# Árbol Generador Mínimo

Ezequiel Companeetz, Agustin Venegas (aka Chimi)

DC - FCEN, UBA

16 de Octubre, 2024

UNIVERSIDAD DE BUENOS AIRES

1 Repaso AGM

2 Ejercicios

1 Repaso AGM
- Árbol Generador
- Árbol Generador Mínimo
- Caminos MaxiMin y MiniMax
- Vínculo entre MiniMax y AGM
- Aplicaciones
- Kruskal
- Prim

2 Ejercicios

1 Repaso AGM
- Árbol Generador
- Árbol Generador Mínimo
- Caminos MaxiMin y MiniMax
- Vínculo entre MiniMax y AGM
- Aplicaciones
- Kruskal
- Prim

2 Ejercicios

# Definiciones

## Árbol Generador (AG)

Sea $G$ un grafo. Decimos que $T$ es un *árbol generador* de $G$ si se cumplen estas condiciones:

- $T$ es subgrafo de $G$
- $T$ es un árbol
- $T$ tiene todos los vértices de $G$

# Definiciones

## Árbol Generador (AG)

Sea $G$ un grafo. Decimos que $T$ es un *árbol generador* de $G$ si se cumplen estas condiciones:

- $T$ es subgrafo de $G$
- $T$ es un árbol
- $T$ tiene todos los vértices de $G$

## Costo de un árbol generador

Sea $G$ un grafo con pesos en sus aristas. Si $T$ es un *árbol generador* de $G$, definimos el *costo* de $T$ como la suma de los pesos de las aristas de $T$.

1 Repaso AGM
- Árbol Generador
- Árbol Generador Mínimo
- Caminos MaxiMin y MiniMax
- Vínculo entre MiniMax y AGM
- Aplicaciones
- Kruskal
- Prim

2 Ejercicios

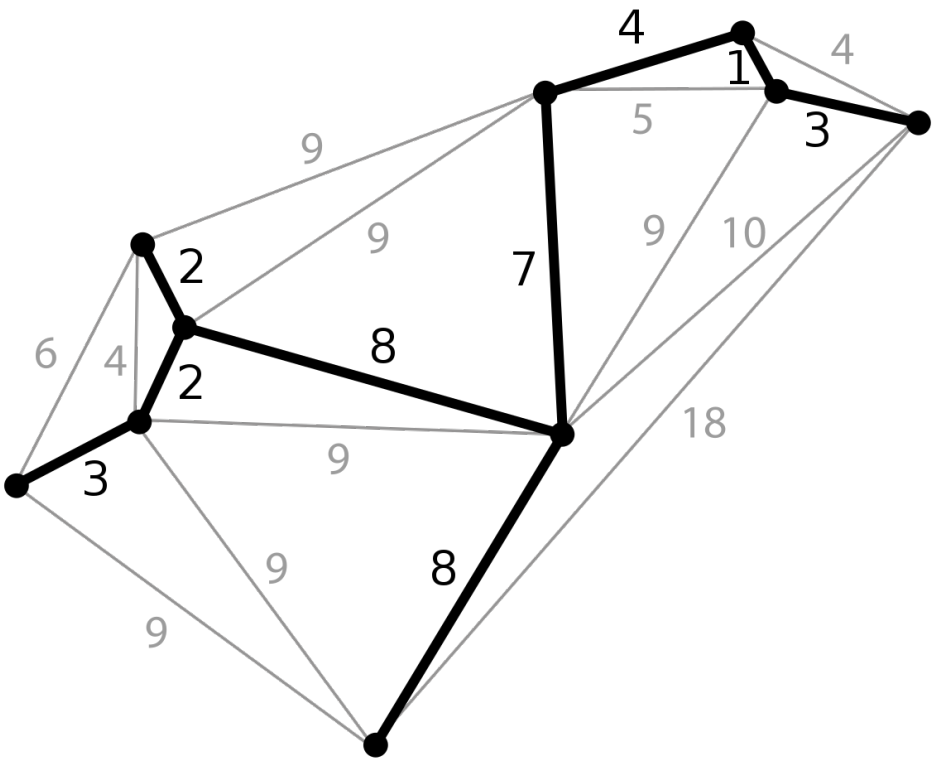
# Definiciones

## Árbol generador mínimo

Sea $G$ un grafo. Decimos que $T$ es un *árbol generador mínimo* de $G$ si se cumplen estas condiciones:

- $T$ es árbol generador de $G$
- El costo de $T$ es mínimo con respecto a todos los árboles generadores de $G$

Acá vemos un grafo $G$ (en gris) y un subgrafo $T$ (en negro) que es AGM de $G$.



1 Repaso AGM
- Árbol Generador
- Árbol Generador Mínimo
- Caminos MaxiMin y MiniMax
- Vínculo entre MiniMax y AGM
- Aplicaciones
- Kruskal
- Prim

2 Ejercicios

# Camino MaxiMin

## Definición (Camino MaxiMin)

Sea $G$ un grafo , $c : E(G) \to \mathbb{R}$ su función de costos y $v, w \in V(G)$. Decimos que $P$ es un *camino MaxiMin* de $v$ a $w$ si se cumple que P maximiza $c_{min}$:

$$c_{min}(P) = \text{mín}\{c(vw) |\ vw \in E(P)\}$$

# Camino MaxiMin

## Definición (Camino MaxiMin)

Sea $G$ un grafo , $c : E(G) \to \mathbb{R}$ su función de costos y $v, w \in V(G)$. Decimos que $P$ es un *camino MaxiMin* de $v$ a $w$ si se cumple que P maximiza $c_{min}$:

$$c_{min}(P) = \text{mín}\{c(vw) |\ vw \in E(P)\}$$

También se lo conoce cómo el problema del *camino más ancho.*[a]

[a] Si nos imaginamos los costos como capacidades, entonces queremos maximizar la capacidad que nuestro camino puede trasladar. Y la capacidad de un camino está dada por la capacidad de su arista más pequeña.

# Camino MiniMax

## Definición (Camino MiniMax)

Sea $G$ un grafo , $c : E(G) \to \mathbb{R}$ su función de costos y $v, w \in V(G)$. Decimos que $P$ es un *camino MiniMax* de $v$ a $w$ si se cumple que P minimiza $c_{max}$:

$$c_{max}(P) = \text{máx}\{c(vw) |\ vw \in E(P)\}$$

1 Repaso AGM
- Árbol Generador
- Árbol Generador Mínimo
- Caminos MaxiMin y MiniMax
- Vínculo entre MiniMax y AGM
- Aplicaciones
- Kruskal
- Prim

2 Ejercicios

## Vínculo entre MiniMax y AGM

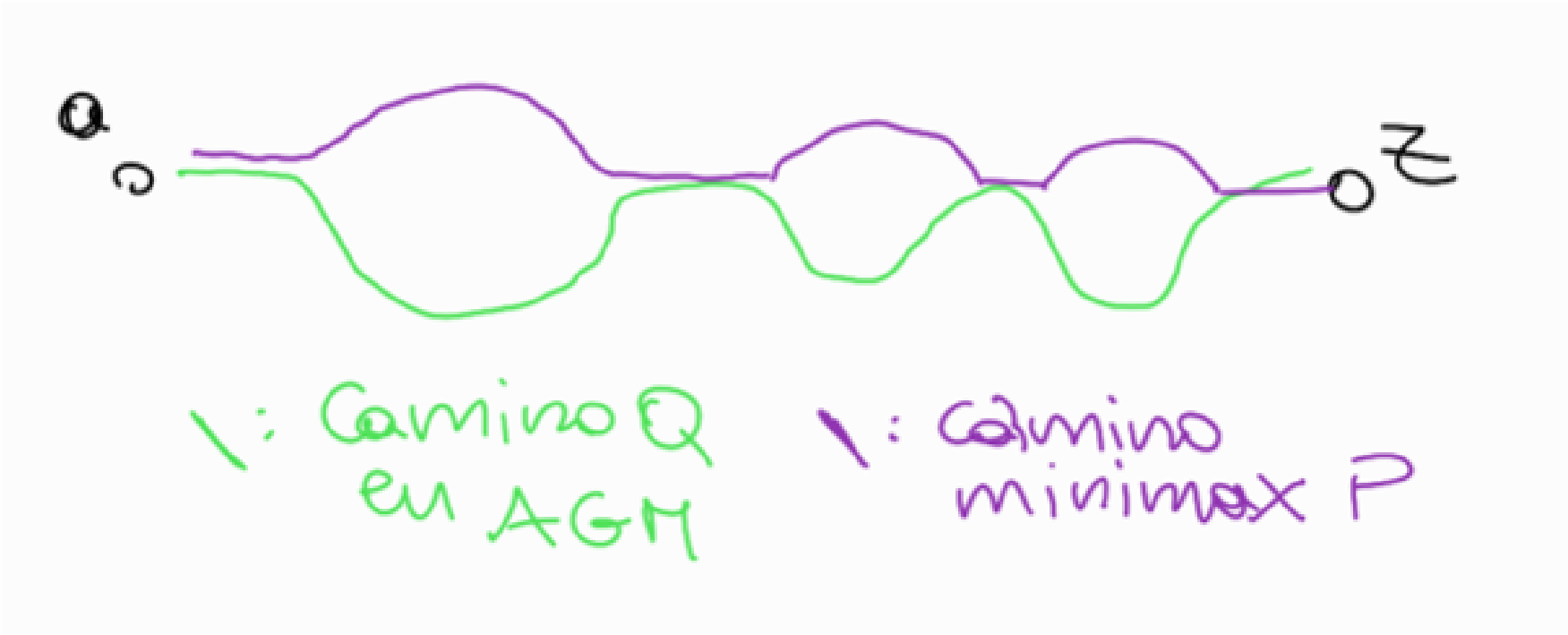
¿Que herramientas de las que vimos creen que podemos utilizar para mostrar que *camino AGM* $\Rightarrow$ *camino Minimax*?

## Vínculo entre MiniMax y AGM

¿Que herramientas de las que vimos creen que podemos utilizar para mostrar que *camino AGM* $\Rightarrow$ *camino Minimax*? ¡Exacto! Podemos usar cualquiera, pero en este caso vamos a utilizar el absurdo.

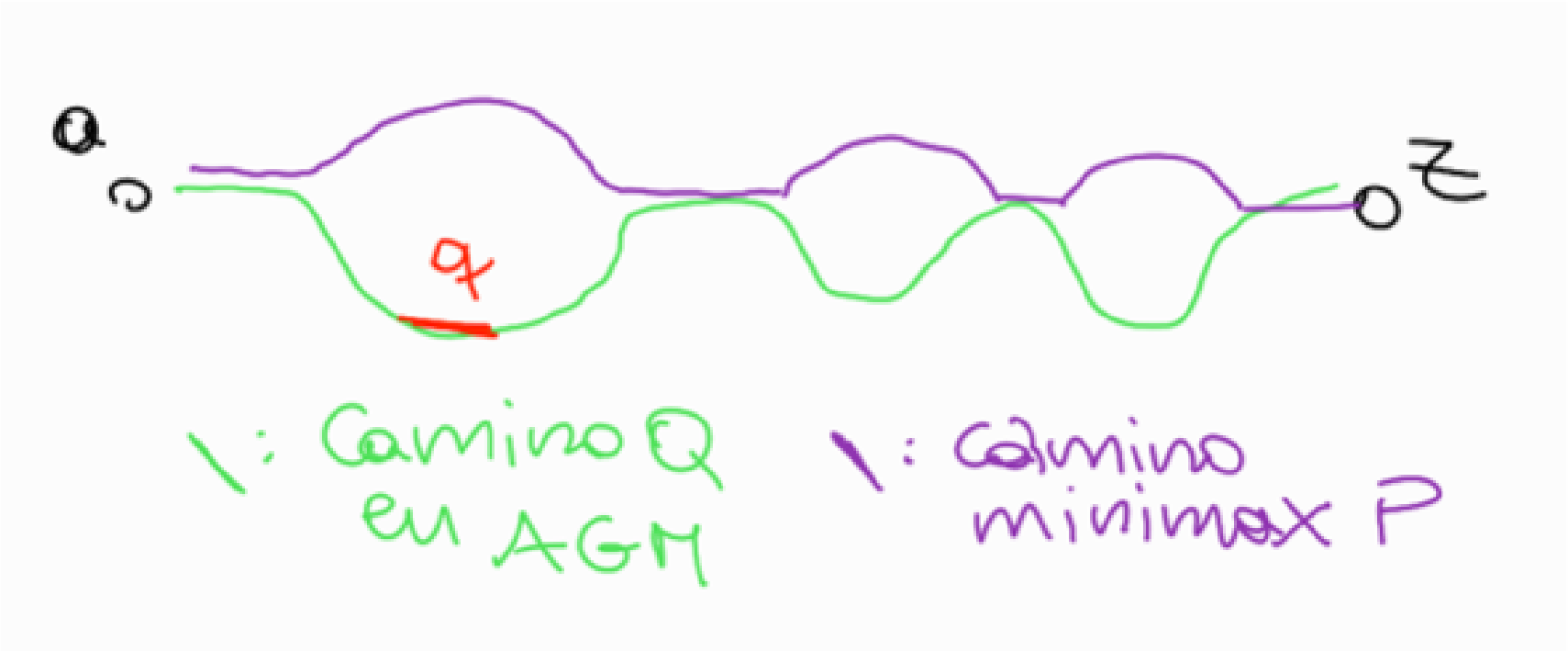
# Vínculo entre MiniMax y AGM

Supongamos que existe un camino $Q$ en un AGM $T$ entre dos vértices $a$ y $z$ que no es minimax. Existe entonces otro camino $P$ entre $a$ y $z$ que es minimax, cuya máxima arista será más chica que la máxima arista de $Q$.



# Vínculo entre MiniMax y AGM

Los caminos $P$ y $Q$ pueden compartir algunas aristas, pero seguro que no comparten la máxima arista de $Q$, porque sino sus máximas aristas serían la misma. Llamemos $q$ a la arista más grande de $Q$.

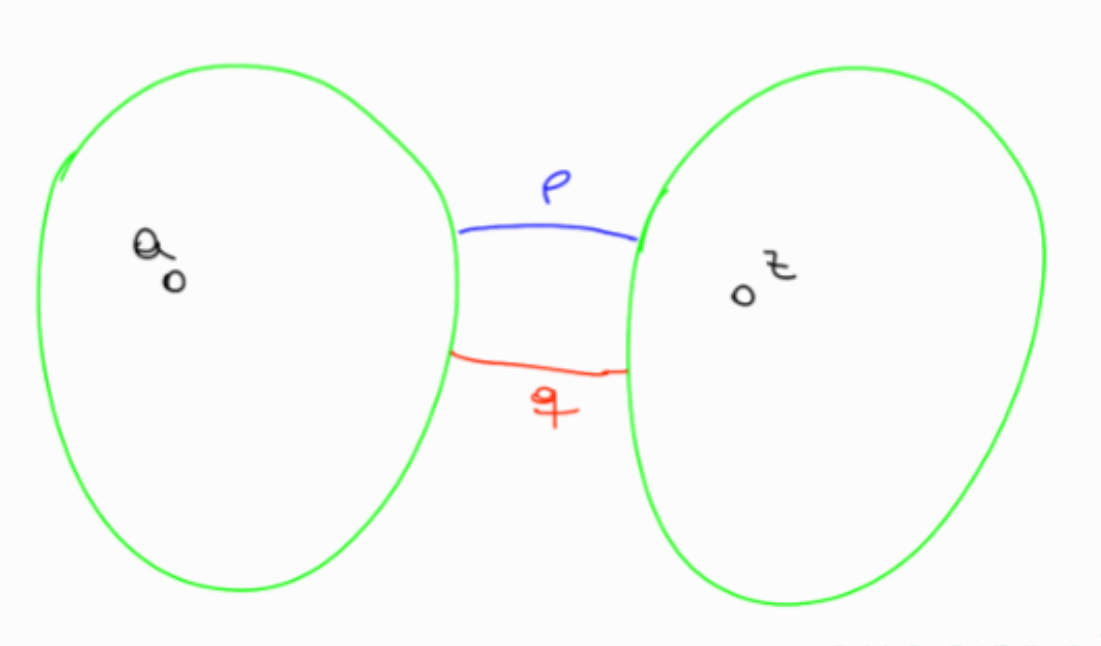


## Vínculo entre MiniMax y AGM

Vamos a querer reemplazar $q$ por alguna arista de $P$ para conseguir otro AG. Como la máxima arista de $P$ es más chica que $q$, esto nos resultaría en un AG de menor peso que $T$, implicando que $T$ no era un AGM, un absurdo. Veamos si podemos encontrar una arista de $P$ que nos sirva.

# Vínculo entre MiniMax y AGM

Si sacamos la arista $q$ del AGM $T$ para conseguir un $T'$, nos van a quedar dos componentes conexas en $T'$: una que contenga $a$, y otra que contenga $z$. Necesariamente hay una arista $p$ en $P$ que conecta estas dos componentes conexas, porque sino $P$ no conectaría a $a$ y $z$. Además, $p$ no está contenida en el AGM $T$, porque sino también estaría en $T'$, o sea que $T'$ no tendría dos componentes conexas.



Nos generamos entonces el grafo $T^*$ que se obtiene de reemplazar la arista $q$ por la arista $p$ en el AGM $T$. Vemos lo siguiente:

1. $T^*$ es generador, porque tiene los mismos vértices que $T$.
2. $T^*$ es conexo, porque $p$ conecta las dos componentes conexas que resultan de eliminar $q$ de $T$.
3. $T^*$ es un árbol, porque tiene la misma cantidad de aristas que $T$, y es conexo.

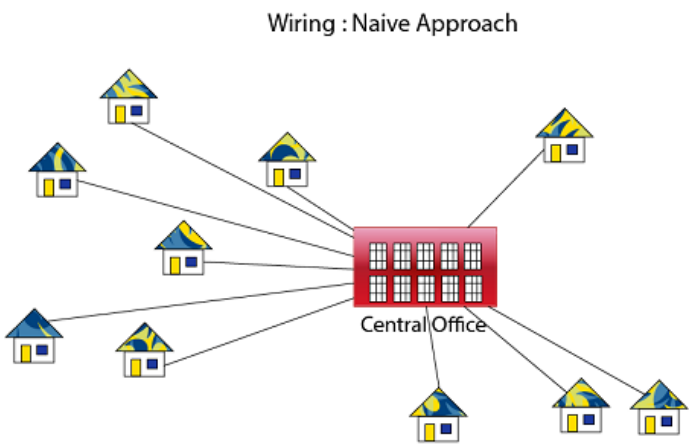
Esto significa que $T^*$ es un AG. Como $p$ es más chica que $q$, el peso de $T^*$ es menor que el de $T$, y entonces encontramos un AG con peso menor que $T$. **ABSURDO,** que vino de suponer que $T$ es un AGM con un camino que no es minimax. □

1 Repaso AGM
- Árbol Generador
- Árbol Generador Mínimo
- Caminos MaxiMin y MiniMax
- Vínculo entre MiniMax y AGM
- Aplicaciones
- Kruskal
- Prim

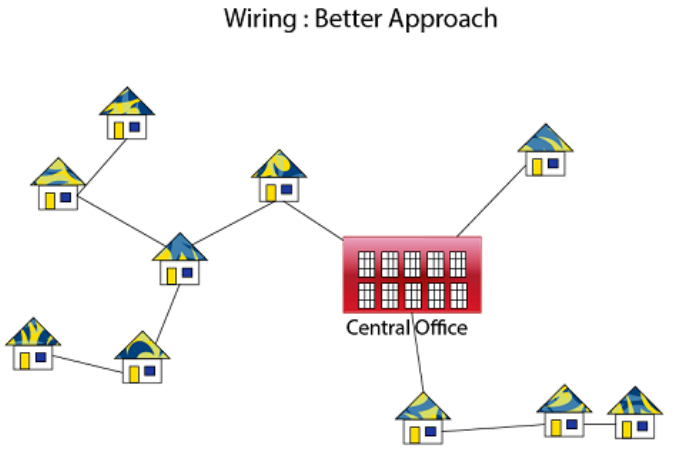
2 Ejercicios

# Aplicaciones del AGM

## Red de electricidad de una ciudad




Expensive!




Minimize the total length of wire connecting the customers

# Más aplicaciones del AGM!

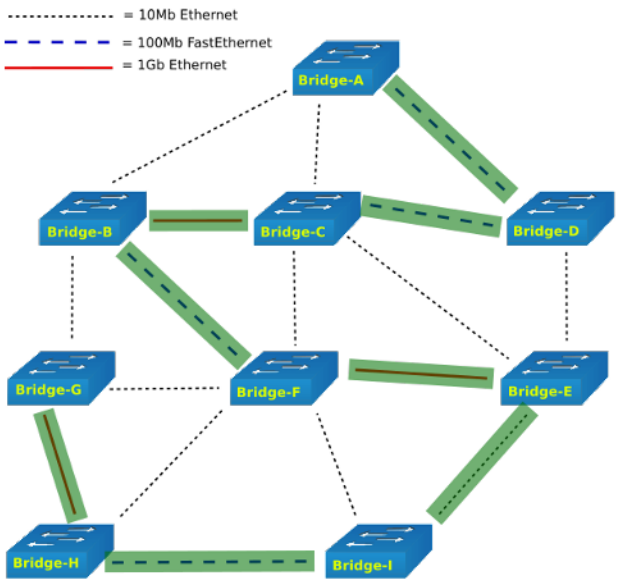



Figura 1: STP (Spanning Tree Protocol)

Figura 2: Aproximación para TSP (Traveling Salesman Problem)

1 Repaso AGM
- Árbol Generador
- Árbol Generador Mínimo
- Caminos MaxiMin y MiniMax
- Vínculo entre MiniMax y AGM
- Aplicaciones
- Kruskal
- Prim

2 Ejercicios

# Algoritmo de Kruskal

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ G.V
    MAKE-SET(v)
create a single list of the edges in G.E
sort the list of edges into monotonically increasing order by weight w
for each edge (u, v) taken from the sorted list in order
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

1 Repaso AGM
- Árbol Generador
- Árbol Generador Mínimo
- Caminos MaxiMin y MiniMax
- Vínculo entre MiniMax y AGM
- Aplicaciones
- Kruskal
- Prim

2 Ejercicios

# Algoritmo de Prim

```
MST-PRIM(G, w, r)
for each vertex u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = ∅
for each vertex u ∈ G.V
    INSERT(Q, u)
while Q ≠ ∅
    u = EXTRACT-MIN(Q)          // add u to the tree
    for each vertex v in G.Adj[u]   // update keys of u's non-tree neighbors
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
            DECREASE-KEY(Q, v, w(u, v))
```

1 Repaso AGM

2 Ejercicios
- Viaje en peligro
- Conjuntos deseables
- Audífonos Defectuosos
- Alimentando hormigas
- Rutas y aeropuertos

1 Repaso AGM

2 Ejercicios
- Viaje en peligro
- Conjuntos deseables
- Audífonos Defectuosos
- Alimentando hormigas
- Rutas y aeropuertos

# Ejercicio 1

## Viaje en peligro

Cifu vive en Kruskal, Rusia, y quiere visitar las ciudades locales, pero su localidad no tiene rutas que lo conecten con el resto. En total hay $n$ ciudades, pero el presidente sólo le ofrece construir $k \ll n$ rutas, poniéndole 2 condiciones:

1. Que queden conectadas $k + 1$ localidades.
2. Que la red resultante sea una subred de la red que conecta a todas las localidades con costo mínimo.

Sabemos las ubicaciones de las localidades y que el costo de construir una ruta entre la localidad $x$ y la $y$ se calcula como $r \cdot \texttt{distEnKm}(x, y) + c_{x,y}$, donde $c_{x,y}$ depende de las localidades. Además sabemos en qué localidad se encuentra Cifu. Queremos una red que cumpla lo pedido. ¿Es única?

## Especificaciones

- El formato del input es una línea que contiene dos enteros $n$ (cantidad de ciudades) y $k$ (cantidad de rutas). Luego tenemos $n$ lineas que tienen el formato $x_i, y_i$, las cuales representan la posición de la $i$-ésima ciudad.
- Tenemos que devolver la red de rutas que cumpla lo pedido.

## Analicemos el problema

- El problema nos pide que queden $k+1$ localidades conectadas a partir de $k$ rutas, por lo que podemos inferir que nos está pidiendo generar un árbol.
- Luego sabemos que $k \ll n$, por lo que el árbol que vamos a generar va a ser un subgrafo del grafo completo.

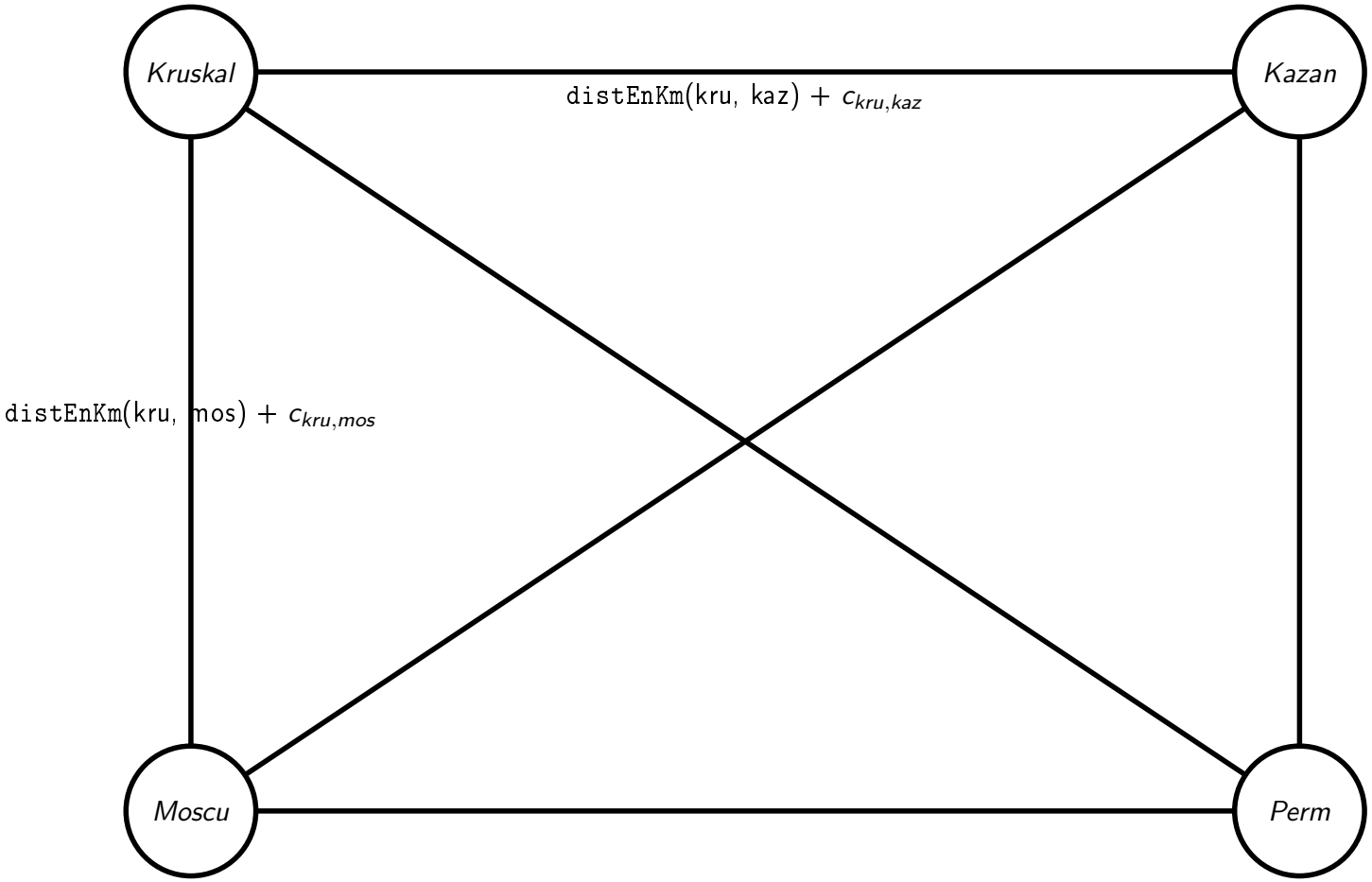
## Analicemos el problema

- El problema nos pide que queden $k+1$ localidades conectadas a partir de $k$ rutas, por lo que podemos inferir que nos está pidiendo generar un árbol.
- Luego sabemos que $k \ll n$, por lo que el árbol que vamos a generar va a ser un subgrafo del grafo completo.
- También nuestro algoritmo va a tener que comenzar desde la localidad de Cifu, puesto que queremos que la misma quede conectada.

## Analicemos el problema

- El problema nos pide que queden $k+1$ localidades conectadas a partir de $k$ rutas, por lo que podemos inferir que nos está pidiendo generar un árbol.
- Luego sabemos que $k \ll n$, por lo que el árbol que vamos a generar va a ser un subgrafo del grafo completo.
- También nuestro algoritmo va a tener que comenzar desde la localidad de Cifu, puesto que queremos que la misma quede conectada.
- Por último como queremos que el presupuesto utilizado sea el menor posible vamos a modelar este problema utilizando *AGM*.

# Dibujemos un ejemplo



## Solución

- Creamos un grafo con un vértice para cada localidad.
- Calculamos las distancias en kilómetros entre las distintas localidades.
- A partir de las distancias y los costos $c_{x,y}$ entre las localidades creamos las aristas con sus pesos asociados.
  $costo(x, y) = \texttt{distEnKm}(x, y) + c_{x,y}$

## Solución

- Creamos un grafo con un vértice para cada localidad.
- Calculamos las distancias en kilómetros entre las distintas localidades.
- A partir de las distancias y los costos $c_{x,y}$ entre las localidades creamos las aristas con sus pesos asociados.
  $costo(x, y) = \texttt{distEnKm}(x, y) + c_{x,y}$
- Corremos *Prim* (¿por qué no Kruskal?) sobre nuestro grafo para obtener el *AGM* parcial de tamaño $k$. ¿Cuál es la complejidad?

## Solución

- Creamos un grafo con un vértice para cada localidad.
- Calculamos las distancias en kilómetros entre las distintas localidades.
- A partir de las distancias y los costos $c_{x,y}$ entre las localidades creamos las aristas con sus pesos asociados.
  $costo(x, y) = \texttt{distEnKm}(x, y) + c_{x,y}$
- Corremos *Prim* (¿por qué no Kruskal?) sobre nuestro grafo para obtener el *AGM* parcial de tamaño $k$. ¿Cuál es la complejidad?
- Construir el grafo nos va a costar $\mathcal{O}(n^2)$ (pues es el grafo completo). Luego si usamos la implementación $\mathcal{O}(n^2)$ de Prim la complejidad nos va a quedar $\mathcal{O}(nk)$. Como $k << n$ la complejidad total es $\mathcal{O}(n^2)$. Con $n =$ cantidad de ciudades.

# Aclaración

## Invariante de Prim

La consigna pide *que la red resultante sea una subred de la red que conecta a todas las localidades con costo mínimo*. ¿Por qué pide eso y no que sea directamente *la red que contenga a Kruskal que conecta a k localidades con costo mínimo*?

# Aclaración

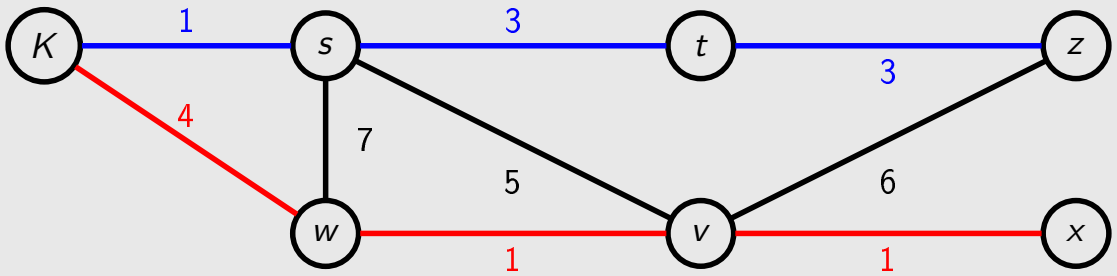
## Invariante de Prim

La consigna pide *que la red resultante sea una subred de la red que conecta a todas las localidades con costo mínimo*. ¿Por qué pide eso y no que sea directamente *la red que contenga a Kruskal que conecta a k localidades con costo mínimo*? Esto se debe a que el invariante de Prim cumple que, en su $i$-ésimo paso, va a tener un subgrafo de tamaño $i$ del AGM, no el árbol de $i$ aristas mínimo entre los que contengan a la raíz.

# Aclaración

## Invariante de Prim

En este grafo, por ejemplo, si arrancamos del nodo $K$ el árbol de $i$ aristas que nos va a generar Prim (el azul) no va a ser el mínimo (el rojo).



1 Repaso AGM

2 Ejercicios
- Viaje en peligro
- Conjuntos deseables
- Audífonos Defectuosos
- Alimentando hormigas
- Rutas y aeropuertos

# Conjuntos deseables

## Conjuntos deseables

Dado $G = (V, E)$ un grafo pesado con función de costos $c : E \to \mathbb{N}$, decimos que un conjunto de nodos $D \subseteq V$ es *deseable* si:

1. el subgrafo inducido por $D$ es conexo; y
2. toda arista de $G$ que *sale* de $D$ tiene peso **estrictamente mayor** que cualquier arista *interna* a $D$.

## Conjuntos deseables - Items

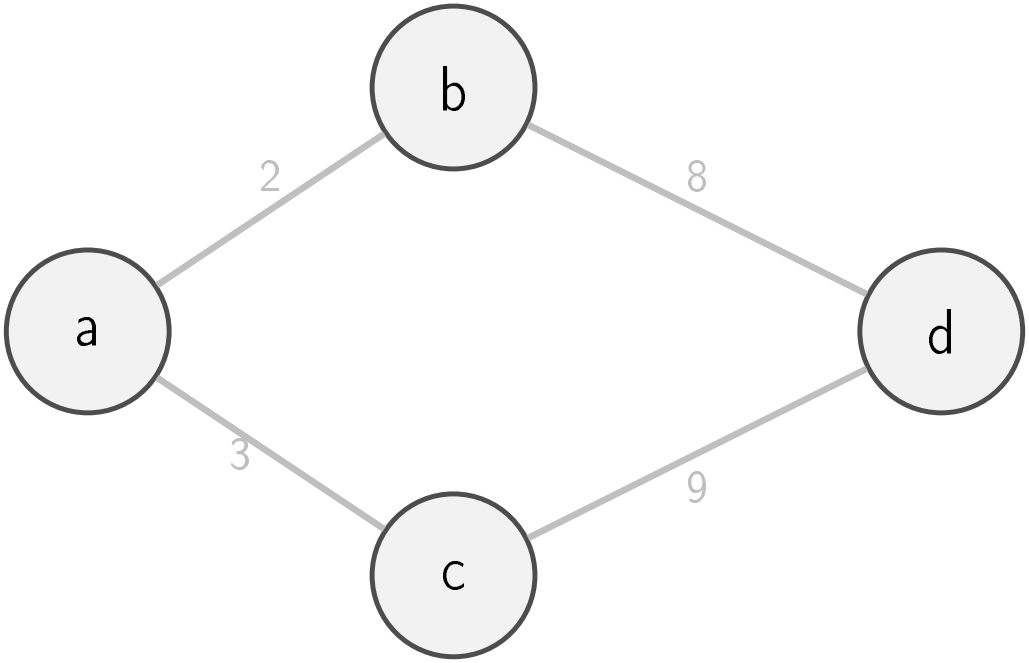
Sean $B_i$ los bosques que obtiene Kruskal tras procesar las primeras $i-1$ aristas (en orden creciente por $c$).

a) Probar que si $D$ es deseable entonces existe alguna iteración $1 \leq j \leq |E|$ tal que $D$ es una de las componentes conexas de $B_j$.

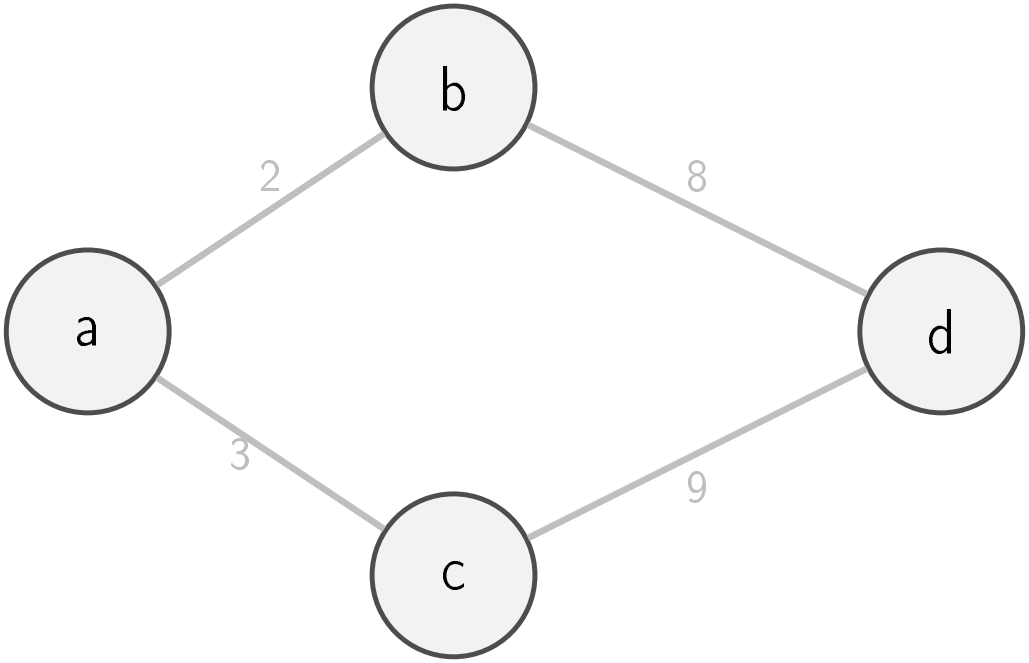
b) Dar un algoritmo eficiente que cuente la cantidad de conjuntos deseables de $G$. Objetivo: $O(mn)$.[1]

[1] La mejor solución conocida es $O(m \log n)$.

# Veamos un ejemplo



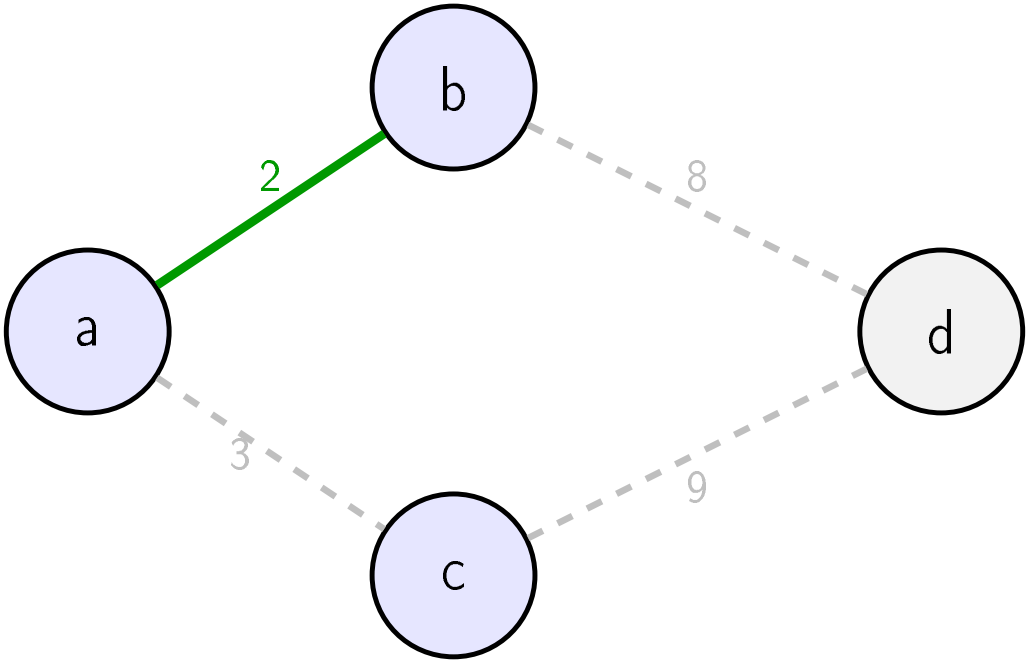
# Veamos un ejemplo




¿Cómo encontrarían un conjunto deseable en este grafo?

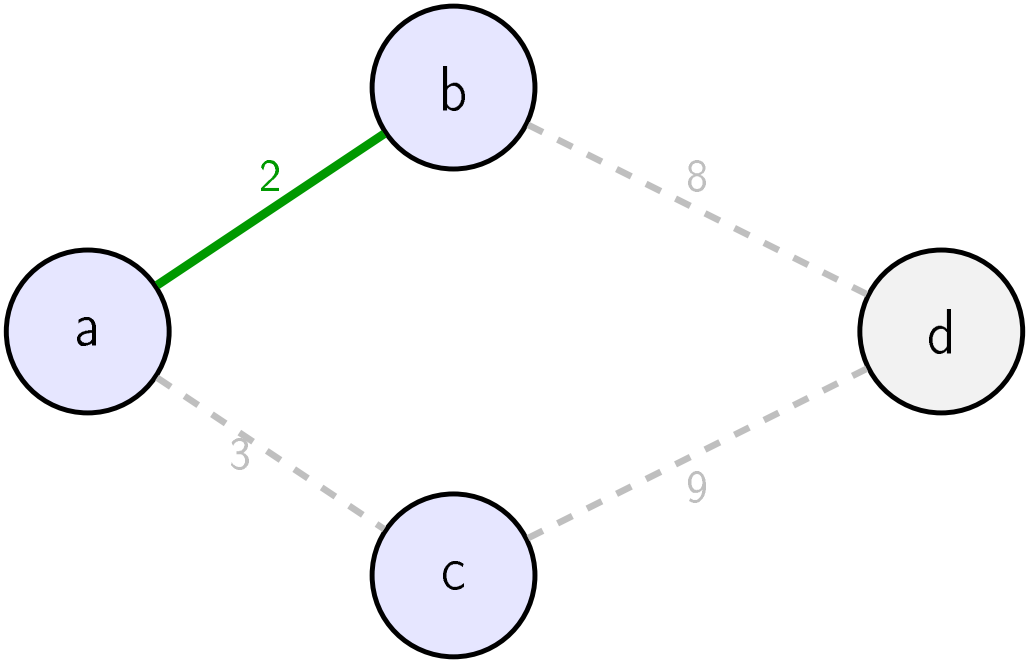
# Primera iteración de Kruskal

- Kruskal va agregando aristas mientras no formen ciclos.



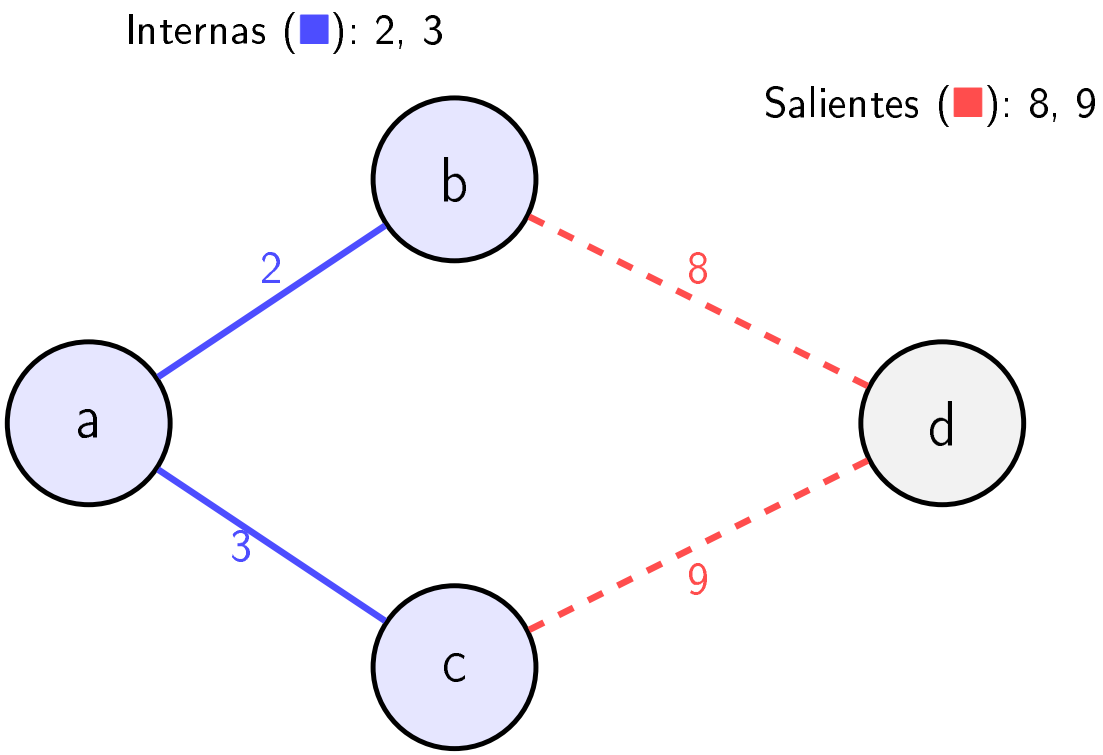
# Primera iteración de Kruskal

- Kruskal va agregando aristas mientras no formen ciclos.

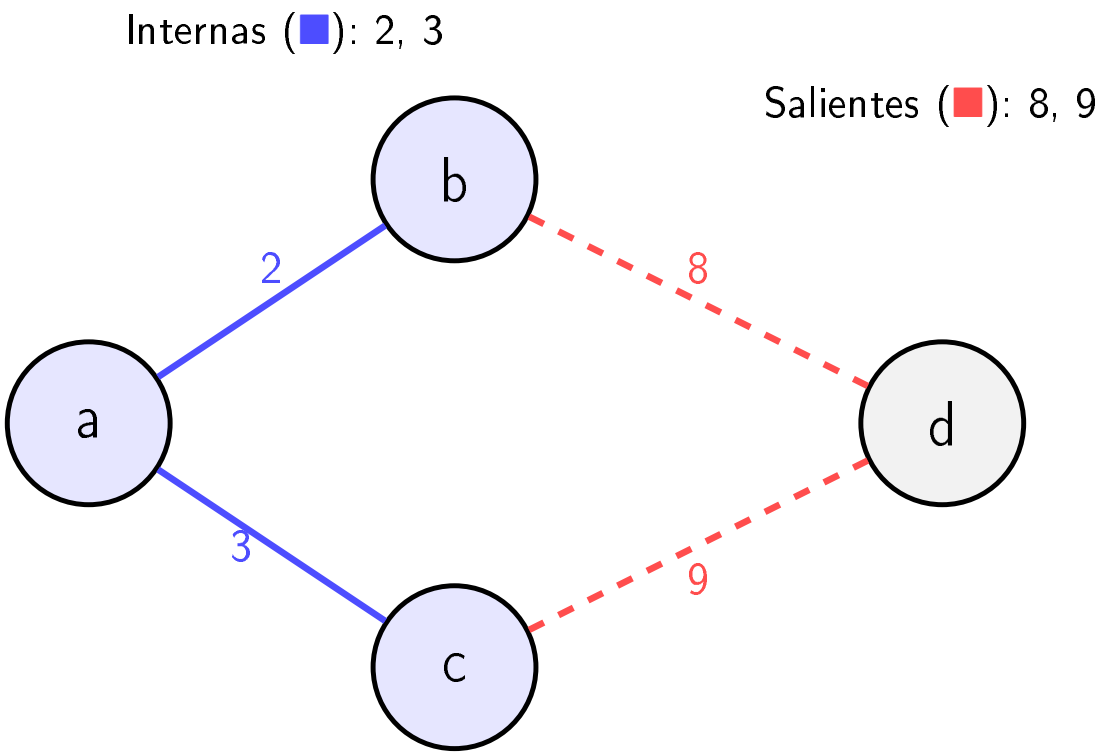



Después de agregar $(a, b)$, Kruskal tiene la componente $\{a, b\}$ y los vértices $c, d$ aún aislados.

# Ejemplo - Conjunto deseable



# Ejemplo - Conjunto deseable




Aquí consideramos el conjunto $D = \{a, b, c\}$. Las aristas internas $(2, 3)$ son más baratas que las salientes $(8, 9)$. Por lo tanto, $D$ es **deseable**: está conectado y ninguna arista externa es más barata que las internas.

# Parte (a): Afirmación

## Afirmación

Si $D \subseteq V$ es deseable, entonces existe un índice $j$ tal que $D$ es una de las **componentes conexas** del bosque $B_j$ obtenido por Kruskal tras procesar las primeras $j-1$ aristas (en orden no decreciente por peso).

–

# Parte (a): Definiciones

## Definiciones

Sea

$$\alpha = \text{mín}\{\, c(e) \;:\; e \text{ cruza el corte } (D, V \setminus D)\,\}$$

$$\beta = \text{máx}\{\, c(e) \;:\; e \text{ es interna a } D\,\}.$$

# Parte (a): Definiciones

## Definiciones

Sea

$$\alpha = \text{mín}\{\, c(e) \;:\; e \text{ cruza el corte } (D, V \setminus D)\,\}$$

$$\beta = \text{máx}\{\, c(e) \;:\; e \text{ es interna a } D\,\}.$$

- $\alpha$ representa el peso más bajo entre las aristas que **salen** de $D$.

# Parte (a): Definiciones

## Definiciones

Sea

$$\alpha = \text{mín}\{\, c(e) \;:\; e \text{ cruza el corte } (D, V \setminus D)\,\}$$

$$\beta = \text{máx}\{\, c(e) \;:\; e \text{ es interna a } D\,\}.$$

- $\alpha$ representa el peso más bajo entre las aristas que **salen** de $D$.
- $\beta$ representa el peso más alto entre las aristas **internas** a $D$.

# Parte (a): Definiciones

## Definiciones

Sea

$$\alpha = \text{mín}\{\, c(e) \;:\; e \text{ cruza el corte } (D, V \setminus D)\,\}$$

$$\beta = \text{máx}\{\, c(e) \;:\; e \text{ es interna a } D\,\}.$$

- $\alpha$ representa el peso más bajo entre las aristas que **salen** de $D$.
- $\beta$ representa el peso más alto entre las aristas **internas** a $D$.
- Como $D$ es deseable, se cumple que para toda arista interna $e_{\text{in}}$ y toda saliente $e_{\text{out}}$:

$$c(e_{\text{in}}) < c(e_{\text{out}}) \quad \Longrightarrow \quad \beta < \alpha.$$

# Parte (a): Estructura de la prueba

**Distinguimos dos casos:**

# Parte (a): Estructura de la prueba

**Distinguimos dos casos:**

## Caso $|D| = 1$

Trivial: $D$ es una componente de $B_0$, ya que al inicio del algoritmo cada vértice forma su propia componente conexa.

# Parte (a): Estructura de la prueba

**Distinguimos dos casos:**

## Caso $|D| = 1$

Trivial: $D$ es una componente de $B_0$, ya que al inicio del algoritmo cada vértice forma su propia componente conexa.

## Caso $|D| \geq 2$

Analizamos la ejecución de Kruskal sobre $G$.

# Parte (a): Estructura de la prueba

**Distinguimos dos casos:**

## Caso $|D| = 1$

Trivial: $D$ es una componente de $B_0$, ya que al inicio del algoritmo cada vértice forma su propia componente conexa.

## Caso $|D| \geq 2$

Analizamos la ejecución de Kruskal sobre $G$.

- Kruskal considera las aristas en orden creciente de peso.
- Como $\beta < \alpha$, **ninguna arista saliente** de $D$ puede ser procesada antes que las internas.
- Por tanto, el algoritmo primero completará las conexiones internas de $D$ antes de examinar aristas que lo unan con el resto de $V$.

# Caso $|D| = 2$

## Idea general

Sea $j'$ la **primera iteración** en la cual el subgrafo inducido en $B_{j'}$ por $D$ es conexo. Es decir, en la iteración $j'$ Kruskal agrega una arista $(d_1, d_2)$ que une dos componentes conexas $D_1$ y $D_2$ contenidas en $D$, tales que $V(D_1) \cup V(D_2) \subseteq D$.

# Caso $|D| = 2$

## Idea general

Sea $j'$ la **primera iteración** en la cual el subgrafo inducido en $B_{j'}$ por $D$ es conexo. Es decir, en la iteración $j'$ Kruskal agrega una arista $(d_1, d_2)$ que une dos componentes conexas $D_1$ y $D_2$ contenidas en $D$, tales que $V(D_1) \cup V(D_2) \subseteq D$.

- En ese momento, $D_1$ y $D_2$ son subconjuntos de $D$, y antes de $j'$ aún no estaban conectados entre sí.

# Caso $|D| = 2$

## Idea general

Sea $j'$ la **primera iteración** en la cual el subgrafo inducido en $B_{j'}$ por $D$ es conexo. Es decir, en la iteración $j'$ Kruskal agrega una arista $(d_1, d_2)$ que une dos componentes conexas $D_1$ y $D_2$ contenidas en $D$, tales que $V(D_1) \cup V(D_2) \subseteq D$.

- En ese momento, $D_1$ y $D_2$ son subconjuntos de $D$, y antes de $j'$ aún no estaban conectados entre sí.
- Asumamos que $D$ *no es una componente conexa* de $B_{j'}$, $V(D_1) \cup V(D_2) \neq D$ .

# Caso $|D| = 2$

## Idea general

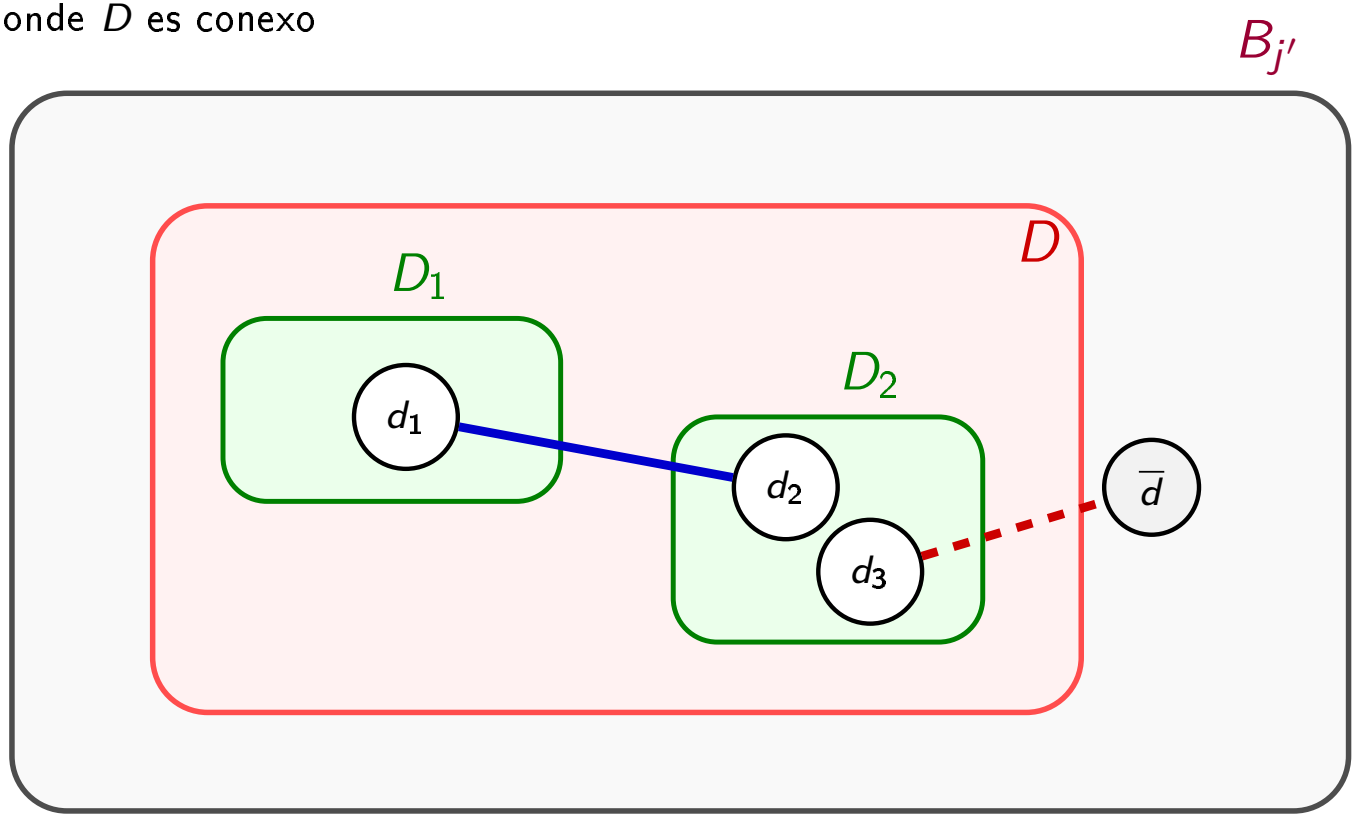
Sea $j'$ la **primera iteración** en la cual el subgrafo inducido en $B_{j'}$ por $D$ es conexo. Es decir, en la iteración $j'$ Kruskal agrega una arista $(d_1, d_2)$ que une dos componentes conexas $D_1$ y $D_2$ contenidas en $D$, tales que $V(D_1) \cup V(D_2) \subseteq D$.

- En ese momento, $D_1$ y $D_2$ son subconjuntos de $D$, y antes de $j'$ aún no estaban conectados entre sí.
- Asumamos que $D$ *no es una componente conexa* de $B_{j'}$, $V(D_1) \cup V(D_2) \neq D$ .
- Entonces, existe una arista $(d_3, \overline{d})$ con $d_3 \in D$ y $\overline{d} \notin D$ que ya estaba disponible (pues $B_{j'}$ no contiene todo $D$).

# Parte (a): Dibujo intuitivo del caso $|D| \geq 2$

Interna: conecta $D_1, D_2$
Saliente: conecta $D$ con $\overline{d}$

$j'$: iteración donde $D$ es conexo



## Demostración (continuación)

- En la iteración $j'$ se agrega una arista **interna** $(d_1, d_2)$ dentro de $D$.

## Demostración (continuación)

- En la iteración $j'$ se agrega una arista **interna** $(d_1, d_2)$ dentro de $D$.
- Sin embargo, $(d_3, \overline{d})$ —una arista **saliente** de $D$— la agregue antes en Kruskal.

## Demostración (continuación)

- En la iteración $j'$ se agrega una arista **interna** $(d_1, d_2)$ dentro de $D$.
- Sin embargo, $(d_3, \overline{d})$ —una arista **saliente** de $D$— la agregue antes en Kruskal.
- Cómo Kruskal recorre las aristas en orden y esa arista fue agregada anteriormente entonces:

$$c(d_3, \overline{d}) \leq c(d_1, d_2).$$

## Demostración (continuación)

- En la iteración $j'$ se agrega una arista **interna** $(d_1, d_2)$ dentro de $D$.
- Sin embargo, $(d_3, \overline{d})$ —una arista **saliente** de $D$— la agregue antes en Kruskal.
- Cómo Kruskal recorre las aristas en orden y esa arista fue agregada anteriormente entonces:

$$c(d_3, \overline{d}) \leq c(d_1, d_2).$$

- Luego, la existencia de una saliente más barata contradice la condición de $D$ deseable.

## Demostración (continuación)

- En la iteración $j'$ se agrega una arista **interna** $(d_1, d_2)$ dentro de $D$.
- Sin embargo, $(d_3, \overline{d})$ —una arista **saliente** de $D$— la agregue antes en Kruskal.
- Cómo Kruskal recorre las aristas en orden y esa arista fue agregada anteriormente entonces:

$$c(d_3, \overline{d}) \leq c(d_1, d_2).$$

- Luego, la existencia de una saliente más barata contradice la condición de $D$ deseable.
- Por lo tanto, llegamos a un absurdo! El cuál surge de asumir que $D$ *no es una componente conexa* de $B_{j'}$.
- Así probamos el caso para $|D| \geq 2$.

## Parte (b): Algoritmo para contar conjuntos deseables

**Estrategia general (Kruskal + DSU):**

## Parte (b): Algoritmo para contar conjuntos deseables

**Estrategia general (Kruskal + DSU):**

- Inicializamos *deseables* $= V$, pues cada vértice es deseable (caso $|D| = 1$).

## Parte (b): Algoritmo para contar conjuntos deseables

**Estrategia general (Kruskal + DSU):**

- Inicializamos *deseables* $= V$, pues cada vértice es deseable (caso $|D| = 1$).
- Recorremos las aristas en orden creciente de peso (como hace Kruskal).

## Parte (b): Algoritmo para contar conjuntos deseables

**Estrategia general (Kruskal + DSU):**

- Inicializamos *deseables* $= V$, pues cada vértice es deseable (caso $|D| = 1$).
- Recorremos las aristas en orden creciente de peso (como hace Kruskal).
- Cada vez que unimos dos componentes $D_1$ y $D_2$:
  1. Formamos $D = D_1 \cup D_2$.

## Parte (b): Algoritmo para contar conjuntos deseables

**Estrategia general (Kruskal + DSU):**

- Inicializamos *deseables* $= V$, pues cada vértice es deseable (caso $|D| = 1$).
- Recorremos las aristas en orden creciente de peso (como hace Kruskal).
- Cada vez que unimos dos componentes $D_1$ y $D_2$:
  1. Formamos $D = D_1 \cup D_2$.
  2. Agregamos $D$ a *deseables*.

## Parte (b): Algoritmo para contar conjuntos deseables

**Estrategia general (Kruskal + DSU):**

- Inicializamos *deseables* $= V$, pues cada vértice es deseable (caso $|D| = 1$).
- Recorremos las aristas en orden creciente de peso (como hace Kruskal).
- Cada vez que unimos dos componentes $D_1$ y $D_2$:
  1. Formamos $D = D_1 \cup D_2$.
  2. Agregamos $D$ a *deseables*.

Perfecto, y así resolvemos el problema en $O(m \cdot \log n)$.

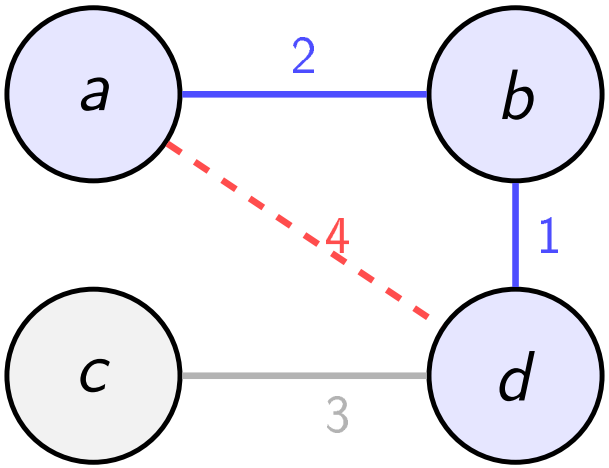
## Parte (b): Algoritmo para contar conjuntos deseables

**Estrategia general (Kruskal + DSU):**

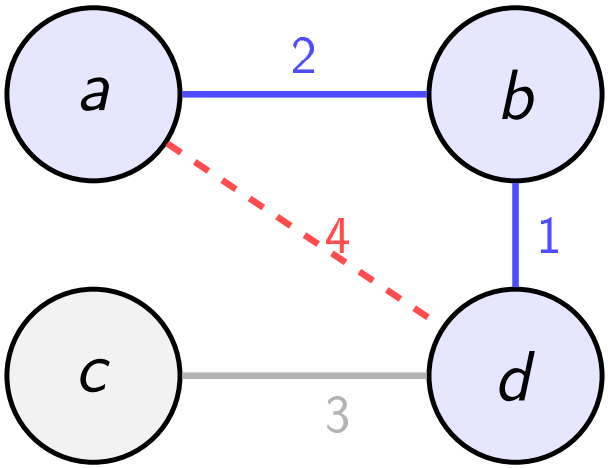
- Inicializamos *deseables* $= V$, pues cada vértice es deseable (caso $|D| = 1$).
- Recorremos las aristas en orden creciente de peso (como hace Kruskal).
- Cada vez que unimos dos componentes $D_1$ y $D_2$:
  1. Formamos $D = D_1 \cup D_2$.
  2. Agregamos $D$ a *deseables*.

Perfecto, y así resolvemos el problema en $O(m \cdot \log n)$. ¿Esta bien este algoritmo? ¿Que problema tiene?

# Contraejemplo: una componente conexa no deseable

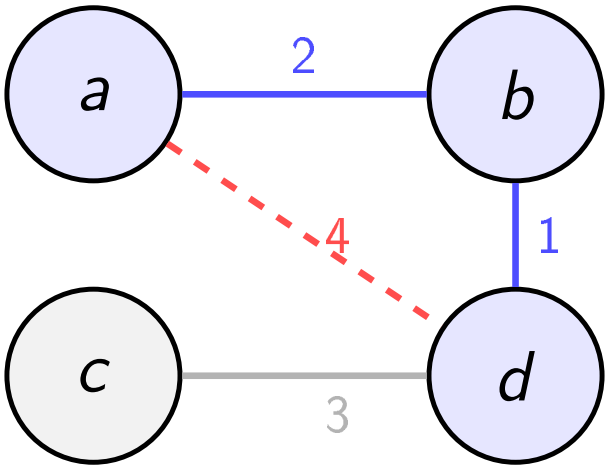


# Contraejemplo: una componente conexa no deseable



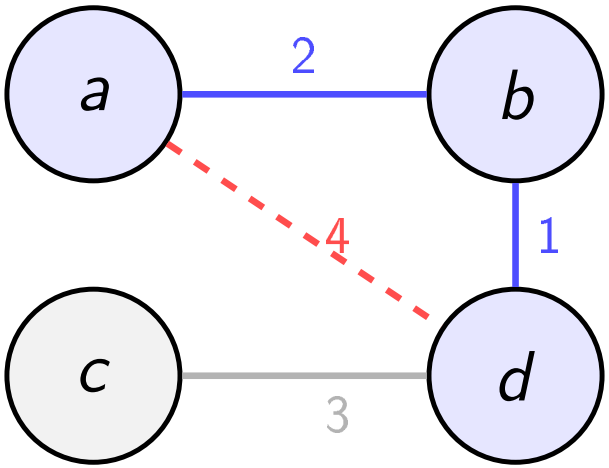

- Consideremos $D = \{a, b, d\}$, componente conexa del grafo.

# Contraejemplo: una componente conexa no deseable



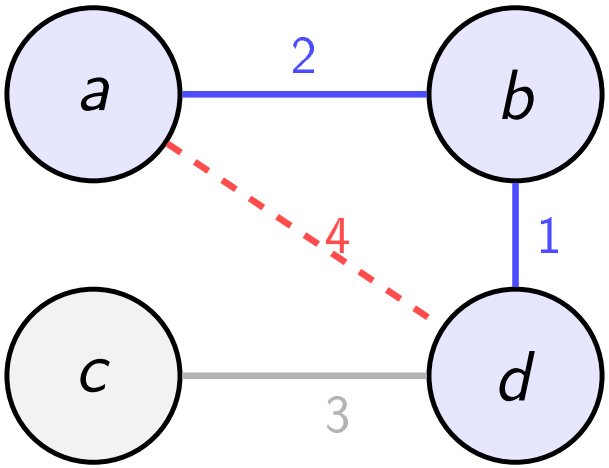

- Consideremos $D = \{a, b, d\}$, componente conexa del grafo.
- Sin embargo, dentro de $D$ hay una arista interna con peso 4.

# Contraejemplo: una componente conexa no deseable




- Consideremos $D = \{a, b, d\}$, componente conexa del grafo.
- Sin embargo, dentro de $D$ hay una arista interna con peso 4.
- Y existe una arista saliente $(c, d)$ con peso 3, que es **menor**.

# Contraejemplo: una componente conexa no deseable




- Consideremos $D = \{a, b, d\}$, componente conexa del grafo.
- Sin embargo, dentro de $D$ hay una arista interna con peso 4.
- Y existe una arista saliente $(c, d)$ con peso 3, que es **menor**.
- Como $c(\text{interna}) < c(\text{saliente})$ no se cumple, $D$ **no es deseable**.

## Algoritmo Correcto

**Estrategia general (Kruskal + DSU):**

## Algoritmo Correcto

**Estrategia general (Kruskal + DSU):**

- Inicializamos *deseables* $= V$, pues cada vértice es deseable (caso $|D| = 1$).

## Algoritmo Correcto

**Estrategia general (Kruskal + DSU):**

- Inicializamos *deseables* $= V$, pues cada vértice es deseable (caso $|D| = 1$).
- Recorremos las aristas en orden creciente de peso (como hace Kruskal).

## Algoritmo Correcto

**Estrategia general (Kruskal + DSU):**

- Inicializamos *deseables* $= V$, pues cada vértice es deseable (caso $|D| = 1$).
- Recorremos las aristas en orden creciente de peso (como hace Kruskal).
- Cada vez que unimos dos componentes $D_1$ y $D_2$:
  1. Formamos $D = D_1 \cup D_2$.

## Algoritmo Correcto

**Estrategia general (Kruskal + DSU):**

- Inicializamos *deseables* $= V$, pues cada vértice es deseable (caso $|D| = 1$).
- Recorremos las aristas en orden creciente de peso (como hace Kruskal).
- Cada vez que unimos dos componentes $D_1$ y $D_2$:
  1. Formamos $D = D_1 \cup D_2$.
  2. Aplicamos un **chequeo de deseabilidad** para verificar si $D$ cumple la propiedad:

     $$c(\text{interna}) < c(\text{saliente})$$

     para todas las aristas que salen de $D$.
  3. Si $D$ es deseable lo agregamos a *deseables*.

# Implementación del chequeo de deseabilidad

## Chequeo de deseabilidad de una componente $D$

Verificar que toda arista saliente de $D$ tenga peso mayor que cualquier arista interna usada en $D$.

# Implementación del chequeo de deseabilidad

## Chequeo de deseabilidad de una componente $D$

Verificar que toda arista saliente de $D$ tenga peso mayor que cualquier arista interna usada en $D$.

- Mantener para cada componente un valor:

$$\texttt{max_interno}(D) = \text{máx}\{\, c(e) \ : \ e \text{ interna a } D \,\}.$$

# Implementación del chequeo de deseabilidad

## Chequeo de deseabilidad de una componente $D$

Verificar que toda arista saliente de $D$ tenga peso mayor que cualquier arista interna usada en $D$.

- Mantener para cada componente un valor:

$$\texttt{max_interno}(D) = \text{máx}\{\, c(e) \ : \ e \text{ interna a } D \,\}.$$

- Cada vez que se unen $D_1$ y $D_2$ obtenemos el $\texttt{max_interno}(D_1 \cup D_2)$, teniendo en cuenta la arista que los une.

# Implementación del chequeo de deseabilidad

## Chequeo de deseabilidad de una componente $D$

Verificar que toda arista saliente de $D$ tenga peso mayor que cualquier arista interna usada en $D$.

- Mantener para cada componente un valor:

$$\texttt{max_interno}(D) = \text{máx}\{\, c(e) \ : \ e \text{ interna a } D \,\}.$$

- Cada vez que se unen $D_1$ y $D_2$ obtenemos el $\texttt{max_interno}(D_1 \cup D_2)$, teniendo en cuenta la arista que los une.
- Luego se recorre todo el conjunto de aristas $(x, y) \in E$:
  - Si exactamente uno de $\{x, y\}$ pertenece a $D$, se exige $c(x, y) > \texttt{max_interno}(D)$.

# Complejidad

## Complejidad

- Cada unión de Kruskal cuesta $O(\alpha(n))$ para la actualización interna.

# Complejidad

## Complejidad

- Cada unión de Kruskal cuesta $O(\alpha(n))$ para la actualización interna.
- El chequeo requiere recorrer $O(m)$ aristas en el peor caso.

# Complejidad

## Complejidad

- Cada unión de Kruskal cuesta $O(\alpha(n))$ para la actualización interna.
- El chequeo requiere recorrer $O(m)$ aristas en el peor caso. .
- Como hay $O(n)$ uniones, el total es $O(nm)$.

## Conclusión

- (a) Todo conjunto deseable aparece como componente en algún bosque intermedio $B_j$.

## Conclusión

- (a) Todo conjunto deseable aparece como componente en algún bosque intermedio $B_j$.
- (b) Podemos contarlos en $O(nm)$ aplicando el chequeo de deseabilidad tras cada unión.

## Conclusión

- (a) Todo conjunto deseable aparece como componente en algún bosque intermedio $B_j$.
- (b) Podemos contarlos en $O(nm)$ aplicando el chequeo de deseabilidad tras cada unión.
- **Intuición:** los conjuntos deseables son *"islas baratas"* que Kruskal completa internamente antes de cruzar aristas más costosas.

1 Repaso AGM

2 Ejercicios
- Viaje en peligro
- Conjuntos deseables
- Audífonos Defectuosos
- Alimentando hormigas
- Rutas y aeropuertos

# Ejercicio 2

## Audífonos Defectuosos

Sasha vino de intercambio a Exactas y quiere ver cómo llegar desde su hogar hasta Ciudad Universitaria. Sus auriculares marca **AGM** (auriculares generalmente malos) están rotos, por lo que no tiene forma de cancelar el sonido. Conocemos todas las calles que conectan una esquina con otra en el mapa y tenemos una función $d$ la cual nos dice el volumen de cada calle. Sasha sufre mucho los sonidos fuertes, por lo que quiere encontrar una ruta que minimice el ruido máximo.

Queremos determinar cuál es el máximo nivel de ruido que tiene que soportar para llegar a Ciudad Universitaria desde su casa.

## Especificaciones

- El formato del input es una línea que contiene dos enteros $C$ (cantidad de calles) y $E$ (cantidad de esquinas). Luego tenemos $C$ lineas que tienen el formato $e_i, e_j, d_{ij}$, las cuales representan el nivel del ruido de la calle que conecta la esquina $i$ con la $j$.
- Tenemos que devolver el umbral mínimo de tolerancia para Sasha.

# Ejemplo

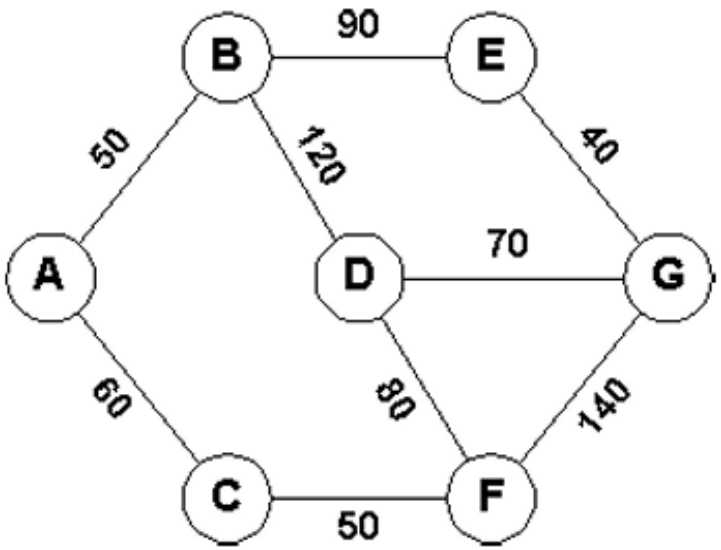



Figura 3: Para ir de A a G el mejor camino es A-C-F-D-G, y la tolerancia que uno necesita es de 80.

## Propiedades a tener en cuenta

- ¿Que propiedad tiene que tener el camino que Sasha quiere encontrar?

## Propiedades a tener en cuenta

- ¿Que propiedad tiene que tener el camino que Sasha quiere encontrar? Que la calle más ruidosa por la que pase sea la menor posible.

## Propiedades a tener en cuenta

- ¿Que propiedad tiene que tener el camino que Sasha quiere encontrar? Que la calle más ruidosa por la que pase sea la menor posible.
- ¿Cómo se llama este tipo de camino?

## Propiedades a tener en cuenta

- ¿Que propiedad tiene que tener el camino que Sasha quiere encontrar? Que la calle más ruidosa por la que pase sea la menor posible.
- ¿Cómo se llama este tipo de camino? *Camino MiniMax*

## Propiedades a tener en cuenta

- ¿Que propiedad tiene que tener el camino que Sasha quiere encontrar? Que la calle más ruidosa por la que pase sea la menor posible.
- ¿Cómo se llama este tipo de camino? *Camino MiniMax*
- ¿Que herramienta tenemos para conocer este tipo de camino?

## Propiedades a tener en cuenta

- ¿Que propiedad tiene que tener el camino que Sasha quiere encontrar? Que la calle más ruidosa por la que pase sea la menor posible.
- ¿Cómo se llama este tipo de camino? *Camino MiniMax*
- ¿Que herramienta tenemos para conocer este tipo de camino? *AGM*

## Propiedades a tener en cuenta

- ¿Que propiedad tiene que tener el camino que Sasha quiere encontrar? Que la calle más ruidosa por la que pase sea la menor posible.
- ¿Cómo se llama este tipo de camino? *Camino MiniMax*
- ¿Que herramienta tenemos para conocer este tipo de camino? *AGM*

**Lema**

Sea $T$ un árbol generador de un grafo $G$ y $c : E(G) \to \mathbb{R}$ su función de costo. Entonces, $T$ es un *AGM* de $(G, c)$ si y sólo si todo camino de $T$ es *MiniMax* de $(G, c)$.

## Solución

- Creamos un grafo con un vértice correspondiente a cada esquina.

## Solución

- Creamos un grafo con un vértice correspondiente a cada esquina.
- Agregamos una arista entre las esquinas que tengan una calle que las conecta y le colocamos un costo igual a su nivel de ruido.

## Solución

- Creamos un grafo con un vértice correspondiente a cada esquina.
- Agregamos una arista entre las esquinas que tengan una calle que las conecta y le colocamos un costo igual a su nivel de ruido.
- Buscamos el *AGM* del grafo con Prim o Kruskal. ¿Cuál es la complejidad? $\mathcal{O}(m + n \log n)$, con $n = E$ y $m = C$.

# Solución

- Nos fijamos la arista de máximo costo del camino *MiniMax* entre la esquina de Sasha y Ciudad Universitaria. Esa es nuestra respuesta.

# Variaciones Ejercicio 2

## Posibles variaciones del ejercicio

- ¿Cómo cambiaría nuestra solución si ahora nos dicen que queremos encontrar los umbrales de tolerancia para todos los lugares (esquinas) a las que puede ir Sasha?

# Variaciones Ejercicio 2

## Posibles variaciones del ejercicio

- ¿Cómo cambiaría nuestra solución si ahora nos dicen que queremos encontrar los umbrales de tolerancia para todos los lugares (esquinas) a las que puede ir Sasha?
  - A partir del *AGM* generado encontramos el camino MiniMax entre todos los nodos. Pues todo camino del *AGM* es MiniMax.
  - Luego nos fijamos para cada nodo cuál es su umbral de tolerancia.

# Variaciones Ejercicio 2

## Posibles variaciones del ejercicio

- ¿Cómo cambiaría nuestra solución si ahora nos dicen que queremos encontrar los umbrales de tolerancia para todos los lugares (esquinas) a las que puede ir Sasha?
  - A partir del *AGM* generado encontramos el camino MiniMax entre todos los nodos. Pues todo camino del *AGM* es MiniMax.
  - Luego nos fijamos para cada nodo cuál es su umbral de tolerancia.
- ¿Qué ocurre si ahora Tasha (hermana de Sasha) se duerme si pasa por lugares con ruido menor a cierto umbral y el umbral es un dato? ¿Cómo conseguimos el camino?

# Variaciones Ejercicio 2

## Posibles variaciones del ejercicio

- ¿Cómo cambiaría nuestra solución si ahora nos dicen que queremos encontrar los umbrales de tolerancia para todos los lugares (esquinas) a las que puede ir Sasha?
  - A partir del *AGM* generado encontramos el camino MiniMax entre todos los nodos. Pues todo camino del *AGM* es MiniMax.
  - Luego nos fijamos para cada nodo cuál es su umbral de tolerancia.
- ¿Qué ocurre si ahora Tasha (hermana de Sasha) se duerme si pasa por lugares con ruido menor a cierto umbral y el umbral es un dato? ¿Cómo conseguimos el camino?
  - Hacemos lo mismo, solo que ahora conseguimos el Árbol Generador Máximo, es análogo.

# Variaciones Ejercicio 2

## Posibles variaciones del ejercicio

- ¿Cómo cambiaría nuestra solución si ahora nos dicen que queremos encontrar los umbrales de tolerancia para todos los lugares (esquinas) a las que puede ir Sasha?
  - A partir del *AGM* generado encontramos el camino MiniMax entre todos los nodos. Pues todo camino del *AGM* es MiniMax.
  - Luego nos fijamos para cada nodo cuál es su umbral de tolerancia.
- ¿Qué ocurre si ahora Tasha (hermana de Sasha) se duerme si pasa por lugares con ruido menor a cierto umbral y el umbral es un dato? ¿Cómo conseguimos el camino?
  - Hacemos lo mismo, solo que ahora conseguimos el Árbol Generador Máximo, es análogo.
- ¿Cómo cambiaría nuestra solución si el grafo resultante es ralo o denso?

# Variaciones Ejercicio 2

## Posibles variaciones del ejercicio

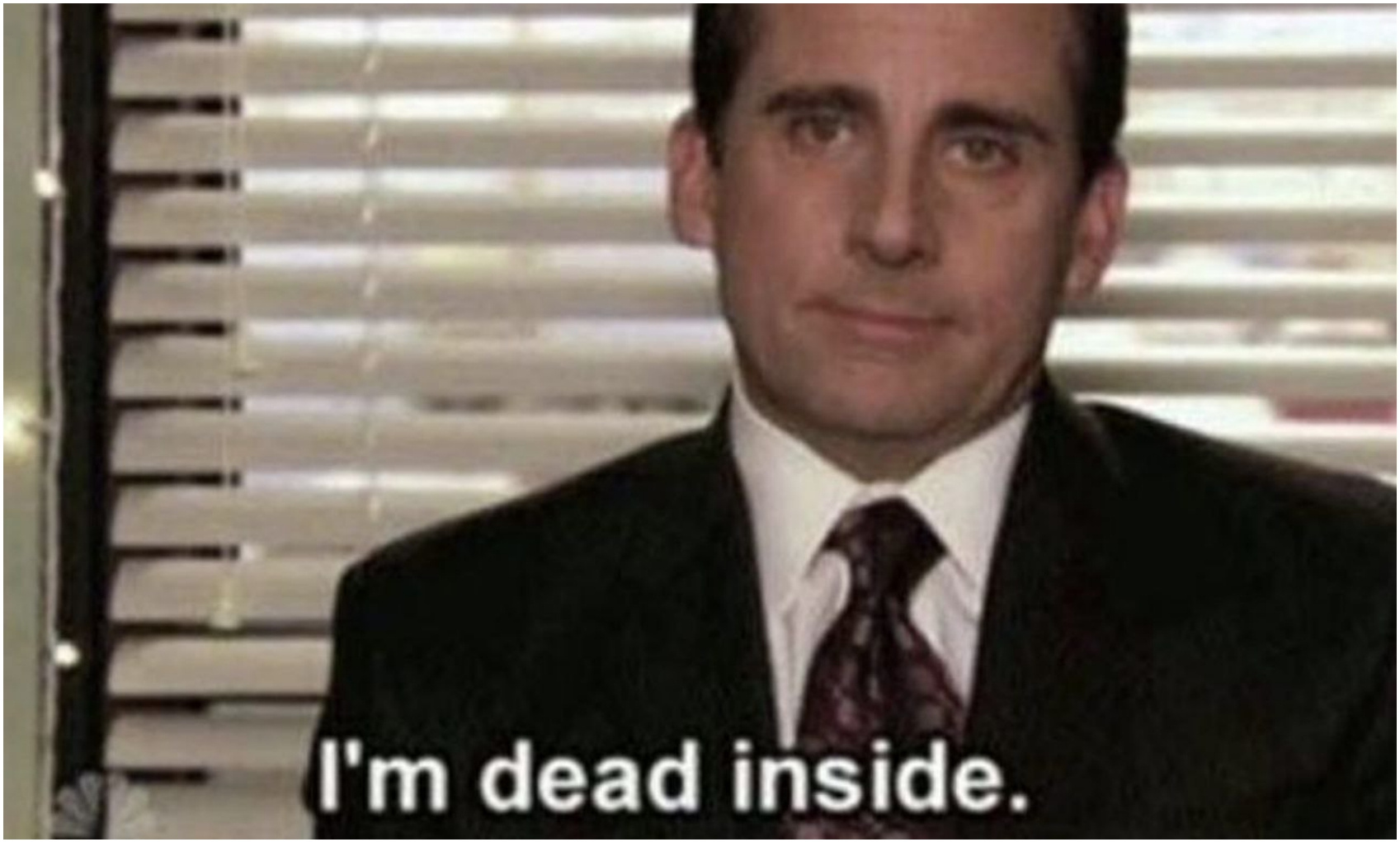
- ¿Cómo cambiaría nuestra solución si ahora nos dicen que queremos encontrar los umbrales de tolerancia para todos los lugares (esquinas) a las que puede ir Sasha?
  - A partir del *AGM* generado encontramos el camino MiniMax entre todos los nodos. Pues todo camino del *AGM* es MiniMax.
  - Luego nos fijamos para cada nodo cuál es su umbral de tolerancia.
- ¿Qué ocurre si ahora Tasha (hermana de Sasha) se duerme si pasa por lugares con ruido menor a cierto umbral y el umbral es un dato? ¿Cómo conseguimos el camino?
  - Hacemos lo mismo, solo que ahora conseguimos el Árbol Generador Máximo, es análogo.
- ¿Cómo cambiaría nuestra solución si el grafo resultante es ralo o denso?
  - Si es ralo, entonces nos conviene usar Kruskal; si es denso, nos conviene usar Prim.

## Bonus track

Tarea: Ahora los auriculares de Sasha le bloquean el sonido pero sólo de 1 calle. ¿Cómo cambia el problema? [2]

[2] Spoiler: Se re pica.

# ¿Hacemos un break?



1 Repaso AGM

2 Ejercicios
- Viaje en peligro
- Conjuntos deseables
- Audífonos Defectuosos
- Alimentando hormigas
- Rutas y aeropuertos

# Ejercicio 3

Martin fue a la NDJ (noche de juegos) y se cruzó con unos estudiantes de Biología. Ahí le contaron el problema que tenían con un hormiguero que estaban intentando mantener.

## Alimentando hormigas

- En el hormiguero hay $n$ cuevas. La i-ésima cueva tiene coordenadas $(x_i, y_i)$, ninguna tiene comida.
- Hay dos formas de proveer comida a una cueva:
  1. Colocarle un tubo de comida.
  2. Construir un túnel entre una cueva i sin comida a otra cueva j que tenga comida. Para unir dos cuevas i,j necesitamos $|x_i - x_j| + |y_i - y_j|$ centímetros de madera.

# Ejercicio 3

## Alimentando hormigas

- Sabemos que construir un tubo de comida cuesta $T$ y un centímetro de madera cuesta $M$.
- ¿Cuál es la manera más barata de que todas las cuevas tengan acceso a la comida?

# Un modelo posible

- Crear un grafo con un vértice correspondiente a cada cueva.

# Un modelo posible

- Crear un grafo con un vértice correspondiente a cada cueva.
- Agregar aristas entre las cuevas con el costo de unir las cuevas con madera.

# Un modelo posible

- Crear un grafo con un vértice correspondiente a cada cueva.
- Agregar aristas entre las cuevas con el costo de unir las cuevas con madera.
- Un *AGM* en este grafo representa una solución al problema, puesto que es indistinto donde colocamos cualquiera de los tubos.

## Un modelo posible

- Crear un grafo con un vértice correspondiente a cada cueva.
- Agregar aristas entre las cuevas con el costo de unir las cuevas con madera.
- Un *AGM* en este grafo representa una solución al problema, puesto que es indistinto donde colocamos cualquiera de los tubos.
- Pero hay un problema...

## Un modelo posible

¡Cuidado! ¡Este modelo no funciona!

Siempre está bueno dibujar algunos ejemplos y pensar casos borde para nuestro algoritmo. En este caso si tenemos dos posibles componentes conexas que estén muy lejos, puede ser mejor colocar dos tubos de comida en vez de uno solo.

# Un modelo posible

¡Cuidado! ¡Este modelo no funciona!

Siempre está bueno dibujar algunos ejemplos y pensar casos borde para nuestro algoritmo. En este caso si tenemos dos posibles componentes conexas que estén muy lejos, puede ser mejor colocar dos tubos de comida en vez de uno solo.

- Entonces puede ser que nos esté faltando agregar información a nuestro grafo. ¿Cómo la podemos agregar?

## Una solución válida

- Crear un grafo con un vértice correspondiente a cada cueva *y crear un vértice adicional, llamémoslo* ***Tubo***.

## Una solución válida

- Crear un grafo con un vértice correspondiente a cada cueva *y crear un vértice adicional, llamémoslo* ***Tubo***.
- Agregar aristas entre las cuevas con el costo de unir las cuevas con madera.

## Una solución válida

- Crear un grafo con un vértice correspondiente a cada cueva *y crear un vértice adicional, llamémoslo* ***Tubo***.
- Agregar aristas entre las cuevas con el costo de unir las cuevas con madera.
- Agregar aristas entre cada cueva y el **Tubo** con costo equivalente a instalar un tubo de comida en la cueva.

# Una solución válida

- Crear un grafo con un vértice correspondiente a cada cueva *y crear un vértice adicional, llamémoslo* ***Tubo***.
- Agregar aristas entre las cuevas con el costo de unir las cuevas con madera.
- Agregar aristas entre cada cueva y el **Tubo** con costo equivalente a instalar un tubo de comida en la cueva.
- Ahora sí: Un *AGM* $T$ en este grafo sí que representa una solución al problema.

## Una solución válida

- A partir de las aristas de $T$ podemos decirles a nuestros amigos de Biología qué hacer. Por cada arista $(c_i, c_j)$ sabemos que tenemos que colocar un túnel entre la cueva $i$ y la cueva $j$. Luego por cada arista $(c_i, \textbf{Tubo})$ sabemos que tenemos que colocar un tubo de comida en la cueva $i$.

## Una solución válida

- A partir de las aristas de $T$ podemos decirles a nuestros amigos de Biología qué hacer. Por cada arista $(c_i, c_j)$ sabemos que tenemos que colocar un túnel entre la cueva $i$ y la cueva $j$. Luego por cada arista $(c_i, \textbf{Tubo})$ sabemos que tenemos que colocar un tubo de comida en la cueva $i$.
- Lo que hicimos fue: Problema $\rightarrow$ Modelado $\rightarrow$ Algoritmo sobre el grafo $\rightarrow$ traducción de la salida del algoritmo a la solución del problema.

1 Repaso AGM

2 Ejercicios
- Viaje en peligro
- Conjuntos deseables
- Audífonos Defectuosos
- Alimentando hormigas
- Rutas y aeropuertos

# Ejercicio 4

Cifu volvio de su estancia de investigacion en Emiratos Arabes. Como había salido tan bien su proyecto el presidente Mohammed bin *Prim* le pidio ayuda.

## Rutas y aeropuertos

Ahora Cifu tiene que conectar todas las ciudades de Emiratos Arabes. Puede poner la cantidad de rutas y aeropuertos que desee. Se puede volar desde una ciudad con aeropuerto a cualquier otra que tenga aeropuerto. Conocemos los costos de colocar una ruta entre la ciudad $i$ y la $j$, que es $c_{i,j}$. También conocemos el costo de colocar un aeropuerto en la ciudad $i$, $a_i$. Con toda esta información queremos lograr conectar todas las ciudades usando el menor presupuesto posible.

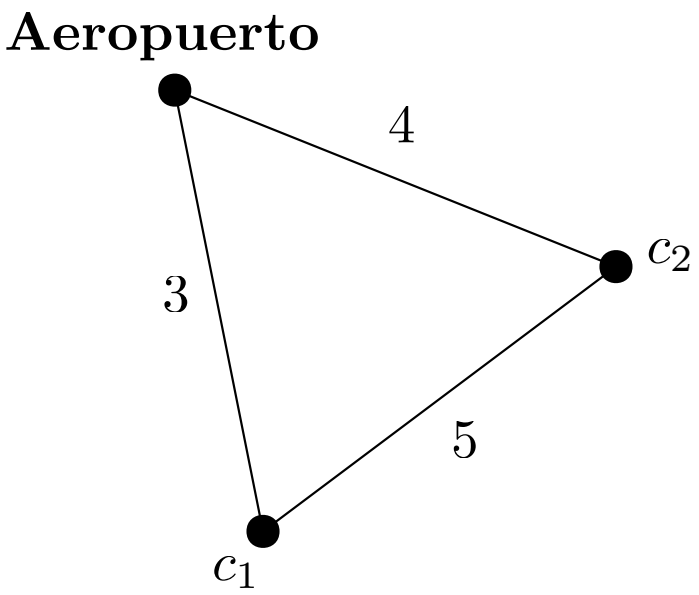
## Analicemos el problema

- Nuestro grafo $G$ seguramente va a tener a las ciudades como vértices y van a estar conectadas con aristas de costo $c_{i,j}$.
- Ahora la pregunta que tenemos es: ¿Cómo modelamos los aeropuertos?

## Analicemos el problema

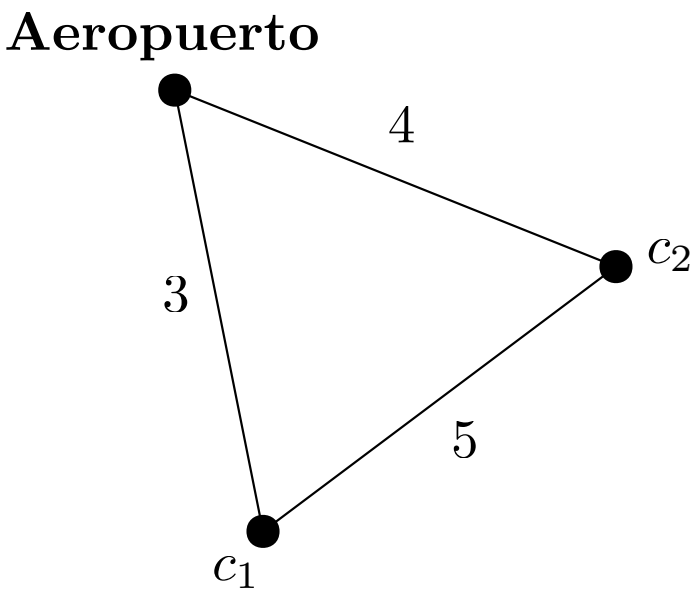
- Nuestro grafo $G$ seguramente va a tener a las ciudades como vértices y van a estar conectadas con aristas de costo $c_{i,j}$.
- Ahora la pregunta que tenemos es: ¿Cómo modelamos los aeropuertos?
- Una opción posible es intentar hacer lo mismo que en el ejercicio anterior. Pero en este caso la relación es distinta, no nos sirve que haya sólo una ciudad con aeropuerto como podía suceder en el caso anterior.

# Analicemos el problema




¿Qué pasa si corremos AGM acá?

# Analicemos el problema




¿Qué pasa si corremos AGM acá? Nos va a decir que la respuesta óptima es agarrar las aristas que conectan con **Aeropuerto**, cuando en realidad la solución óptima del problema era agregar solamente la calle entre $c_1$ y $c_2$.

## Solución

Separemos entonces las posibles soluciones en 2: las que usan algún aeropuerto y las que no. Vamos a encontrar la mejor que no use ningún aeropuerto, y la mejor que use al menos uno, y compararlas.

## Solución

Si una solución no usa ningún aeropuerto, necesariamente tiene que conectar todas las ciudades con rutas. Por lo tanto, debe ser un subgrafo del grafo que contiene solo las rutas como aristas. Como todos los pesos de las rutas son positivos, el AGM va a ser la solución óptima que **no** use aeropuertos.

## Solución

Usemos el modelo $G'$ en el que agregamos un nodo que represente a todos los aeropuertos, llamémoslo **Aeropuerto**. ¿Qué pasa con el AGM de este grafo?
Supongamos que una solución $S$ sí usa un aeropuerto. Por cada calle usada en $S$, seleccionamos la arista correspondiente en $G'$, y por cada aeropuerto agregado a una ciudad, seleccionamos la arista correspondiente que incida en **Aeropuerto**. Al grafo inducido por las aristas que seleccionamos lo llamamos $H$.

Sabemos que $H$ es conexo, porque en $S$ cada ciudad debe estar conectada a alguna ciudad que tenga un aeropuerto, y todas las ciudades que tienen un aeropuerto están conectadas a **Aeropuerto**. Además, si tenemos una solución $S$ tal que $H$ no es un árbol, podemos sacar alguna calle o aeropuerto de la solución y obtener una solución mejor. Por último, un árbol generador de $G'$ es una solución válida.

Por lo tanto, la solución óptima que usa al menos un aeropuerto es el árbol generador mínimo de $G'$.

## Algoritmo

1. Crear un grafo $G$ con un vértice correspondiente a cada ciudad.
2. Agregar aristas entre las ciudades con el costo de colocar una ruta entre ellas.
3. Correr Prim o Kruskal sobre $G$ para obtener el AGM, y guardar el peso en `pesoSinAeropuertos`.
4. Agregar a $G$ un nodo **Aeropuerto** y agregar aristas entre **Aeropuerto** y cada ciudad con el costo de colocar un aeropuerto en la ciudad.
5. Correr Prim o Kruskal sobre $G'$ y guardar el peso en `pesoConAeropuertos`.
6. Devolver mín(`pesoSinAeropuertos`, `pesoConAeropuertos`).

# Kruskal - Idea

Pregunta: Dada una arista e = uv, cómo sabemos si u y v están en componentes conexas diferentes?

# Kruskal - Idea

# Disjoint Set

## Disjoint Set

Es una estructura de datos que nos provee las siguientes operaciones eficientemente:

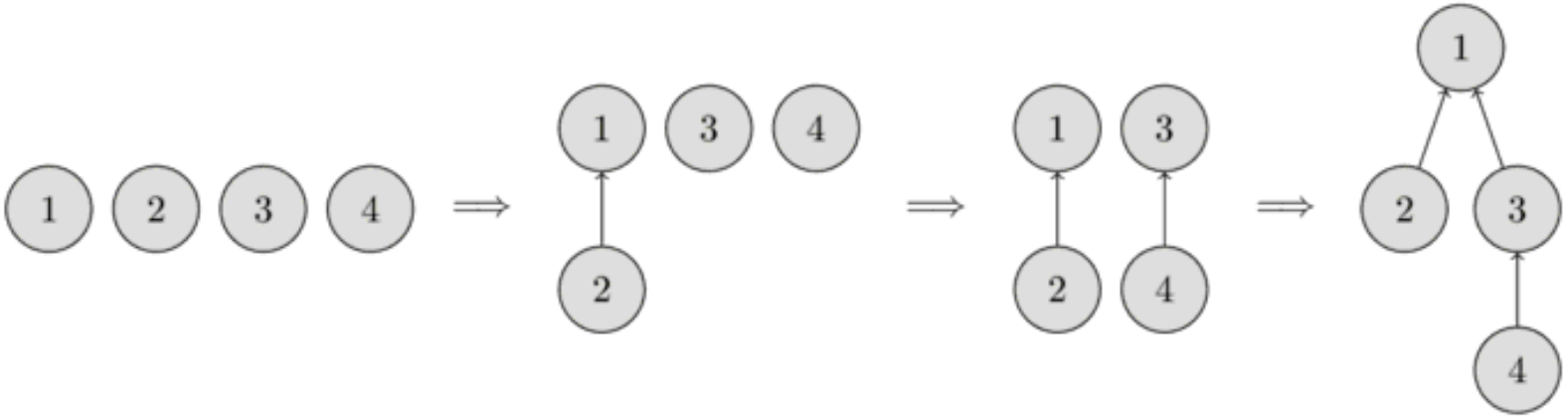
- *find-set(u)*: Dado un vértice nos dice a qué componente conexa pertenece.
- *union(u,v)*: Une las componentes conexas a las que pertenecen u y v.

# Disjoint Set

## Disjoint Set

Vamos a representar a las componentes conexas en forma de árboles: Cada árbol va a corresponder a una componente conexa. La raíz del árbol representará al representante de la componente conexa. Al principio empezamos con $n$ nodos donde cada uno es su propia componente conexa, donde el representante es uno mismo. Luego, cuando hacemos la unión entre dos nodos, ambos nodos van a tener el mismo representante. La complejidad del find() es O(n) en peor caso, por lo tanto la complejidad total sería O(m * n).

# Disjoint Set



# Disjoint Set

Se suelen implementar dos optimizaciones a esta estructura de datos:

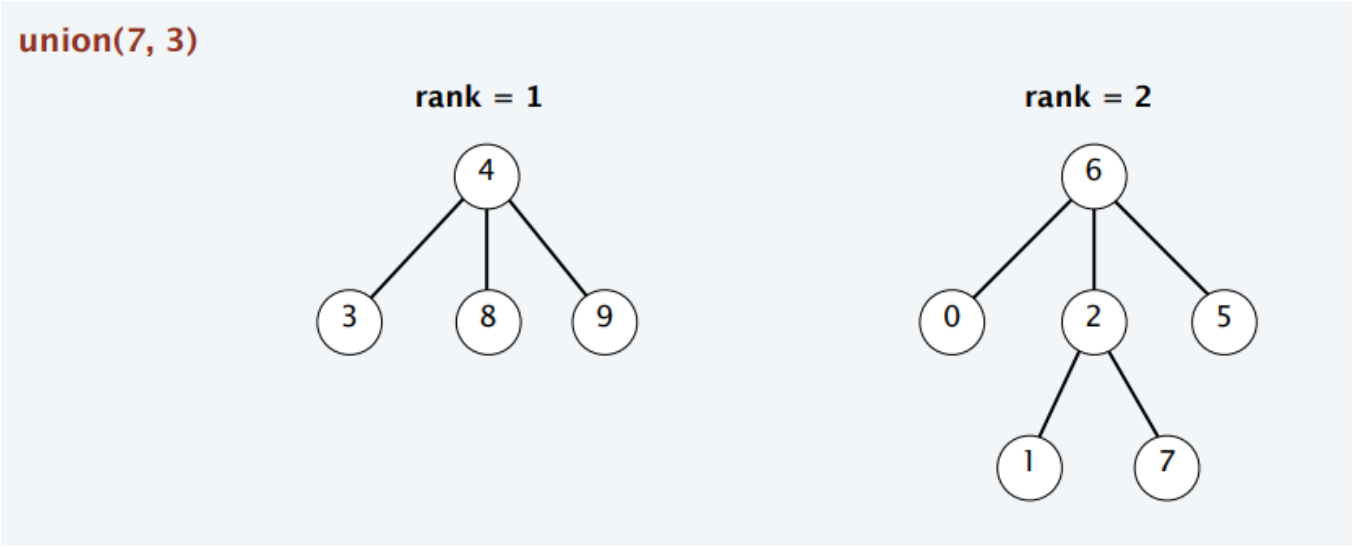
- *Union by rank*
- *Path Compression*

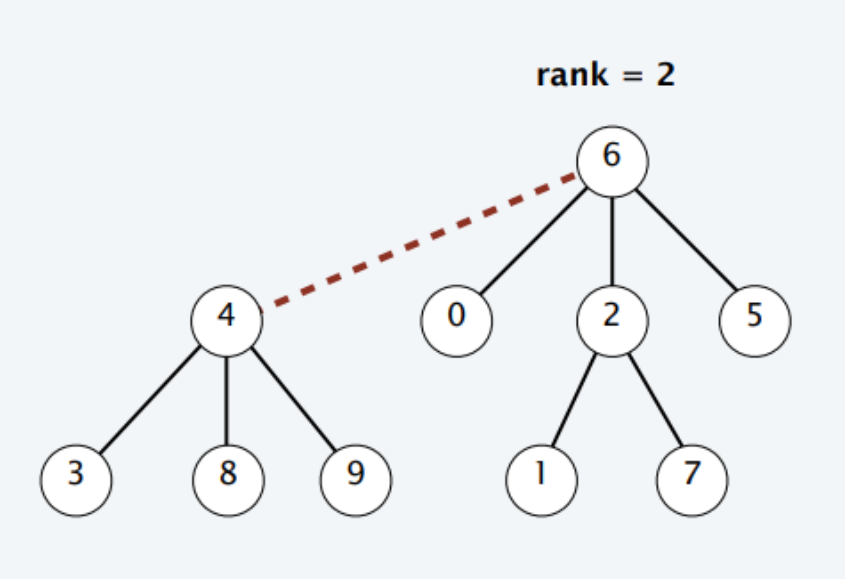
# Union by rank

A cada nodo se le asigna un *rank*, que indica el tamaño de su subarbol mas grande.

- Al principio, cada nodo tiene *rank* 0.
- Cuando se hace una union, si el *rank* de ambos nodos es igual, se incremente el rank del representante en 1.
- En caso contrario, el nodo con *rank* menor apunta hacia el de *rank* mayor, sin alterar el valor del *rank* del representante.

# Union by rank



## Union by rank

La complejidad en el peor caso para:

- Find queda $O(log\ n)$
- union queda $O(1)$, ya que solamente hacemos apuntar un representante a otro.

Veamos por qué para Find es esa complejidad. La estrategia será ver en que el árbol resultante que nos queda de unir dos componentes conexas, es un árbol que tiene una altura acotada.

# Union by rank

## Lema

*Cada nodo raiz de rank k tiene al menos $2^k$ nodos en su arbol.*

## Demostración.

Hagamos induccion sobre $k$. Caso base: $k = 0$, entonces tiene al menos un nodo en su árbol, que es cierto Caso inductivo: Considero *P(n-1)* verdadero, quiero ver que *P(n)* es cierto. Un nodo de rank k es solo creado mergeando dos nodos raíz de rank $k - 1$. Por hipótesis inductiva, cada uno de esos nodos raíz tiene al menos $2^{k-1}$ nodos en sus árboles. Si los juntamos nos queda que $2^{k-1} + 2^{k-1} = 2^k$, que es lo queríamos. □

# Union by rank

## Lema

*El rank mas alto de un nodo es menor o igual a* $log_2(n)$

## Demostración.

Por la proposición anterior, un nodo de *rank* x tiene a lo sumo $2^x$. Sea $n$ la cantidad de nodos totales dentro del árbol, entonces se cumple que $2^x \leq n$ que es lo mismo que $x \leq log_2 n$. □

## Union by rank

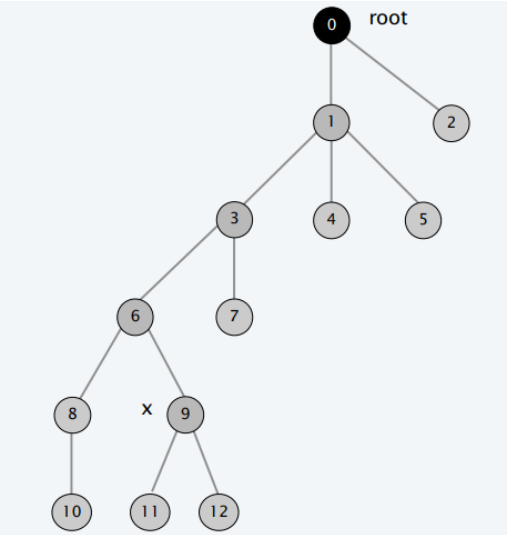
Esto nos dice que el *rank* máximo está acotado por el logaritmo de la cantidad de nodos. Con lo cual, la altura máxima de nuestro árbol es a lo sumo logarítmica. Entonces si queremos buscar el representante de un elemento, basta que recorra todo el árbol hasta la raíz. Por eso nos queda en complejidad $O(logn)$.

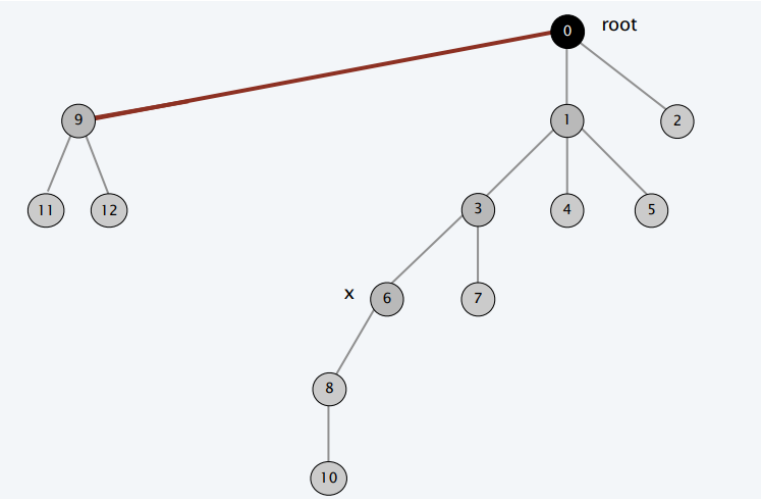
# Path Compression

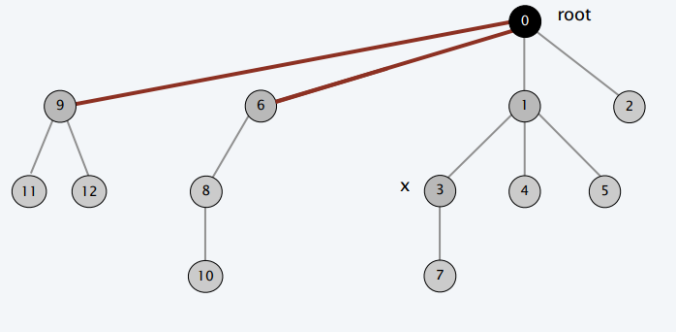
## Path Compression

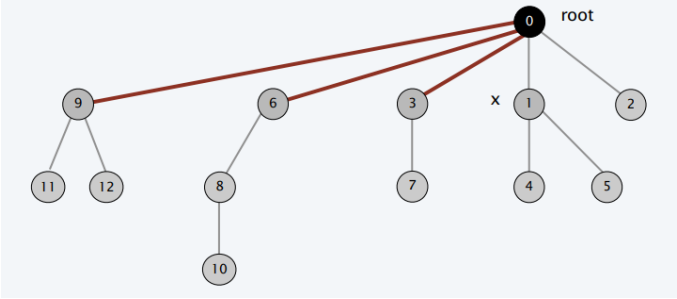
Empezando desde un nodo $x$, ir hasta la raíz para encontrar el representante. En el camino, cambiar los punteros de todos los nodos dentro del camino para apuntar directamente al representante.

# Path Compression



## Resumen de complejidades

- Sin optimizaciones: la complejidad del find() es O(n) en peor caso, por lo tanto la complejidad total sería O(m * n).
- Con *union by size/by rank* la complejidad de cada find es O(log n) en peor caso.
- Si además se usa *Path Compression* la complejidad amortizada de cada find() es O($\alpha$(n)), donde $\alpha$ representa a la función inversa de Ackermann. Como esta funcion crece extremadamente lento, entonces en la practica es O(1).