

---

---

# Ayudatón I3

---

---

# Outline

- Ordenamiento
- Heaps
- Estrategias Algorítmicas
- DFS y Kosaraju
- Kruskal
- Prim
- Árboles
- Dijkstra + BellmanFord

# Ordenamiento

# II 2019-2 P2

Tienes a tu disposición 4 ejecutables compilados E1, E2, E3 y E4 correspondientes a 4 algoritmos de ordenación: **InsertionSort, SelectionSort, QuickSort con pivote aleatorio, QuickSort con pivote del primer elemento**. El problema es que no sabes cual archivo corresponde a que algoritmo. Para descubrir a qué algoritmo corresponde cada archivo creas 3 arreglos con 1 millón de números para ordenar: **El primer arreglo tiene los números ya ordenados, el segundo tiene los números de manera aleatoria y el tercero tiene los números ordenados en el orden inverso.**

# II 2019-2 P2

Al ejecutar cada programa con cada set de números obtienes la siguiente tabla de tiempos (en segundos):

Archivo	Datos Ordenados	Datos Aleatorios	Ordenados a la inversa
E1	44.88	48.72	43.35
E2	40.87	9.97	43.22
E3	7.35	25.47	46.11
E4	10.05	9.81	10.56

Diga cuál archivo corresponde a qué algoritmo y justifique cómo llegó a esa conclusión a partir de los datos obtenido

# II 2019-2 P2

Tenemos que evaluar los mejores y peores casos de cada algoritmo/ejecutable

## **InsertionSort**

- tiempo lineal cuando los datos están ordenados
- tiempo cuadrático cuando los datos están en el orden inverso
- tiempo cuadrático en el caso promedio
- El caso en el que realiza más intercambios es cuando el arreglo está al revés ya que es el caso en el que todos los pares están invertidos.

# II 2019-2 P2

Tenemos que evaluar los mejores y peores casos de cada algoritmo/ejecutable

## **InsertionSort**

- tiempo lineal cuando los datos están ordenados
- tiempo cuadrático cuando los datos están en el orden inverso
- tiempo cuadrático en el caso promedio
- El caso en el que realiza más intercambios es cuando el arreglo está al revés ya que es el caso en el que todos los pares están invertidos.

## **SelectionSort**

- tiempo cuadrático en todos los casos

# II 2019-2 P2

Tenemos que evaluar los mejores y peores casos de cada algoritmo/ejecutable

## **QuickSort aleatorio**

- caso promedio toma tiempo  $O(N \log N)$
- peor caso tiempo cuadrático
- pivote es aleatorio lo más probable es que el tiempo en los 3 inputs sea similar



# II 2019-2 P2

Tenemos que evaluar los mejores y peores casos de cada algoritmo/ejecutable

## **QuickSort aleatorio**

- caso promedio toma tiempo  $O(N \log N)$
- peor caso tiempo cuadrático
- pivote es aleatorio lo más probable es que el tiempo en los 3 inputs sea similar

## **QuickSort con pivote primer elemento**

- El peor caso es cuadrático (cuando el pivote es el mayor o menor elemento del intervalo).
- caso promedio toma tiempo  $O(N \log N)$

# II 2019-2 P2

Archivo	Datos Ordenados	Datos Aleatorios	Ordenados a la inversa
E1	44.88	48.72	43.35
E2	40.87	9.97	43.22
E3	7.35	25.47	46.11
E4	10.05	9.81	10.56

# II 2019-2 P2

Archivo	Datos Ordenados	Datos Aleatorios	Ordenados a la inversa
E1	44.88	48.72	43.35
E2	40.87	9.97	43.22
E3	7.35	25.47	46.11
E4	10.05	9.81	10.56

E1 => **QuickSort con pivote primer elemento** ya que toma más tiempo cuando elegimos el elemento mayor o menor

# II 2019-2 P2

Archivo	Datos Ordenados	Datos Aleatorios	Ordenados a la inversa
E1	44.88	48.72	43.35
E2	40.87	9.97	43.22
E3	7.35	25.47	46.11
E4	10.05	9.81	10.56

E1 => **QuickSort con pivote primer elemento** ya que toma más tiempo cuando elegimos el elemento mayor o menor

E2 => **SelectionSort** ya que siempre se demora

# II 2019-2 P2

Archivo	Datos Ordenados	Datos Aleatorios	Ordenados a la inversa
E1	44.88	48.72	43.35
E2	40.87	9.97	43.22
E3	7.35	25.47	46.11
E4	10.05	9.81	10.56

E1 => **QuickSort con pivote primer elemento** ya que toma más tiempo cuando elegimos el elemento mayor o menor

E2 => **SelectionSort** ya que siempre se demora

E3 => **InsertionSort** ya que mientras más ordenado menos tiempo toma.}

# II 2019-2 P2

Archivo	Datos Ordenados	Datos Aleatorios	Ordenados a la inversa
E1	44.88	48.72	43.35
E2	40.87	9.97	43.22
E3	7.35	25.47	46.11
E4	10.05	9.81	10.56

E1 => **QuickSort con pivot primer elemento** ya que toma más tiempo cuando elegimos el elemento mayor o menor

E2 => **SelectionSort** ya que siempre se demora

E3 => **InsertionSort** ya que mientras más ordenado menos tiempo toma.

E4 => **QuickSort con pivot aleatorio** ya que en todos los casos tuvo un comportamiento eficiente independiente del orden

# Heaps

## II 2019-2 P4

Queremos extender la funcionalidad de un Heap Binario. Asumiendo que tienes las funciones **insert**, **extract**, **siftUp**, **siftDown**, escribe un algoritmo para las siguientes operaciones:

- Modificar la prioridad del elemento en la posición  $i$
- Eliminar del Heap el elemento en la posición  $i$



# II 2019-2 P4

Queremos extender la funcionalidad de un Heap Binario. Asumiendo que tienes las funciones **insert**, **extract**, **siftUp**, **siftDown**, escribe un algoritmo para las siguientes operaciones:

- Modificar la prioridad del elemento en la posición  $i$

***updatePriority***( $H, i, \text{value}$ ):

$H[i] = \text{value}$

$\text{siftUp}(i)$

$\text{siftDown}(i)$

# II 2019-2 P4

Queremos extender la funcionalidad de un Heap Binario. Asumiendo que tienes las funciones **insert**, **extract**, **siftUp**, **siftDown**, escribe un algoritmo para las siguientes operaciones:

- Eliminar del Heap el elemento en la posición  $i$

**pop**(H,  $i$ ):

value = H[i]

H[i] = H[H.count]

H.count -= 1

siftUp(i)

siftDown(i)

## II 2018-1 P2

Tienes dos heaps: A, de tamaño  $m$ , y B, de tamaño  $n$ . Los heaps están almacenados explícitamente como árboles binarios y no como arreglos. Si los  $m+n$  elementos son todos distintos, si  $m = 2^k - 1$  para algún entero  $k$ , y si  $m/2 < n \leq m$ , explica cómo construir un heap C con los elementos de  $A \cup B$ .

- a) ¿Cuál es la profundidad del árbol A y cuál es la profundidad del árbol B ?
- b) ¿Cuál va a ser la profundidad del árbol C ? Justifica.
- c) Describe un algoritmo eficiente para la construcción de C. Explica cuál es la complejidad de tu algoritmo.

## II 2018-1 P2

Tienes dos heaps: A, de tamaño  $m$ , y B, de tamaño  $n$ . Los heaps están almacenados explícitamente como árboles binarios y no como arreglos. Si los  $m+n$  elementos son todos distintos, si  $m = 2^k - 1$  para algún entero  $k$ , y si  $m/2 < n \leq m$ , explica cómo construir un heap C con los elementos de  $A \cup B$ .

a) ¿Cuál es la profundidad del árbol A y cuál es la profundidad del árbol B ?

Si consideramos la profundidad de la raíz como 0, entonces la profundidad tanto de A como de B es  $k-1$ .

## II 2018-1 P2

Tienes dos heaps: A, de tamaño  $m$ , y B, de tamaño  $n$ . Los heaps están almacenados explícitamente como árboles binarios y no como arreglos. Si los  $m+n$  elementos son todos distintos, si  $m = 2^k - 1$  para algún entero  $k$ , y si  $m/2 < n \leq m$ , explica cómo construir un heap C con los elementos de  $A \cup B$ .

b) ¿Cuál va a ser la profundidad del árbol C ? Justifica.

### **Solución 1**

Dado que tanto A como B tienen profundidad  $k-1$ , al unirlos en un nuevo árbol, C, mediante una raíz común, este nuevo árbol necesariamente tiene profundidad  $k$ .

## II 2018-1 P2

Tienes dos heaps: A, de tamaño  $m$ , y B, de tamaño  $n$ . Los heaps están almacenados explícitamente como árboles binarios y no como arreglos. Si los  $m+n$  elementos son todos distintos, si  $m = 2^k - 1$  para algún entero  $k$ , y si  $m/2 < n \leq m$ , explica cómo construir un heap C con los elementos de  $A \cup B$ .

b) ¿Cuál va a ser la profundidad del árbol C ? Justifica.

### **Solución 2**

En el extremo, es decir, si  $n = m$ , entonces el árbol resultante de la unión tiene  $2^k - 1 + 2^k - 1 = 2^{k+1} - 2$  elementos. Como  $2^{k+1} - 2 = (2^{k+1} - 1) - 1$ , se trata de un árbol binario completo en todos sus niveles menos en el último (le falta un elemento), por lo que su profundidad es  $k$ .

## II 2018-1 P2

Tienes dos heaps: A, de tamaño  $m$ , y B, de tamaño  $n$ . Los heaps están almacenados explícitamente como árboles binarios y no como arreglos. Si los  $m+n$  elementos son todos distintos, si  $m = 2^k - 1$  para algún entero  $k$ , y si  $m/2 < n \leq m$ , explica cómo construir un heap C con los elementos de  $A \cup B$ .

c) Describe un algoritmo eficiente para la construcción de C. Explica cuál es la complejidad de tu algoritmo.

Saquemos el último elemento de B, es decir, el de más a la derecha en el nivel de más abajo; llamémoslo  $x$ .

Creamos un nuevo árbol binario con  $x$  como raíz y A y B como subárboles. Todo esto es  $O(1)$ . Finalmente, aplicamos heapify sobre  $x$ . Esta operación, como vimos en clase, es  $O(k) = O(\log m)$ .

# Estrategias Algorítmicas



# Knapsack problem

El problema de la mochila consiste en seleccionar un subconjunto de objetos (indivisibles y unitarios), cada uno con un **valor  $v$**  y un **peso  $w$** , de manera que se maximice el valor total de los objetos elegidos sin superar la capacidad **máxima  $W$**  de la mochila.

Cree una solución con **backtracking**, **PD** y **greedy** (Para este último no se pide que la solución sea globalmente óptima).



# Knapsack problem - Backtracking

Variables y Restricciones

$$\sum_{i=1}^n x_i w_i \leq W$$

$x_i$ : 1 si se considera el objeto  $i$  en la solución, 0 en otro caso.

# Knapsack problem - Backtracking

KnapsackBacktracking ( $w, v, W$ ) :

1.  $n \leftarrow \text{len}(w)$
2.  $\text{max\_value} \leftarrow 0$
3. Backtrack ( $i, \text{current\_weight}, \text{current\_value}$ ) :
4.     **if**  $\text{current\_weight} > W$  :
5.         **return**
6.     **if**  $\text{current\_value} > \text{max\_value}$  :
7.          $\text{max\_value} \leftarrow \text{current\_value}$
8.     **if**  $i = n$  :
9.         **return**
10.         Backtrack ( $i + 1, \text{current\_weight} + w[i], \text{current\_value} + v[i]$ )
11.         Backtrack ( $i + 1, \text{current\_weight}, \text{current\_value}$ )
12.     Backtrack ( $0, 0, 0$ )
13.     **return**  $\text{max\_value}$

$O(2^n)$

# Knapsack problem - PD

$$DP[i][w] = \max\{DP[i-1][w], \text{valor}[i] + DP[i-1][w - \text{peso}[i]]\}$$

Valor máximo considerando los primeros  $i$  objetos y una capacidad  $w$

Subproblema donde no se considera el objeto  $i$

Subproblema donde sí se considera el objeto  $i$ , entonces hay menos capacidad

# Knapsack problem - PD

KnapsackDP ( $w, v, W$ ) :

1.  $n \leftarrow \text{len}(w)$
2. Crear una matriz  $dp[0 \dots n][0 \dots W] \leftarrow 0$
3. **for**  $i = 1 \dots n$  :
4.     **for**  $j = 0 \dots W$  :
5.         **if**  $w[i - 1] \leq j$  :
6.              $dp[i][j] \leftarrow \max(dp[i - 1][j], dp[i - 1][j - w[i - 1]] + v[i - 1])$
7.         **else** :
8.              $dp[i][j] \leftarrow dp[i - 1][j]$
9. **return**  $dp[n][W]$

**$O(n \times W)$**

# Knapsack problem - Greedy

Probar insertar el elemento con el valor más denso (valor/peso) y repetir hasta no cumplir con la restricción de carga.

# Knapsack problem - Greedy

KnapsackGreedy ( $v, w, W$ ) :

1.  $n \leftarrow \text{len}(w)$
2. Crear una lista  $density[0 \dots n - 1]$  donde  $density[i] \leftarrow \frac{v[i]}{w[i]}$
3. Ordenar los objetos en orden descendente según  $density$
4.  $current\_weight \leftarrow 0$
5.  $total\_value \leftarrow 0$
6. **for**  $i = 0 \dots n - 1$  :
7.     **if**  $current\_weight + w[i] \leq W$  :
8.          $current\_weight \leftarrow current\_weight + w[i]$
9.          $total\_value \leftarrow total\_value + v[i]$
10.     **else** :
11.         **break**
12.     **return**  $total\_value$

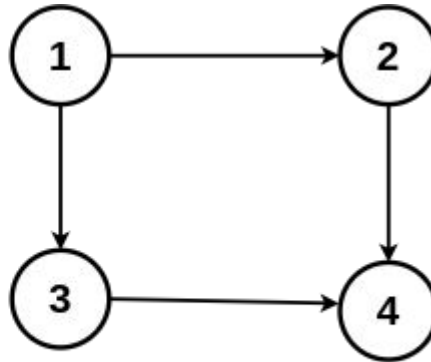
**$O(n \log n)$**

# DFS y Kosaraju



# I3 2017-1 P1

- a) Sea  $G = (V, E)$  un grafo dirigido acíclico. ¿Cuántas componentes fuertemente conexas encuentra el algoritmo de Kosaraju al ser ejecutado en  $G$ ? Justifique.



Direct Acyclic Graph

# Componentes Fuertemente Conexas (CFC)

## Definición

- En un grafo **dirigido**  $G$ , una **CFC** es un conjunto **maximal** de nodos  $C \subseteq G$  de tal manera que dados  $u, v \in C$  existe un camino dirigido desde  $u$  hasta  $v$

# Kosaraju: Pseudocódigo

**input** : grafo  $G$

Kosaraju( $G$ ):

```
1    $L \leftarrow \text{TopSort}(G)$ 
2   for  $u \in L$  :
3       Assign( $u, u$ )
```

**input** : grafo  $G$ , nodo  $u \in V(G)$ , nodo representante  $r$

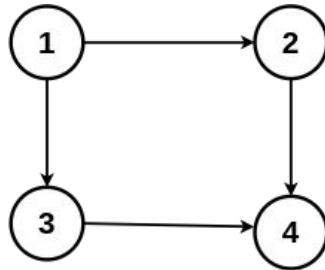
Assign( $G, u, r$ ):

```
1   if  $u.rep = \emptyset$  :
2        $u.rep \leftarrow r$ 
3       for  $v \in N_{G^T}(u)$  :
4           Assign( $G, v, r$ )
```

# I3 2017-1 P1

- a) Sea  $G = (V, E)$  un grafo dirigido acíclico. ¿Cuántas componentes fuertemente conexas encuentra el algoritmo de Kosaraju al ser ejecutado en  $G$ ? Justifique.

El algoritmo de Kosaraju encuentra todas las componentes fuertemente conexas de un grafo



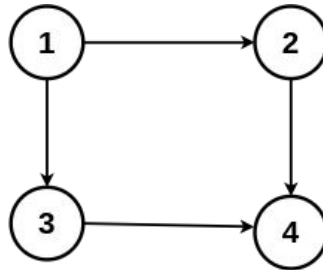
Direct Acyclic Graph

# I3 2017-1 P1

- a) Sea  $G = (V, E)$  un grafo dirigido acíclico. ¿Cuántas componentes fuertemente conexas encuentra el algoritmo de Kosaraju al ser ejecutado en  $G$ ? Justifique.

**Grafo dirigido acíclico:** no posee ningún ciclo y sus nodos se conectan en un solo sentido. Al no existir ciclos, no existe camino que permita a un nodo conectarse con sí mismo.

Dado esto, no puede haber componentes fuertemente conexas si no se permiten ciclos en un grafo. Por ende, El grafo  $G$  no posee componentes fuertemente conexas. Si se define que un nodo es alcanzable por sí mismo, entonces la cantidad de componentes fuertemente conexas es  $|V|$ .



Direct Acyclic Graph

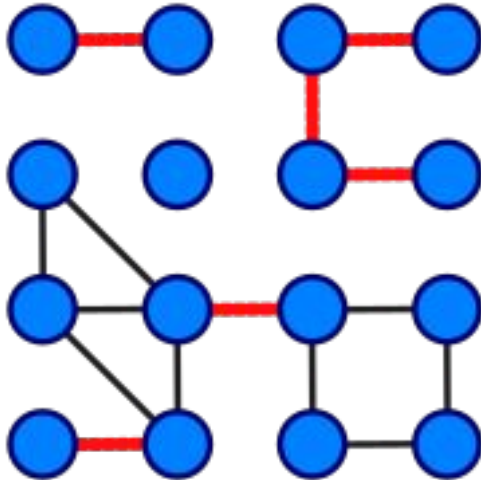
# Control 4 2019 - DFS

1. Un **punto en un grafo no direccional es una arista  $(u, v)$**  tal que al sacarla del grafo hace que el grafo quede desconectado -o, más precisamente, aumenta el número de componentes conectadas del grafo; en otras palabras, la única forma para ir de  $u$  a  $v$  en el grafo es a través de la arista  $(u, v)$ .

Explica cómo usar el **algoritmo DFS** para **encontrar eficientemente los puentes de un grafo no direccional**; y justifica qué tan eficientemente. Recuerda que DFS asigna tiempos de descubrimiento (y finalización) a cada vértice que visita.

Un **punto en un grafo no direccional es una arista  $(u, v)$**  tal que al sacarla del grafo hace que el grafo quede desconectado -o, más precisamente, aumenta el número de componentes conectadas del grafo; en otras palabras, la única forma para ir de  $u$  a  $v$  en el grafo es a través de la arista  $(u, v)$ .

Explica cómo usar el **algoritmo DFS** para **encontrar eficientemente los puentes de un grafo no direccional**; y justifica qué tan eficientemente. Recuerda que DFS asigna tiempos de descubrimiento (y finalización) a cada vértice que visita.



Grafo con 6 puentes o aristas de corte (marcadas en rojo)

## Modificación a DFS para encontrar aristas de corte:

Buscar aquellas **aristas que no pertenezcan a algún ciclo** dentro del grafo. Es posible encontrar dichas aristas utilizando el algoritmo DFS.

Teniendo nuestro bosque, generado por el algoritmo, supongamos que tenemos una **arista (u, v)** y que en el bosque **no es posible llegar desde un descendiente de v hasta un ancestro de u**, entonces diremos que dicha arista no pertenece a ningún ciclo y en consecuencia, **es un puente**.

Para detectar de manera eficiente si una arista es un puente, lo que se hará es **guardar, en cada nodo, el menor de los tiempos de descubrimiento de cualquier nodo alcanzable** (no necesariamente en un paso) desde donde me encuentre, llamémoslo **v.low**. Si la arista es la (u, v), esto es:

$$u. low = \min \{ u. d, v. low \}$$



## Modificación a DFS para encontrar aristas de corte (Pt. 2):

$$u. low = \min \{ u. d, v. low \}$$

Ahora, al momento de retornar el método **dfsVisit**, lo que se hará es comparar dicho valor con el tiempo de descubrimiento del nodo en el que estoy parado. Digamos estoy **parado en el nodo u** y **visité v**, si se tiene que  **$u. d < v. low$**  entonces no se puede alcanzar ningún ancestro de u desde algún descendiente de v, en consecuencia **dicha arista no pertenece a ningún ciclo** y es un puente.

Esta forma de detectar puentes en el grafo posee la misma complejidad que el algoritmo DFS,  **$O(|V| + |E|)$** . Lo único que se está haciendo es **actualizar un valor y compararlo**, lo que no suma mayor complejidad (se hace en tiempo constante).

**PRIM**

# 2018-2

## 3. MSTs

Considera el siguiente grafo no direccional con costos, representado matricialmente:

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>
<i>a</i>		2	4	1			
<i>b</i>	2			3	10		
<i>c</i>	4			2		5	
<i>d</i>	1	3	2		7	8	4
<i>e</i>		10		7			6
<i>f</i>			5	8			1
<i>g</i>				4	6	1	

	<i>¿sol?</i>	<i>dist</i>	<i>padre</i>
<i>a</i>	<i>F</i>	0	—
<i>b</i>	<i>F</i>	$\infty$	—
<i>c</i>	<i>F</i>	$\infty$	—
<i>d</i>	<i>F</i>	$\infty$	—
<i>e</i>	<i>F</i>	$\infty$	—
<i>f</i>	<i>F</i>	$\infty$	—
<i>g</i>	<i>F</i>	$\infty$	—

Ejecuta paso a paso el algoritmo de Prim para determinar un árbol de cobertura de costo mínimo, tomando *a* como vértice de partida. En cada paso muestra la versión actualizada de la tabla a la derecha, en que *¿sol?* indica si el vértice ya está en la solución: *V* o *F*.

# HUH?

	0		
	¿sol ?	dis t	pad re
a	<i>F</i>	0	—
b	<i>F</i>	∞	—
c	<i>F</i>	∞	—
d	<i>F</i>	∞	—
e	<i>F</i>	∞	—
f	<i>F</i>	∞	—
g	<i>F</i>	∞	—

	1		
	¿so l?	dist	pad re
a	<i>V</i>	0	—
b	<i>F</i>	2	a
c	<i>F</i>	4	a
d	<i>F</i>	1	a
e	<i>F</i>	∞	—
f	<i>F</i>	∞	—
g	<i>F</i>	∞	—

	2		
	¿sol ?	dist	pad re
a	<i>V</i>	0	—
b	<i>F</i>	2	a
c	<i>F</i>	2	d
d	<i>V</i>	1	a
e	<i>F</i>	7	d
f	<i>F</i>	8	d
g	<i>F</i>	4	d

	3		
	¿sol ?	dist	pad re
a	<i>V</i>	0	—
b	<i>V</i>	2	a
c	<i>F</i>	2	d
d	<i>V</i>	1	a
e	<i>F</i>	7	d
f	<i>F</i>	8	d
g	<i>F</i>	4	d



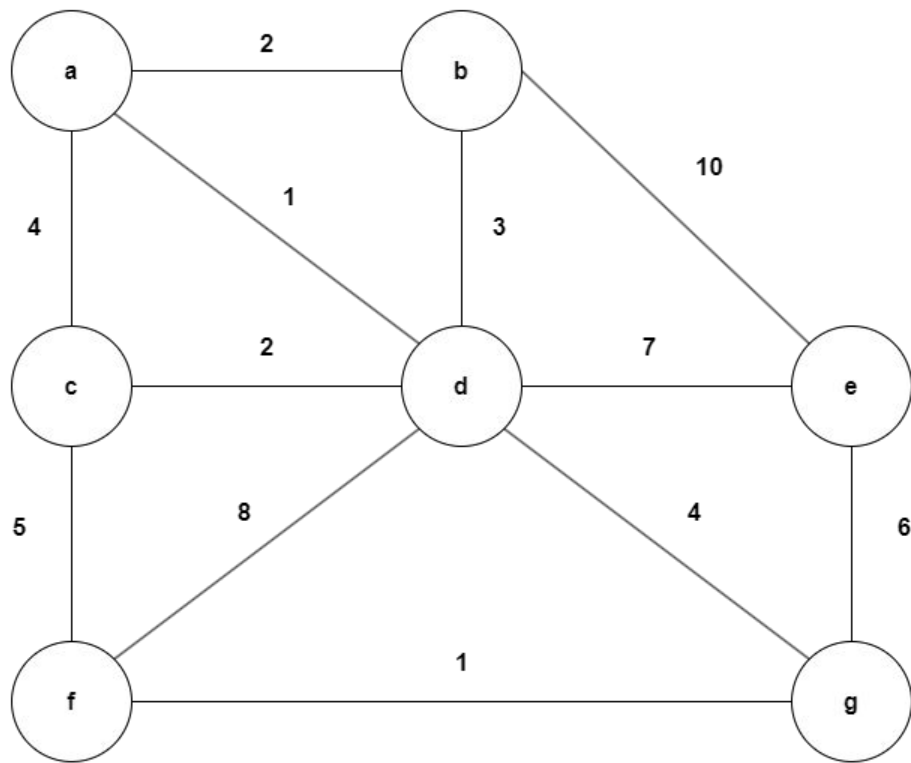
# 2018-2 Solución

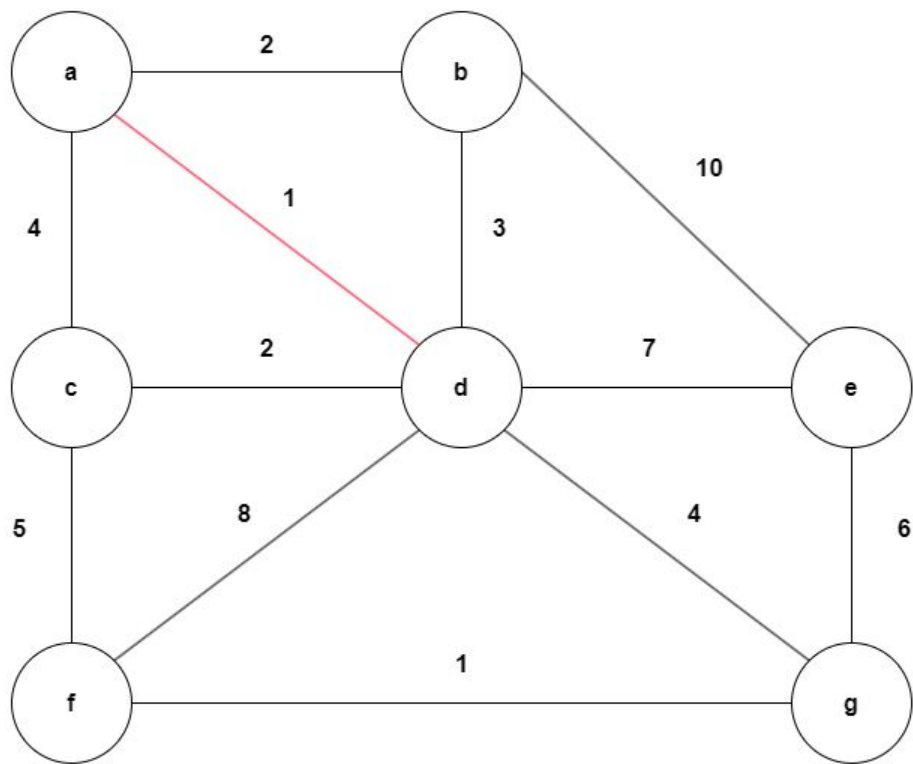
Lo PRIMero que tienen que hacer:

- A partir de la matriz armar el dibujo del grafo (Es fácil marearse y equivocarse mirando la matriz).
- Darse cuenta de que es V y que es F.
- Padre** = nodo del MST que "da acceso con el menor costo a este nodo que aún no está en el MST"
- Dist** = Distancia al nodo padre.

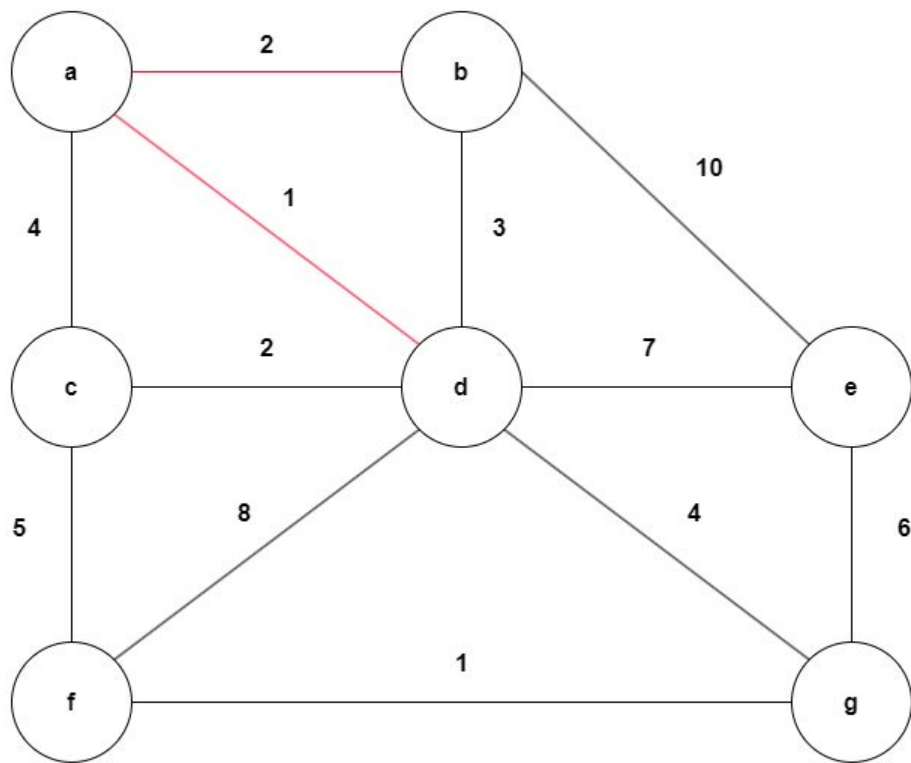


# Ahora sipue



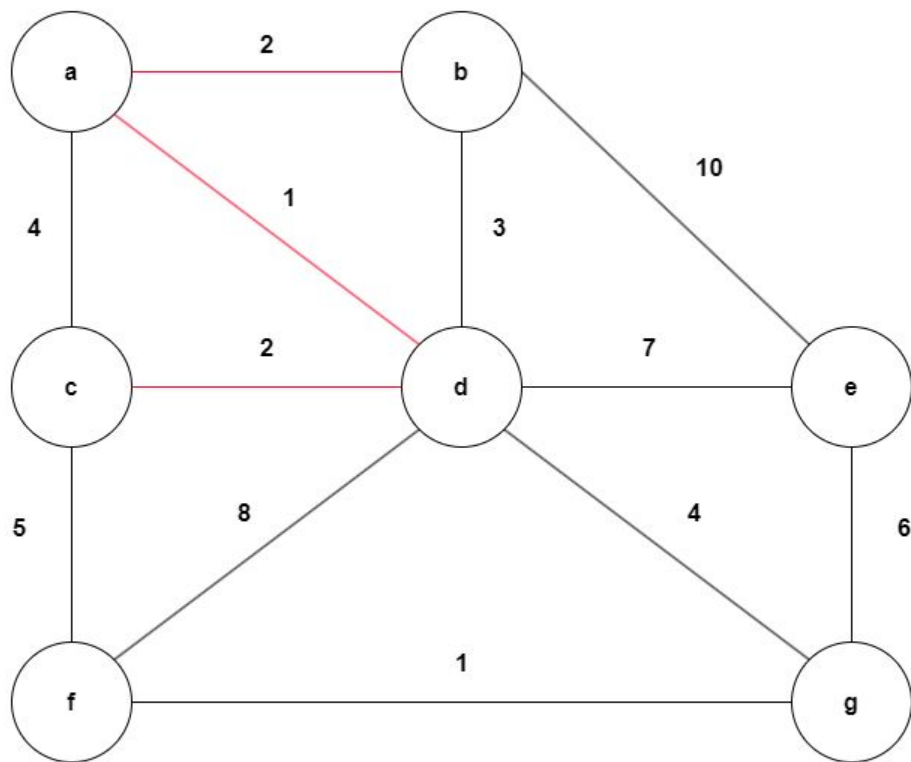


1				2			
	is l?	dist	pa- re		is sol ?	dist	pa- re
a	V	0	—	a	V	0	—
b	F	2	a	b	F	2	a
c	F	4	a	c	F	2	d
d	F	1	a	d	V	1	a
e	F	$\infty$	—	e	F	7	d
f	F	$\infty$	—	f	F	8	d
g	F	$\infty$	—	g	F	4	d

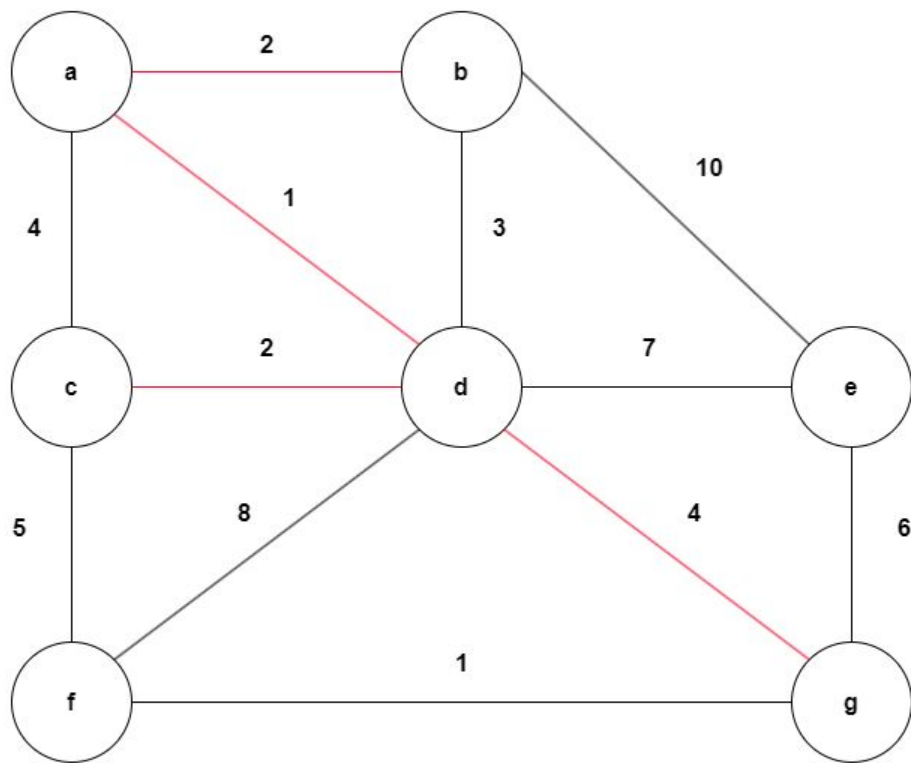


	3		
	¿sol ?	dist	pad re
a	V	0	—
b	V	2	a
c	F	2	d
d	V	1	a
e	F	7	d
f	F	8	d
g	F	4	d

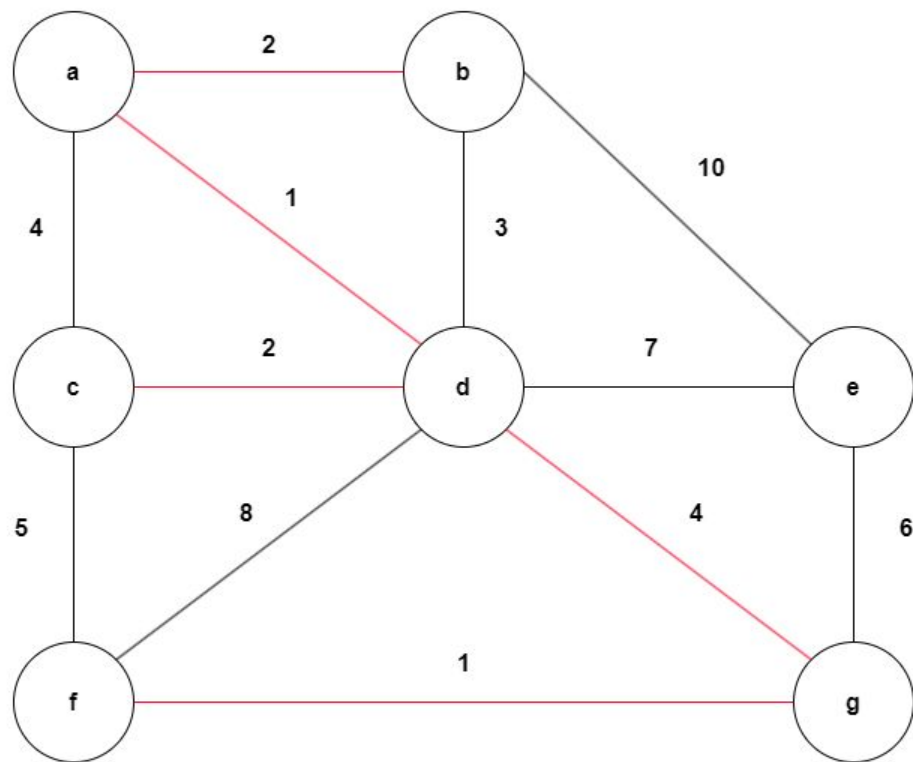




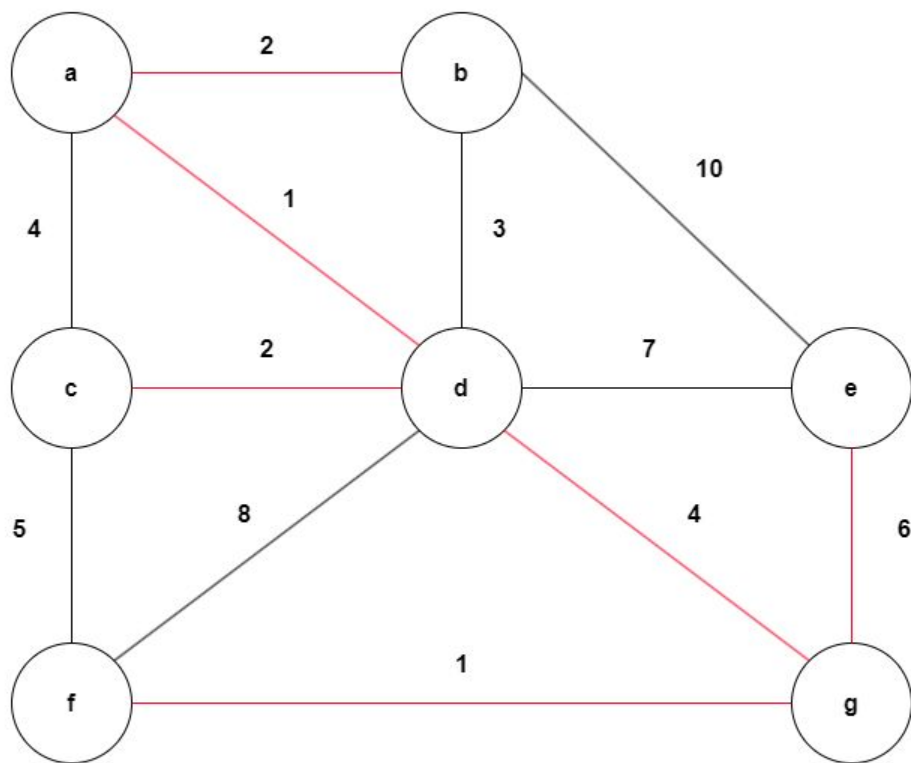
	4		
	sol ?	dis t	pad re
a	V	0	—
b	V	2	a
c	V	2	d
d	V	1	a
e	F	7	d
f	F	5	c
g	F	4	d



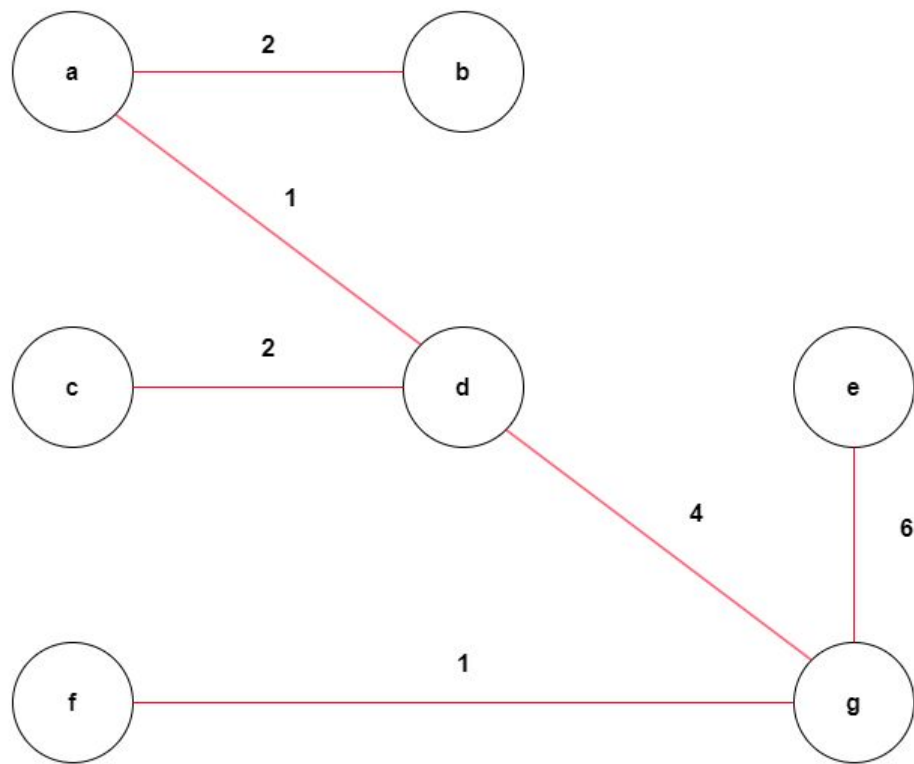
5			
	is l?	dist	parent
a	V	0	—
b	V	2	a
c	V	2	d
d	V	1	a
e	F	6	g
f	F	1	g
g	V	4	d



6			
	sol ?	dist	pad re
a	V	0	—
b	V	2	a
c	V	2	d
d	V	1	a
e	F	6	g
f	V	1	g
g	V	4	d



	7		
	is sol ?	dist	parent
a	V	0	—
b	V	2	a
c	V	2	d
d	V	1	a
e	V	6	g
f	V	1	g
g	V	4	d



# Una pregunta:

¿Cuándo es Prim Mejor que Kruskal?

# La diferencia PRIMordial:

Kruskal  $\rightarrow O(E * \log V)$

Prim  $\rightarrow O(E * \log V)$  (Para el caso que le entregamos los nodos como un heap binario o una LISTA de adyacencia)

PERO, hay una implementación de Prim (Heap Fibonacci) que resulta en  $O(E + V \log V)$  (No es necesario que se aprendan como, esto es más conceptual)

Prim es mejor en grafos de alta densidad (Muchas aristas), dado que Prim recorre los nodos, mientras que Kruskal recorre las Aristas.

# Soluciones

- Algoritmo de Prim
- **Algoritmo de Kruskal**



# KRUSKAL

y Union Find

# Kruskal

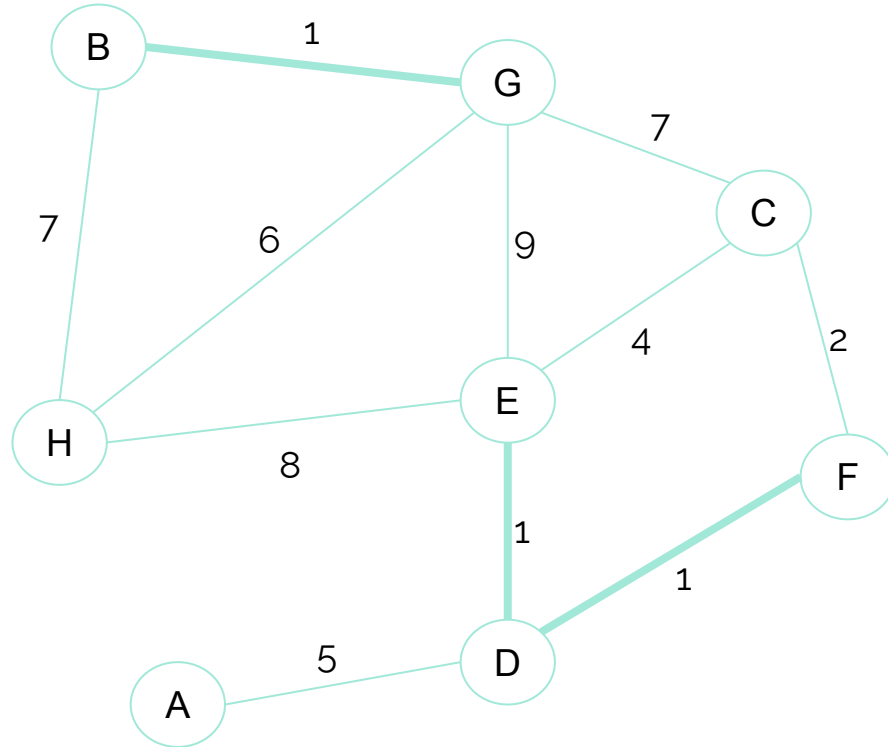
Idea base: Crear un bosque que converge en un único árbol

Sea  $G = (V, E)$  grafo no dirigido iteramos sobre las aristas **e** en orden no decreciente de costo

1. Si **e** genera un ciclo al agregarla a **T**, la ignoramos
2. Si no genera ciclo, se agrega

¿Será necesario revisar **todas** las aristas de  $E$ ?

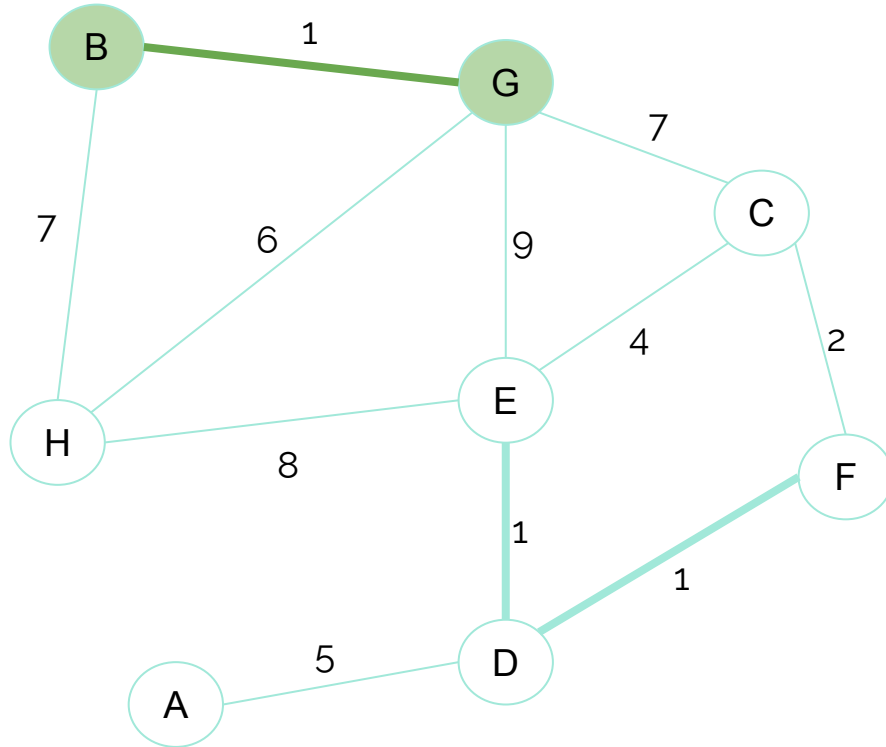
# Ejemplo:



— Aristas candidatas

— Aristas en MST

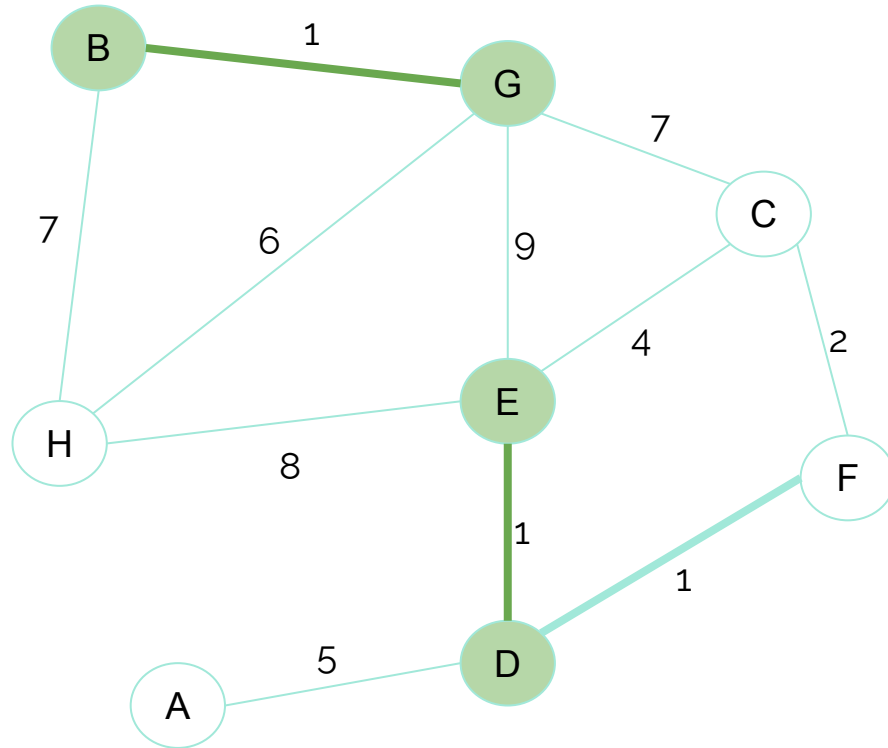
# Ejemplo:



— Aristas candidatas

— Aristas en MST

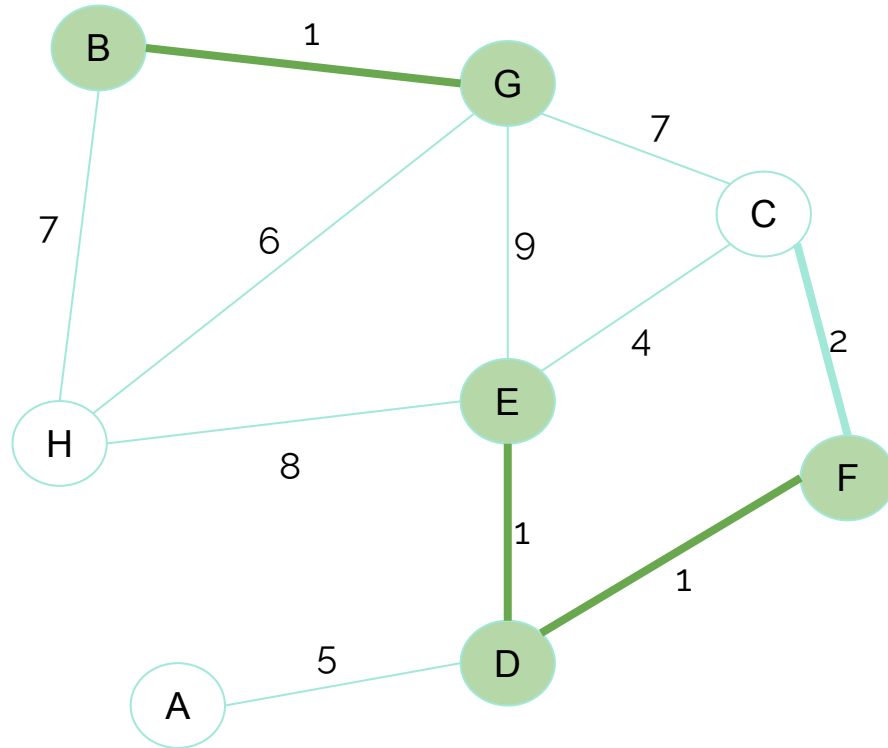
# Ejemplo:



— Aristas candidatas

— Aristas en MST

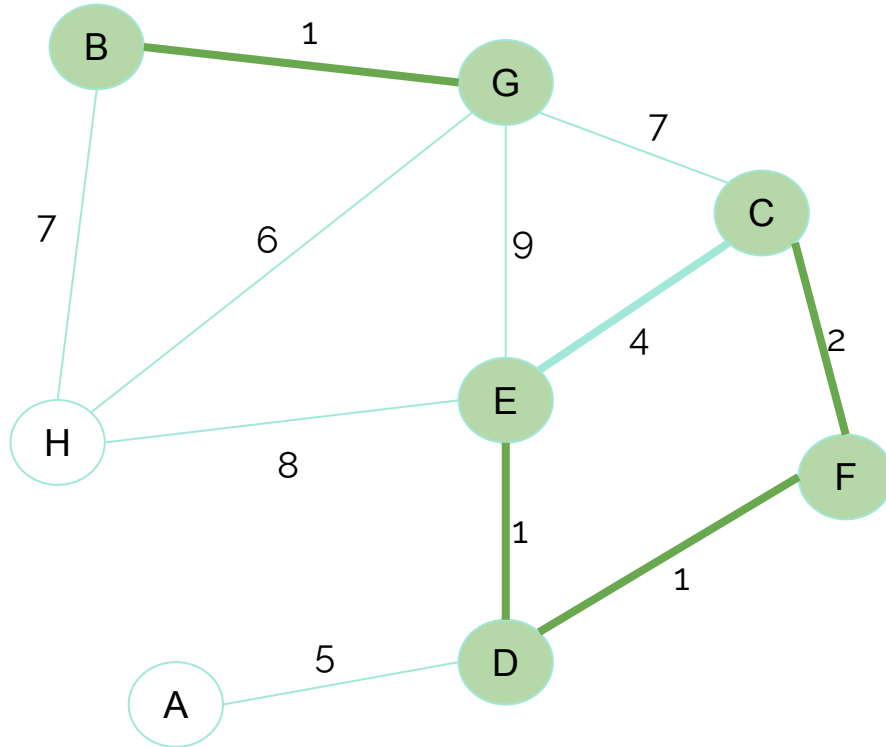
# Ejemplo:



— Aristas candidatas

— Aristas en MST

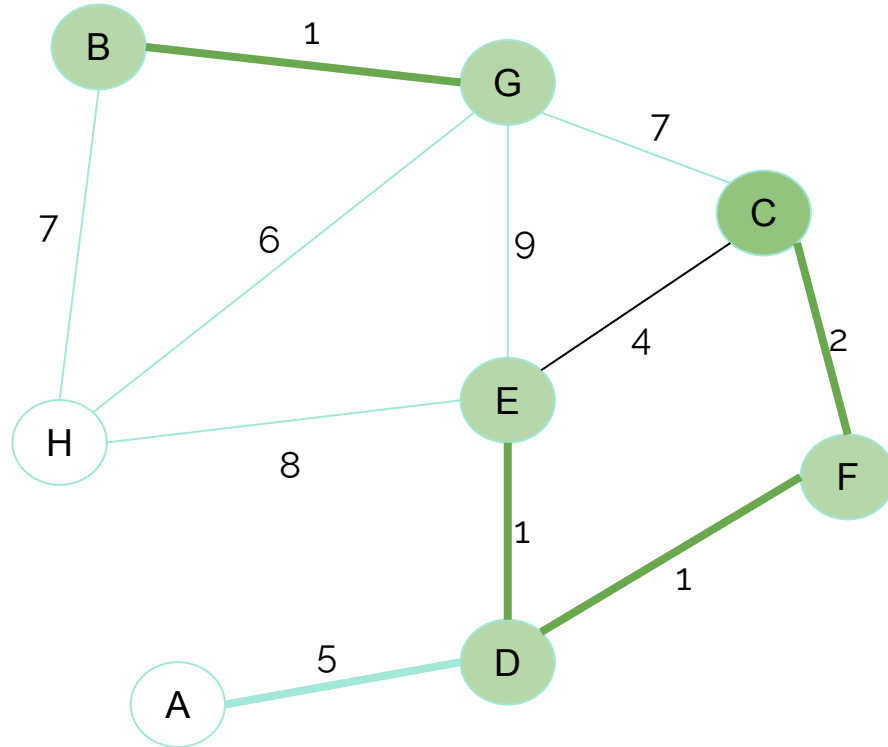
# Ejemplo:



— Aristas candidatas

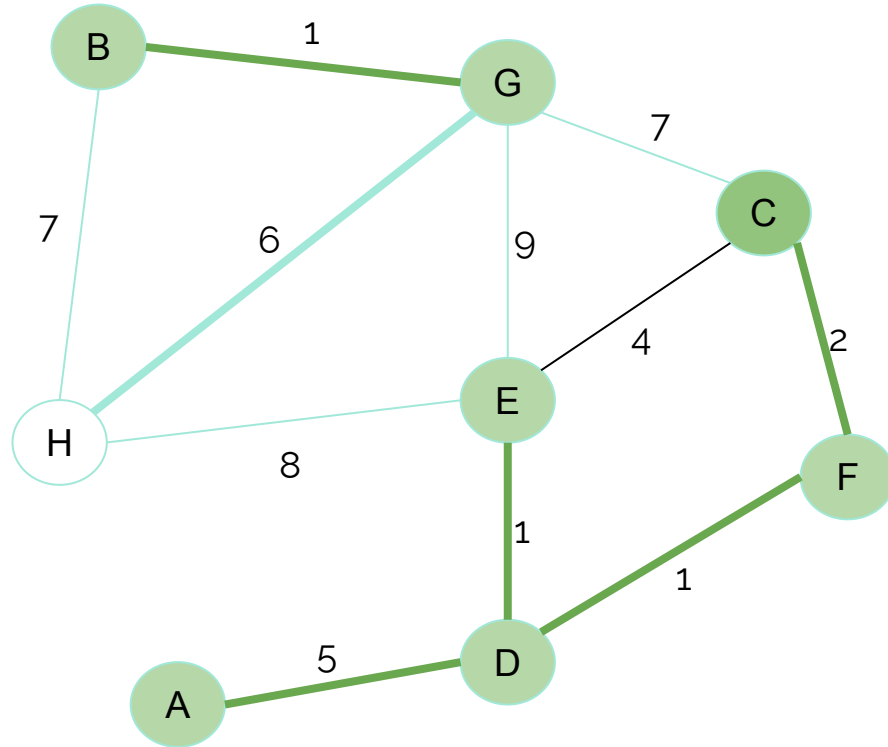
— Aristas en MST


# Ejemplo:






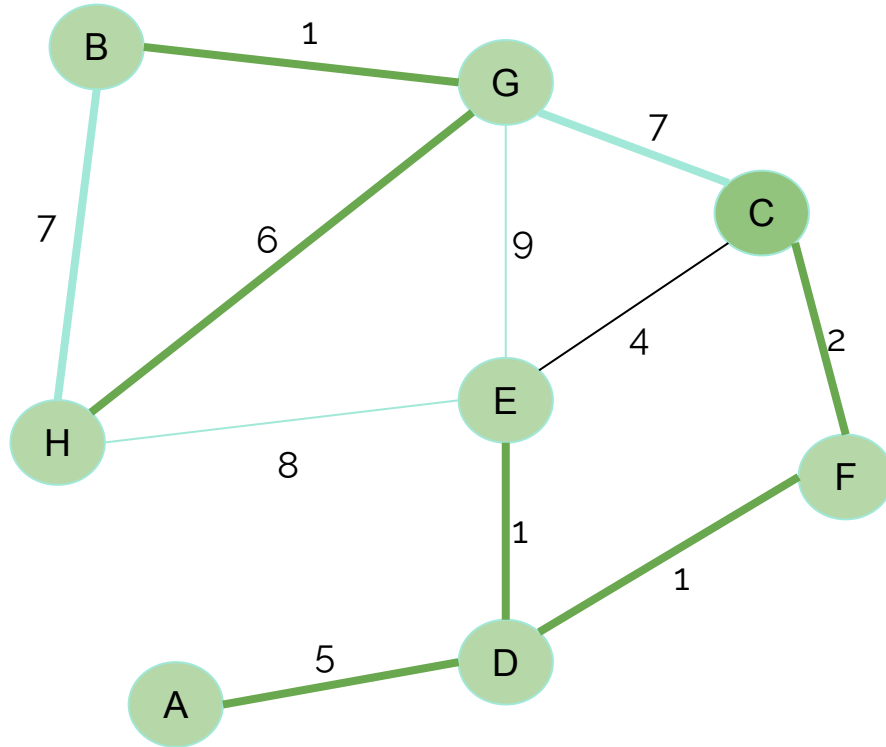
# Ejemplo:



 Aristas candidatas

 Aristas en MST

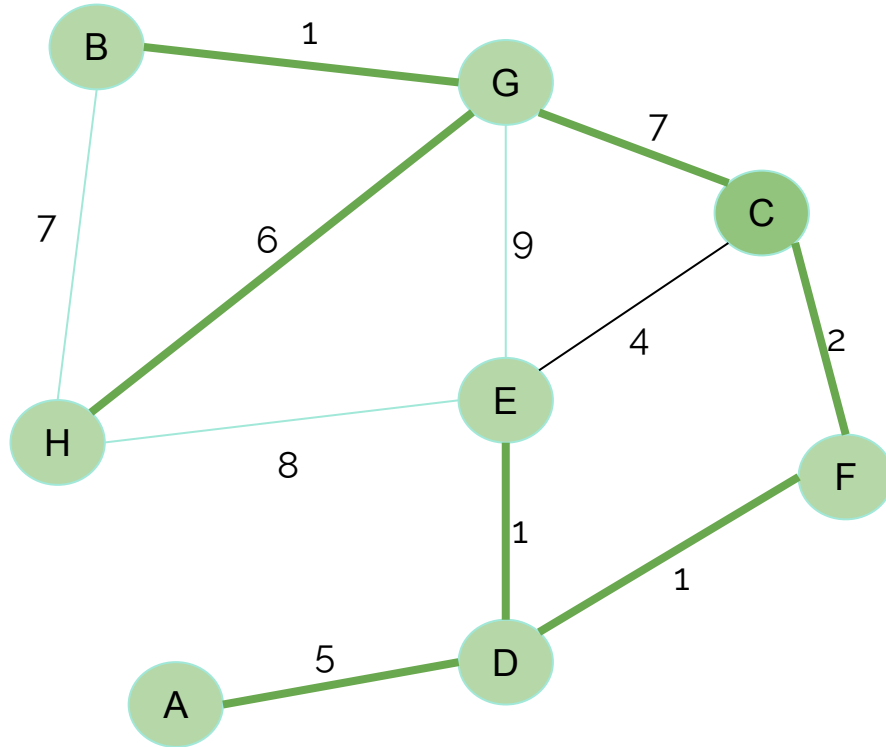
# Ejemplo:



— Aristas candidatas

— Aristas en MST

# Ejemplo:



— Aristas candidatas

— Aristas en MST

# Union Find

**¿Cómo detectamos si una arista produce o no un ciclo?**

**¡Conjuntos disjuntos!**

1. Asignamos un set a cada nodo dentro del grafo
2. Cuando agregamos una arista unimos sus sets
3. Al revisar más aristas a futuro, sabemos que si dos nodos se encuentran en el mismo set, entonces ya existe un camino que los conecta, por lo que agregar esa arista nos generaría un ciclo

Kruskal( $G$ ):

```
1   $E \leftarrow E$  ordenada por costo, de menor a mayor
2  for  $v \in V$  :
3      MakeSet( $v$ )
4   $T \leftarrow$  lista vacía
5  for  $(u, v) \in E$  :
6      if Find( $u$ )  $\neq$  Find( $v$ ) :
7           $T \leftarrow T \cup \{(u, v)\}$ 
8          Union( $u, v$ )
9  return  $T$ 
```

\* Find( $u$ ) entrega el conjunto al que pertenece el nodo  $u$

# Kruskal

kruskal(Grafo G)

UnionFind T

E = G.edges

sort(E) //Por el peso

for w,u,v in E

if not T.same\_set(u,v)

T.join(u,v)

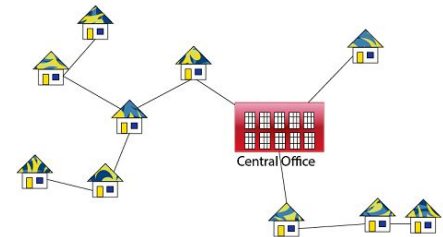
//añadir arista u,v

# I3 2023-1 P3

Una estrategia base para **asegurar cobertura de comunicaciones** es contar con **diferentes servicios de conexión**. Por ejemplo, el servicio de Tesorería (TGR) puede conectar sus oficinas y puntos de atención mediante:

**Redes** de fibra de proveedores, Redes de telefonía móvil, Red StarLink, Redes Satelitales, Radioaficionados y Banda Local.

Naturalmente, cada uno de ellos tienen diferentes costos, velocidades y latencias, y pueden estar disponibles o no entre diferentes oficinas y puntos de atención, o perderse al ocurrir una emergencia.



## I3 2023-1 P3

a) ¿De qué forma puede TGR decidir la **mejor forma de conectar sus oficinas y puntos de atención** en función de los servicios de conexión disponibles en cada oficina o punto?

Indique la forma de representar el problema y las estrategias conocidas para determinar la solución.

Pista: Define primero como determinar la mejor conexión en cada caso.



# I3 2023-1 P3

El problema se puede representar como un grafo no dirigido con costos en las aristas.

- **Nodos:** oficinas y puntos de atención
- **Aristas:** conexiones existentes entre nodos, con una arista por cada tipo de conexión disponible (puede haber más de una arista entre dos nodos).
- **Costo de cada arista:** función que relaciona costo, velocidad y latencia como un valor que describe el costo real de un tipo de conexión entre nodos.

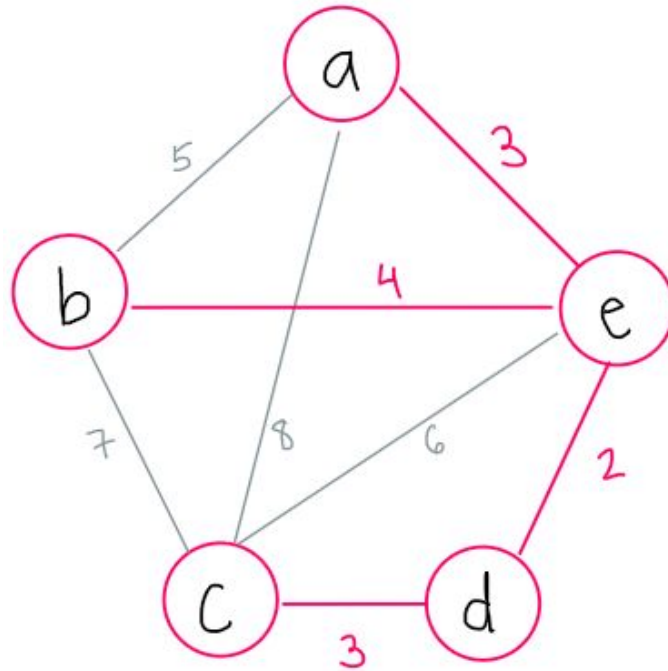
La mejor forma de conectar, basándose en minimizar costo total, es determinar el **MST** de este grafo con algún algoritmo estudiado como Prim o Kruskal.

## I3 2023-1 P3

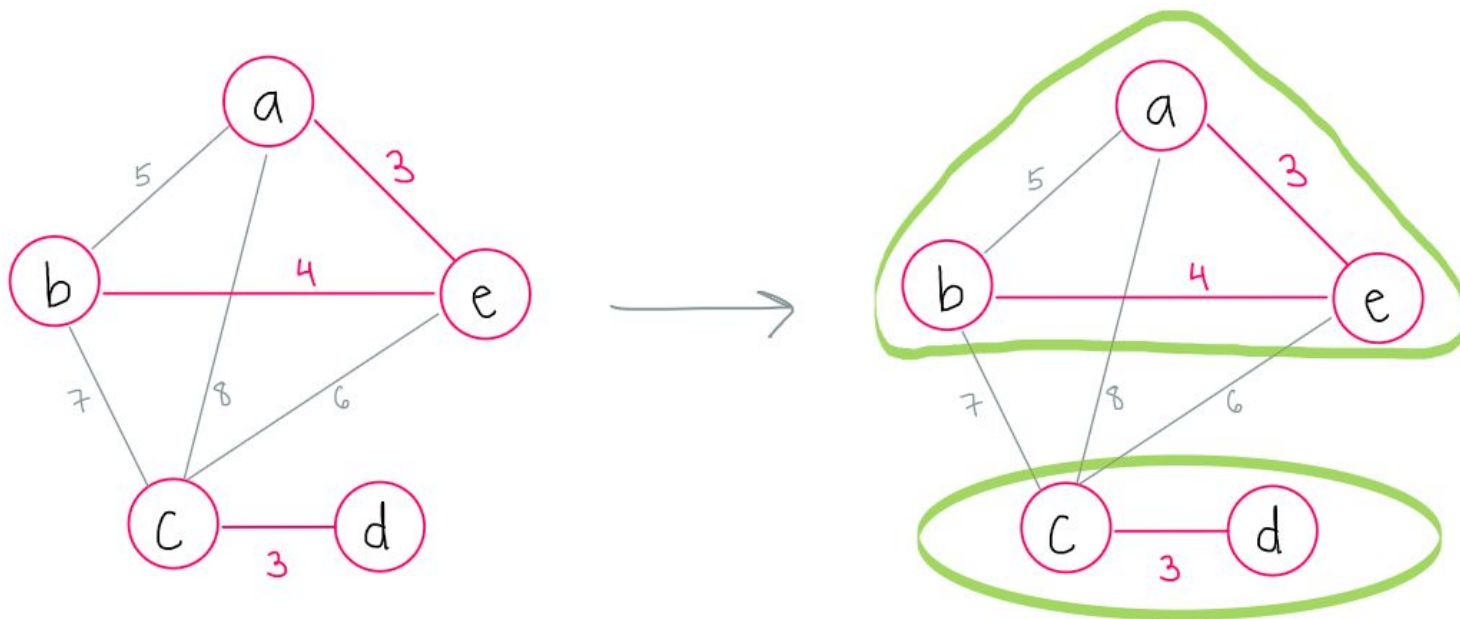
b) Al ocurrir una emergencia, **TGR pierde la conectividad entre dos oficinas a través de uno de los enlaces que es parte de la mejor forma de conectar ya escogida**. Proponga una forma de recuperar la conectividad de la mejor forma disponible.

Justifique por qué es la mejor opción.

Supongamos que este es el grafo que representa el MST de las oficinas y puntos de atención.

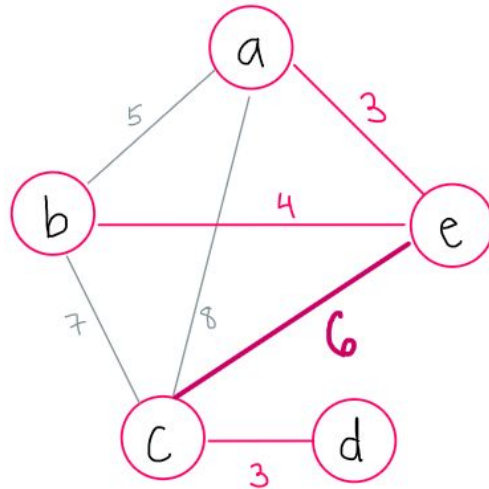


Al perder un enlace el MST del grafo se parte en dos MST independientes.



Sabemos por la estrategia de construcción de los MST que ambos MST son óptimos para conectar los nodos que cubren.

Luego, si consideramos que lo que tenemos al perder la arista es un “corte” entre los MST (como en Kruskal) si determinamos la **siguiente arista de las disponibles que tiene el menor costo y no genera un ciclo** podemos recuperar el MST con las aristas disponibles.



## I3 2023-1 P3

c) Si la emergencia **afecta simultáneamente a múltiples enlaces de la forma de conectar ya escogida**, ¿La estrategia que se propuso antes sigue siendo válida?

Argumente su respuesta.

## I3 2023-1 P3

Sí, es posible generalizar la estrategia anterior cuando el MST del grafo pierde múltiples aristas que eran parte del MST. Podemos aplicar un enfoque similar para reconstruir el MST después de perder una arista.

## I3 2020-2 P2

La gente de la tierra de Omashu se toma los grupos de amigos muy en serio. Tan en serio, que podemos describirlos matemáticamente (son clases de equivalencia):

- Si  $a$  es amigo de  $b$  entonces  $b$  es amigo de  $a$
- Cada persona es amiga de sí misma y cada persona pertenece a un solo grupo de amigos
- Si  $a$  forma parte del grupo  $X$ , y  $b$  es amigo de  $a$ , entonces necesariamente  $b$  forma parte de  $X$ .



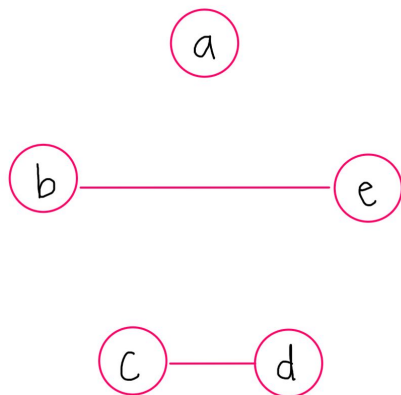
## I3 2020-2 P2

La gente de la tierra de Omashu se toma los grupos de amigos muy en serio. Tan en serio, que podemos describirlos matemáticamente (son clases de equivalencia):

- Si  $a$  es amigo de  $b$  entonces  $b$  es amigo de  $a$
  - Cada persona es amiga de sí misma y cada persona pertenece a un solo grupo de amigos
  - Si  $a$  forma parte del grupo  $X$ , y  $b$  es amigo de  $a$ , entonces necesariamente  $b$  forma parte de  $X$ .
- a) Dada una **lista F de pares de forma  $(a, b)$  que indican amistad** entre la persona  $a$  y la persona  $b$ , describe un **algoritmo lineal en el número de pares** que **calcule la cantidad de grupos de amigos distintos** que existen en Omashu.

## I3 2020-2 P2

- Si  $a$  es amigo de  $b$  entonces  $b$  es amigo de  $a$
- Cada persona es amiga de sí misma y cada persona pertenece a un solo grupo de amigos
- Si  $a$  forma parte del grupo  $X$ , y  $b$  es amigo de  $a$ , entonces necesariamente  $b$  forma parte de  $X$ .



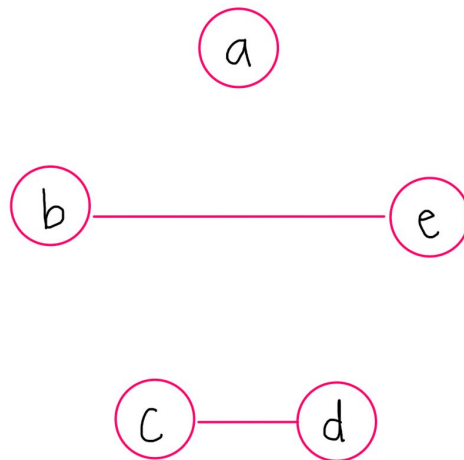
## I3 2020-2 P2

- a) Dada una **lista F de pares de forma (a, b) que indican amistad** entre la persona a y la persona b, describe un **algoritmo lineal en el número de pares** que **calcule la cantidad de grupos de amigos distintos** que existen en Omashu.

$F = [ (a,a), (b,b), (c,c), (d,d), (e,e), (b,e), (c,d) ]$

Grupos de amigos:

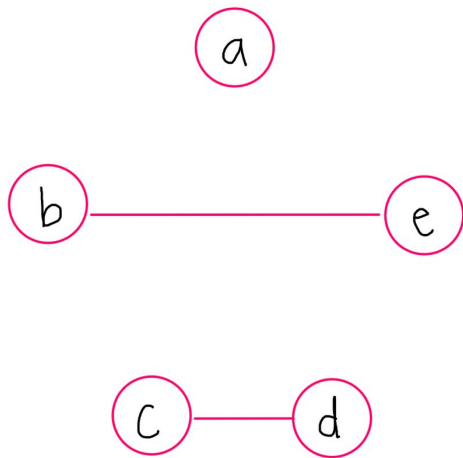
1. a
2. b, e
3. c, d



Cada grupo de amigos = un conjunto disjunto

(nunca va existir un amigo que esté en dos grupos distintos)

Para la resolución del problema, se cuenta con un grafo donde cada nodo es una persona y las aristas son los pares  $(a, b)$  de la lista  $F$

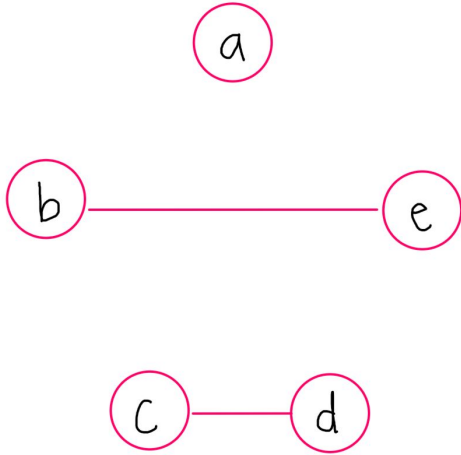


**¿Cómo calculamos la cantidad de conjuntos disjuntos?**

Cada grupo de amigos = un conjunto disjunto

(nunca va existir un amigo que esté en dos grupos distintos)

Para la resolución del problema, se cuenta con un grafo donde cada nodo es una persona y las aristas son los pares  $(a, b)$  de la lista  $F$

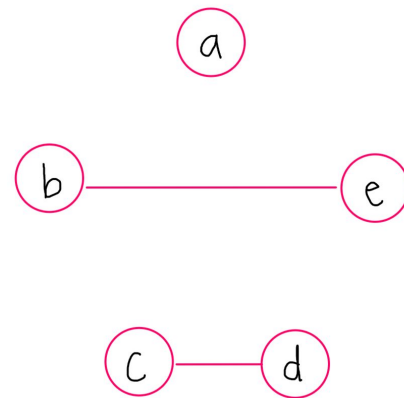


**¿Cómo calculamos la cantidad de conjuntos disjuntos?**

**KRUSKAL!! (con modificaciones)**

```
1: procedure OMASHU( $F$ )
2:   groups := 0
3:   for  $(a, b) \in F$  do
4:     if  $a = b$  then
5:       make_set( $a$ )
6:       groups := groups + 1
7:     end if
8:   end for
9:   for  $(a, b) \in F$  do
10:    if find_set( $a$ )  $\neq$  find_set( $b$ ) then
11:      union( $a, b$ )
12:      groups := groups - 1
13:    end if
14:  end for
15:  return groups
16: end procedure
```

$F = [ (a,a), (b,b), (c,c), (d,d), (e,e),$   
 $(b,e), (c,d) ]$

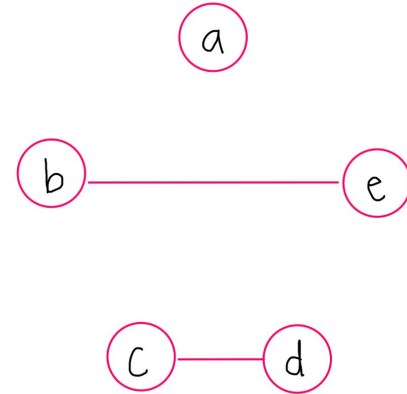


```

1: procedure OMASHU( $F$ )
2:    $groups := 0$ 
3:   for  $(a, b) \in F$  do
4:     if  $a = b$  then
5:       make_set( $a$ )
6:        $groups := groups + 1$ 
7:     end if
8:   end for
9:   for  $(a, b) \in F$  do
10:    if find_set( $a$ )  $\neq$  find_set( $b$ ) then
11:      union( $a, b$ )
12:       $groups := groups - 1$ 
13:    end if
14:  end for
15:  return  $groups$ 
16: end procedure

```

$F = [ (a,a), (b,b), (c,c), (d,d), (e,e), (b,e), (c,d) ]$



$groups = 5$

Sets iniciales:

$\{a\}, \{b\}, \{c\}, \{d\}, \{e\}$

```
1: procedure OMASHU( $F$ )
2:   groups := 0
3:   for  $(a, b) \in F$  do
4:     if  $a = b$  then
5:       make_set( $a$ )
6:       groups := groups + 1
7:     end if
8:   end for
9:   for  $(a, b) \in F$  do
10:    if find_set( $a$ )  $\neq$  find_set( $b$ ) then
11:      union( $a, b$ )
12:      groups := groups - 1
13:    end if
14:  end for
15:  return groups
16: end procedure
```

$F = [ (a,a), (b,b), (c,c), (d,d), (e,e),$   
 $(b,e), (c,d) ]$

$S : \{a\}, \{b\}, \{c\}, \{d\}, \{e\}$

Groups = 5

**Revisamos (a,a):**

find\_set( $a$ ) = { $a$ }

find\_set( $a$ ) = { $a$ }

Son iguales, pasamos al siguiente.

Lo mismo para:

$(b,b), (c,c), (d,d), (e,e)$



```
1: procedure OMASHU( $F$ )
2:   groups := 0
3:   for  $(a, b) \in F$  do
4:     if  $a = b$  then
5:       make_set( $a$ )
6:       groups := groups + 1
7:     end if
8:   end for
9:   for  $(a, b) \in F$  do
10:    if find_set( $a$ )  $\neq$  find_set( $b$ ) then
11:      union( $a, b$ )
12:      groups := groups - 1
13:    end if
14:  end for
15:  return groups
16: end procedure
```

$F = [ (a,a), (b,b), (c,c), (d,d), (e,e),$   
 $(b,e), (c,d) ]$

$S : \{a\}, \{b\}, \{c\}, \{d\}, \{e\}$

Groups = 5

**Revisamos (b,e):**

find\_set( $b$ ) =  $\{b\}$

find\_set( $e$ ) =  $\{e\}$

Union( $b, e$ )

**$S : \{a\}, \{b, e\}, \{c\}, \{d\}$**

**Groups = 4**

```
1: procedure OMASHU( $F$ )
2:   groups := 0
3:   for  $(a, b) \in F$  do
4:     if  $a = b$  then
5:       make_set( $a$ )
6:       groups := groups + 1
7:     end if
8:   end for
9:   for  $(a, b) \in F$  do
10:    if find_set( $a$ )  $\neq$  find_set( $b$ ) then
11:      union( $a, b$ )
12:      groups := groups - 1
13:    end if
14:  end for
15:  return groups
16: end procedure
```

$F = [ (a,a), (b,b), (c,c), (d,d), (e,e),$   
 $(b,e), (c,d) ]$

$S : \{a\}, \{b\}, \{c\}, \{d\}, \{e\}$

Groups = 5

**Revisamos (c,d):**

find\_set(c) = {c}

find\_set(d) = {d}

Union(c,d)

**$S : \{a\}, \{b, e\}, \{c, d\}$**

**Groups = 3**

```

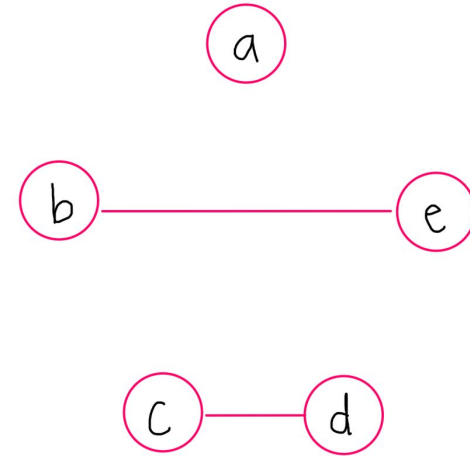
1: procedure OMASHU( $F$ )
2:    $groups := 0$ 
3:   for  $(a, b) \in F$  do
4:     if  $a = b$  then
5:       make_set( $a$ )
6:        $groups := groups + 1$ 
7:     end if
8:   end for
9:   for  $(a, b) \in F$  do
10:    if find_set( $a$ )  $\neq$  find_set( $b$ ) then
11:      union( $a, b$ )
12:       $groups := groups - 1$ 
13:    end if
14:  end for
15:  return  $groups$ 
16: end procedure

```

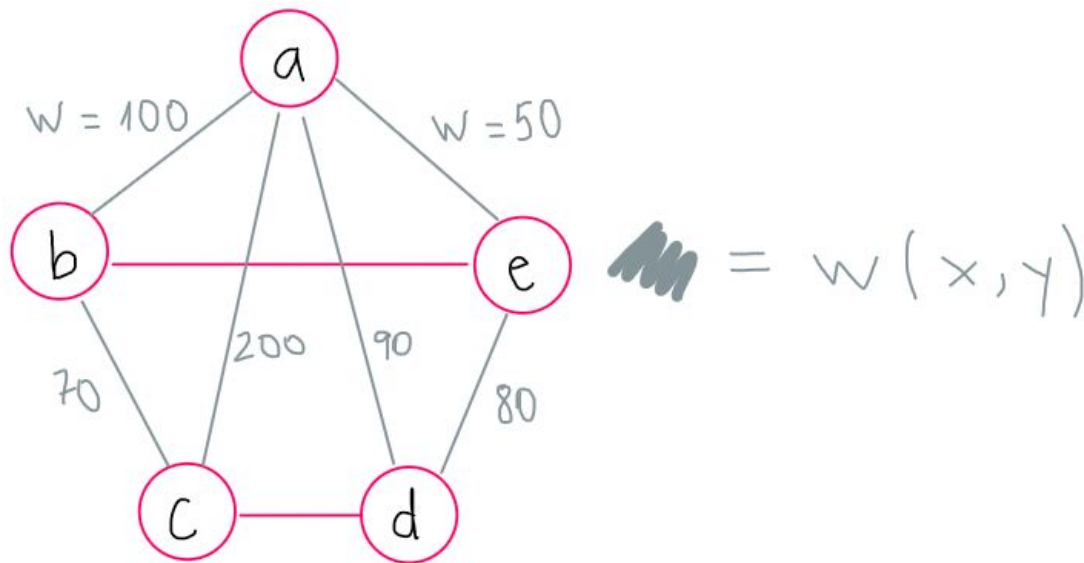
$F = [ (a,a), (b,b), (c,c), (d,d), (e,e), (b,e), (c,d) ]$

$S : \{a\}, \{b, e\}, \{c, d\}$

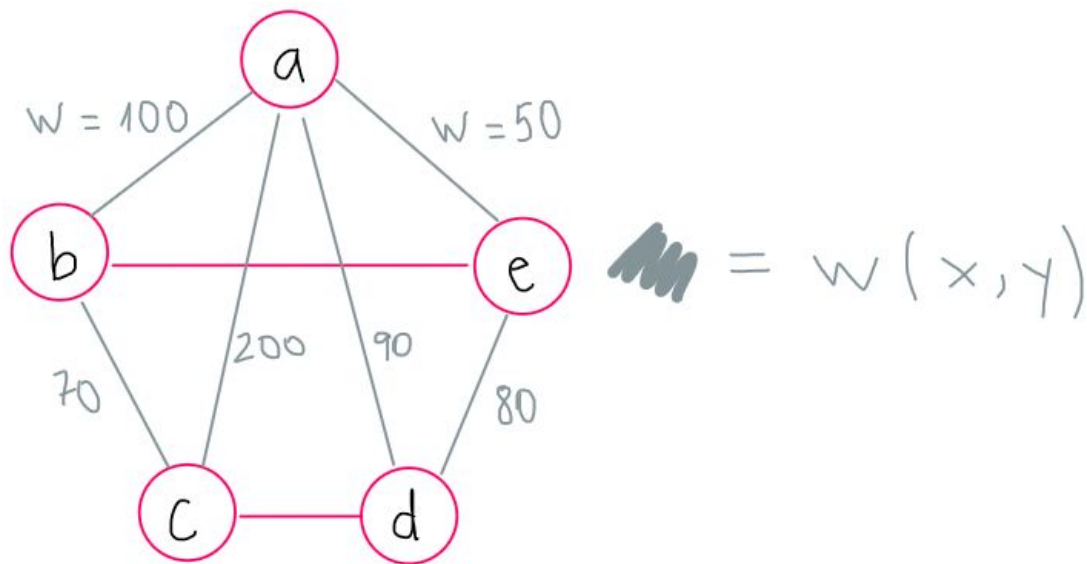
**Groups = 3**



b) Además de la lista F anterior, se te da una lista U con tríos de la forma  $(a, b, w)$ . Cada trío indica que **a y b no son amigos**, pero **podrían serlo** si se les paga una **cantidad positiva w de dinero**. Describe un algoritmo a lo más lineárítmico (es decir, del tipo  $n \log(n)$ ) que calcule el **costo mínimo** necesario para que todos los habitantes de Omashu **formen un solo gran grupo de amigos**.

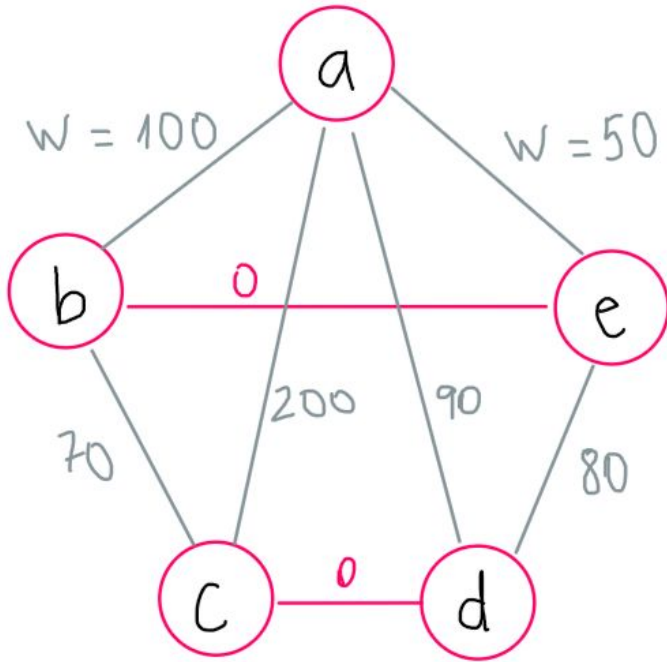


b) Además de la lista F anterior, se te da una lista U con tríos de la forma  $(a, b, w)$ . Cada trío indica que **a y b no son amigos**, pero **podrían serlo** si se les paga una **cantidad positiva w de dinero**. Describe un algoritmo a lo más lineárítmico (es decir, del tipo  $n \log(n)$ ) que calcule el **costo mínimo** necesario para que todos los habitantes de Omashu **formen un solo gran grupo de amigos**.



**QUEREMOS UN MST!!**

**¿Le falta algo al grafo?**



**Agregar costos faltantes  
de las amistades  
existentes.**

```
1: procedure OMASHU( $F, U$ )
2:   for  $(a, b) \in F$  do
3:     union( $U, (a, b, 0)$ )
4:   end for
5:   sort( $U$ ) de menor a mayor  $w$ 
6:    $cost := 0$ 
7:   for  $(a, b, w) \in U$  do
8:     if  $a = b$  then
9:       make_set( $a$ )
10:    end if
11:  end for
12:  for  $(a, b, w) \in U$  do
13:    if find_set( $a$ )  $\neq$  find_set( $b$ ) then
14:      union( $a, b$ )
15:       $cost := cost + w$ 
16:    end if
17:  end for
18:  return  $cost$ 
19: end procedure
```

Muy parecido a Kruskal.

Los que ya son amigos  
tienen costo 0.

```

1: procedure OMASHU( $F, U$ )
2:   for  $(a, b) \in F$  do
3:     union( $U, (a, b, 0)$ )
4:   end for
5:   sort( $U$ ) de menor a mayor  $w$ 
6:    $cost := 0$ 
7:   for  $(a, b, w) \in U$  do
8:     if  $a = b$  then
9:       make_set( $a$ )
10:    end if
11:  end for
12:  for  $(a, b, w) \in U$  do
13:    if find_set( $a$ )  $\neq$  find_set( $b$ ) then
14:      union( $a, b$ )
15:       $cost := cost + w$ 
16:    end if
17:  end for
18:  return  $cost$ 
19: end procedure

```

Luego los ordenamos de menor a mayor (por Kruskal).

Guardamos en la variable ***cost*** el costo hasta el momento.



```

1: procedure OMASHU( $F, U$ )
2:   for  $(a, b) \in F$  do
3:     union( $U, (a, b, 0)$ )
4:   end for
5:   sort( $U$ ) de menor a mayor  $w$ 
6:    $cost := 0$ 
7:   for  $(a, b, w) \in U$  do
8:     if  $a = b$  then
9:       make_set( $a$ )
10:    end if
11:  end for
12:  for  $(a, b, w) \in U$  do
13:    if find_set( $a$ )  $\neq$  find_set( $b$ ) then
14:      union( $a, b$ )
15:       $cost := cost + w$ 
16:    end if
17:  end for
18:  return  $cost$ 
19: end procedure

```

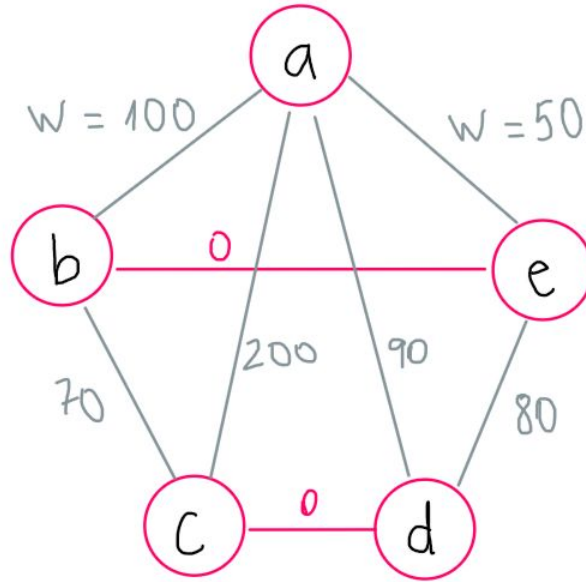
Se crean los sets iniciales,  
(uno para cada nodo).

```
1: procedure OMASHU( $F, U$ )
2:   for  $(a, b) \in F$  do
3:     union( $U, (a, b, 0)$ )
4:   end for
5:   sort( $U$ ) de menor a mayor  $w$ 
6:    $cost := 0$ 
7:   for  $(a, b, w) \in U$  do
8:     if  $a = b$  then
9:       make_set( $a$ )
10:    end if
11:  end for
12:  for  $(a, b, w) \in U$  do
13:    if find_set( $a$ )  $\neq$  find_set( $b$ ) then
14:      union( $a, b$ )
15:       $cost := cost + w$ 
16:    end if
17:  end for
18:  return  $cost$ 
19: end procedure
```

Modificación a Kruskal:  
Sumar el costo

# Ejercicio propuesto

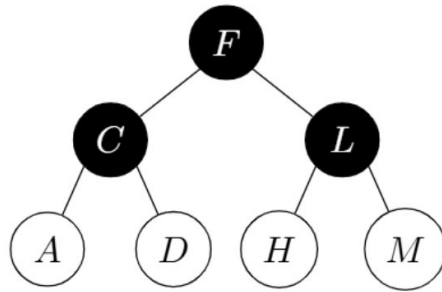
Encontrar un MST y su costo con el algoritmo anterior para el siguiente grafo.



# Árboles

## I2 2023-1 P1

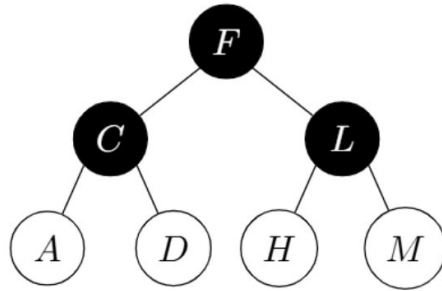
Considere el siguiente árbol  $T$  que es AVL y además rojo-negro (tomando las hojas  $A$ ,  $D$ ,  $H$ ,  $M$  como rojos)



Muestre el resultado de insertar, en el mismo orden, la secuencia de llaves  $J$ ,  $N$ ,  $G$ ,  $K$ ,  $E$  cuando  $T$  se considera...

# I2 2023-1 P1

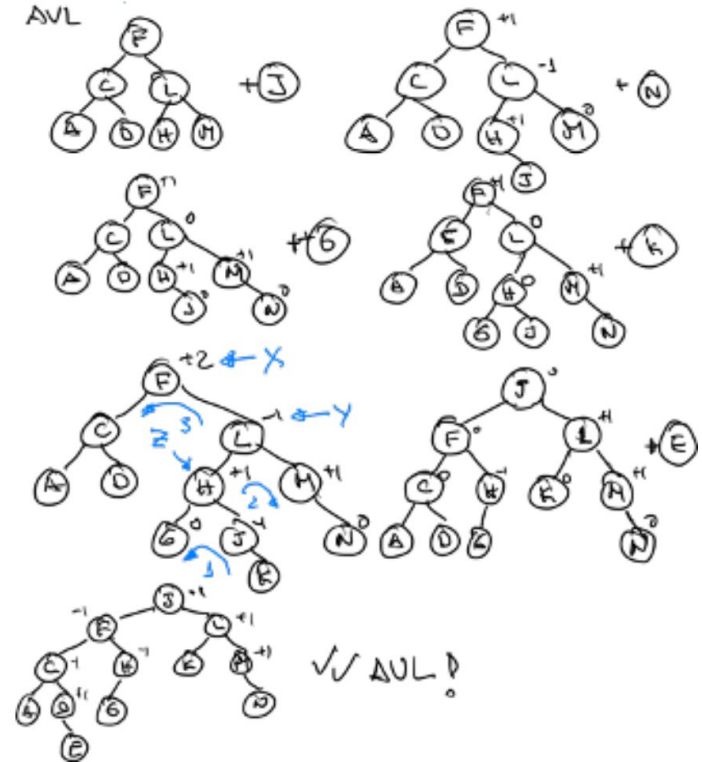
a) Un AVL (llaves J, N, G, K, E)



(En pizarra)

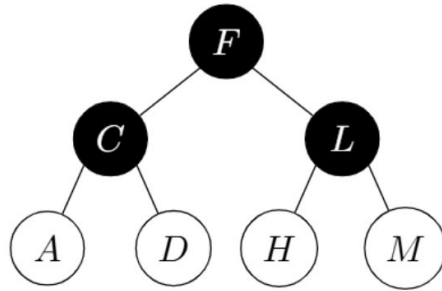
# I2 2023-1 P1

Yo iba a poner una solución más linda que esta, pero estamos en periodo de exámenes y no me dio el tiempo para hacer el grafo de los puffles esta vez sepan perdonar - Isa



# I2 2023-1 P1

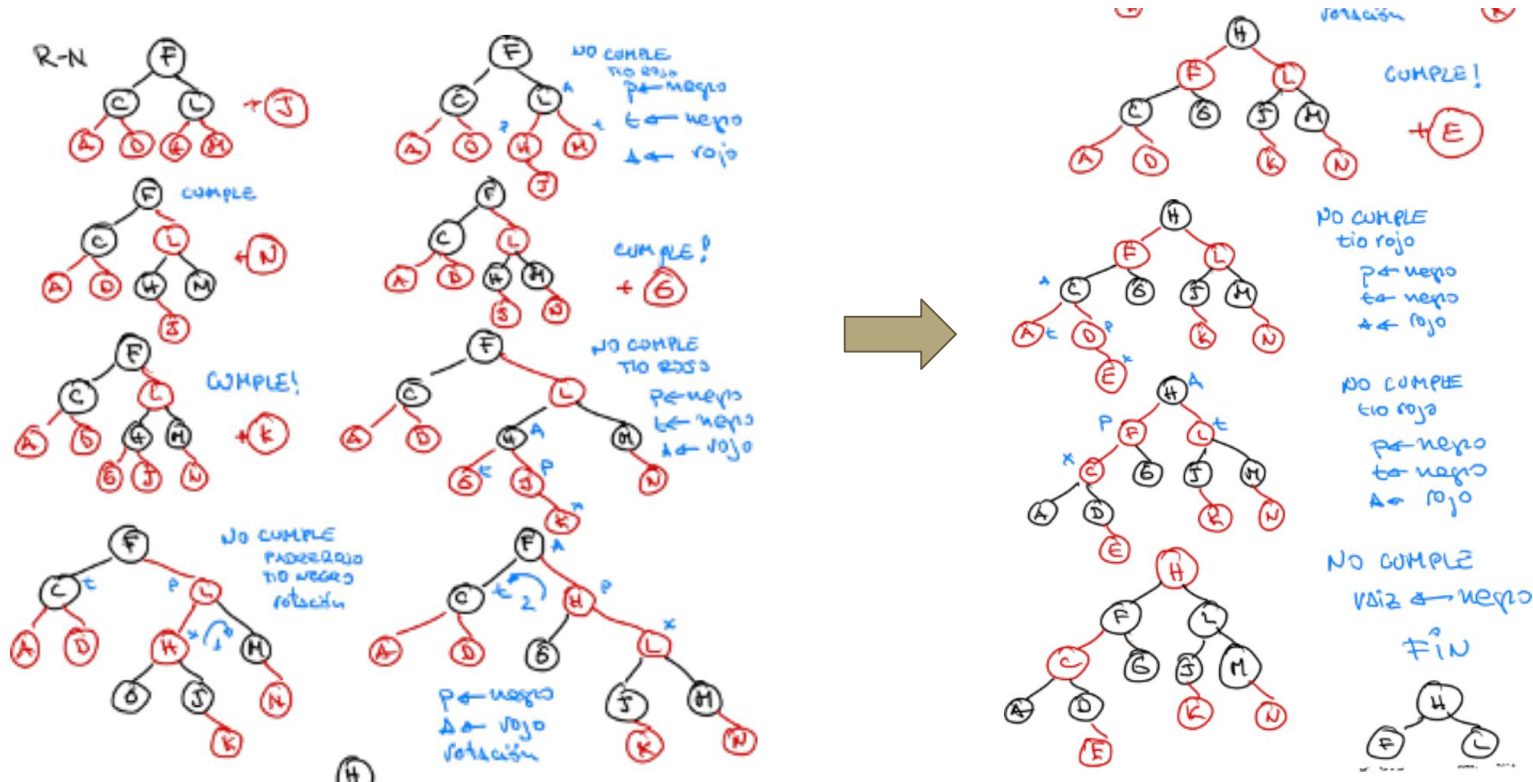
b) Un árbol RN (llaves J, N, G, K, E)



(En pizarra)



# I2 2023-1 P1



(no se alcanza a ver en la pauta de la prueba, pero al final cambiamos colores para quedar con nodo raíz negro)

## I2 2023-1 P1

c) Suponga que deseamos implementar un árbol binario de búsqueda de gran tamaño (muchos valores de llaves diferentes) y un volumen muy grande de inserciones. ¿Qué tipo de ABB elegiría entre AVL y rojo-negro? Justifique su respuesta.

## I2 2023-1 P1

Esta respuesta depende del contexto y de supuestos que puedan considerarse. Si nos centramos solo en la diferencia comparativa de la inserción, los árboles rojo-negro son más rápidos en la práctica por cuanto requieren menos operaciones para rebalancear. Si la secuencia de inserciones produce un árbol más desbalanceado que el correspondiente AVL, entonces puede haber una desventaja al usar rojonegro. Si asumimos que las inserciones son “uniformes”, entonces rojo-negro permite tener inserciones más rápidas de forma general.

# Dijkstra

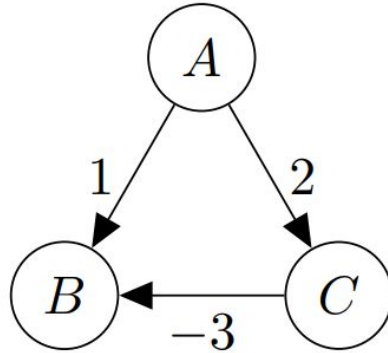
## I3 2023-1 P2

(a) Demuestre que el algoritmo de Dijkstra no siempre es óptimo cuando el grafo dirigido sobre el que opera tiene costos negativos.

## I3 2023-1 P2

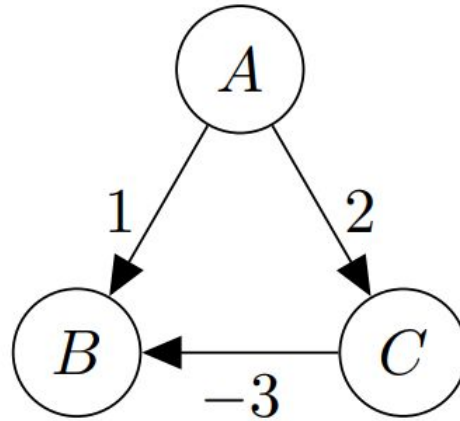
Demuestre que el algoritmo de Dijkstra no siempre es óptimo cuando el grafo dirigido sobre el que opera tiene costos negativos.

El algoritmo de Dijkstra entrega como solución de A hasta B el camino A, B con costo 1, mientras que el óptimo real es el camino A, C, B con costo  $-1$ .



## I3 2023-1 P2

¿Cómo podríamos calcular la ruta más barata para este ejemplo?



## I3 2023-1 P2

(b) Sea  $G$  un grafo dirigido con costos. Proponga el pseudocódigo de un algoritmo que entregue una lista con los nodos de algún ciclo de costo negativo, en caso que  $G$  tenga tal ciclo. En caso contrario, retorna una lista vacía.



# I3 2023-1 P2

```
NegativeCycle(s):
1  for  $u \in V$  :
2       $d[u] \leftarrow \infty$ ;  $\pi[u] \leftarrow \emptyset$ 
3   $d[s] \leftarrow 0$ 
4  for  $k = 1 \dots |V| - 1$  :
5      for  $(u, v) \in E$  :
6          if  $d[v] > d[u] + cost(u, v)$  :
7               $d[v] \leftarrow d[u] + cost(u, v)$ 
8               $\pi[v] \leftarrow u$ 
9  for  $(u, v) \in E$  :
10     if  $d[v] > d[u] + cost(u, v)$  :
11          $L \leftarrow$  lista con elemento  $v$ 
12          $current \leftarrow u$ 
13         while  $current \neq v$  :
14             agregar a  $L$  el nodo  $current$ 
15              $current \leftarrow \pi[current]$ 
16  return Lista vacía
```

Notemos que el algoritmo es esencialmente *ValidBellmanFord*, al cual se le cambia su retorno booleano. En caso que exista ciclo negativo (resultado false del algoritmo visto en clases), se construye la lista visitando ancestros del nodo que genera el ciclo. En caso contrario, se retorna lista vacía.

## i3 2023-1 P2

Ahora, este método solo indica ciclo negativo desde una fuente. Si existe un ciclo negativo, pero no es alcanzable desde un nodo específico, tenemos que asegurarnos de detectarlo. Para esto, se puede ejecutar este método desde todos los vértices. Es decir

```
NegativeCycleGlobal():  
1   for  $u \in V$  :  
2        $L \leftarrow \text{NegativeCycle}(u)$   
3       if  $L \neq \emptyset$  :  
4           return  $L$   
5   return Lista vacía
```

## i3 2023-1 P2

(c) Determine la complejidad de su algoritmo.

El algoritmo de detección desde una sola fuente *NegativeCycle* es  $O(EV)$  al igual que BellmanFord. Los  $V$  llamados en el método global hacen que esta solución sea  $O(EV^2)$