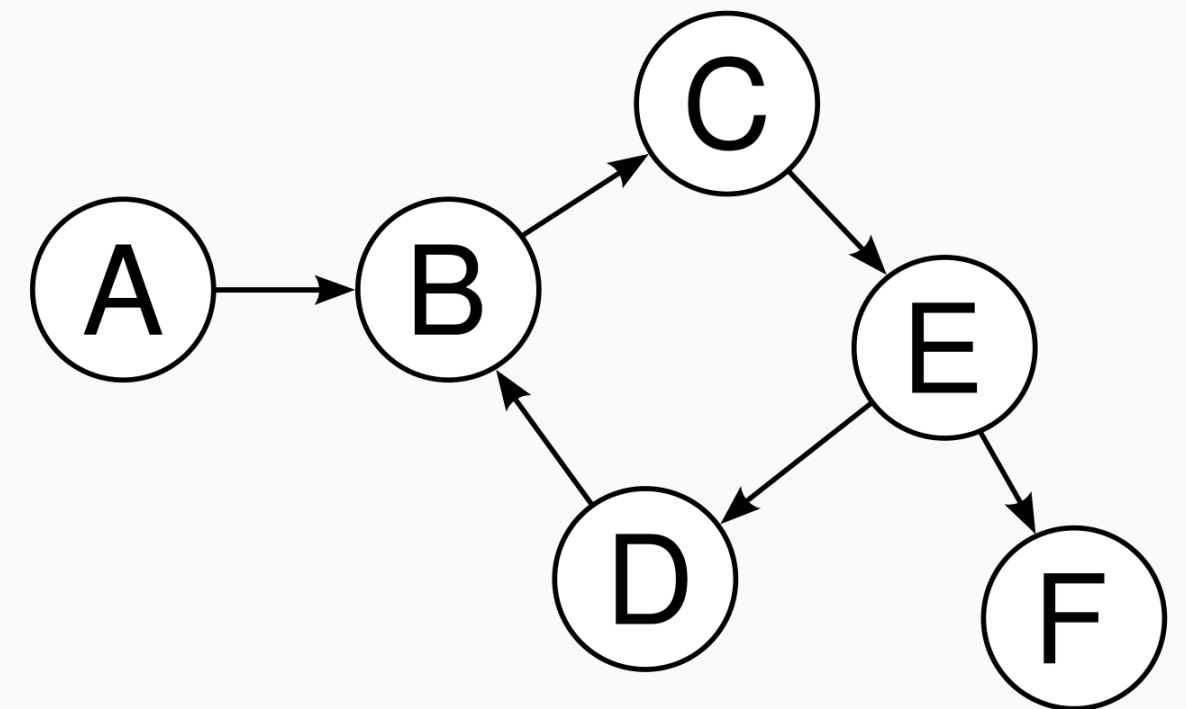
Qué es un grafo?

- Un grafo es un par de conjuntos finitos (V, E)
(Vértices y Aristas)
 - ◆ $V = \{A, B, C, D, E\}$
 - ◆ $E = \{\{A, B\}, \{A, C\}, \{A, E\}, \{B, D\}, \{B, E\}\}$
- Cada arista establece una relación entre dos vértices
- Puede existir más de un camino entre un vértice a otro
- Un vértice puede relacionarse consigo mismo
- Un grafo vacío es el que no tiene vértices

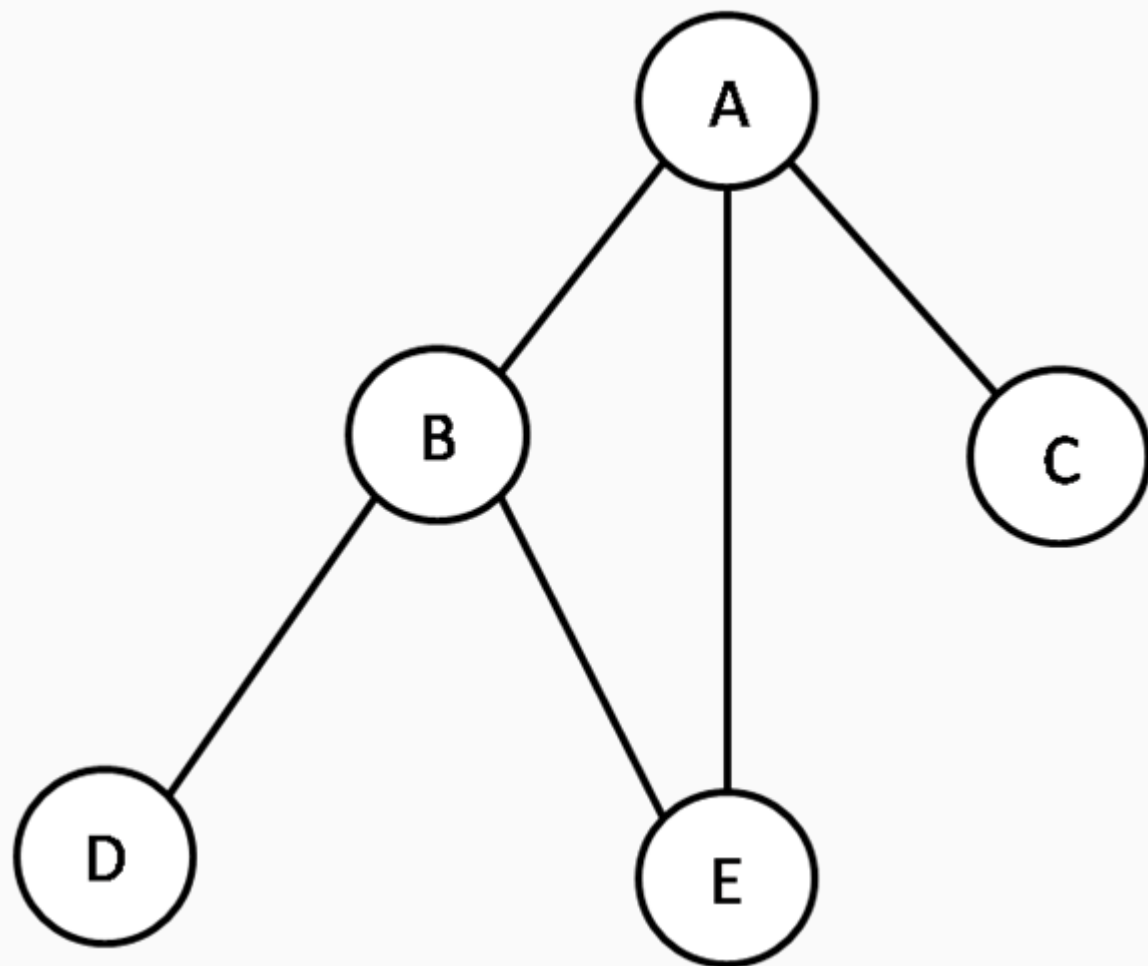


Qué es un grafo?

- Un camino en un grafo es una lista ordenada de nodos consecutivos que van de un nodo inicial a uno final. El inicial y final pueden ser el mismo
- En el camino, todos los nodos (n_i, n_{i+1}) deben ser adyacentes
- Un **camino simple** es aquel en el que no se repiten las aristas
- Un **ciclo** es un camino en el que el nodo inicial y el final son iguales e.g. $C \rightarrow E \rightarrow D \rightarrow B \rightarrow C$



Tipos de grafos



→ Es un **grafo no direccionado**

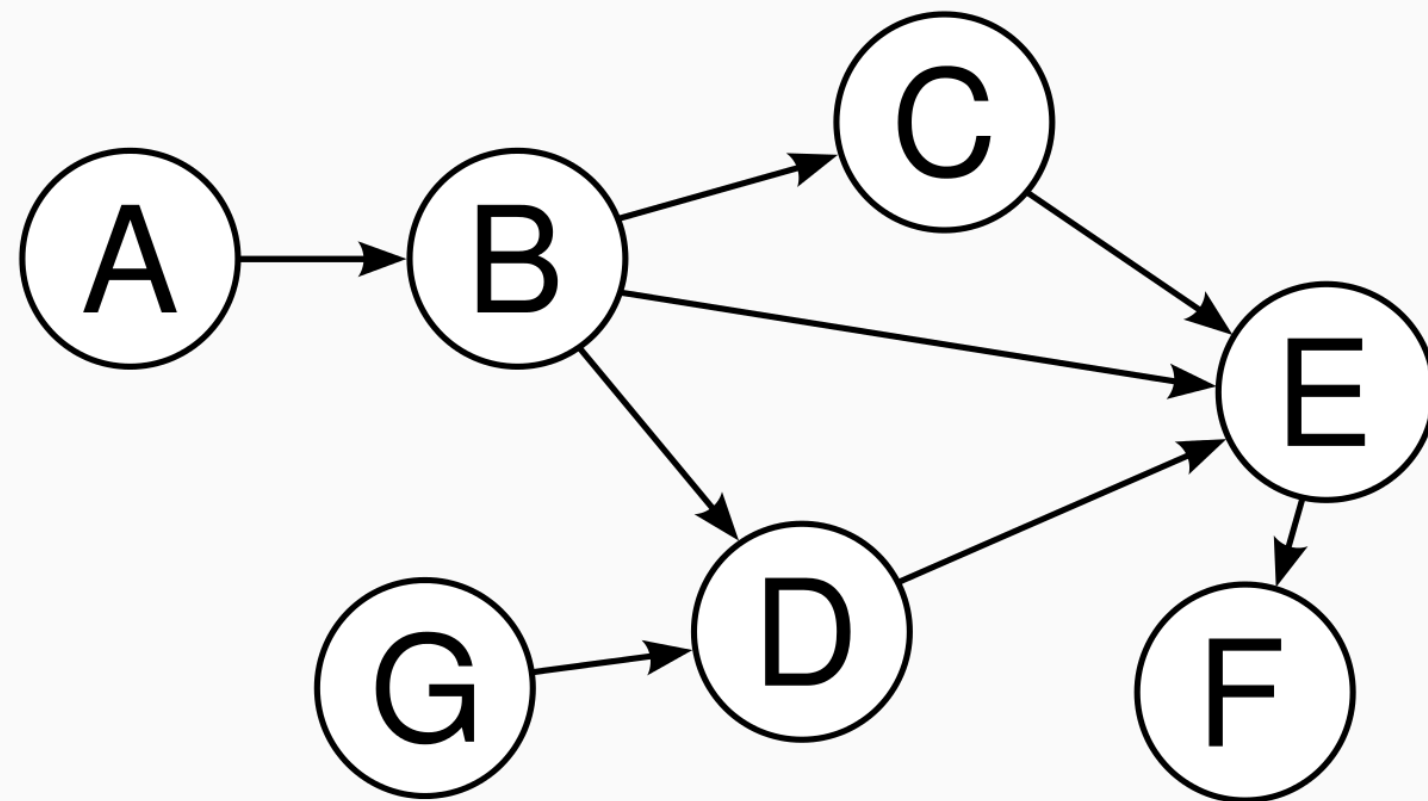
◆ $V = \{A, B, C, D, E\}$

◆ $E = \{\{A, B\}, \{A, C\}, \{A, E\}, \{B, D\}, \{B, E\}\}$

→ Se utilizan llaves ($\{\}$) para la notación de las aristas

→ En las aristas:
 $\{a, b\} = \{b, a\}$

Tipos de grafos



→ Es un **grafo direccional** o digrafo

◆ $V = \{A, B, C, D, E, F, G\}$

◆ $E = \{(A, B), (B, C), (B, D), (B, E), (C, E), (D, E), (E, F), (G, D)\}$

→ Se utilizan paréntesis (()) para la notación de las aristas

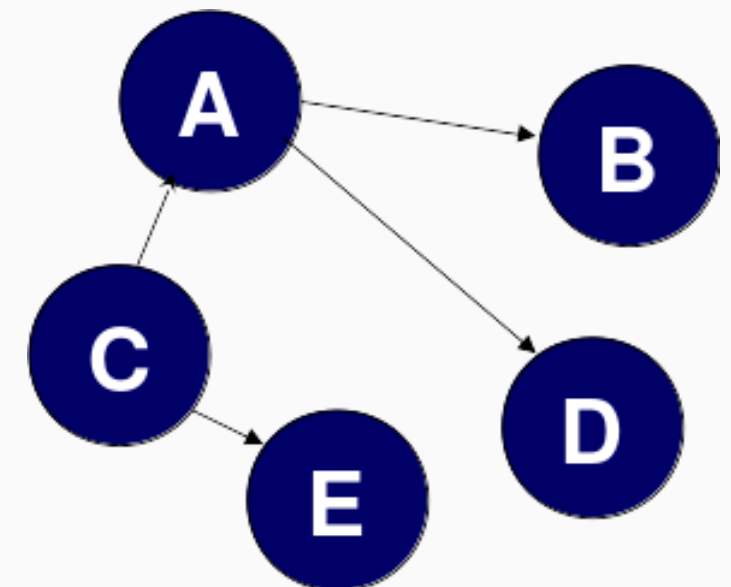
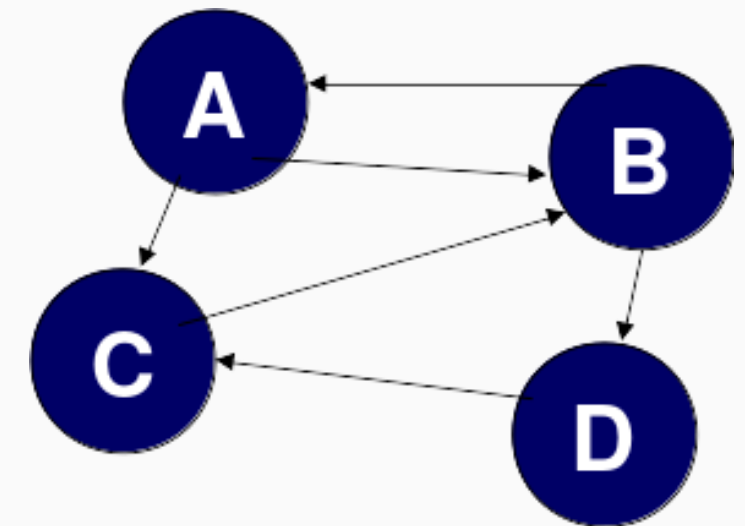
→ En las aristas:
 $(a, b) \neq (b, a)$

Grafo cíclico y acíclico

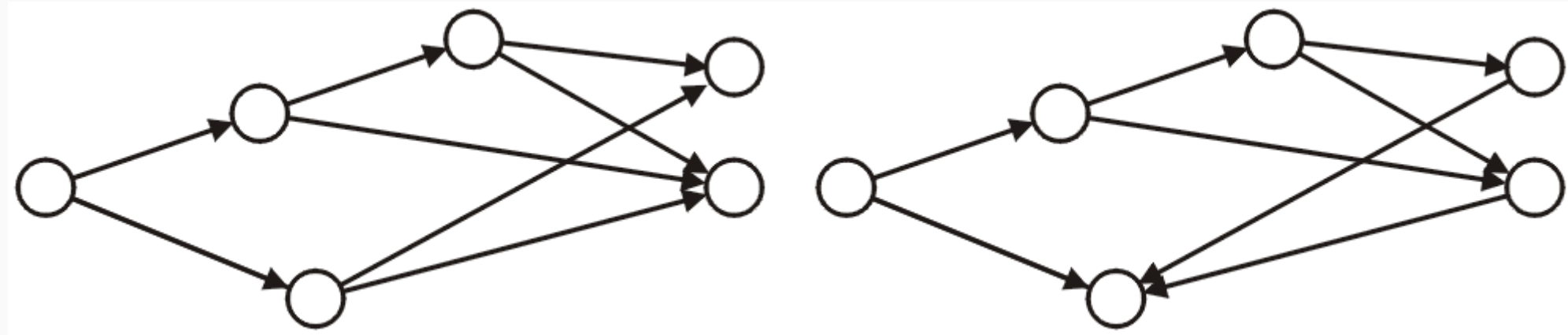
- Si el grafo contiene al menos un ciclo se llama cíclico
- Entonces un grafo acíclico será aquel que no tiene ningún ciclo

Qué tipo de grafo serían los siguientes?

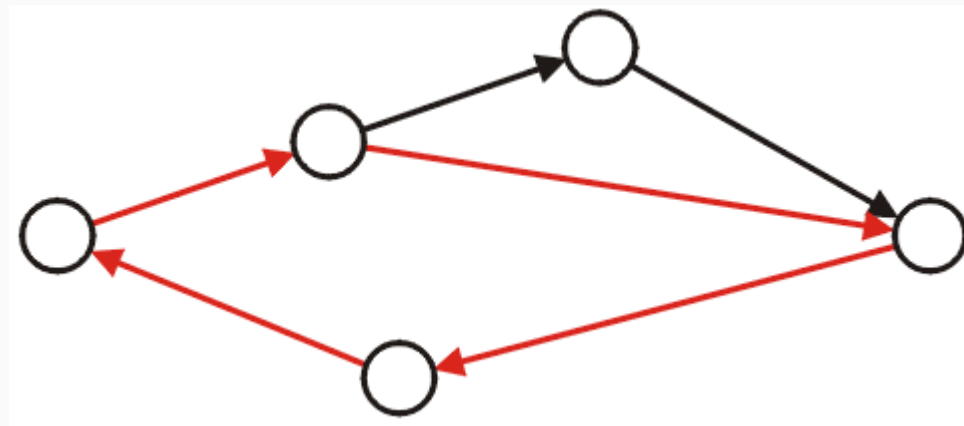
Den ejemplos de usos de un grafo direccionado acíclico



Grafo cíclico y acíclico



En este caso si tenemos un ciclo:



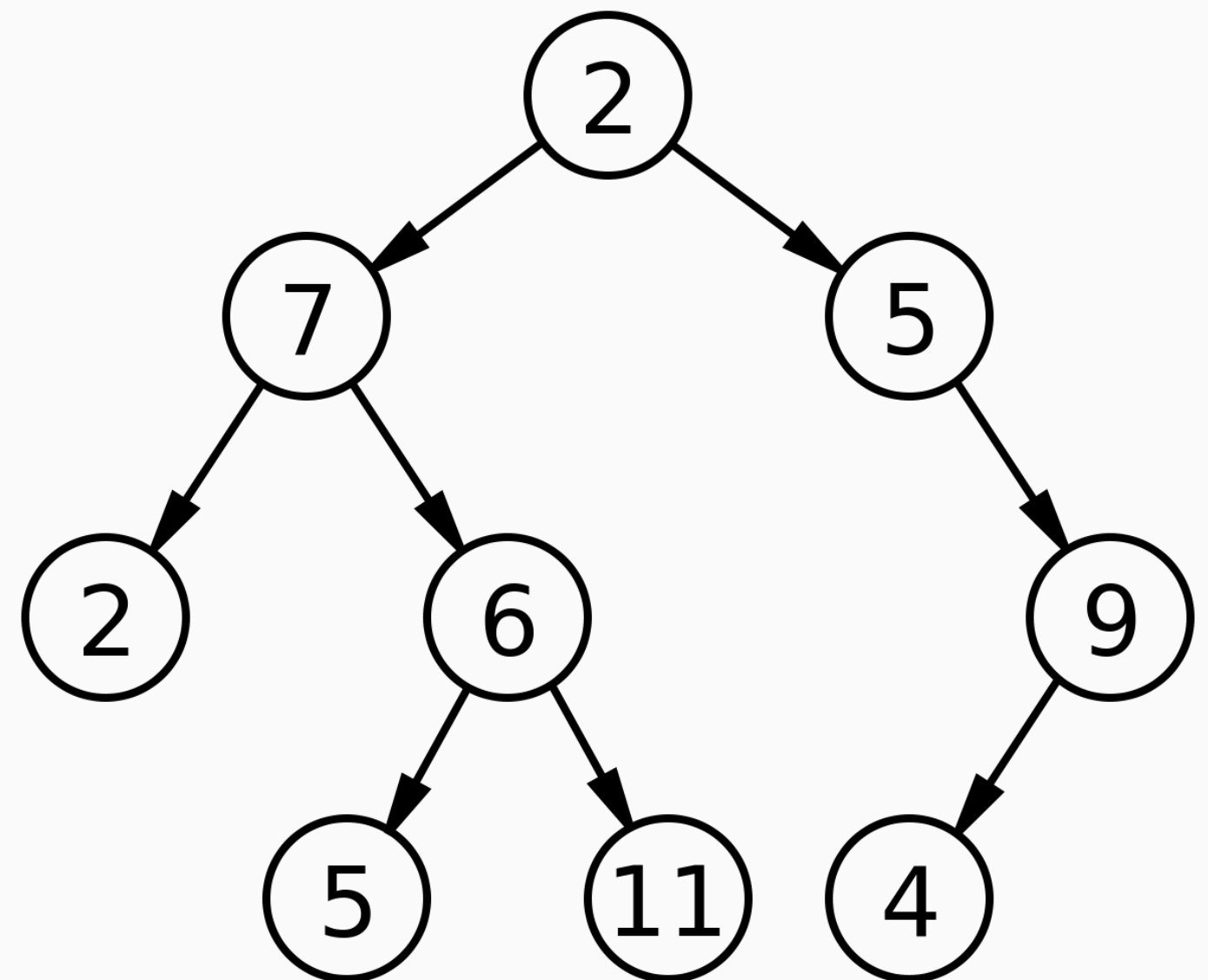
Entonces recordemos, qué es un árbol?

Un árbol es un tipo especial de grafo donde se tiene un nodo raíz y una relación de paternidad entre nodos

Los nodos hijos (excluyendo la raíz) son particionados en sub-árboles

Los nodos que no tienen hijos se les denomina hojas

Sólo existe un camino entre dos nodos



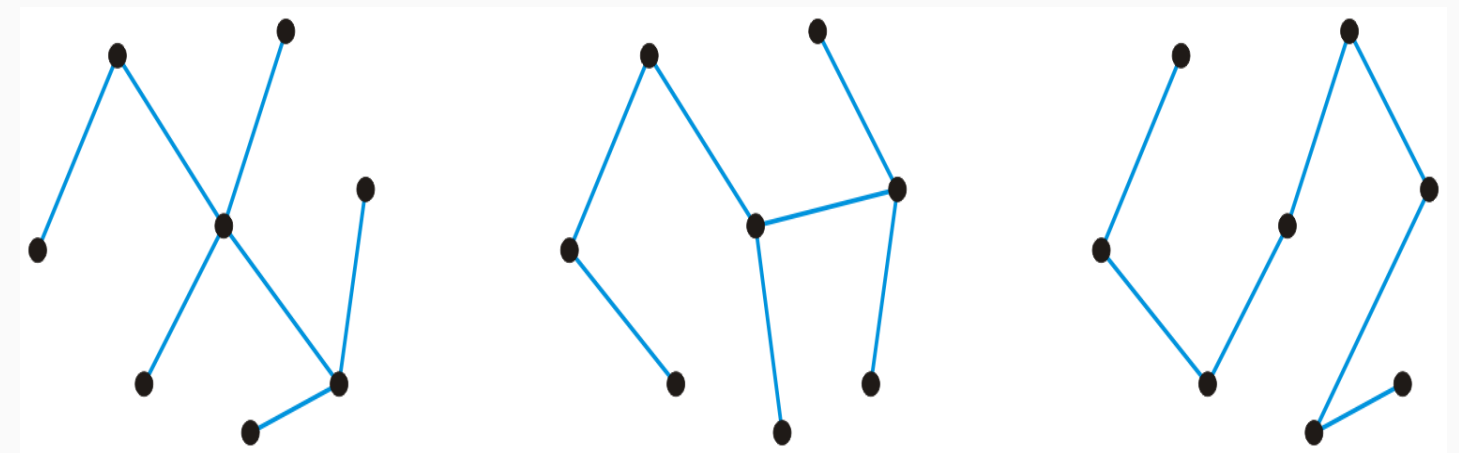
Árbol

Para que sea un árbol, el grafo debe estar conectado

No debe contener ciclos, y si se agrega una arista debería aparecer un ciclo

Remover cualquier arista generaría un grafo disjunto (separado)

El número de aristas será $|E| = |V| - 1$

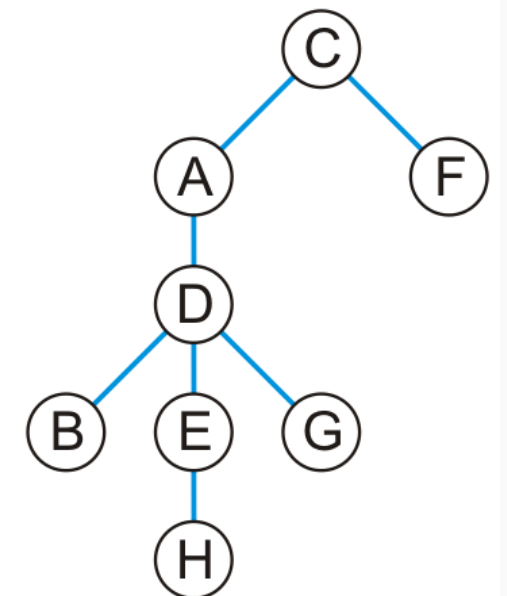
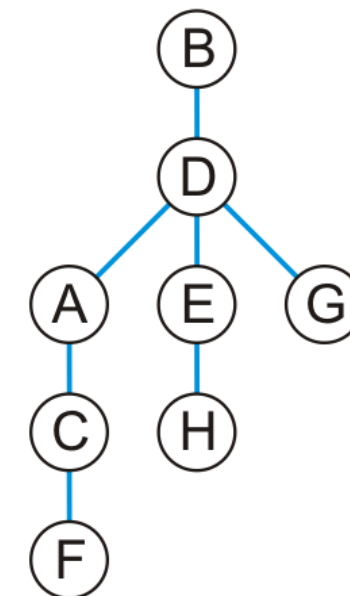
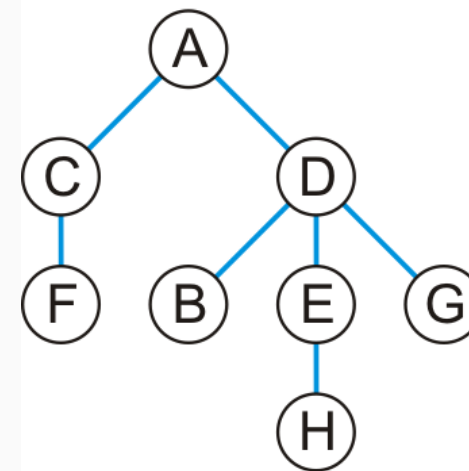
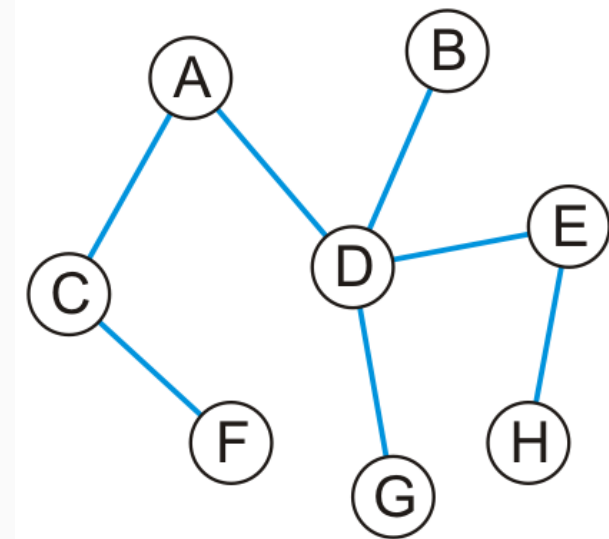


Árbol

Un grafo (árbol) puede ser transformado en un árbol al elegir un nodo como su raíz y definiendo sus vértices vecinos como su hijos

Entonces, qué sería un árbol de expansión?

Cómo crearían uno?



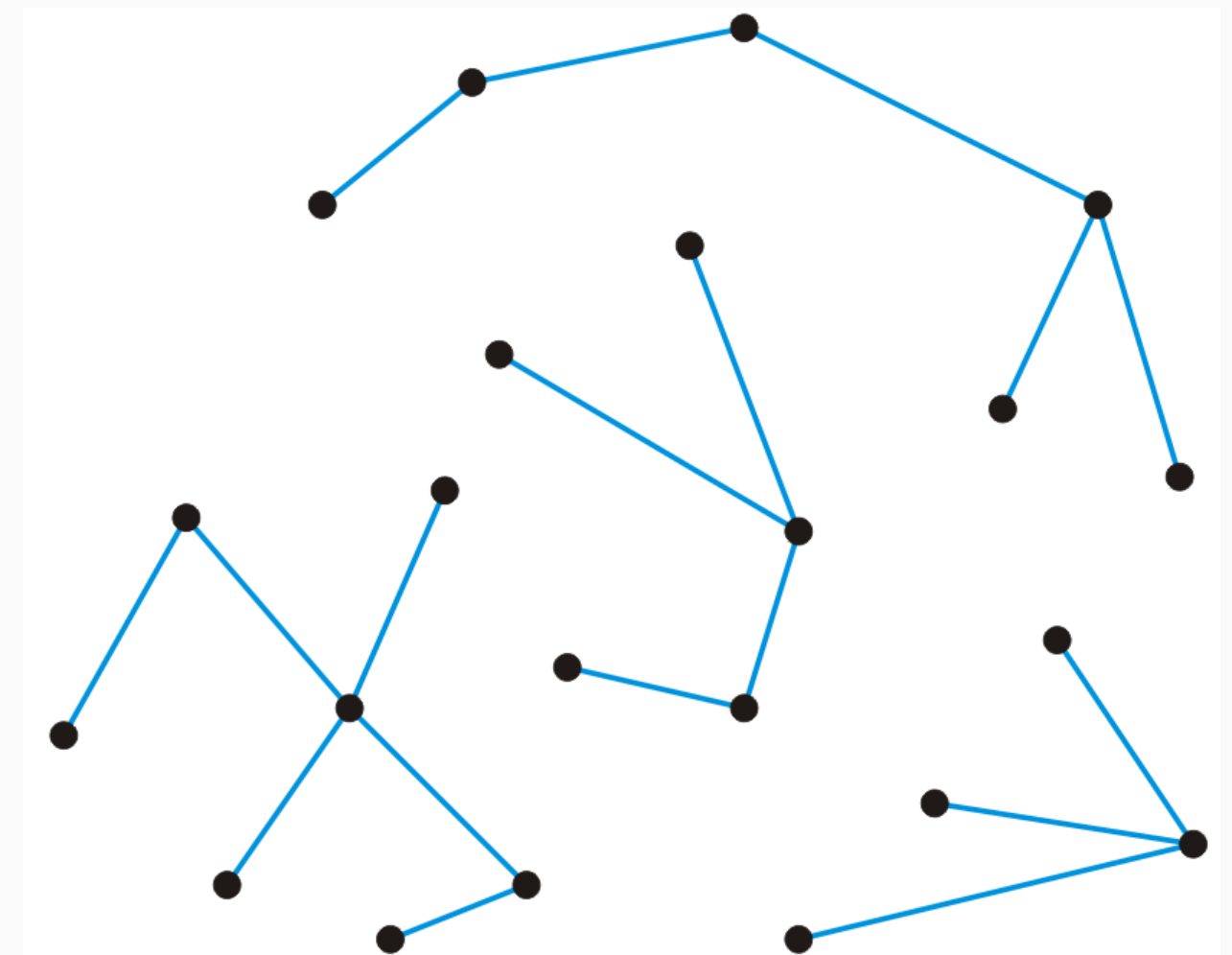
Bosque

Un bosque es cualquier grafo que no tiene ciclos y contiene uno o más árboles

Propiedades:

- El número de aristas es $|E| < |V|$
- El número de árboles es $|V| - |E|$
- Al remover una arista agregas un árbol al bosque

Qué es un bosque de expansión?



Ejercicio

→ Dibujar el siguiente grafo:

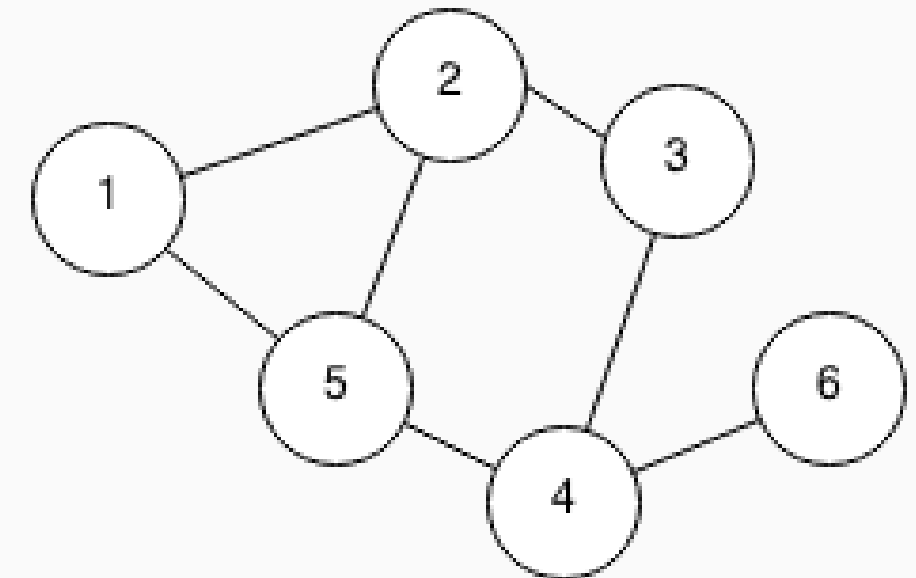
- ◆ $V = \{A, B, C, D, E, F\}$
- ◆ $E = \{\{A, B\}, \{A, C\}, \{B, D\}, \{C, E\}, \{D, F\}, \{E, F\}\}$

→ Dibujar el siguiente grafo:

- ◆ $V = \{A, B, C, D, E, F\}$
- ◆ $E = \{(A, B), (A, C), (A, D), (B, D), (C, E), (D, E), (E, F)\}$

Propiedades

- **Adyacencia:** Si dos vértices x_i y x_j están conectados por una arista son adyacentes (e.g. 1, 2)
- **Incidencia:** Una arista es incidente a sus vértices x_i y x_j (e.g. {1,2} es incidente a 1 y 2)
- **Grado:** El número de aristas incidentes a un vértice, es su grado. Un vértice aislado tiene grado 0, y un vértice hoja es el que tiene grado 1. En un grafo dirigido se puede tener:
 - ◆ Grado de salida
 - ◆ Grado de entradaLos vértices son fuente si su grado de entrada es 0, o hundido si su grado de salida es 0
- **Grafo simple:** No tiene loops ni múltiples aristas
- **Grafo completo:** Cada par de vértices está conectado por una arista
- **Multigrafo:** Las aristas pueden tener más de una conexión



Loop: Arista a sí mismo
Múltiple: Varias aristas entre dos nodos

Propiedades

→ **Densidad:** Podemos tener un grafo denso, o disperso dependiendo de este campo. La densidad máxima es 1 (grafo completo) y la mínima es 0. Usualmente se define una cota superior para el tipo de grafo.

◆ Grafo no dirigido: $D = \frac{2|E|}{|V|(|V| - 1)}$

◆ Grafo dirigido: $D = \frac{|E|}{|V|(|V| - 1)}$

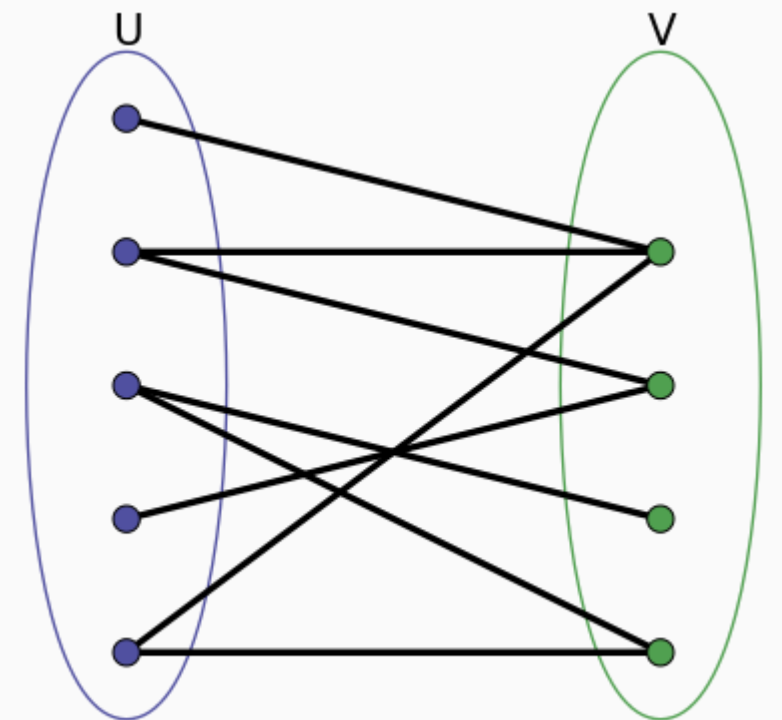
→ **Bipartito:** Un grafo bipartito es todo aquel que sus vértices se pueden separar en dos conjuntos disjuntos, donde:

$$U \cup V = N$$

$$U \cap V = \emptyset$$

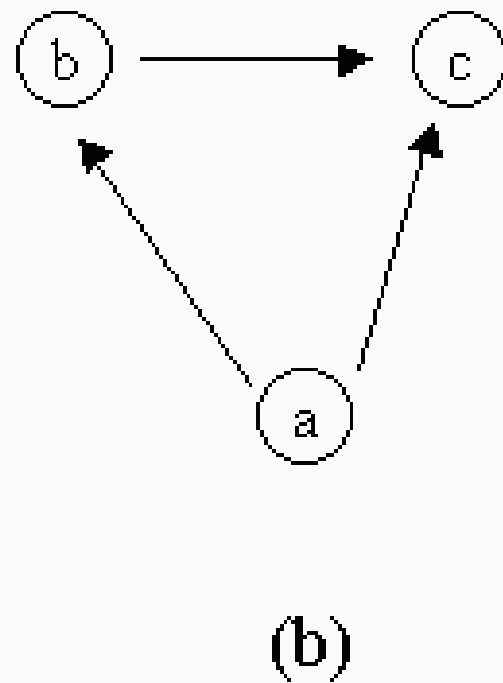
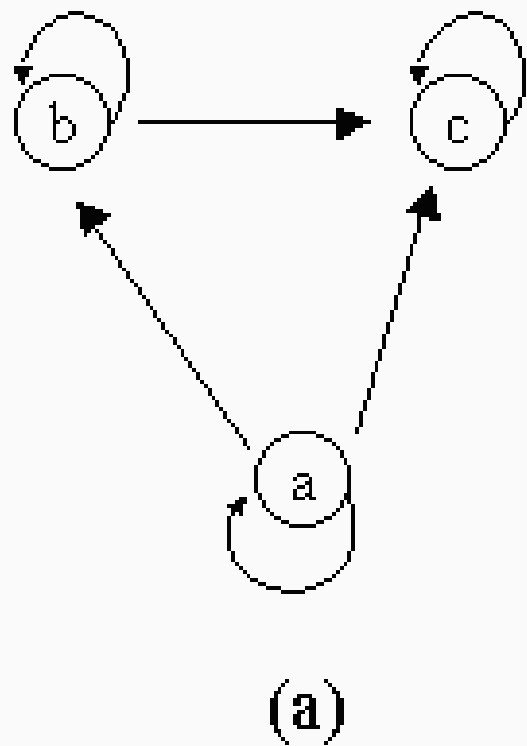
Se resuelve alternando el pintado de los vértices para finalmente tener dos conjuntos con colores diferentes

Recuerden: Todo árbol es bipartito, grafos sin ciclos con vértices impares etc.

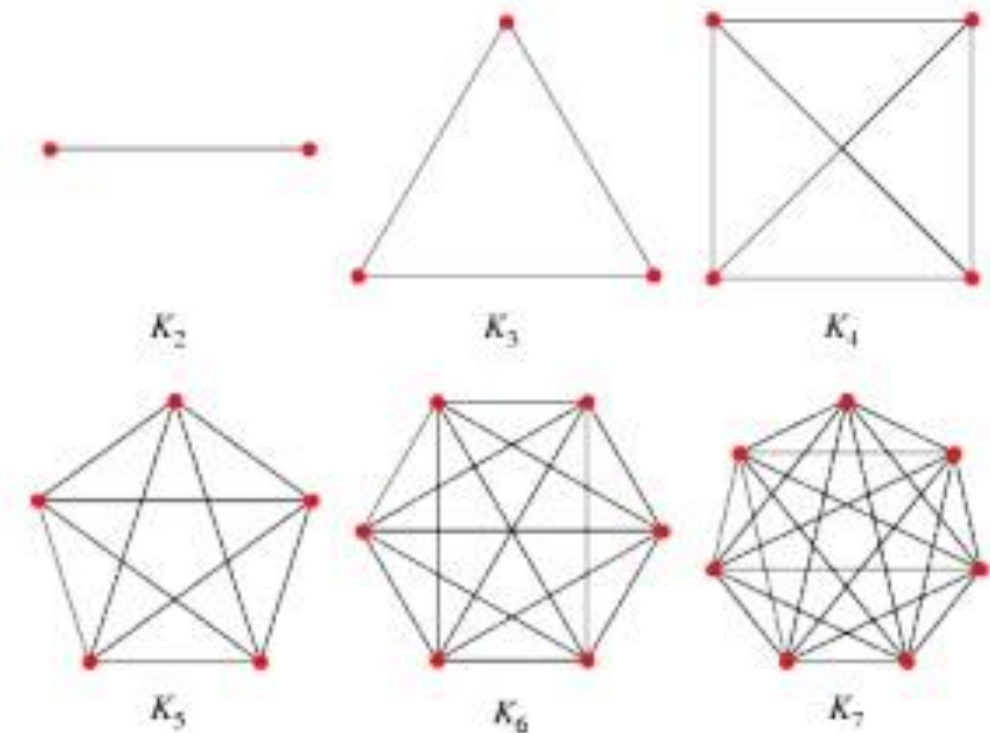


Ejercicios:

Cuál de los siguientes grafos es un grafo simple?

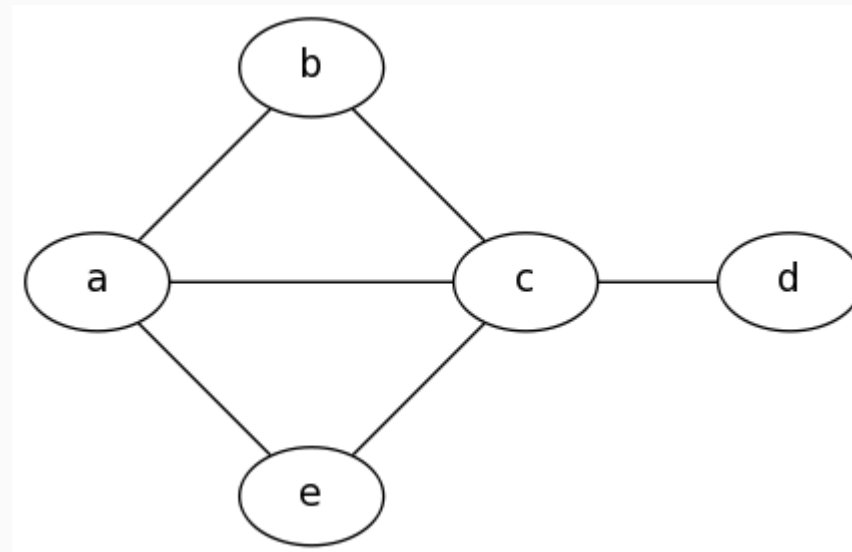
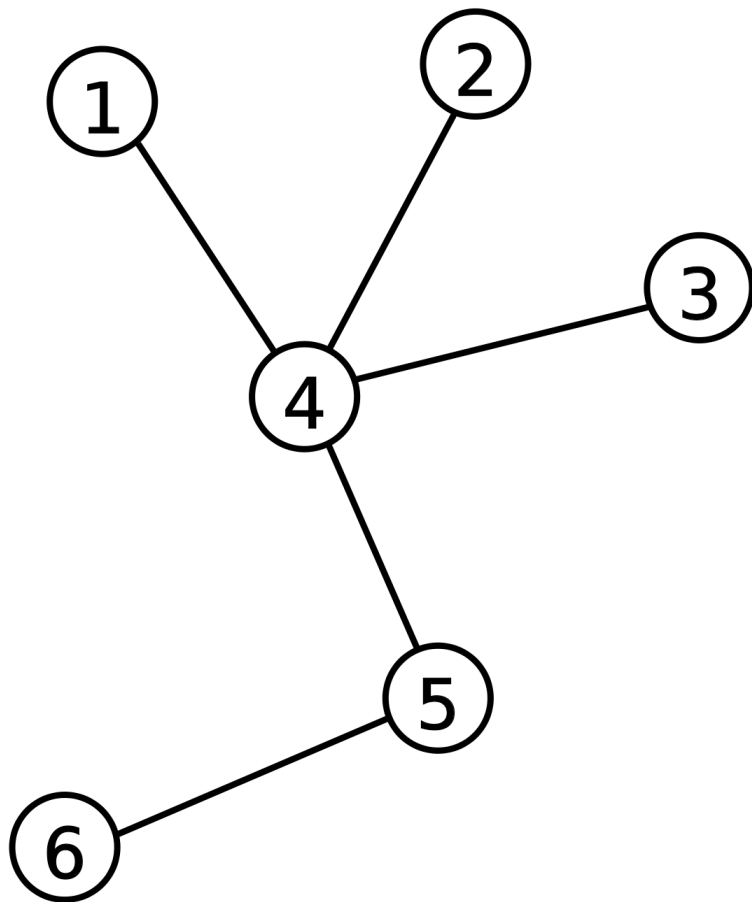


Y cuál es un grafo completo?

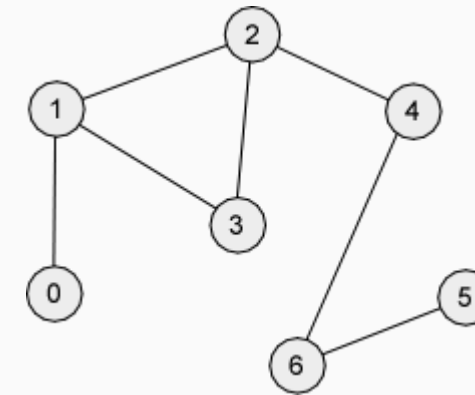


Ejercicios:

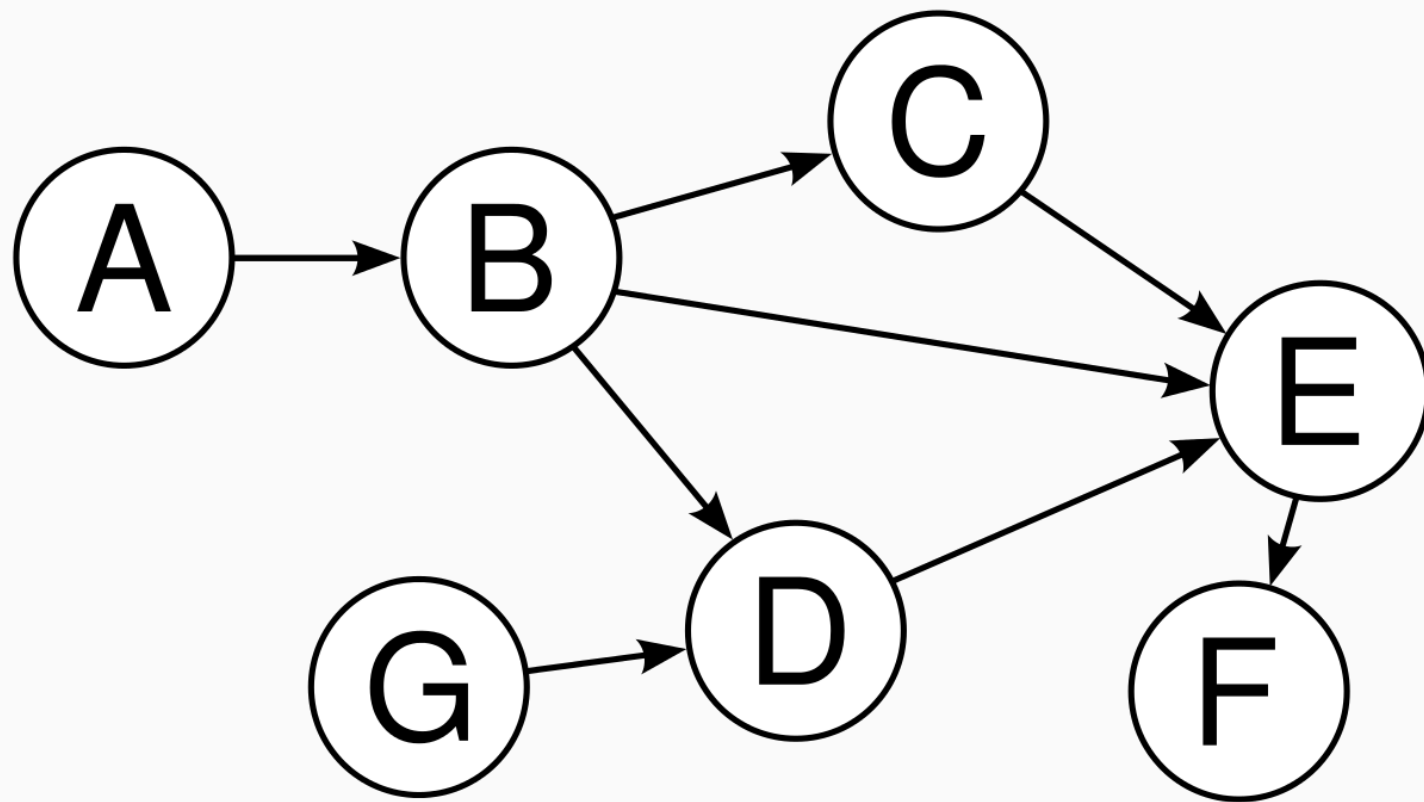
Si la cota para definir un grafo denso es 0.7, los siguientes grafos serán?



Y son bipartitos?



Ejercicio:



- **Cuál es el grado de la vértice B?**
Entrada: 1 Salida: 3 Grado: 4
- **Es un grafo simple?**
Si, no hay loops ni múltiples aristas
- **El vértice A es adyacente al vértice C?**
No, ya que no comparten una arista
- **Qué tipo de vértice es F?**
Hundido
- **Qué vértices fuente hay en el grafo?**
G, A

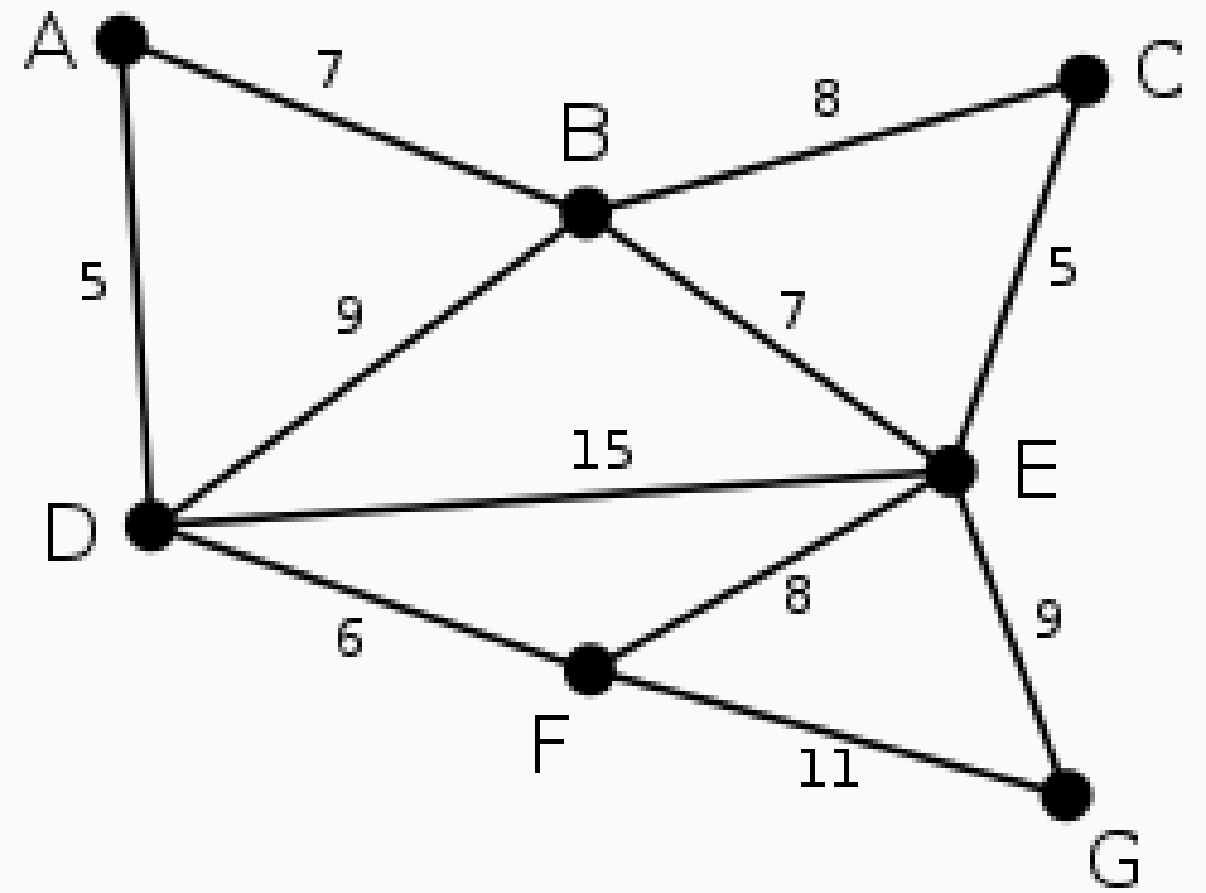
Grafos ponderados

En muchos casos es de interés asignar valores o pesos al grafo para representar por ejemplo: distancias, tráfico, precio, peligrosidad, consumo de energía, etc.

1. Grafos ponderados por aristas
2. Grafos ponderados por vértices

Recuerden:

- Máximo número de aristas en un grafo no dirigido: $(|V| * (|V| - 1)) / 2 \approx O(|V|^2)$
- Máximo número de aristas en un grafo dirigido: $|V| * (|V| - 1) \approx O(|V|^2)$

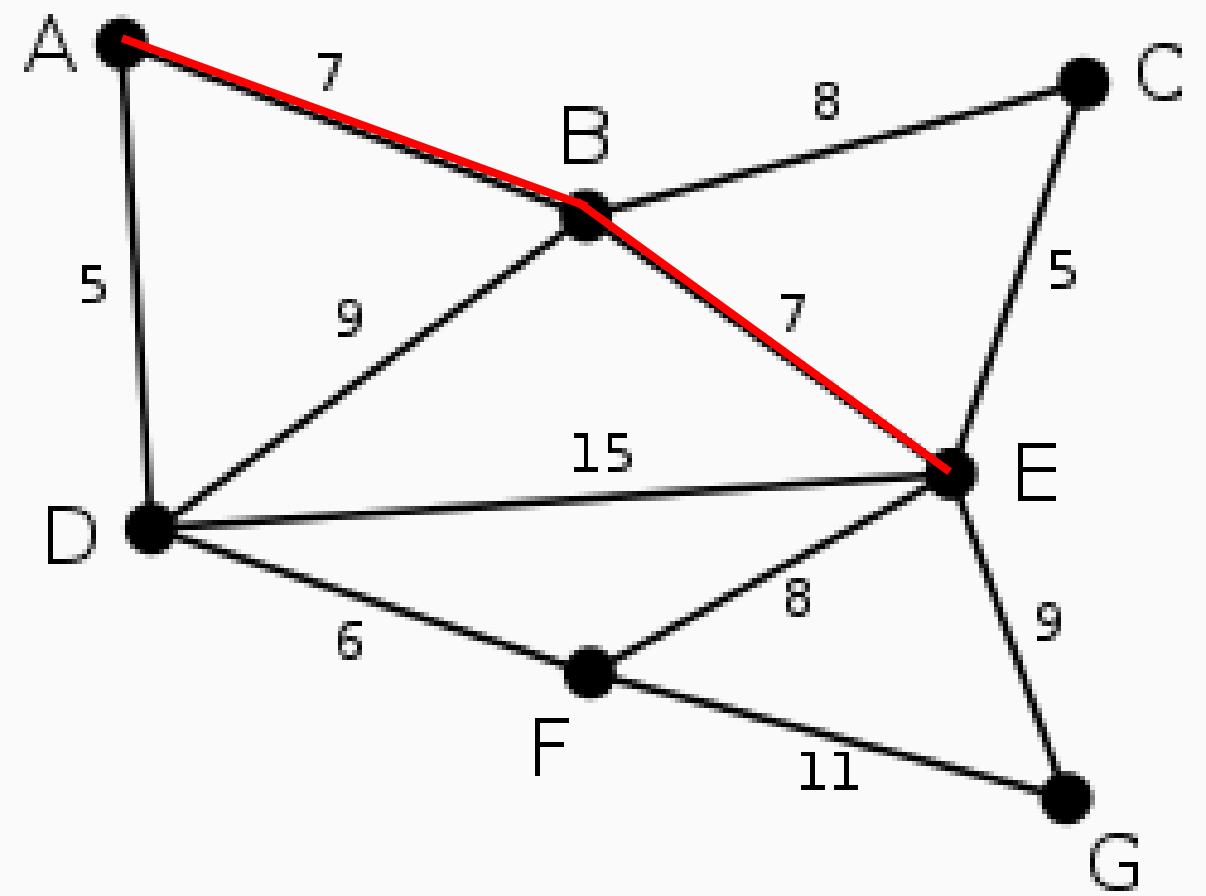


Grafos ponderados

En este tipo de grafos, el camino será dado por la sumatoria del peso de las aristas (o vértices)

El peso del camino A, B, E en el grafo de ejemplo será de 14

En la mayoría de casos nos interesará el camino con menor peso



Conectividad

❑ Grafo conexo:

Es un grafo en el que para cualquier par de vértices i y j existe al menos un camino entre ellos

❑ Grafo no conexo:

Existe al menos un par de vértices que no tengan un camino entre ellos

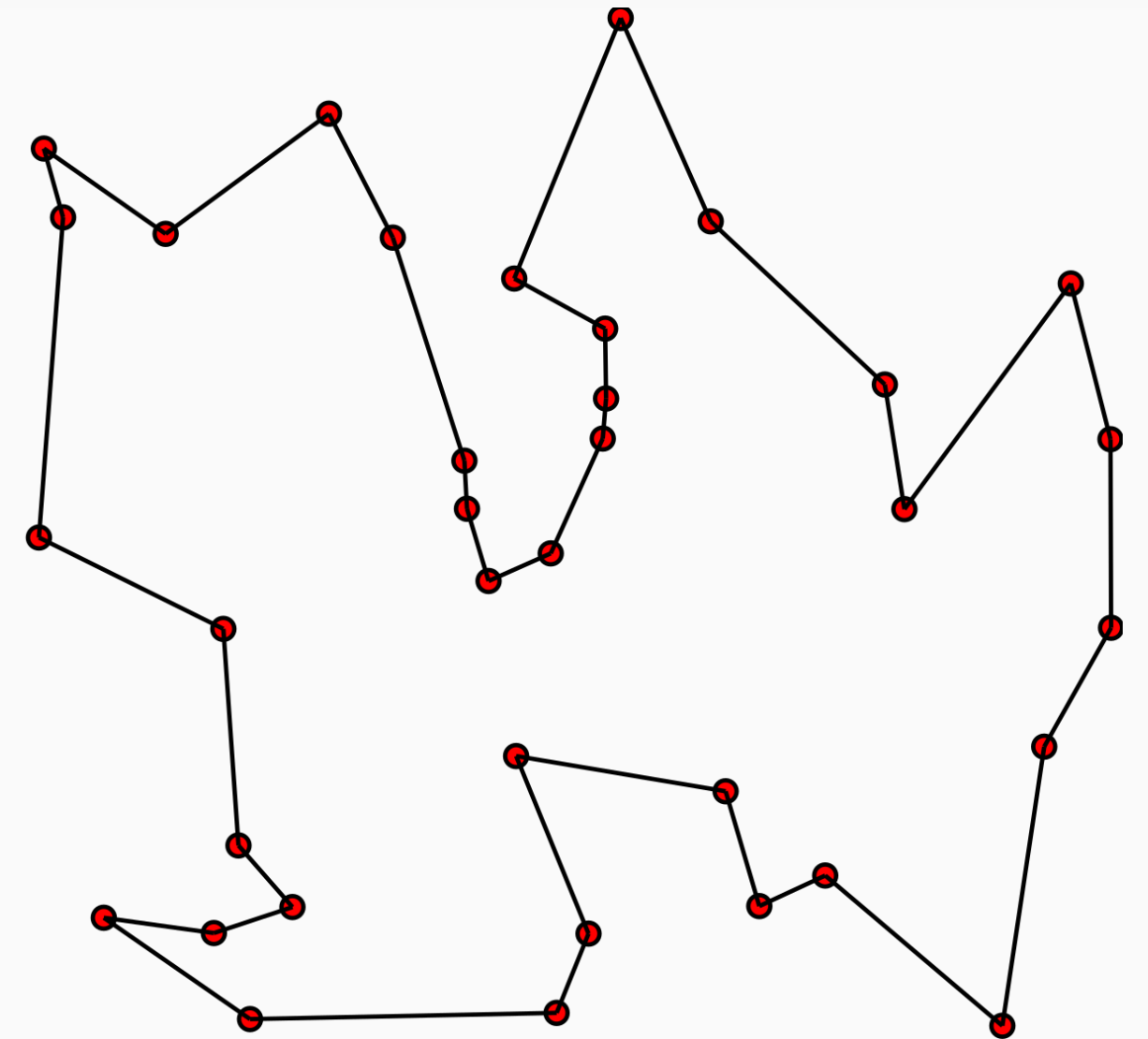
En grafos dirigidos, qué sería un grafo fuertemente conexo?

Cuando existe un camino dirigido de i a j , y de j a i

Problema del agente viajero

Un agente debe repartir paquetes en diferentes ciudades. Se conoce la distancia entre las ciudades, y las ciudades a las cuales debe repartir los paquetes.

El problema consiste en descubrir la mejor ruta a seguir por el agente:



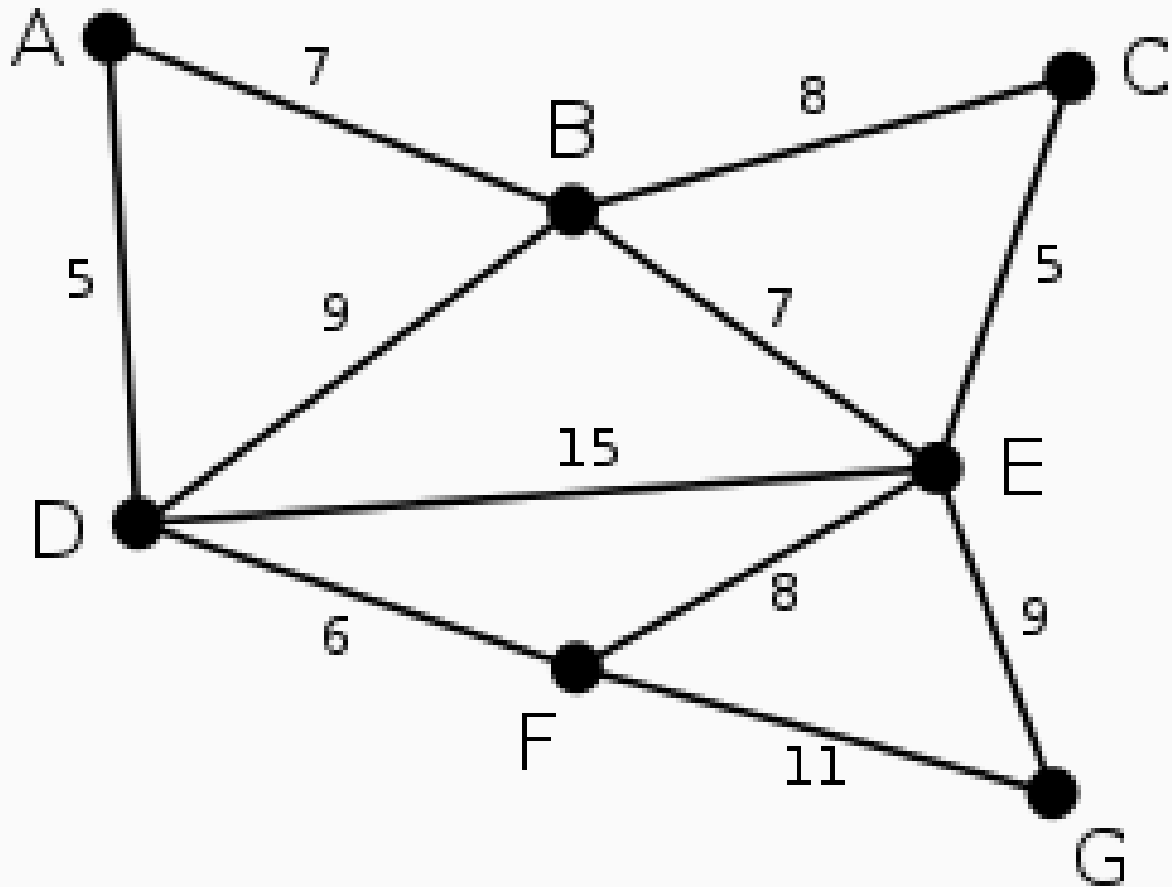
Problema del agente viajero

¿Cómo modelarían este problema?

- Las ciudades como vértices
- Caminos entre ciudades como aristas ponderadas
- Se puede usar un grafo no dirigido si se cumple que la distancia de la ciudad a a b es igual a la distancia b a a

Representación de Grafos

Grafo = (V , E)

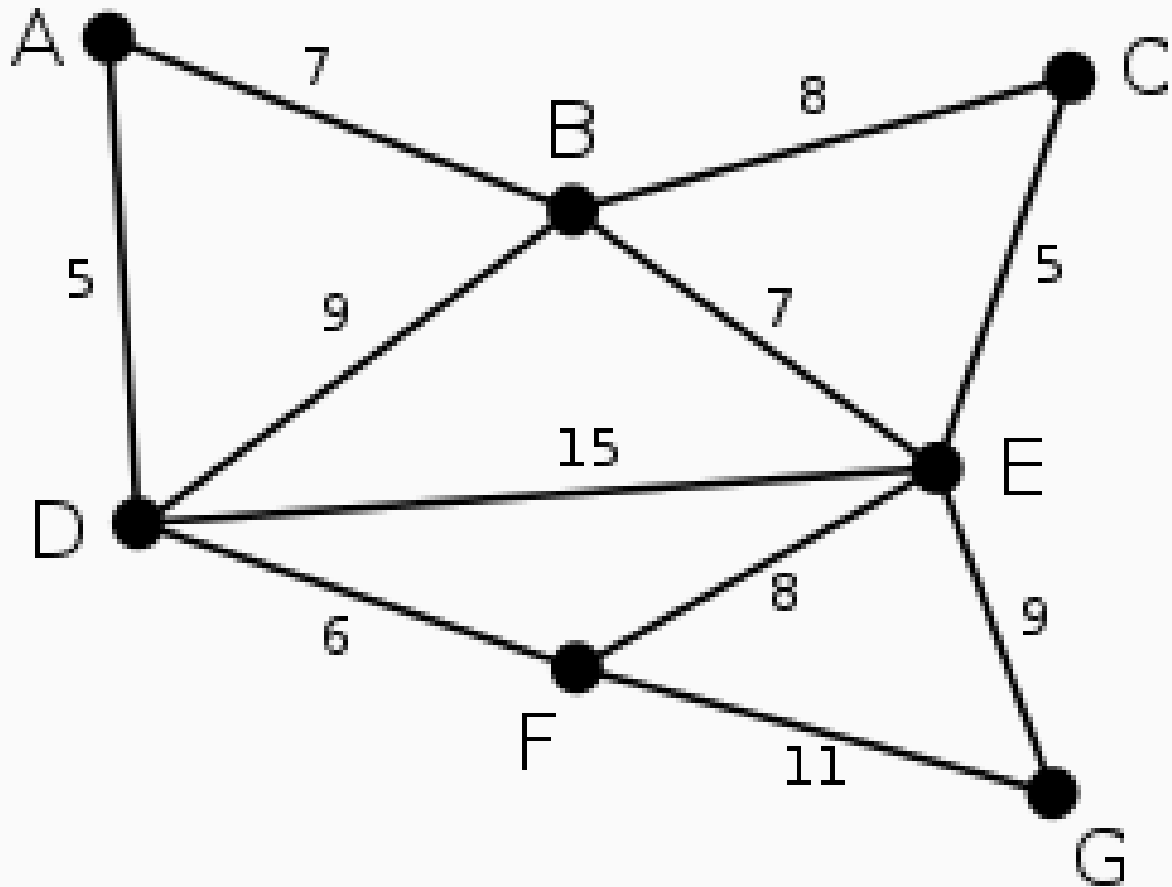


¿Un ejemplo de cómo podría implementarse ?

El grafo como estructura que contenga una lista de vértices y otra de aristas.

Representación ingenua (lista de aristas)

Grafo = (V , E)



¿Un ejemplo de cómo podría implementarse ?

```
struct Edge {  
    string startVertex;  
    string endVertex;  
    float weight;  
}  
  
class Graph{  
    vector<string> vertex;  
    vector<Edge*> edges;  
}
```

Representación ingenua (lista de aristas)

Espacio: $O(V) + O(E)$

Por qué esto no es eficiente?

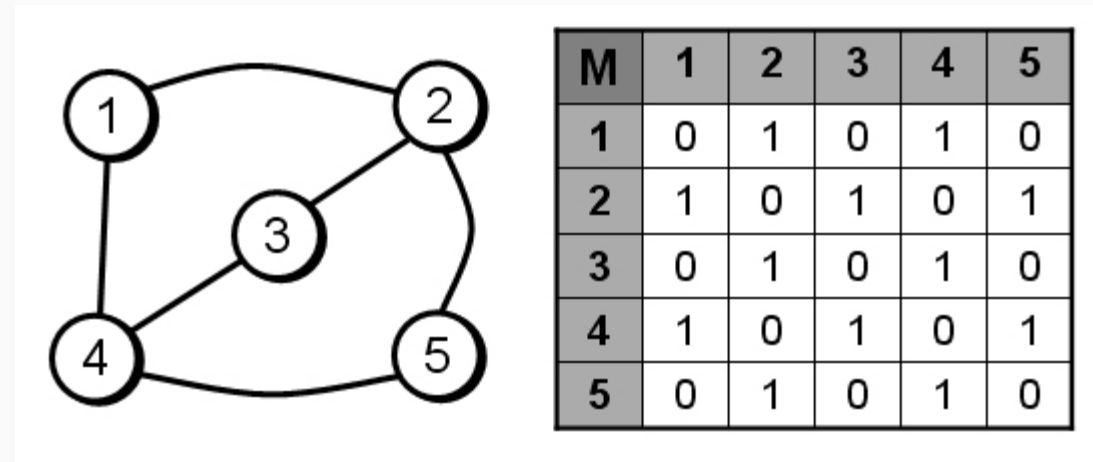
- Cuánto toma encontrar los nodos adyacentes a un nodo?
- Saber si dos vértices están conectados?

Un ejemplo de cómo podría implementarse:

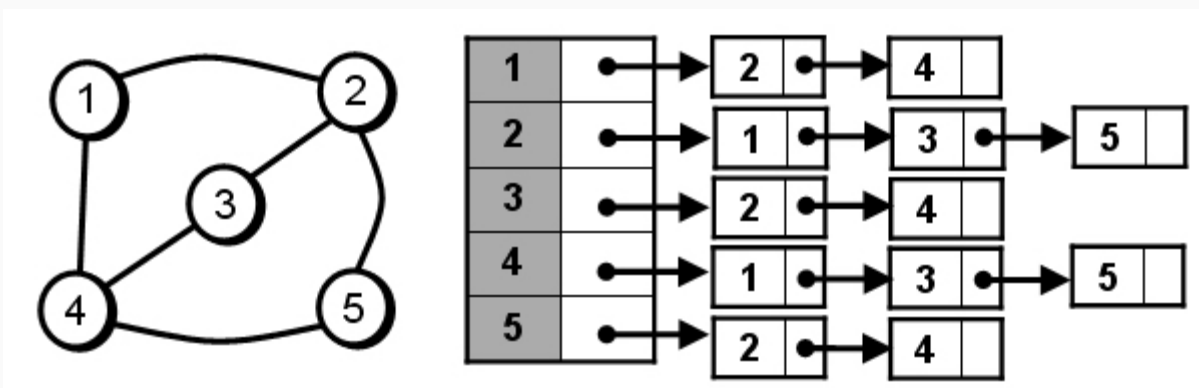
```
struct Edge {  
    string startVertex;  
    string endVertex;  
    float weight;  
}  
  
class Graph{  
    vector<string> vertex;  
    vector<Edge*> edges;  
}
```

Representación de grafos

❑ Matrices de adyacencia



❑ Listas de adyacencia



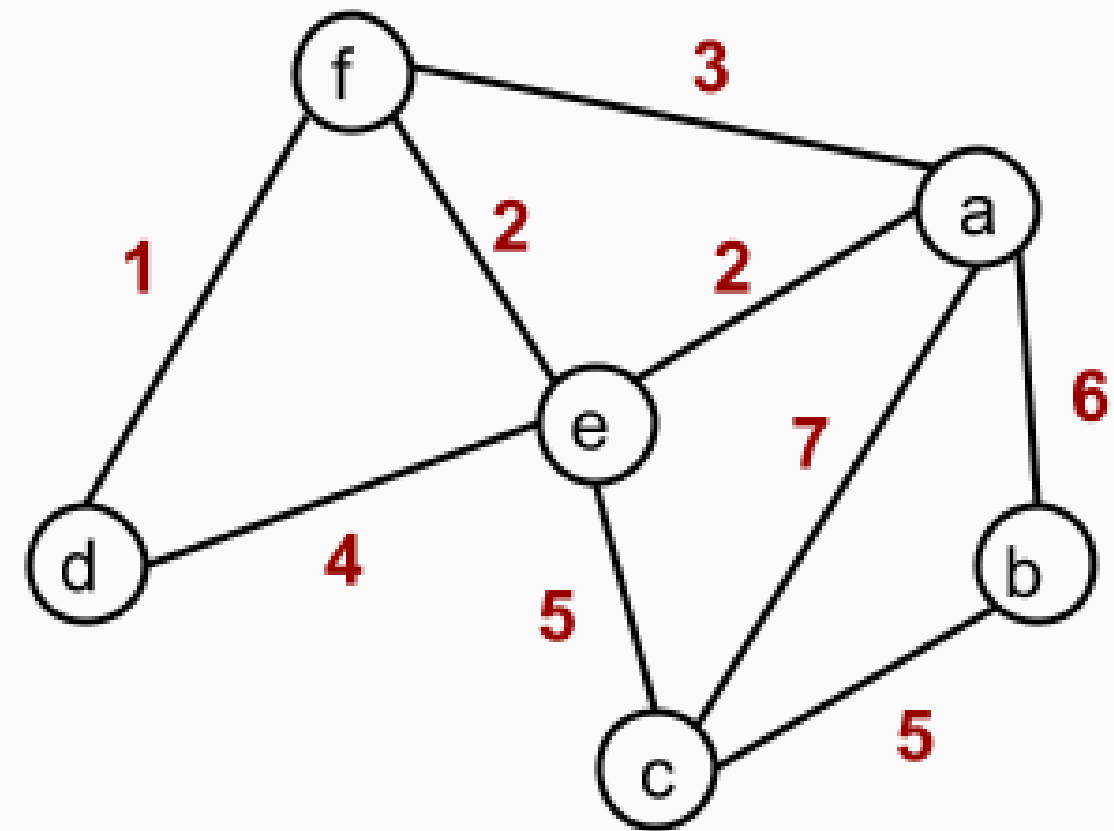
Matrices de adyacencia

Se utiliza una matriz cuadrada (Array 2D)

La matriz representa para cada vértice sus vértices adyacentes

Cada fila y columna de la matriz corresponde a un vértice del grafo

Para grafos no dirigidos, la matriz siempre será simétrica ($M_{ij} = M_{ji}$)



Matrices de adyacencia

1. **Cuánto nos costará encontrar todos los vértices adyacentes a un vértice?**

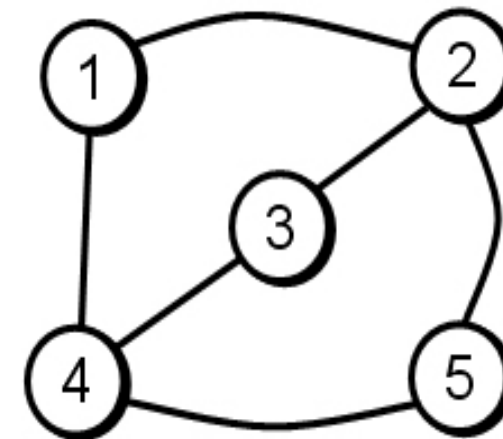
$O(V)$

2. **Saber si dos vértices están conectados?**

$O(1)$

2. **Cuál es el trade-off?**

Uso de memoria $O(V^2)$



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

Mientras más vértices, más espacio se consume. Te imaginas cuánto espacio ocuparía representar una red social como Facebook? Una red social es un grafo denso o disperso? Es un grafo direccionado o no?

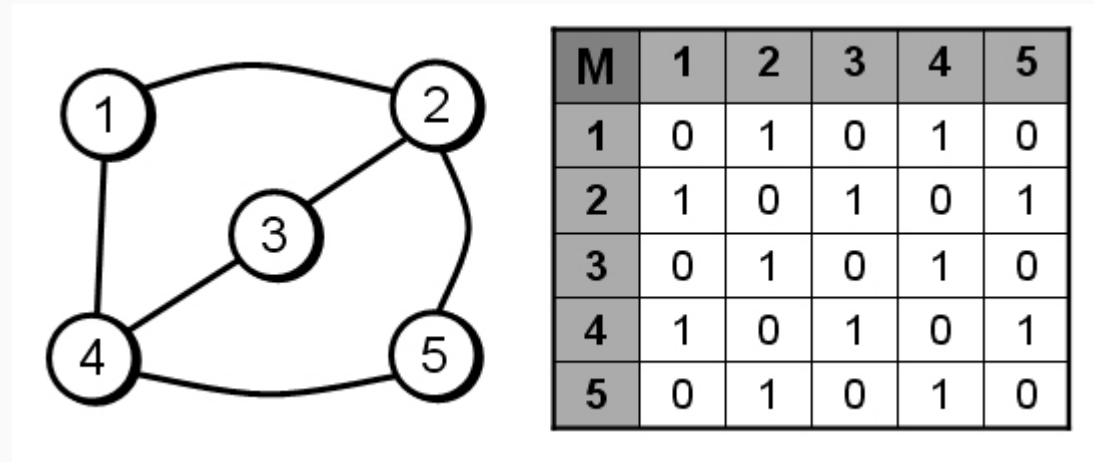
Por tanto, una matriz de adyacencia será útil si el grafo es denso o si la cantidad de vértices es pequeña

Nota: **Grafo denso**, grupo de aristas cercano al máximo posible. **Grafo disperso**, contiene pocas aristas.

Matrices de adyacencia

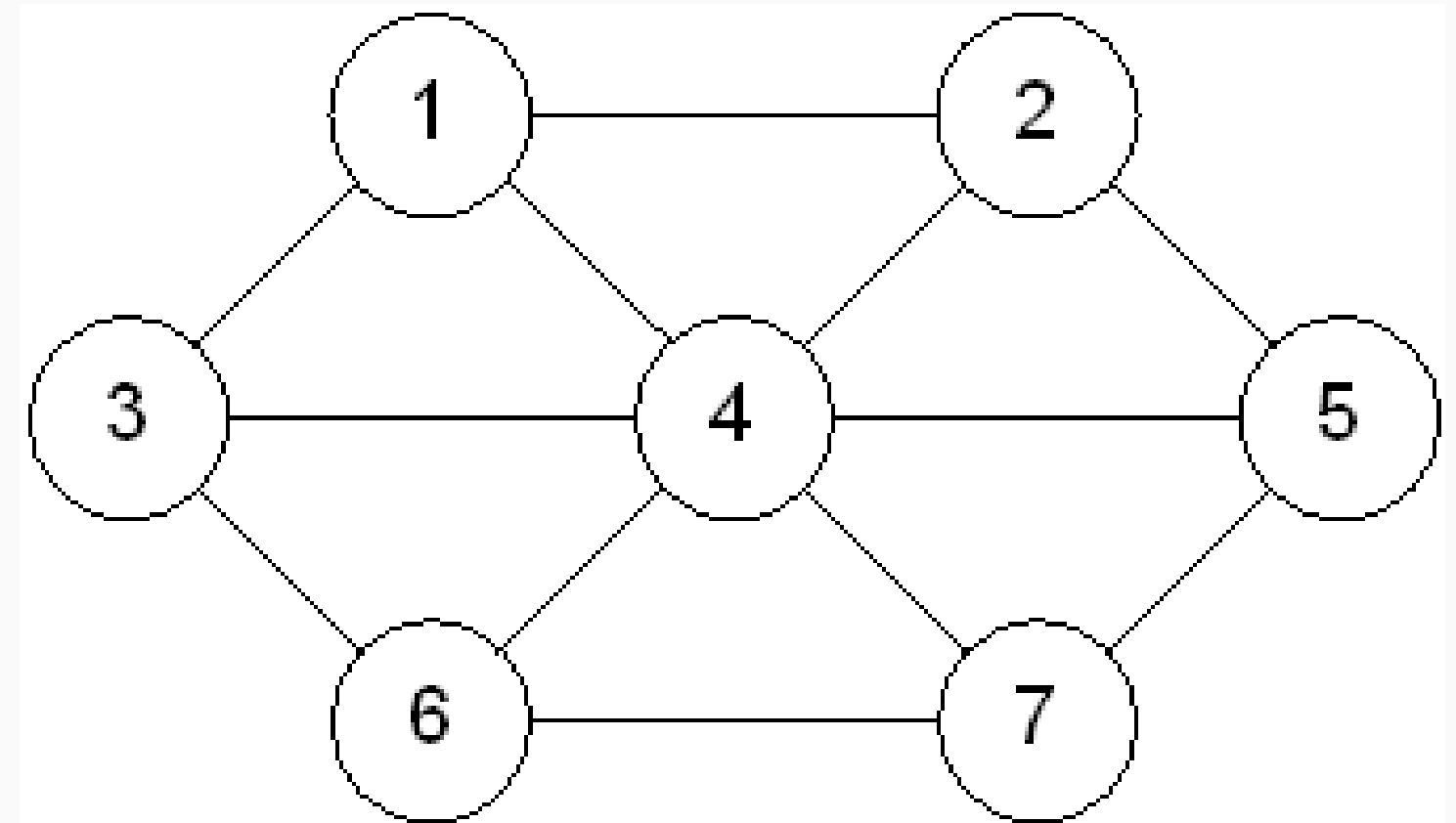
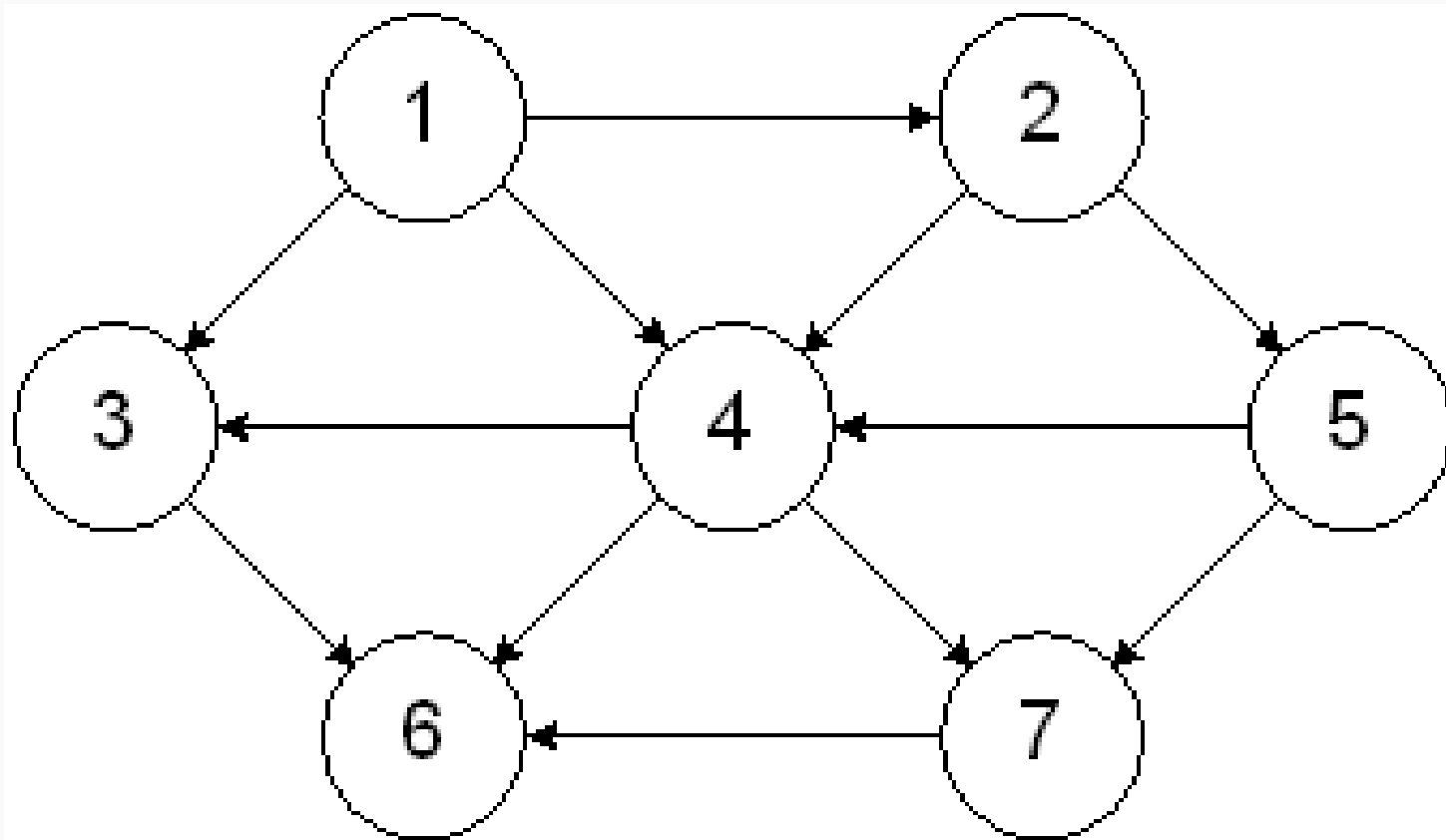
Para utilizar una matriz de adyacencia, debemos conocer el tamaño del grafo primero

Termina siendo una solución costosa, ya que no es muy flexible

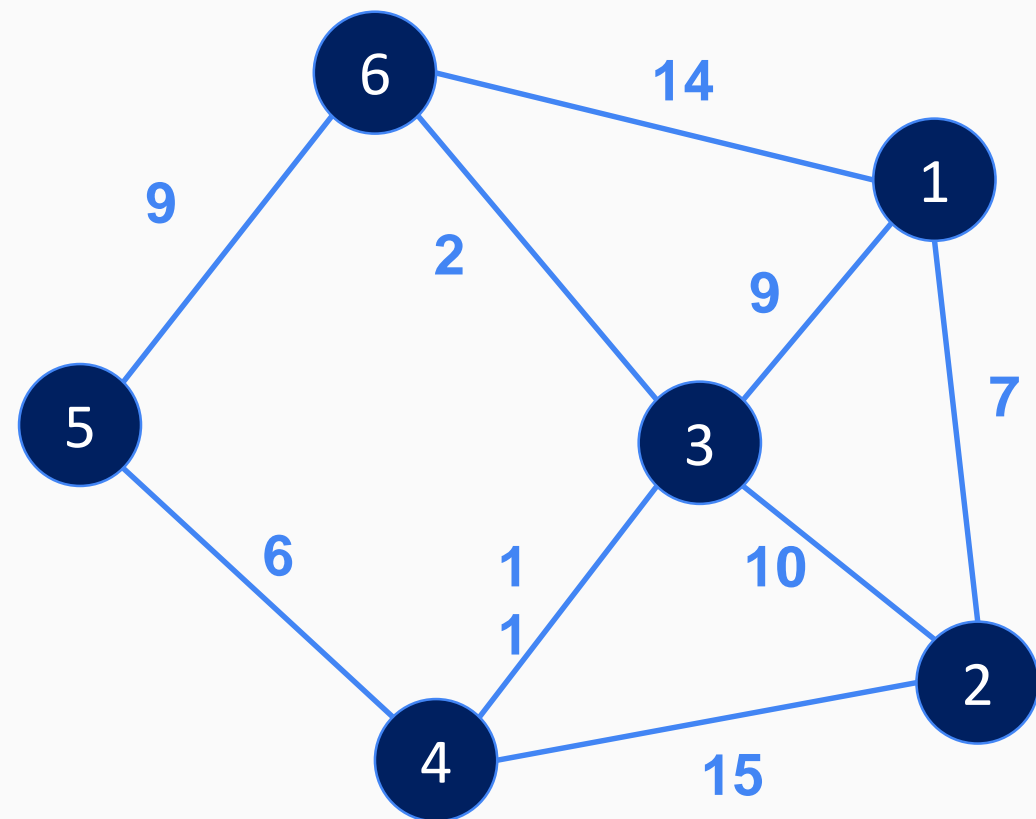


Matrices de adyacencia

Representen ambos grafos en su matriz de adyacencia:

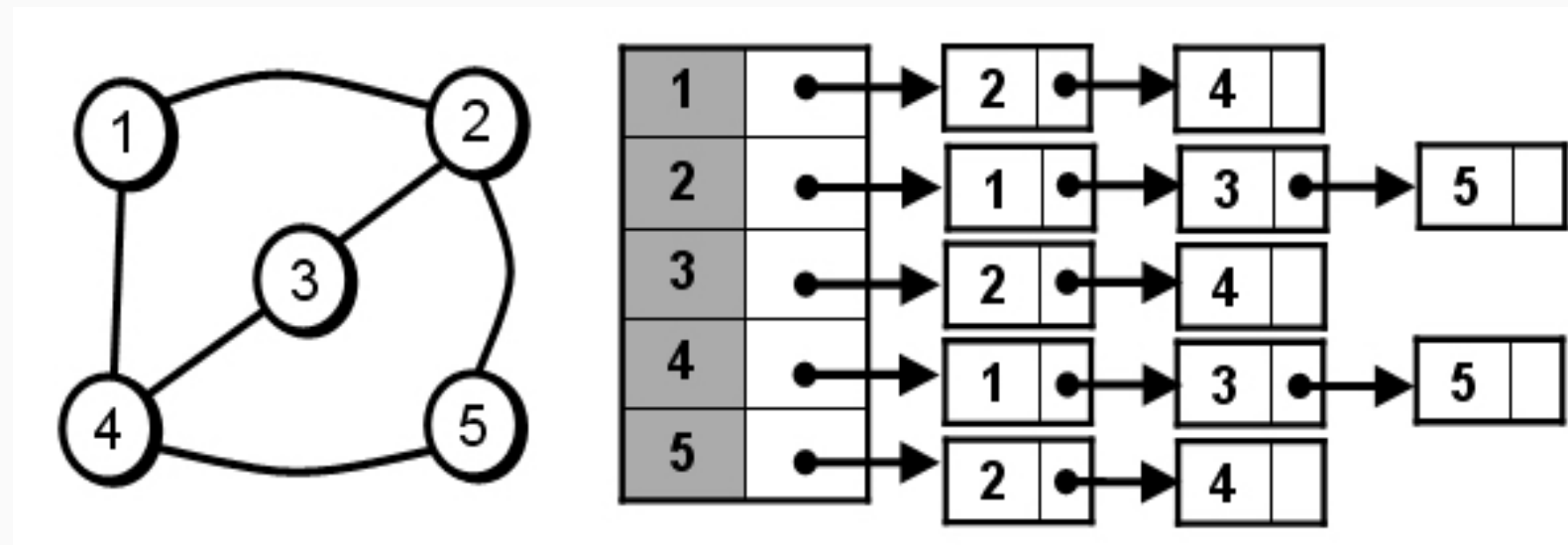


Lista de adyacencia



1	→	6,14	3,9	2,7	
2	→	1,7	3,10	4,15	
3	→	4,11	1,9	2,10	6,2
4	→	5,6	3,11	2,15	
5	→	4,6	6,9		
6	→	3,2	1,14	5,9	

Listas de adyacencia



Un ejemplo de cómo podría implementarse:

```
class Graph{  
    vector<list<string>> ady;  
}
```

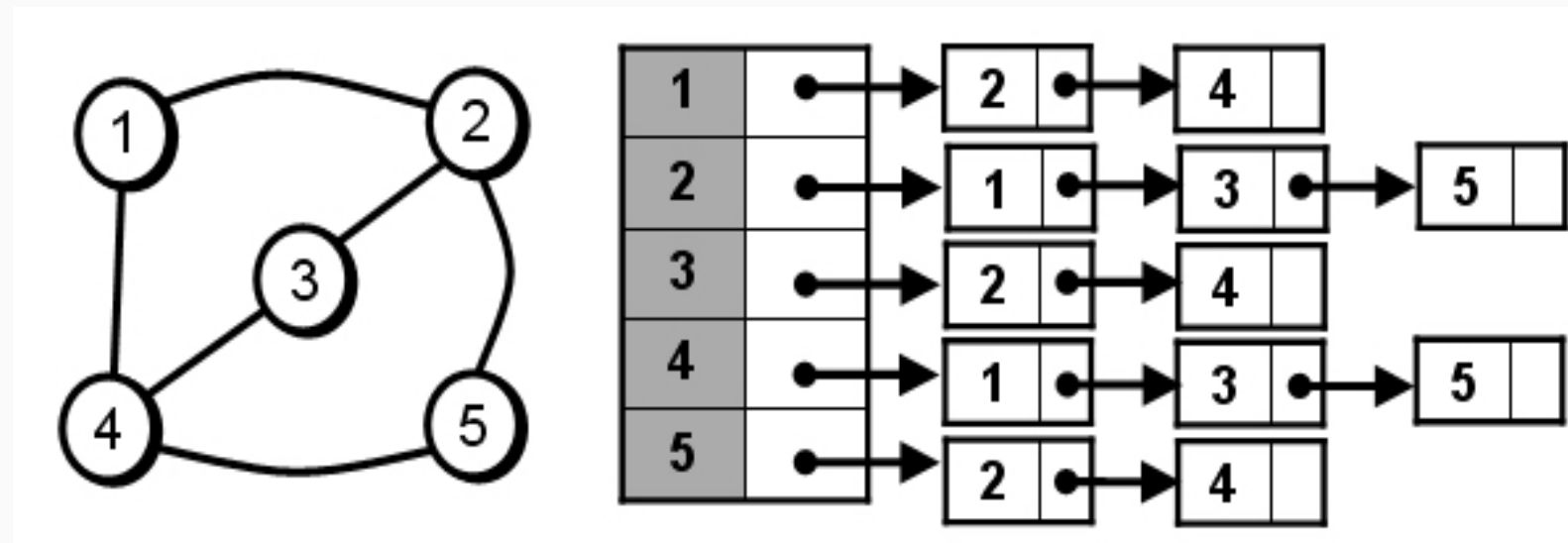
Es una lista de vértices, donde cada vértices tiene una lista de sus vértices adyacentes o vecinos (aristas)

Cuánto espacio requiere?

$O(|V| + |E|)$

Y para grafo ponderado?

Listas de adyacencia



Es una lista de vértices, donde cada vértices tiene una lista de sus vértices adyacentes o vecinos (aristas)

Cuánto espacio requiere?

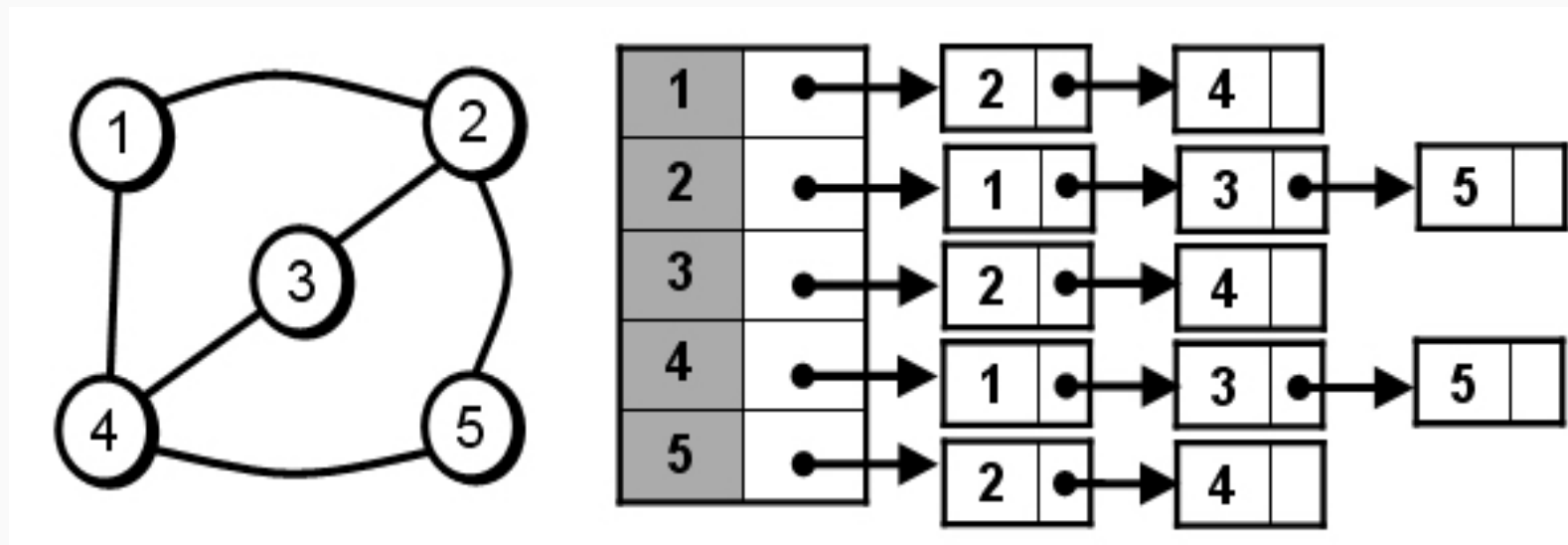
$O(|V| + |E|)$

Un ejemplo de cómo podría implementarse:

```
struct Edge {  
    string startVertex;  
    string endVertex;  
    float weight;  
}  
  
class Graph{  
    vector<list<Edge>> ady;  
}
```

Y si el vértice contiene un objeto?

Listas de adyacencia



Es una lista de vértices, donde cada vértices tiene una lista de sus vértices adyacentes o vecinos (aristas)

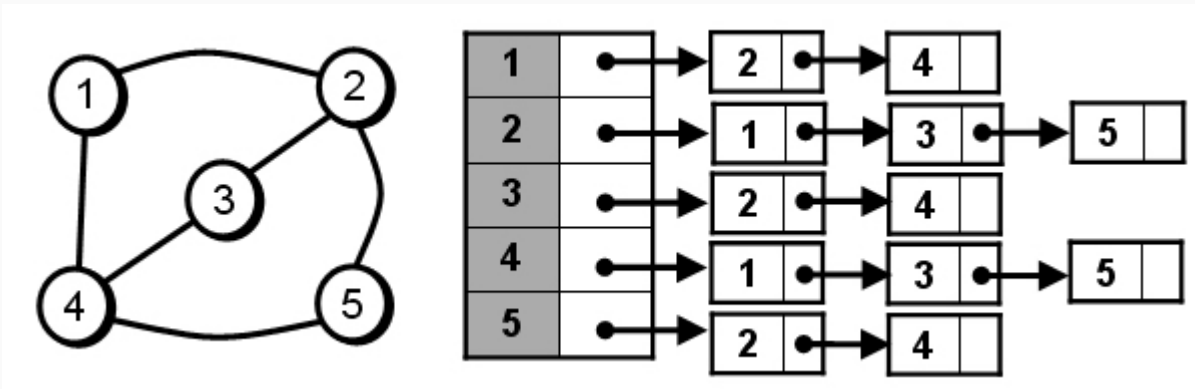
Cuánto espacio requiere?

$O(|V| + |E|)$

Un ejemplo de cómo podría implementarse:

```
struct Edge {  
    Vertex* startVertex;  
    Vertex* endVertex;  
    TE weight;  
}  
  
struct Vertex {  
    TV data;  
    list<Edge*> edges;  
}  
  
class Graph{  
    vector<Vertex> vertices;  
}
```

Listas de adyacencia



```
template <typename V, typename E>
class Graph
{
public:
    typedef Graph<V, E> self;
    typedef Node<self> Node;
    typedef Edge<self> Edge;
    typedef vector<Node *> NodeSeq;
    typedef list<Edge *> EdgeSeq;
private:
    NodeSeq nodes;
    EdgeSeq edges;
}
```

```
template <typename G>
class Node
{
public:
    typedef typename G::V V;
    typedef typename G::Edge Edge;
    typedef typename G::EdgeSeq EdgeSeq;
private:
    EdgeSeq edges;
    V data;
}

-----

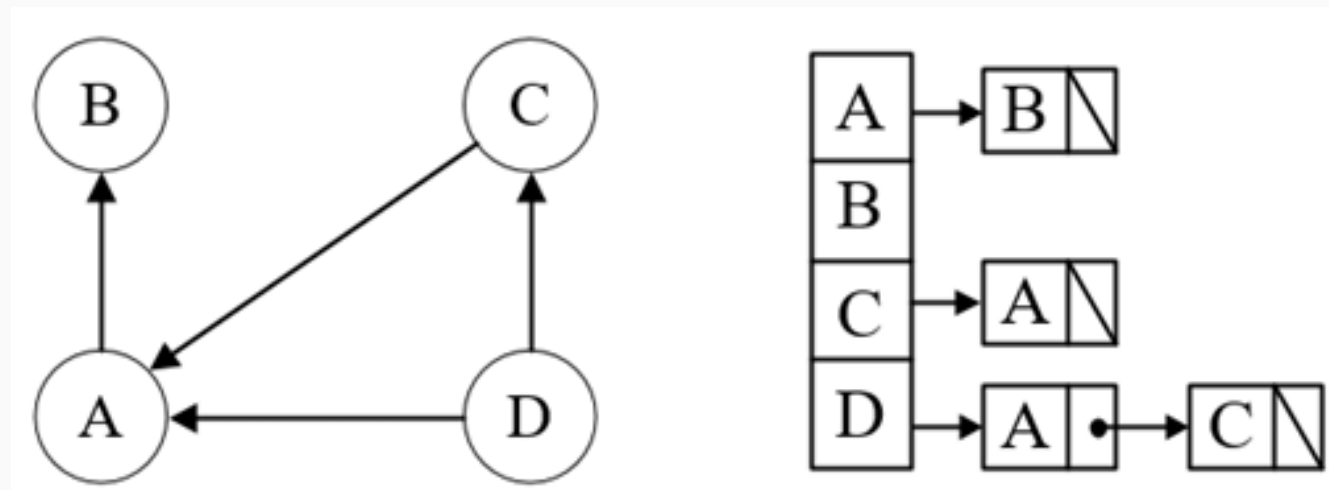
template <typename G>
class Edge
{
public:
    typedef typename G::E E;
    typedef typename G::Node Node;
private:
    Node *nodes[2];
    E weight;
}
```


Listas de adyacencia

Cómo sabrían cuántas aristas llegan a un nodo en un grafo dirigido?

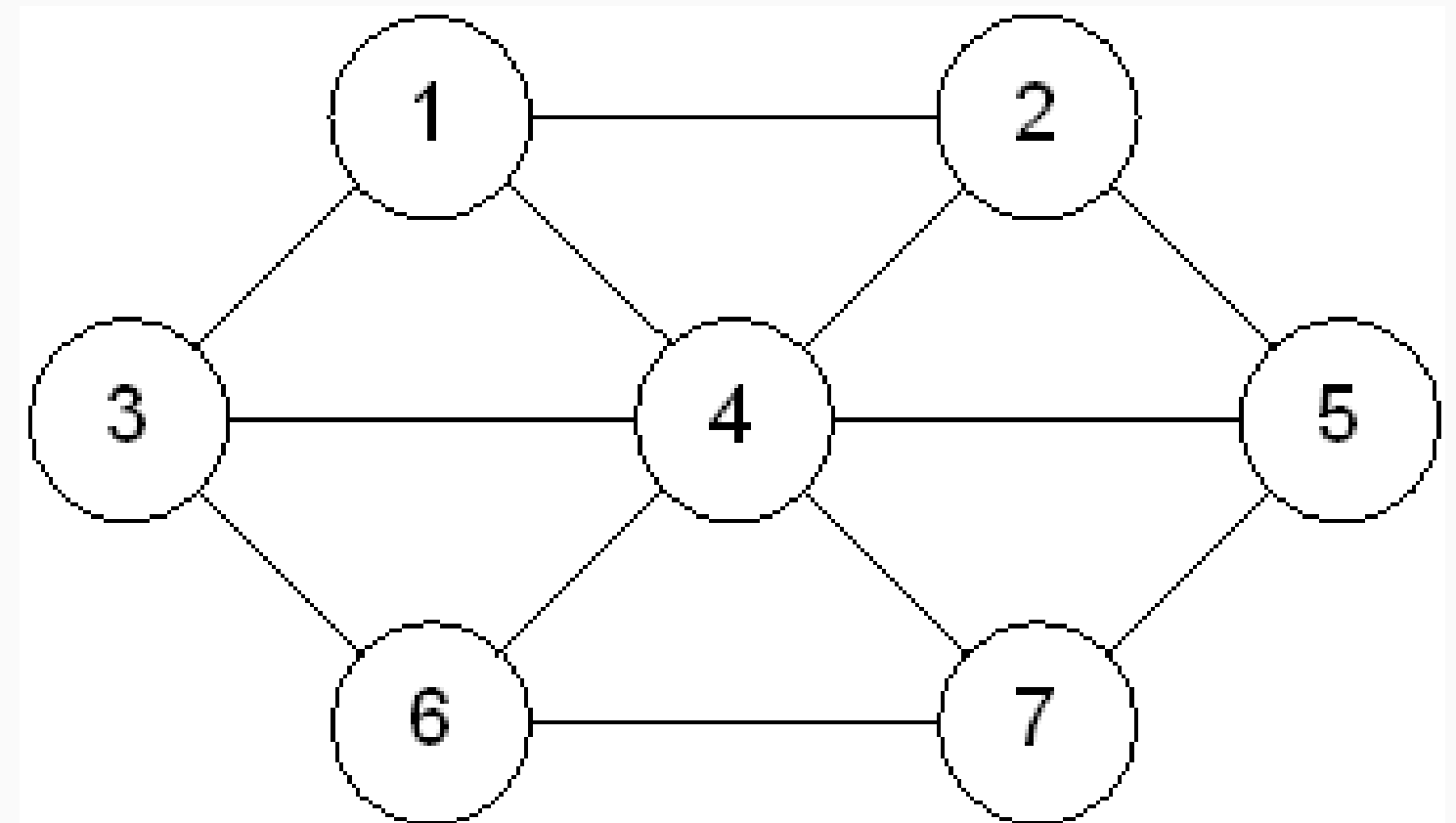
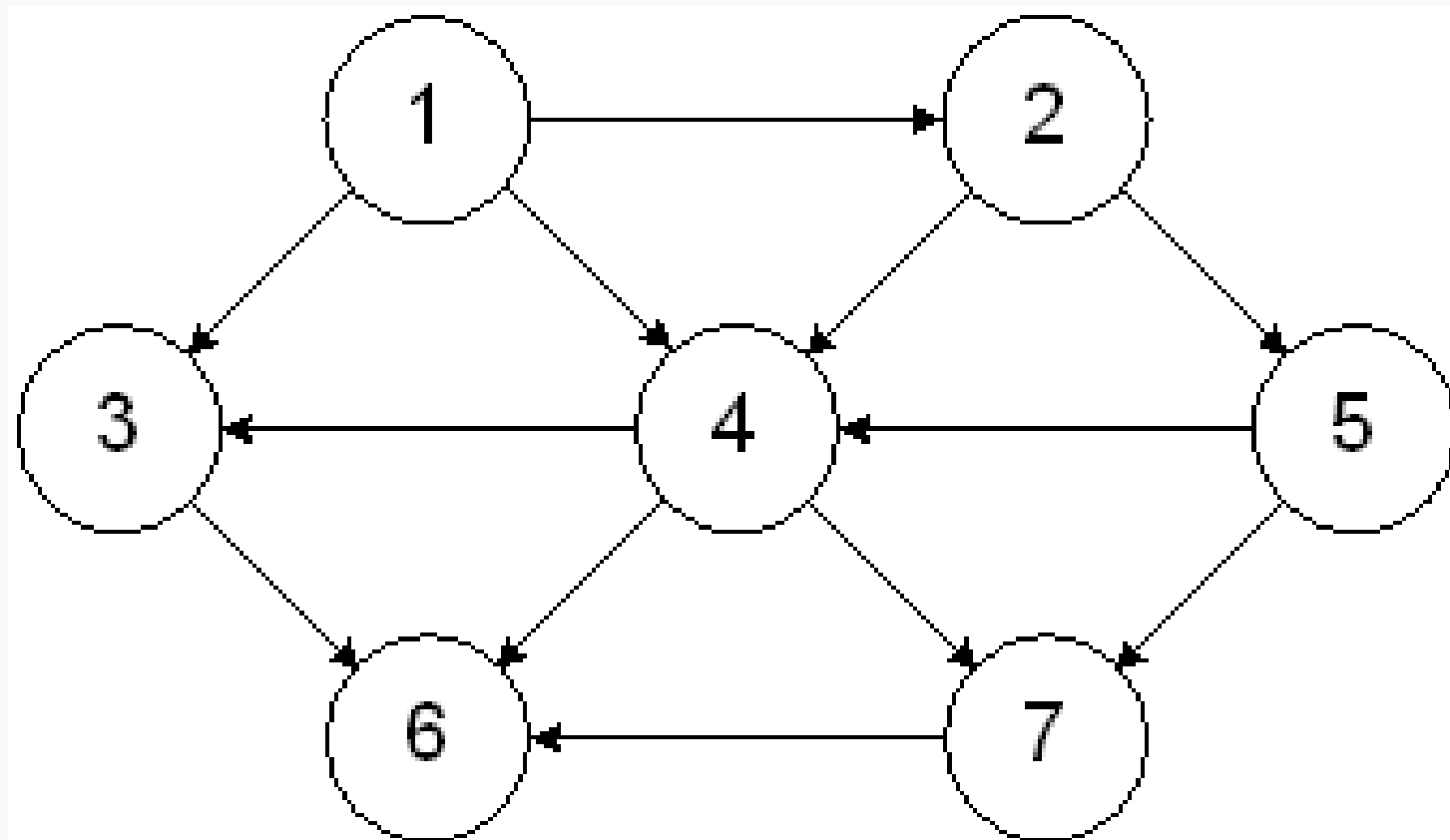
Cuánto toma conocer todos los vértices a los que se llega directamente desde otro vértice?

$O(|E|)$



Listas de adyacencia

Representen ambos grafos en su lista de adyacencia:

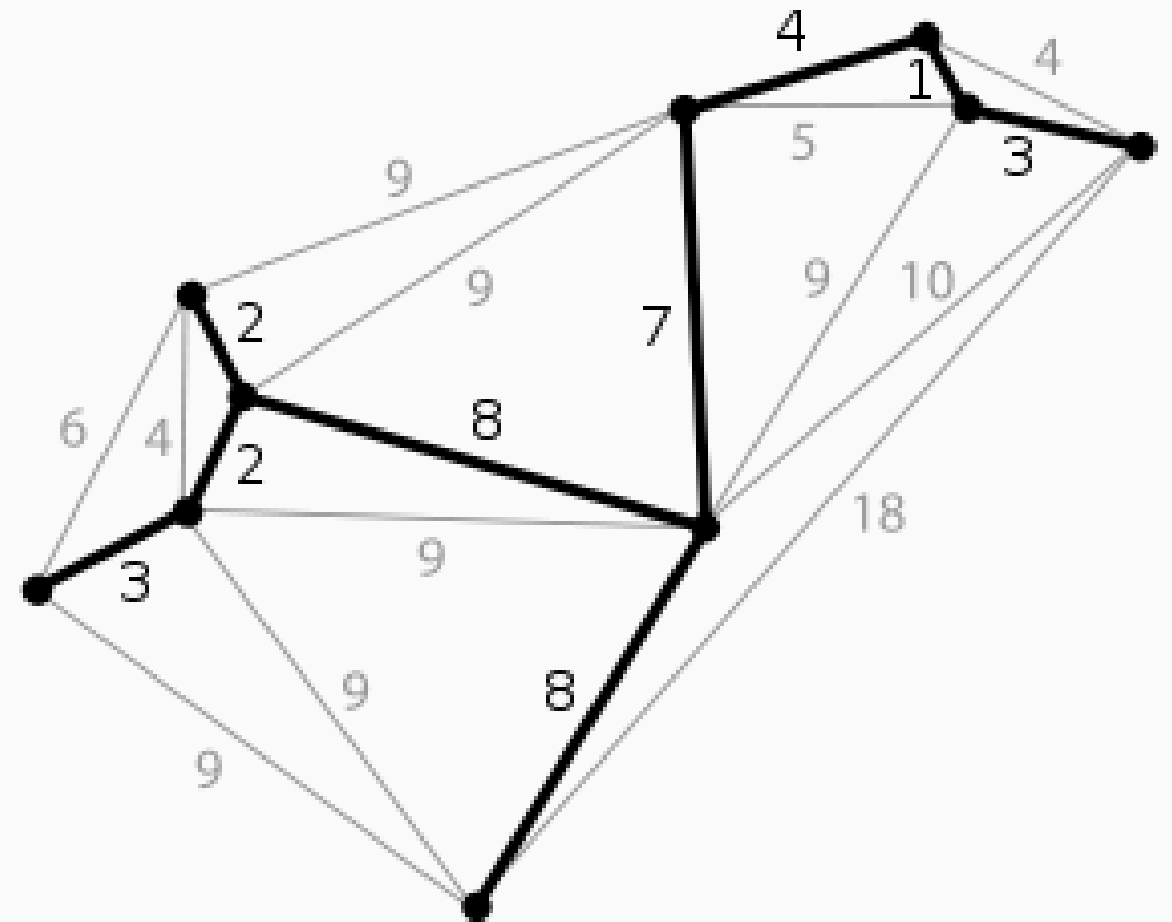


Árbol de mínima expansión (MST)

Solo se dan en grafos no dirigidos, pueden ser ponderados o no

Es un subconjunto de aristas conectadas que unen todos los vértices, sin ciclos y con el menor peso posible en la sumatoria de las aristas (puede haber más de uno)

En el caso de grafos dirigidos se tienen otros algoritmos. **¿Cómo crearían uno?**



Arborescencia

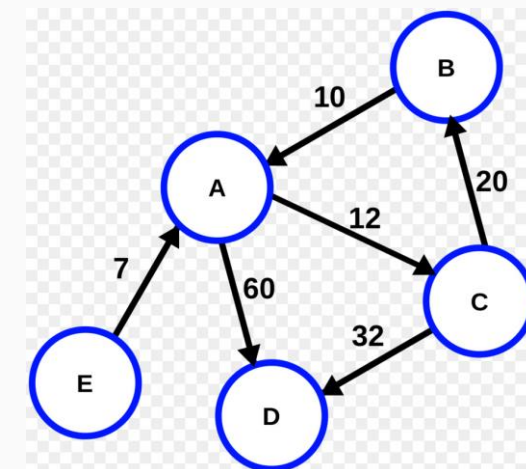
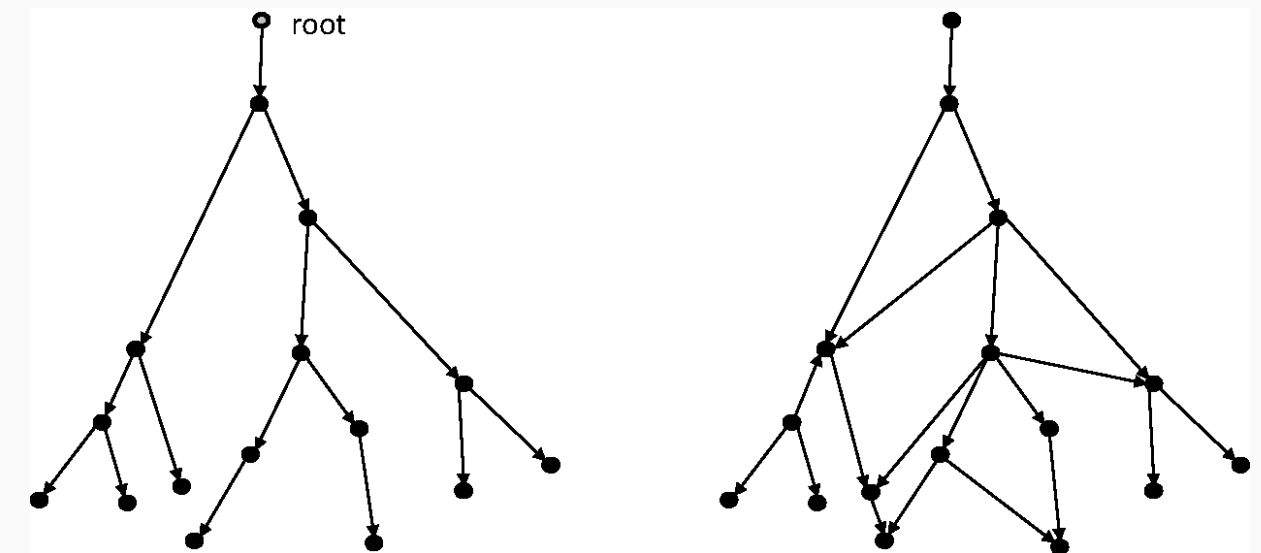
Los grafos dirigidos no tienen un árbol de mínima expansión

Arborescencia (arborescence) es un grafo dirigido en el cual un vértice u llamado raíz, y cualquier otro vértice v , tienen un solo camino dirigido.

Entonces todos los vértices tienen un camino alejándose de la raíz

Muchos autores definen este tipo de grafo como un árbol dirigido de expansión de un digrafo

Toda arborescencia es un grafo dirigido acíclico (DAG), pero no todo DAG es una arborescencia



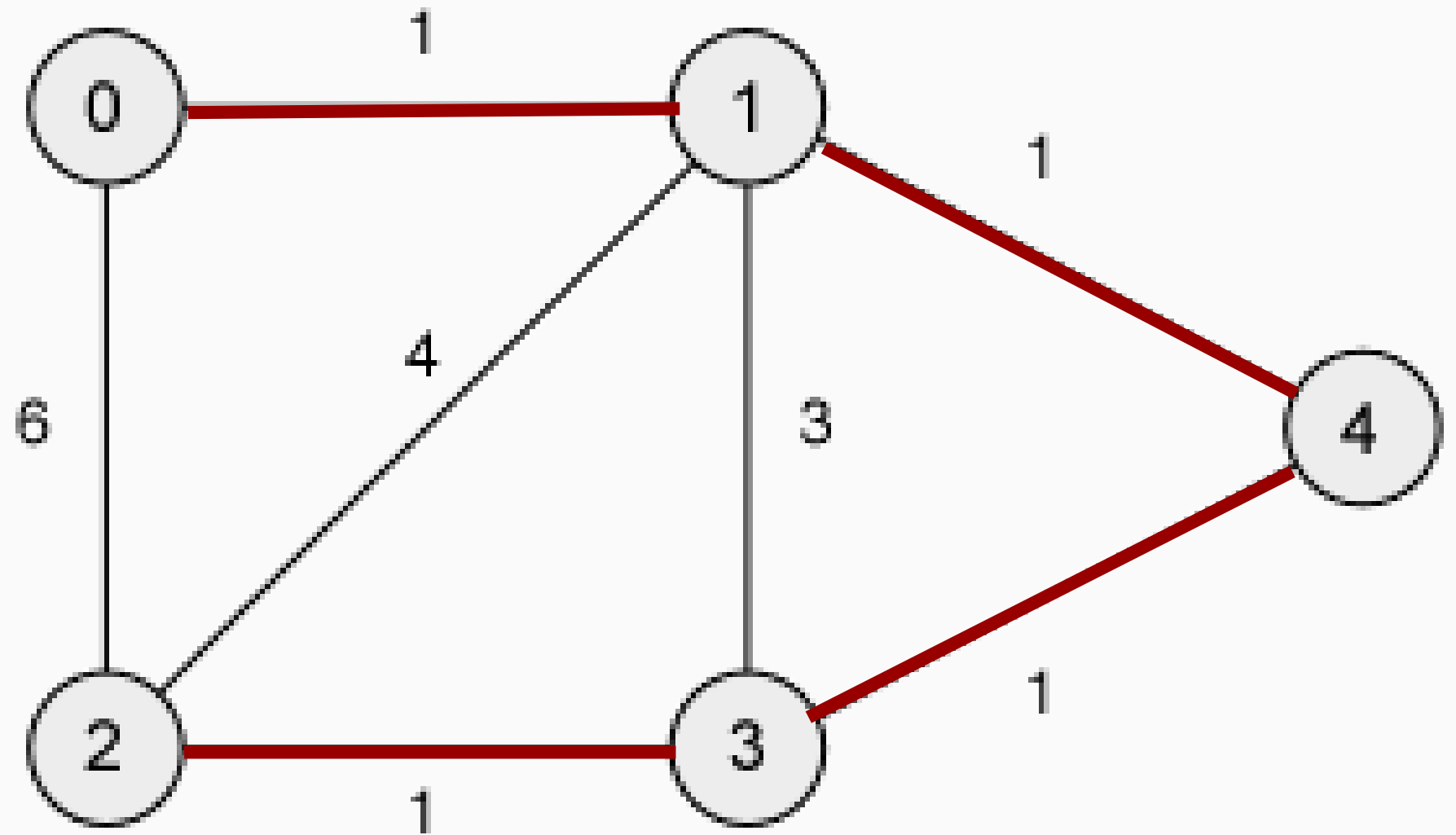
Prim

Es un algoritmo goloso para encontrar el MST en un grafo conexo no dirigido

El algoritmo empieza en un vértice arbitrario

Tiempo de ejecución?

$O(|E| \log |V|)$ Mejorado

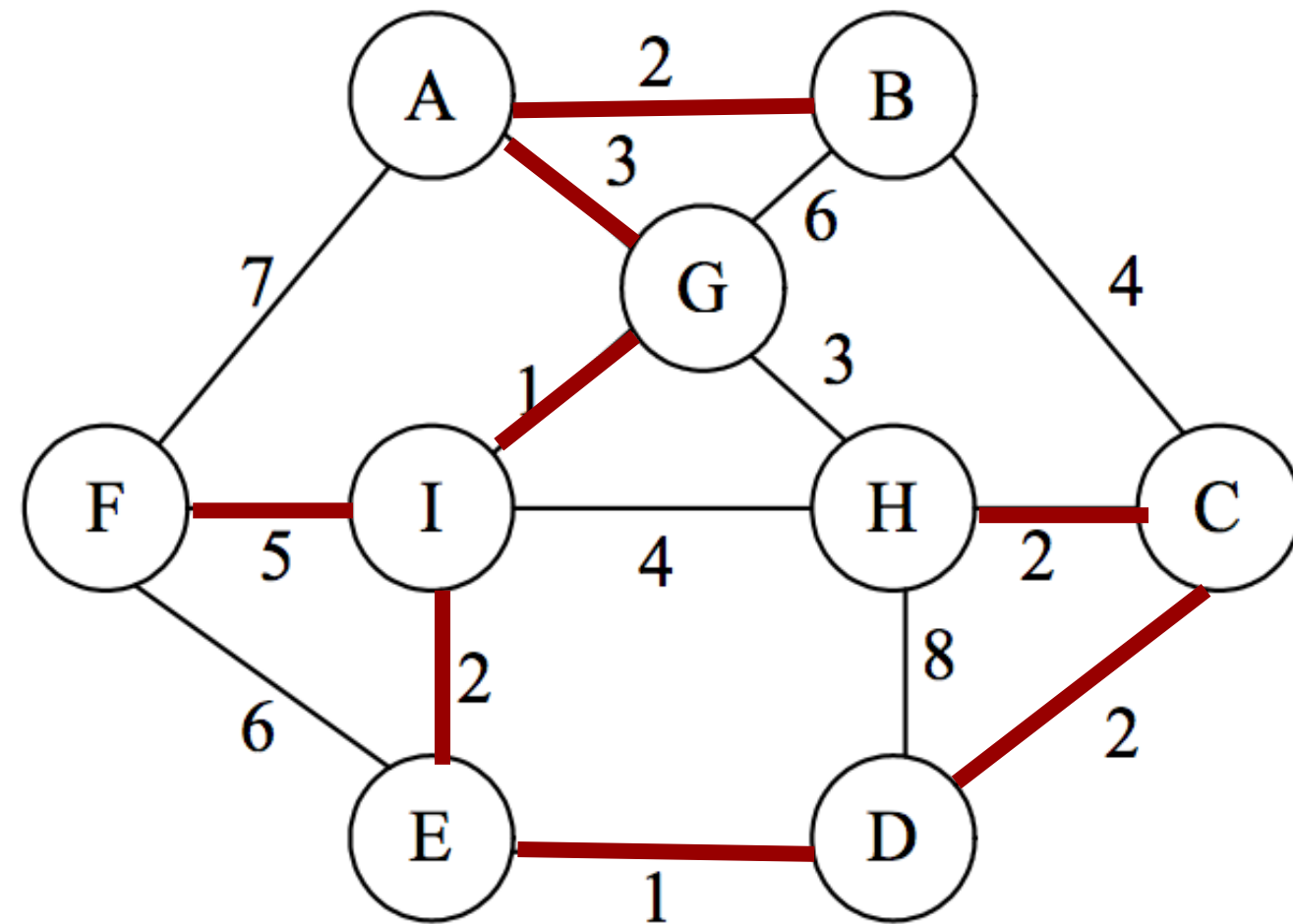


Prim

El árbol se va expandiendo de acuerdo a los vértices visitados

Cada expansión contendrá la arista de menor peso que sea incidente a uno de los vértices ya visitados.

Lo pasos se repiten hasta visitar todos los vértices



Prim

PRIM (Grafo G, nodo_fuente s)

// Inicializamos todos los nodos del grafo. La distancia la ponemos a infinito y el padre de cada nodo a NULL

// Encolamos, en una cola de prioridad donde la prioridad es la distancia, todas las parejas <nodo,distancia> del grafo

por cada u en V[G] hacer

 distancia[u] = INFINITO

 padre[u] = NULL

 Añadir(cola,<u,distancia[u]>)

distancia[s]=0

mientras cola != 0 do

 // OJO: Se entiende por mayor prioridad aquel nodo cuya distancia[u] es menor.

 u = extraer_minimo(cola) //devuelve el minimo y lo elimina de la cola.

 por cada v adyacente a 'u' hacer

 si ((v ∈ cola) && (distancia[v] > peso(u, v)) entonces

 padre[v] = u

 distancia[v] = peso(u, v)

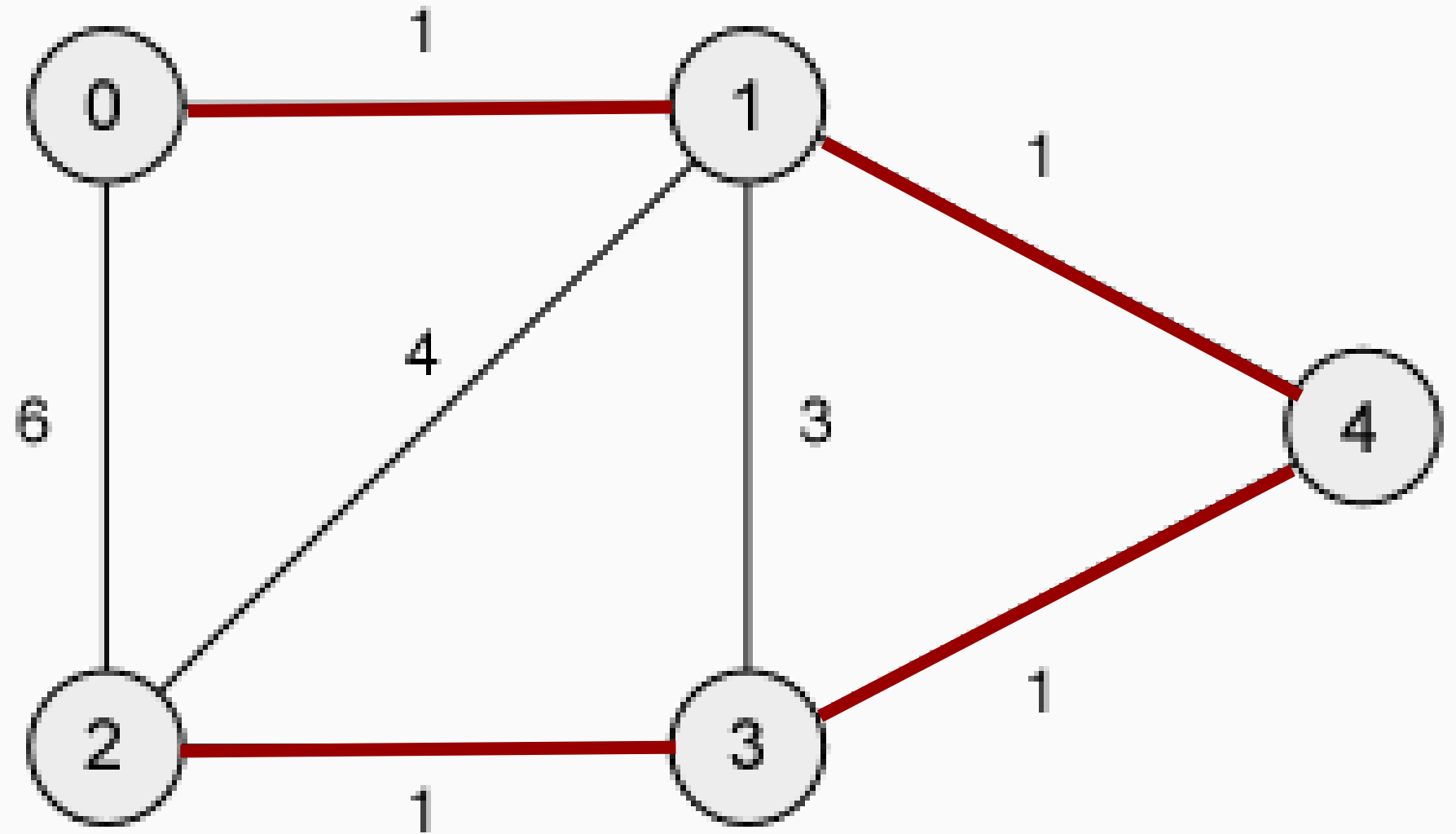
 Actualizar(cola,<v,distancia[v]>)

Kruskal

Al igual que Prim es un algoritmo voraz para encontrar el MST en un grafo conexo no dirigido

Cada vértice es un árbol inicialmente.

Se van agregando vértices que unen los árboles

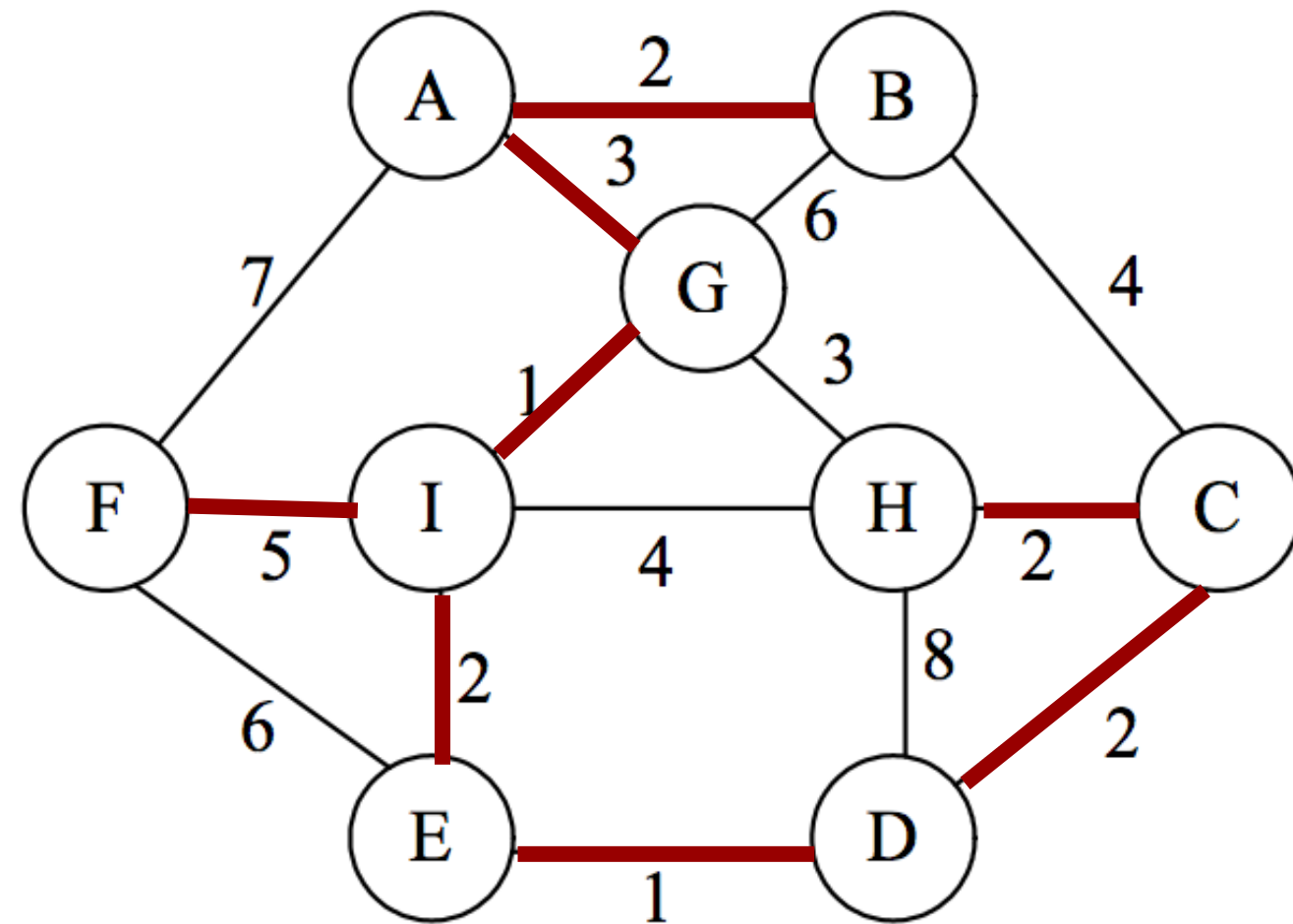


Kruskal

Si es un grafo no conexo,
entonces busca un bosque
expandido mínimo

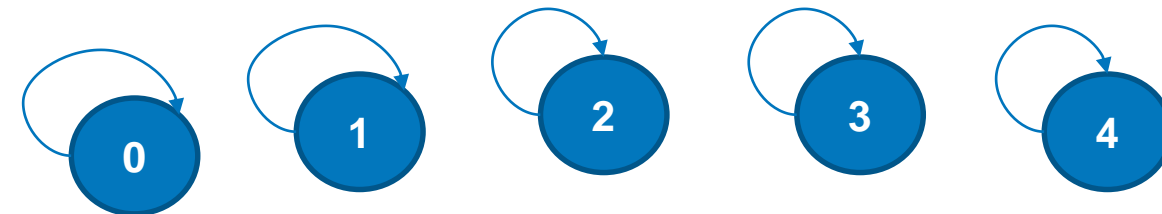
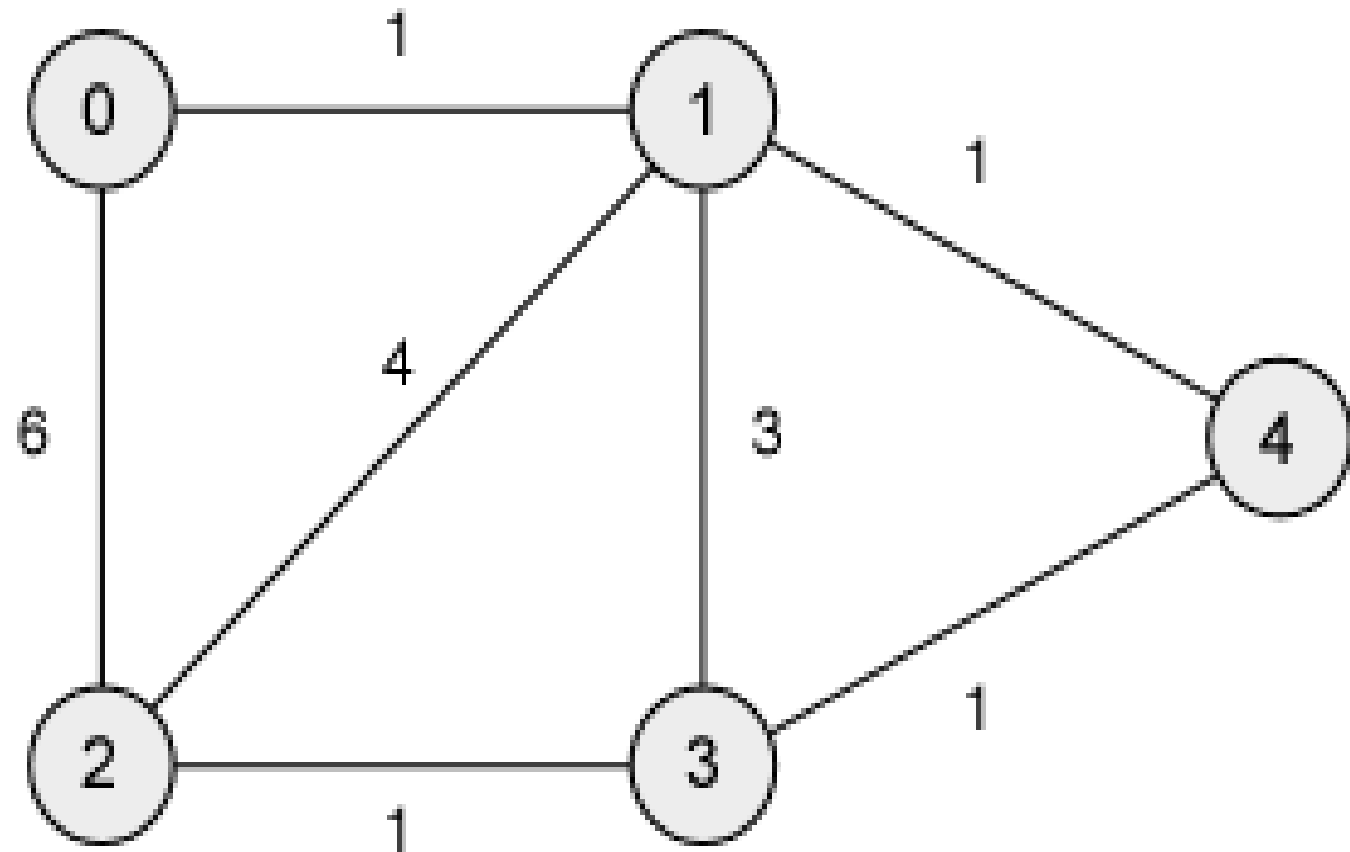
No se debe formar un ciclo

**¿Prim y Kruskal siempre dan una
respuesta óptima?**



Kruskal

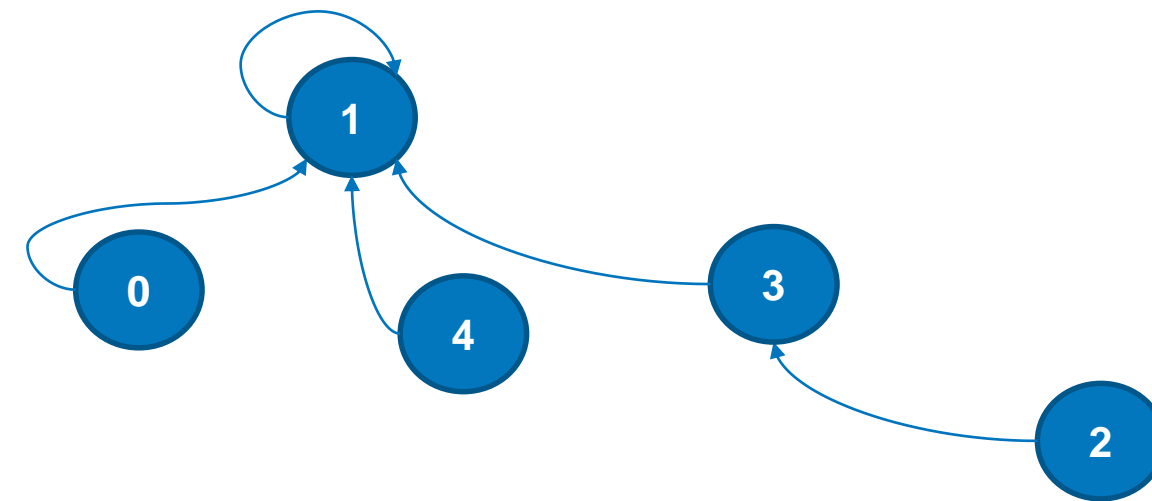
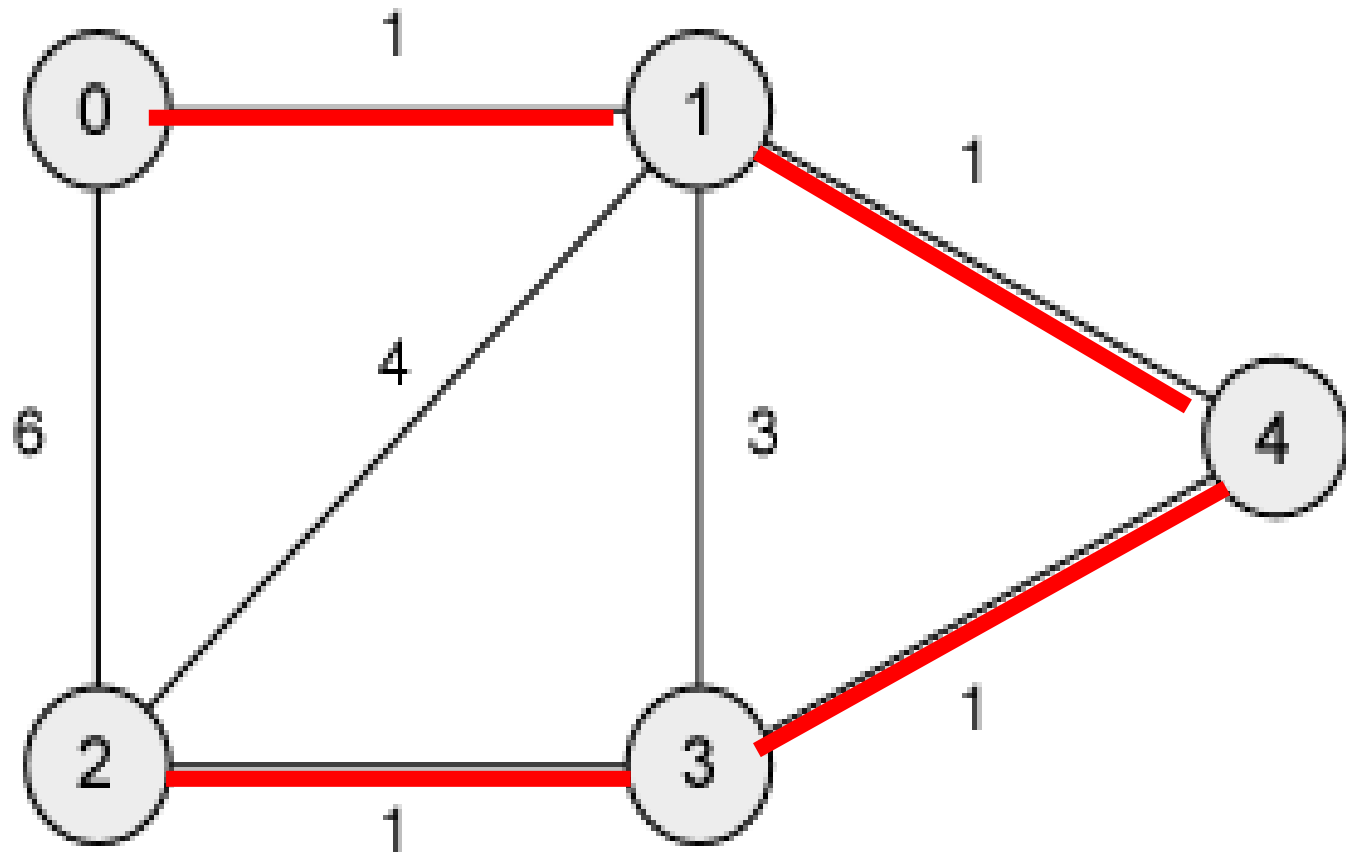
- 1- Ordenar las aristas en función de su peso
- 2- Aplicar MakeSet a todos los vértices
- 3- Recorrer todas las aristas y agregar al árbol MST
 - 3.1 - Agregar aristas $e=\{v1, v2\}$ a la solución si cumple:
 $\text{Find}(v1) \neq \text{Find}(v2)$
 - 3.2 - Luego Unir $v1$ y $v2$
 $\text{Union}(v1, v2)$



DisjoinSet:
MakeSet(x)
Find(x)
Union(x,y)

Kruskal

- 1- Ordenar las aristas en función de su peso
- 2- Aplicar MakeSet a todos los vértices
- 3- Recorrer todas las aristas y agregar al árbol MST
 - 3.1 - Agregar aristas $e=\{v1, v2\}$ a la solución si cumple:
 $\text{Find}(v1) \neq \text{Find}(v2)$
 - 3.2 - Luego Unir $v1$ y $v2$
 $\text{Union}(v1, v2)$

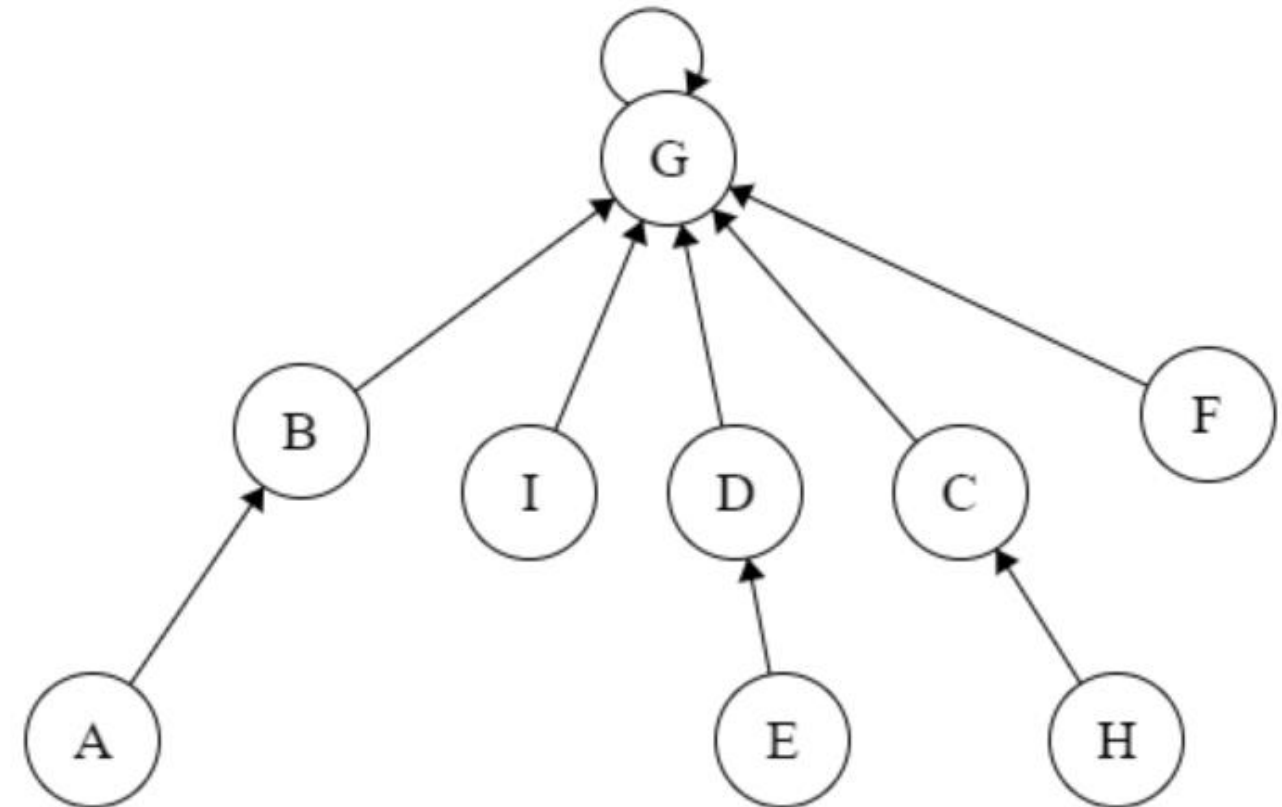
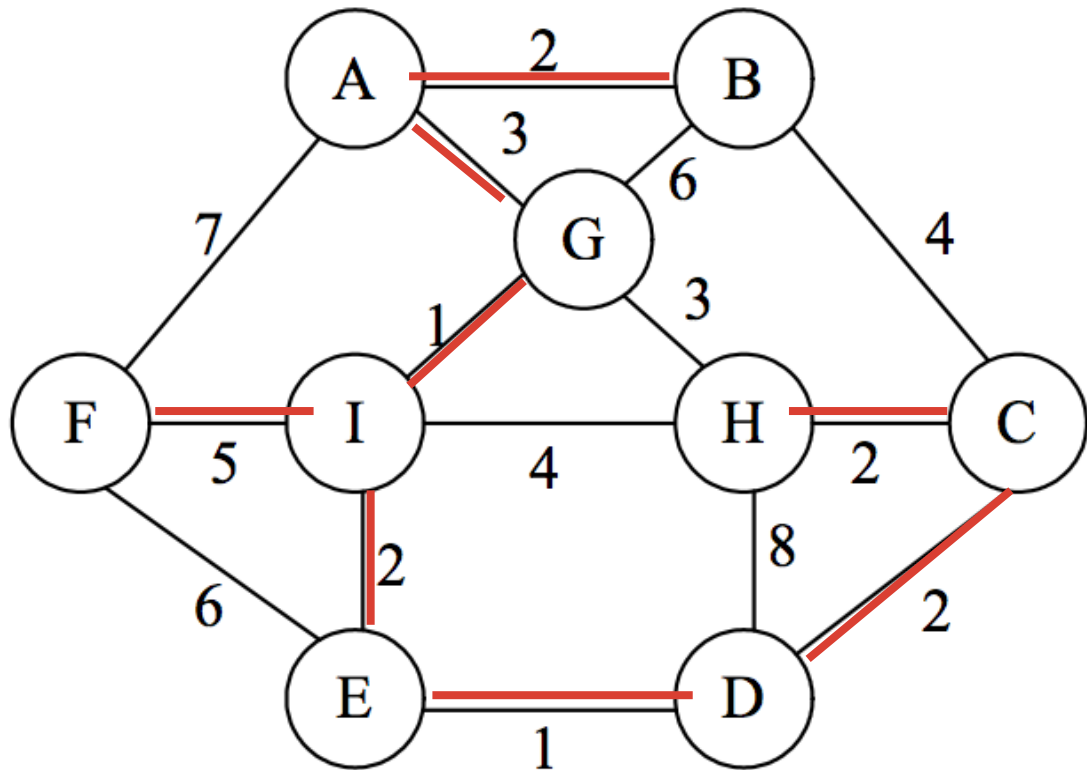


DisjoinSet:
MakeSet(x)
Find(x)
Union(x,y)

Kruskal

- 1- Ordenar las aristas en función de su peso
- 2- Aplicar MakeSet a todos los vértices
- 3- Recorrer todas las aristas y agregar al árbol MST
 - 3.1 - Agregar aristas $e=\{v1, v2\}$ a la solución si cumple:
Find(v1) \neq Find(v2)
 - 3.2 - Luego Unir v1 y v2
Union(v1, v2)

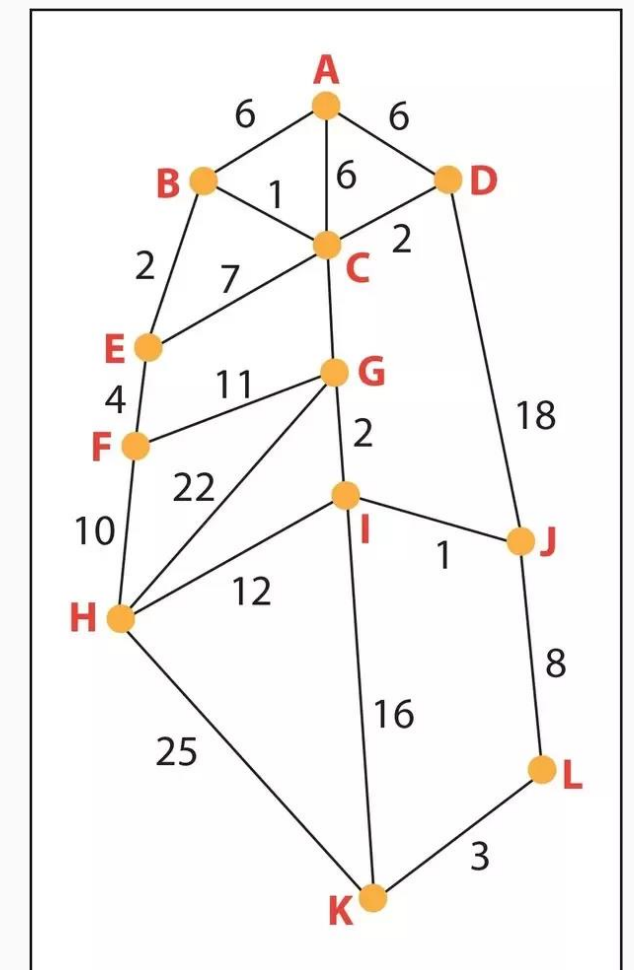
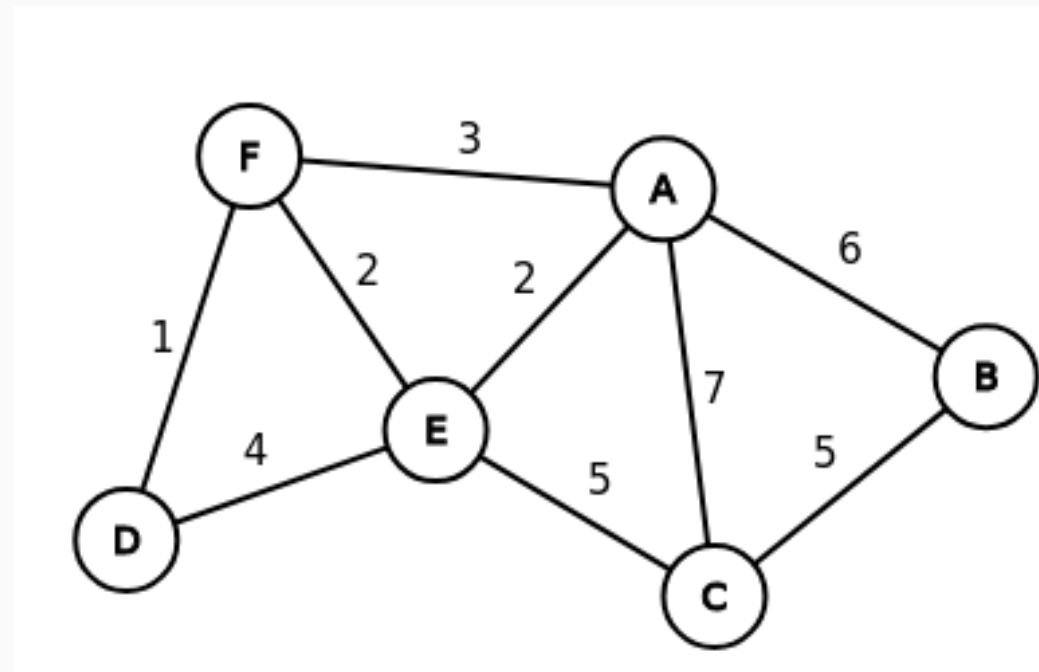
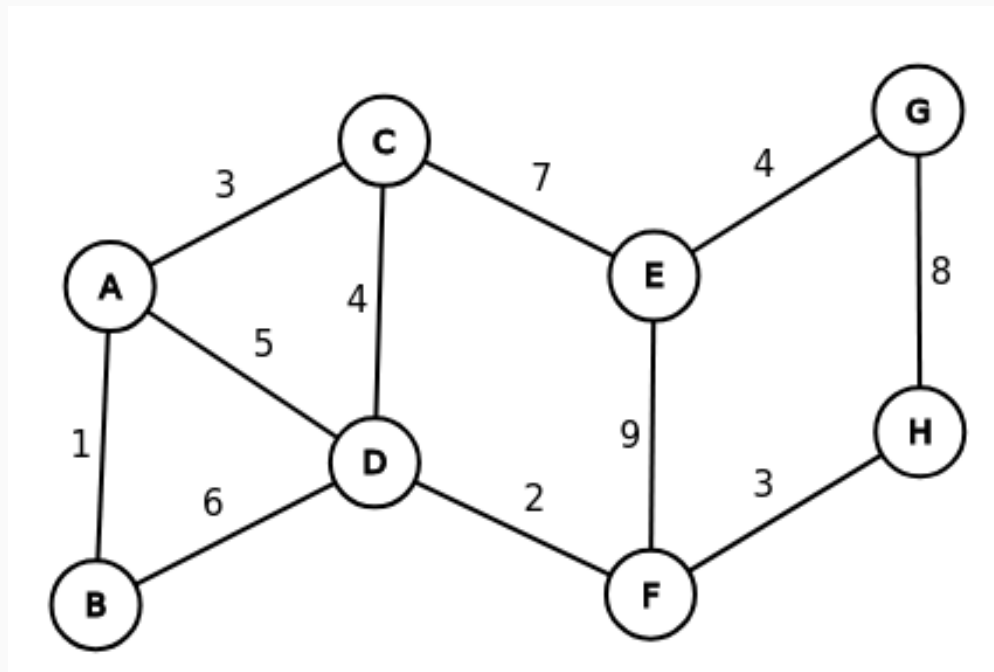
DisjoinSet:
MakeSet(x)
Find(x)
Union(x,y)



Conjuntos disjuntos (disjoint sets)

Ejercicios

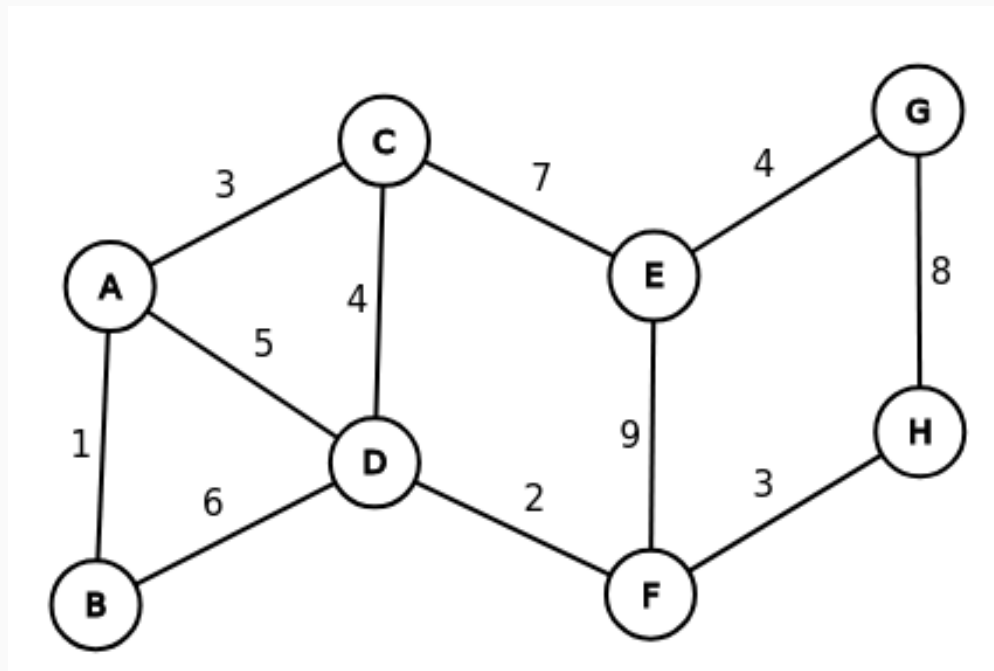
Apliquen el algoritmo de Kruskal a los siguientes grafos utilizando disjoint sets:



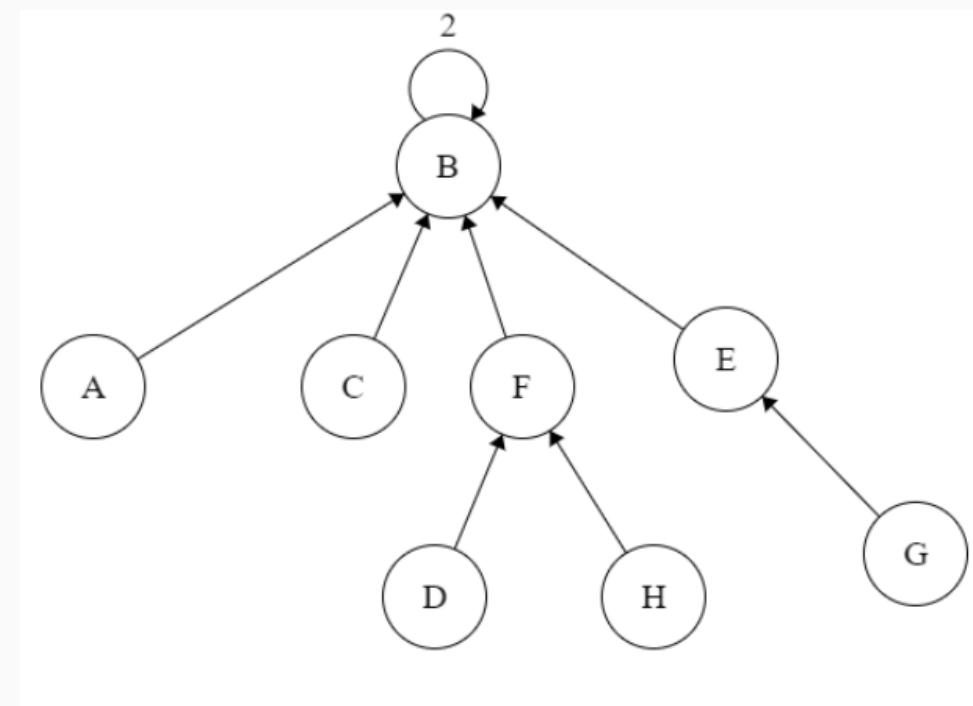
Conjuntos disjuntos (disjoint sets)

Ejercicios

Apliquen el algoritmo de Kruskal a los siguientes grafos utilizando disjoint sets:

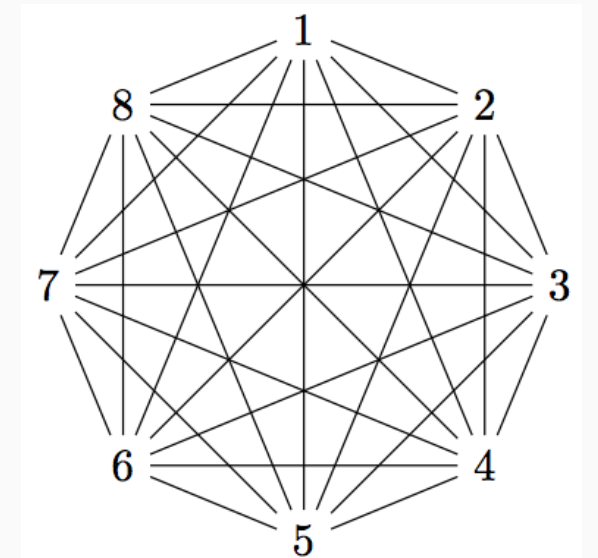
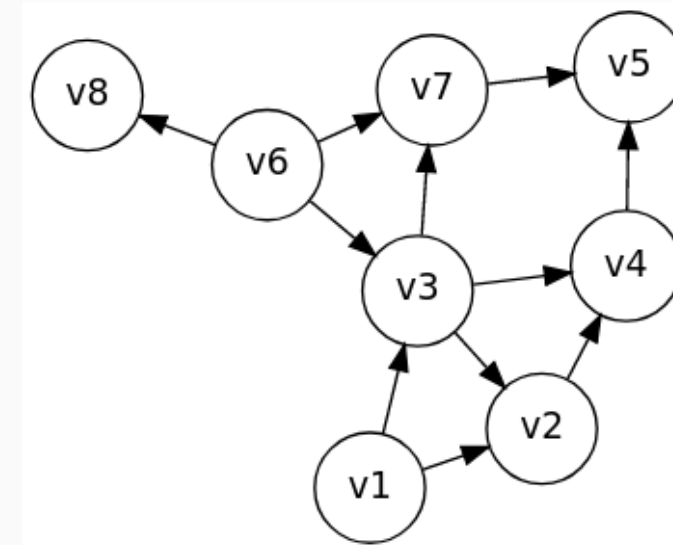
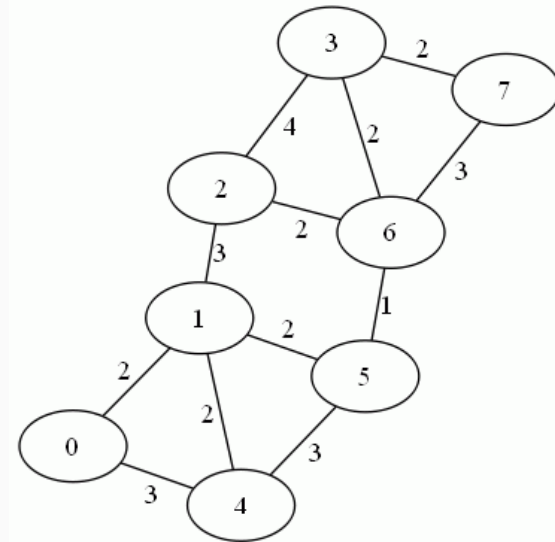
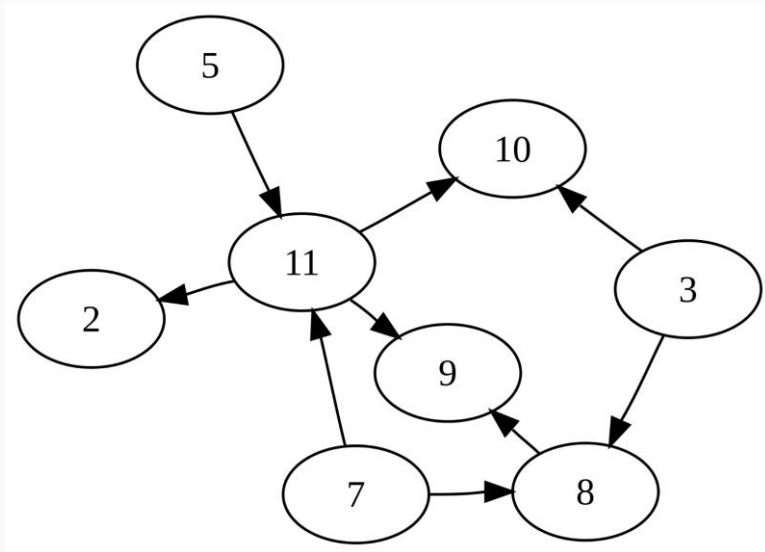


$\{A,B\}\{D,F\}\{A,C\}\{F,H\}\{C,D\}\{E,G\}\{C,E\}$



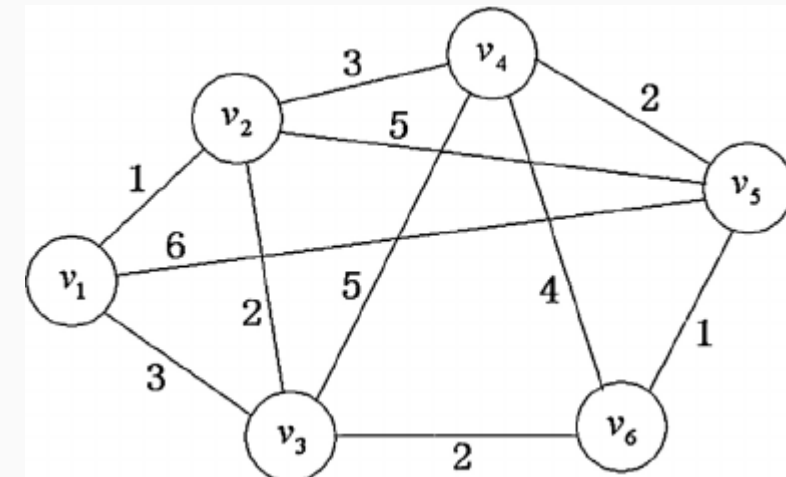
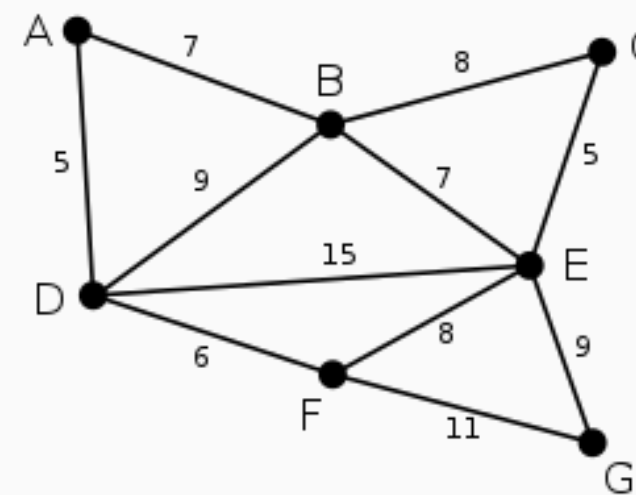
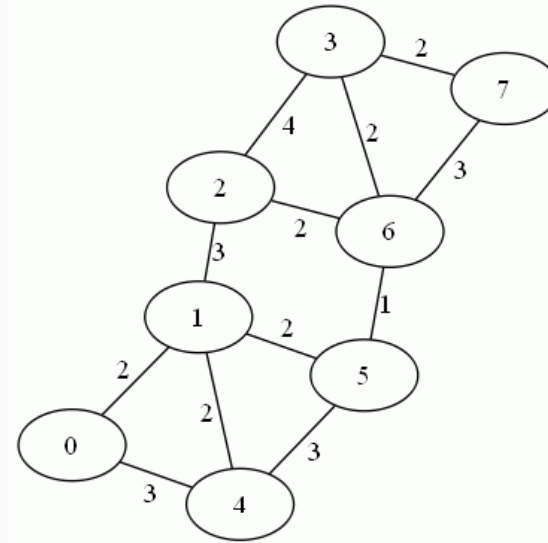
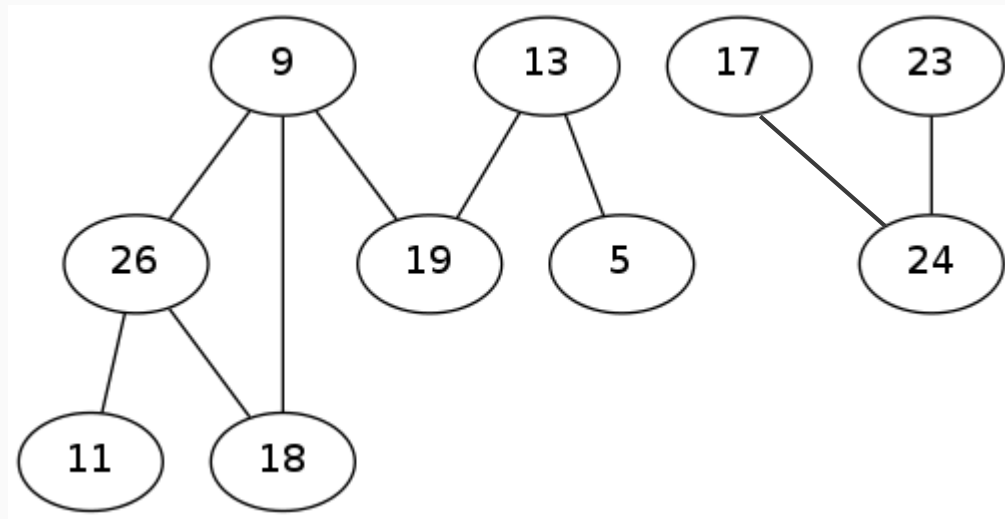
Ejercicios

Generar las matrices y listas de adyacencia de los siguientes grafos:



Ejercicios

Obtenga el MST aplicando Prim y Kruskal en los siguientes grafos (si se puede)



Ejercicios

Por qué Prim y Kruskal no se pueden usar en grafos dirigidos?

Fallan porque asumen que todos los vértices tienen caminos a todos los otros vértices, lo cual solo es válido para grafos no dirigidos

Cuál es la diferencia de usar Prim o Kruskal?

Kruskal puede encontrar el bosque de mínima expansión en caso de grafos no conexos. Prim llegaría a un punto en el cual no podría avanzar

Welcome to Algorithms and Data Structures! - CS2100