Arboles Abarcadores de Costo Mínimo

Algoritmos de Kruskal y PRIM

Bibliografía: "Introduction to Algorithms". Third Edition.

The MIT Press. Massachusetts Institute of Technology. Cambridge, Massachusetts 02142.

http://mitpress.mit.edu

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

Facultad de Matemática y Computación
Universidad de La Habana

Una Aplicación

Diseño de circuitos eléctricos

Problema

Conectar por cables los pines de un componente electrónico usando la menor cantidad de cable posible

Objetivo: De todas las posibles formas de conectar, determinar la que use la menor cantidad de cable

Idea:
interconectar un conjunto de n pines usando $n-1$ cables

Propuesta de Solución:
Modelar el problema de cableado mediante un grafo no dirigido $G=(V,E)$, V : conjunto de pines; E: conjunto de las posibles conexiones entre pares de pines

Grafo ponderado

Grafo ponderado:

$\forall$<u,v>$\in$E, $\exists$ ω(<u,v>)
Donde, generalmente, ω: E $\rightarrow$ $\Re$
ω<u,v> representa ***el costo*** *o* ***el peso*** para la arista <u, v>

Ejemplo: **costo** → cantidad de **cable** necesario para conectar los **pines** u y v

Objetivo:

Encontrar un **subconjunto E' de E**, **que conecte <u>todos los vértices</u>** en V, cuyo ***costo total*** (la suma de todos los *costos* o *pesos* de las aristas) **sea <u>mínimo</u>.** Con todo ello, formar un árbol **T=<V, E'>**

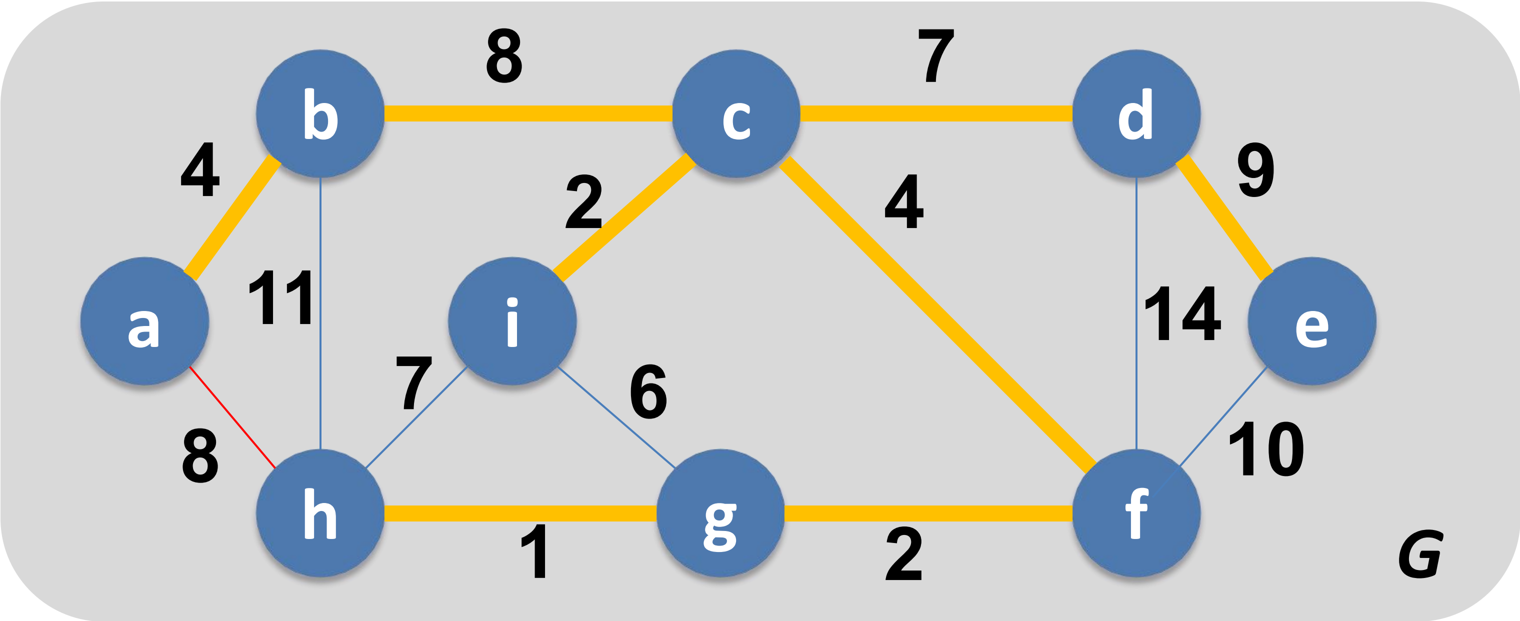
Árbol Abarcador de costo mínimo - *AACM*

- Como T es **acíclico y conexo (conecta todos los vértices)**, entonces forma **un árbol libre**, al que se le llama, además, ***árbol abarcador*** porque en él están todos los vértices del grafo

- Si, además, **la suma de todos los *costos* o *pesos* de las aristas de T es mínimo, entonces T es *abarcador de costo mínimo***

Mínimo: se refiere a la suma de los costos de las aristas de T y **NO** a la cantidad de aristas, ya que **la cantidad de aristas del árbol T siempre serán |V|-1**

Árbol Abarcador de costo mínimo - *AACM*



- ***AACM*** para el grafo conexo ***G***

- El **peso total** del *AACM* es: 37

- En algunos casos: El *AACM* **no es único**

En este caso, removiendo la arista <b, c> y reemplazándola por la <a, h>, se obtiene otro *AACM* con el mismo costo

Algoritmo genérico para obtener un *AACM*

Sea G=(V,E) **conexo**, **no dirigido** y con una función de *costo* asociada w : $E \rightarrow \Re$

Problema
Encontrar el *AACM* de G

Los dos algoritmos que se presentarán en la clase:
PRIM y **KRUSKAL**

se basan en una **estrategia *glotona***, aunque difieren en la forma de atacar el problema

Estrategia **Glotona**: En cada paso del algoritmo, ante varias decisiones posibles a tomar, se escoge la que mejor resultado da en un momento dado

Algoritmo genérico para obtener un *AACM*

La estrategia *glotona* se expresa en el siguiente **algoritmo genérico**:

- El ***AACM*** crece **añadiendo una arista en cada iteración**

- El algoritmo maneja un conjunto ***A*** de aristas que siempre es **subconjunto de las aristas de algún *AACM*** de *G*

- **En cada iteración, se determina una arista <u,v> que puede ser añadida a *A* sin violar la siguiente <u>invariante</u>:**

A $\cup$ {<u, v>} $\subseteq$ de algún *AACM* de G

A la arista <u,v> se le llama **arista segura** para *A*, porque al ser añadida a *A*, se sigue cumpliendo la invariante

Algoritmo genérico para obtener un *AACM*

```
AACM_Genérico (G, w)
{
1  A ← ϕ
2  While (A no sea un AACM)
   {
3     encontrar <u, v> segura para A
4     A ← A∪{<u, v>}
   }
5  return A
}
```

Itera |V|-1 veces

- **p1**: *A* satisface la invariante: $\phi \subset$ ***AACM***
- **p(2-4)**: se mantiene la invariante
- **p5**: cuando se retorna ***A*** tiene que ser un ***AACM***
- **p3: La parte ingeniosa está en encontrar una *arista segura***

Teorema para reconocer aristas seguras

Definiciones:

- Un **corte** (S, V-S) de G=(V,E) no dirigido es una partición de los vértices del conjunto V

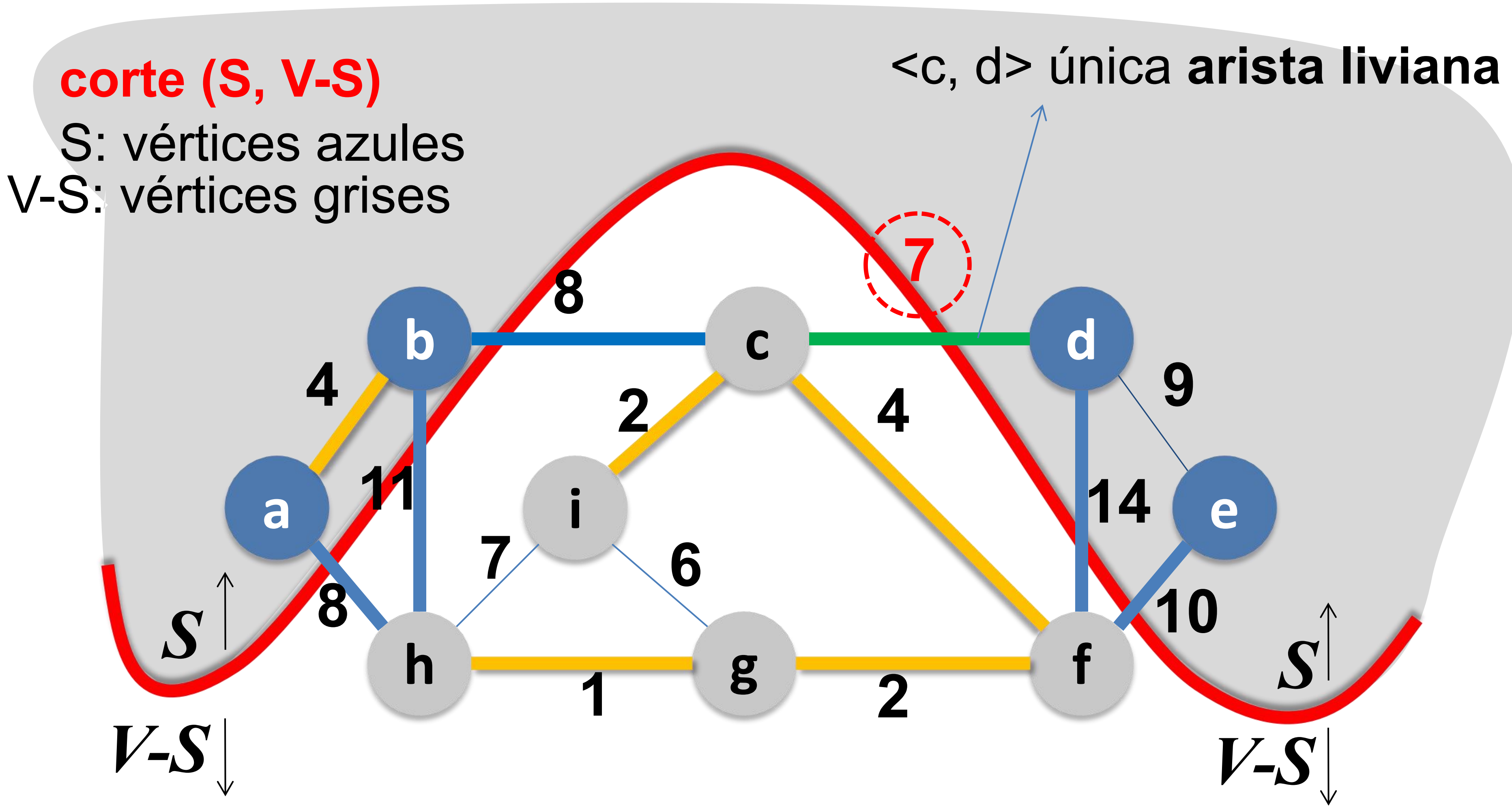
- Una arista <u,v> **cruza el corte** (S, V-S) si uno de los extremos de la misma está en S y el otro en V-S

- Un **corte, respeta un conjunto de aristas A**, si NO existen aristas en **A** que crucen el corte

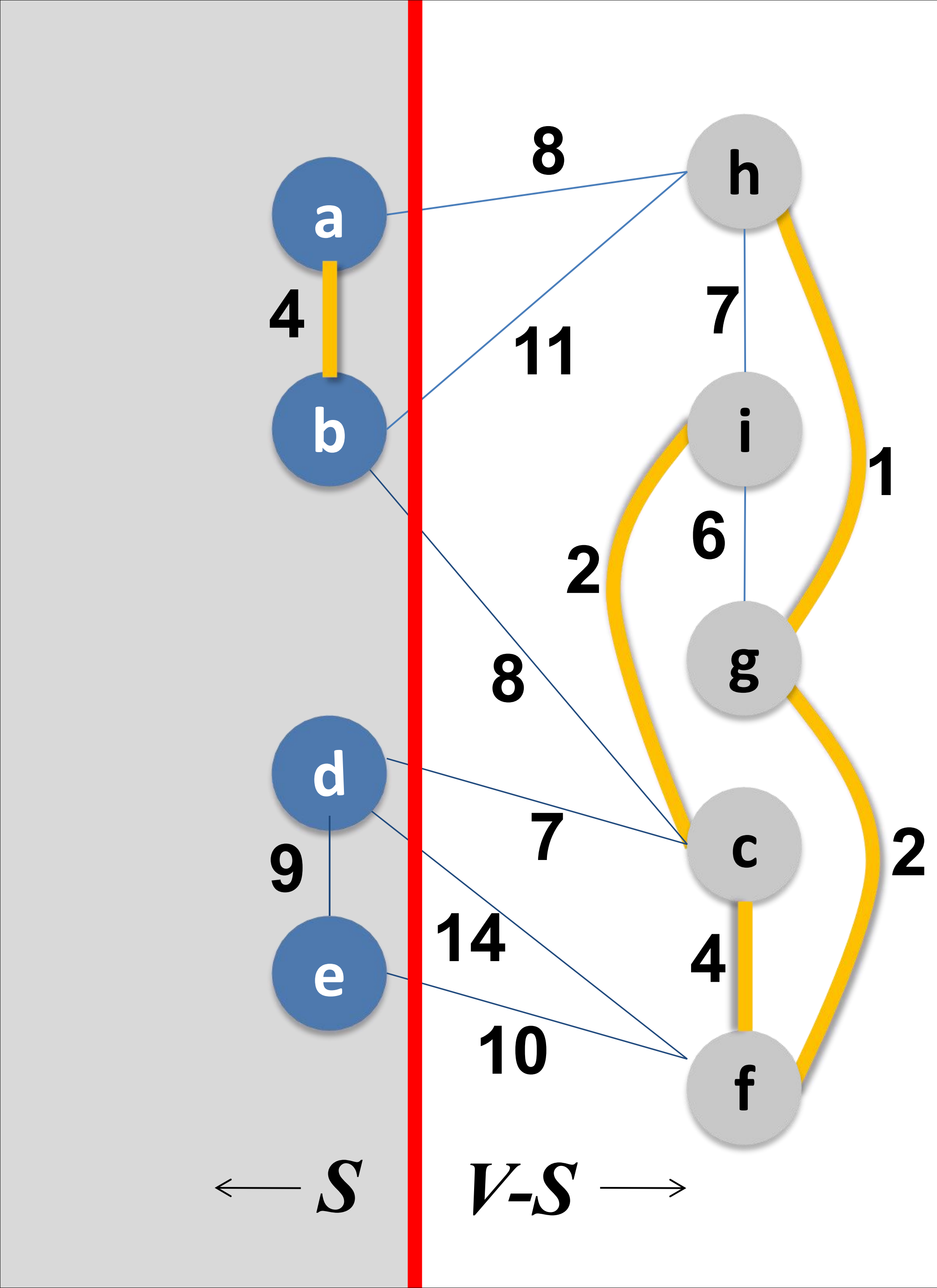
propiedad asociada a la arista

- Una arista es **liviana** **cruzando el corte**, si tiene el ***menor peso*** entre todas las que lo cruzan

Una **arista** es **liviana**, **satisfaciendo una propiedad** dada, si **tiene el menor *peso* entre todas las aristas que satisfacen dicha propiedad**



- Conjunto *A***:** aristas **amarillas**
- El corte **respeta *A*** porque no hay ninguna arista de *A* que lo cruza
- Las aristas que **cruzan el corte** son las que conectan vértices azules con grises



El mismo grafo con los vértices en S en la parte izquierda, y los de V-S en la derecha

Una arista **cruza el corte** si conecta a un vértice de la parte izquierda con uno de la derecha

Teorema 1: para reconocer aristas seguras

Teorema 1

- Sea G=(V,E) un grafo **conexo, no dirigido y ponderado**

- Sea **A** $\subseteq$ **E incluido en algún *AACM*** de ***G***

- Sea (S, V-S) **un corte de *G* que respeta A**

- Sea <u,v>$\in$ *E* una **arista liviana que cruza el corte** (S, V-S)

entonces,

<u,v> es segura para *A*

Esto significa que A $\cup$ {<u, v>} $\subseteq$ de algún *T AACM* de *G*

Demostración del Teorema 1

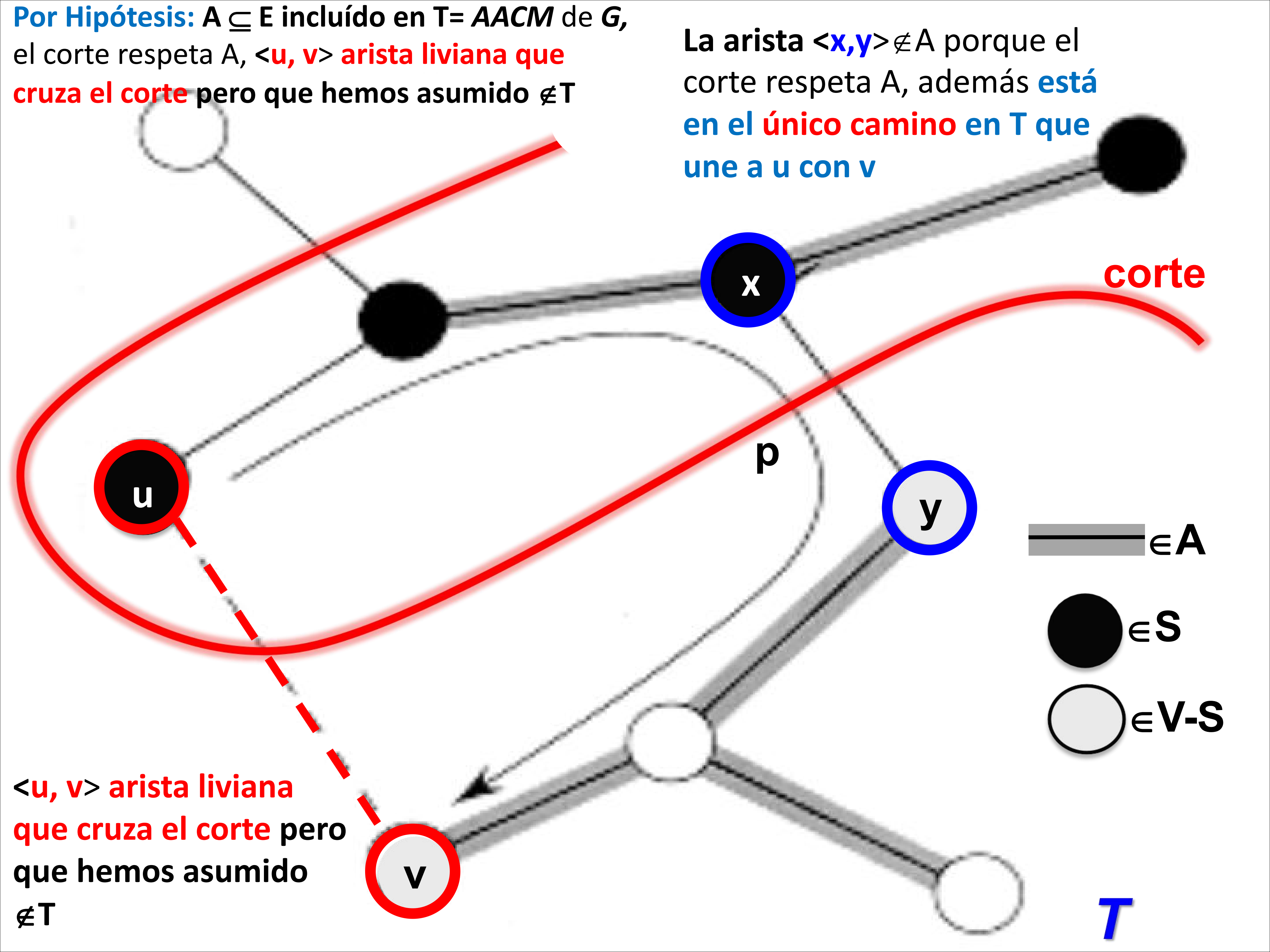
Sea *T* un *AACM* de *G* tal que $A \subseteq T$ ($A \subseteq T$ - por Hipótesis)

Si <u, v> $\in T \Rightarrow$ <u, v> es segura y demostrado el Teorema 1

Si <u, v> $\notin T$ entonces:

Como T es un árbol abarcador para el grafo *G* conexo, entonces <u>tiene</u> que existir un camino en T entre el vértice *u* y el vértice *v* y dicho camino tiene que tener alguna arista que cruce el corte, sea dicha arista <x, y>

Ilustremos con un ejemplo la situación descrita hasta ahora teniendo en cuenta las hipótesis del Teorema:


Por Hipótesis: A ⊆ E incluído en T= AACM de G, el corte respeta A, <u, v> arista liviana que cruza el corte pero que hemos asumido ∉T
La arista <x,y>∉A porque el corte respeta A, además está en el único camino en T que une a u con v
corte
x
p
u
y
∈A
∈S
∈V-S
<u, v> arista liviana que cruza el corte pero que hemos asumido ∉T
v
T

Demostración del Teorema 1 (continuación)

Podemos construir otro *AACM* T’ que incluya a A $\cup${<u, v>} y demostrar que <u,v> es segura para A

Cómo lo construiríamos ?

- La arista <*u*, *v*> formaría un ciclo con las aristas que están en el camino de *u* a *v* en T y la arista <x, y> pertenece a dicho ciclo

- Sustituyamos en T la arista <x, y> por la arista <u, v>

- Sea T’ el árbol resultante

Teorema 1

$$T' = T - \{<x, y>\} + \{<u, v>\}$$

$$\omega(T') = \omega(T) - \omega(<x, y>) + \omega(<u, v>)$$

Se cumple que:

$\omega(<u, v>) \leq \omega(<x, y>)$ --- ***porque <u, v> es liviana***

entonces, $\omega(T') \leq \omega(T)$

- $\omega(T') < \omega(T)$ no es posible porque entonces T no sería un AACM

Por tanto, $\omega(T') = \omega(T) \therefore T'$ AACM

El Teorema 1 y el algoritmo genérico

El **Teorema 1** permite comprender mejor el funcionamiento del algoritmo genérico

- En cualquier momento de la ejecución del algoritmo genérico, las aristas en *A* forman un grafo G_A = *(V, A).*

- Las componentes conexas de grafo G_A son árboles, por tanto, constituyen un bosque.

- Algunos de los árboles de dicho bosque pueden tener un solo nodo y son los vértices para los cuales ninguna arista que incide en ellos ha sido aun insertada en el conjunto *A*

Por ejemplo, tras las inicializaciones del algoritmo genérico:
A=ϕ y el bosque G_A tiene |*V*| árboles, cada uno de un vértice

El Teorema 1 y el algoritmo genérico

- Cualquier arista segura <*u*, *v*> para *A* tiene que conectar dos componentes conexas diferentes de G_A, ya que $G_A \cup \{<u, v>\}$ tiene que ser acíclico
- Incialmente, cuando A=ϕ , en G_A hay |V| árboles de un solo vértice y en cada iteración, se reduce en 1 el número de estos
- Cuando G_A, finalmente, contiene un solo árbol, entonces concluye la ejecución del algoritmo genérico

Corolario 2

Colorario 2

-Sea $G=(V, E)$ **conexo, no dirigido y ponderado**

-Sea $A \subseteq E$ **incluido en algún *AACM* de *G***

-Sea $C=(Vc, Ec)$ **comp. conexa** (árbol) en el bosque

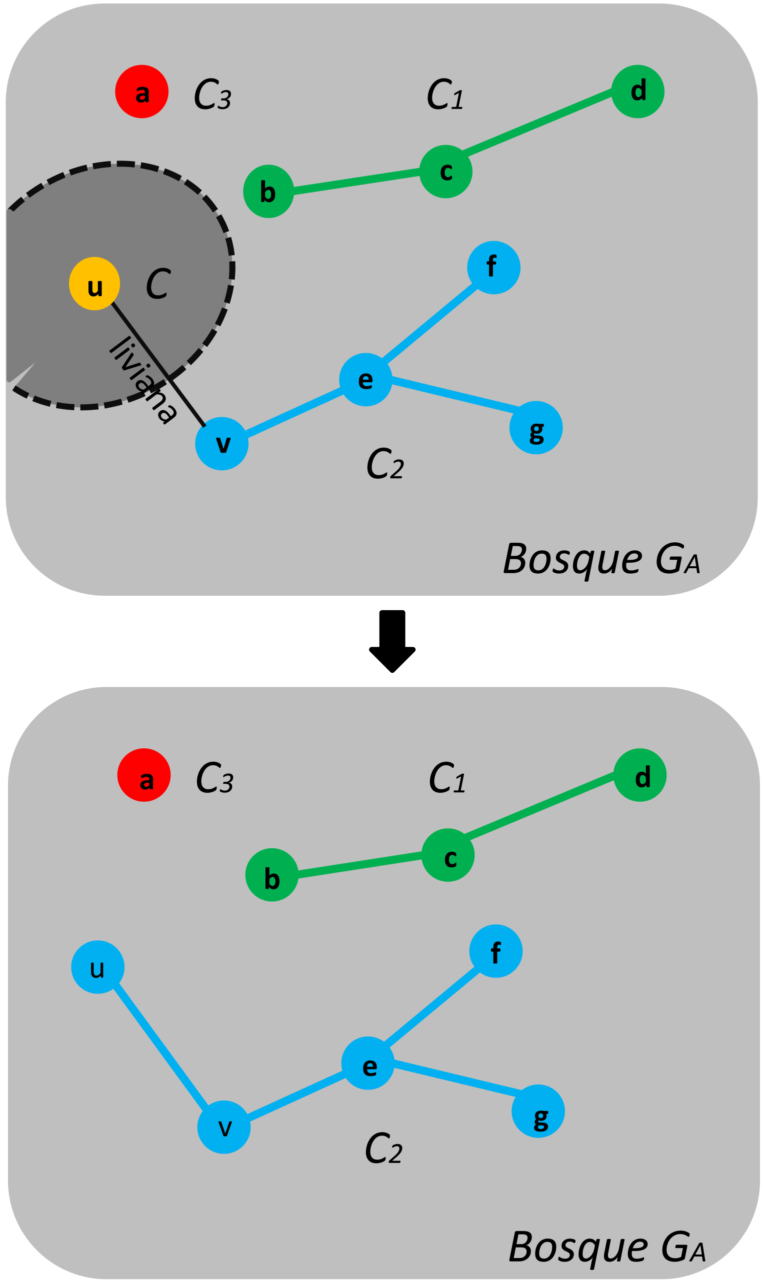
$$G_A = (V, A)$$

-Si <u, v> es una arista liviana que conecta a *C* con cualquier otra componente en G_A, entonces

<*u*, *v*> **segura** para A

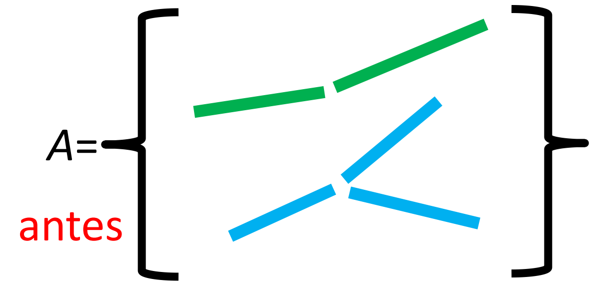
Ilustremos lo expresado en el Corolario 2 a partir del ejemplo que habíamos visto en la demostración del Teorema 1:

El corte
$(V_C, V- V_C)$
respeta A

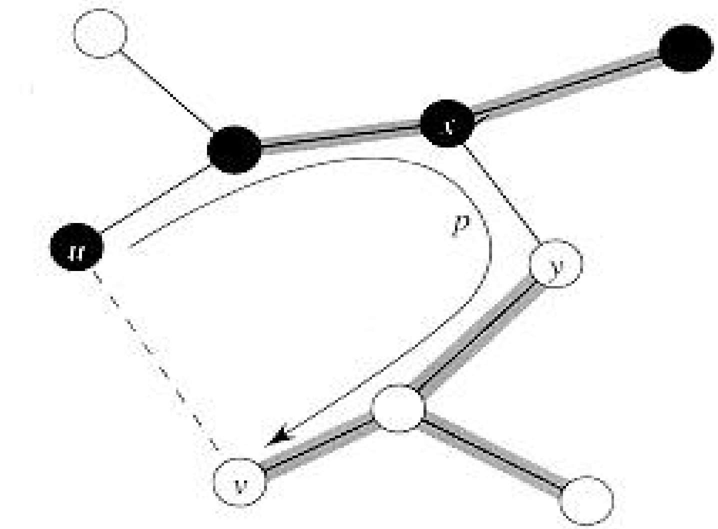


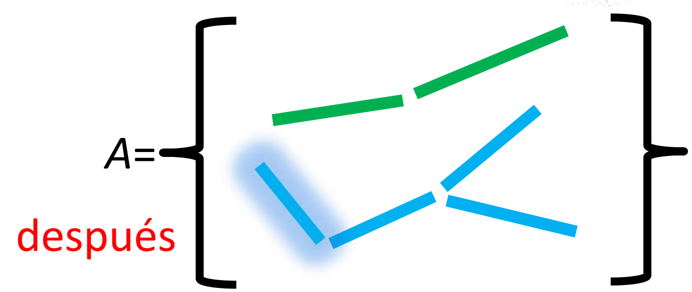
$G=(V, E)$ $A \subset E$

$V=\{a, b, c, d, e\ f, g, u, v\}$



Bosque $G_A = \{C1=\mathbf{T1}, C2=\mathbf{T2}, C3=\mathbf{T3}, C=\mathbf{T4}\}$





Bosque $G_A = \{C1=\mathbf{T1}, C2=\mathbf{T2}, C3=\mathbf{T3}\}$

Demostración del Corolario 2

Demostración

El corte (Vc, V - Vc) respeta a A y <u, v> es una arista liviana de las que cruzan el corte, entonces por **Teorema 1**

<u, v> es **segura** para A

El Colorario 2 sirve de marco teórico para los algoritmos de Kruskal y PRIM

Algoritmo de PRIM y Kruskal

Los dos algoritmos **determinan un AACM** de *G*

Son **variantes del algoritmo genérico**: cada uno tiene **una forma distinta de determinar la arista segura**

Kruskal: El conjunto *A*, en un momento intermedio de la ejecución del algoritmo, es **un conjunto de árboles** dentro del bosque G_A (los restantes vértices que aun no forman parte de las aristas de A constituyen árboles de un solo vértice en el bosque G_A)

Prim: Las aristas en A, en un momento intermedio de la ejecución del algoritmo inducen un Grafo $\boldsymbol{G_A = < V_A , A >}$ **.** $\boldsymbol{G_A}$ es un árbol. V_A es el conjunto de los vértices sobre los que incide alguna arista de *A*

Algoritmo de Kruskal

Estrategia (*glotona*) de funcionamiento

Encuentra la **arista segura,** para añadir a *A* bajo el siguiente criterio:

Seleccionar, **entre todas las aristas que enlazan árboles distintos** del bosque G_A, la arista <u, v> **de menor peso (*liviana*)**

Lo siguiente justifica la credibilidad del criterio planteado:

Sean C_1 y C_2 dos árboles del bosque G_A

Sea u$\in C_1$, v$\in C_2$ y <u, v>$\in$E (**<u, v> conecta a C1 con C2**).

Sea el corte (C_1, V-C_1) (**<u, v> cruza el corte**),

entonces,

si <u, v> es **liviana**, **entonces por Corol. 2**$\Rightarrow$es **segura para A**

Algoritmo de Kruskal

La implementación de Kruskal que se ofrece utiliza la estructura de datos: **Conjuntos Disjuntos** para representar los árboles del bosque G_A

Los conjunto están formados, en cada caso, por los vértices que pertenecen a un mismo árbol en el bosque G_A

SetOf : permite saber si dos vértices pertenecen al mismo árbol

Merge: permite *unir* dos árboles

Algoritmo de Kruskal

```
Kruskal
A ← ϕ
Inicializar una estructura de
conjunto disjunto donde cada vértice
de G es un conjunto
Ordenar las aristas de E en orden no
creciente con respecto a su costo
Para cada arista <u, v>∈E, tomadas
según la ordenación:
  if SetOf(u)≠ SetOf(v)
      A ← A ∪{<u, v>}
      Merge(u, v)
Return A
```

Algoritmo de Kruskal

L 1-2: Inicializan a *A* vacío y crear |V| árboles de un nodo cada uno

L 3: Las aristas en E se ordenan para ir tomándolas en orden no decreciente con respecto al peso

L 4-7: Chequear para cada arista <*u*, *v*> seleccionada si sus extremos pertenecen al mismo árbol:

SI - <*u*, *v*> no puede añadirse al bosque, formaría un ciclo, entonces la arista se descarta

NO - los extremos pertenecen a árboles diferentes y la arista se añade a A, mezclando en un mismo conjunto a los vértices del árbol al que pertenece *u* con los del árbol de *v*

8: retornar A = *AACM*

Complejidad Temporal - Algoritmo de Kruskal

Complejidad temporal:

- El tiempo de ejecución del algoritmo de Kruskal depende de la implementación que se tenga de la estructura de conjuntos disjuntos

- Asumiremos que se tiene la implementación con la *unión por cantidad o altura* (*Vea Conferencia Conj. Disj. EDA I*)

- Inicializar la estructura de Conjunto Disjunto es: O(|V|)

- Ordenar las aristas es O(|E| log |E|)

Se realizan O(|E|) operaciones (setOf ó Merge) sobre la estructura, en O(log |V|) cada una, para el caso peor

Complejidad Temporal - Algoritmo de Kruskal

```
Kruskal
A ← ϕ
Inicializar una estructura de
conjunto disjunto donde cada vértice
de G es un conjunto                      O(|V|)
Ordenar las aristas de E en orden
creciente con respecto a su costo O(|E|log|E|)
Para cada arista <u, v>∈E, tomadas
según la ordenación O(|E|)
  if SetOf(u)≠ SetOf(v) O(log|V|)
      A ← A ∪{<u, v>}
      Merge(u, v) O(log|V|)              O(|E| log |V|)
Return A
```

- G representado por una **Lista de Adyacencia**

Kruskal

O(|V|)

Como G es conexo, entonces ,
$|V| - 1 \leq |E| \Rightarrow |V|$ es $O(|E|)$ $\therefore$
este costo no es el preponderante
en el orden general del algoritmo

O(|E|log|E|)

O(|E|log|V|)

$|E| < |V|^2$

$\Rightarrow \quad \log|E| < \log|V|^2$

$\Rightarrow \quad \log|E| < 2\log|V|$

por tanto, log|E| es $O(\log|V|)$
entonces, el costo del algoritmo es

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

- G denso o representado por una **Matriz de Adyacencia**

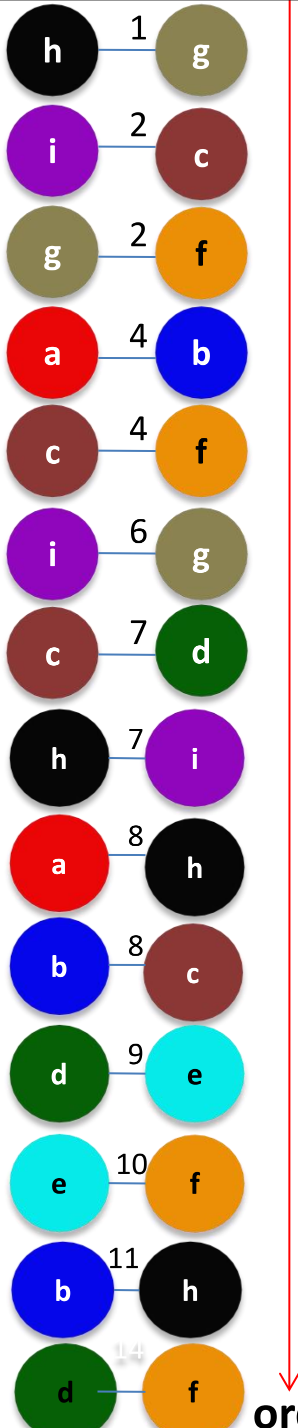
Entonces |E| es $O(|V|^2)$, entonces el costo quedaría

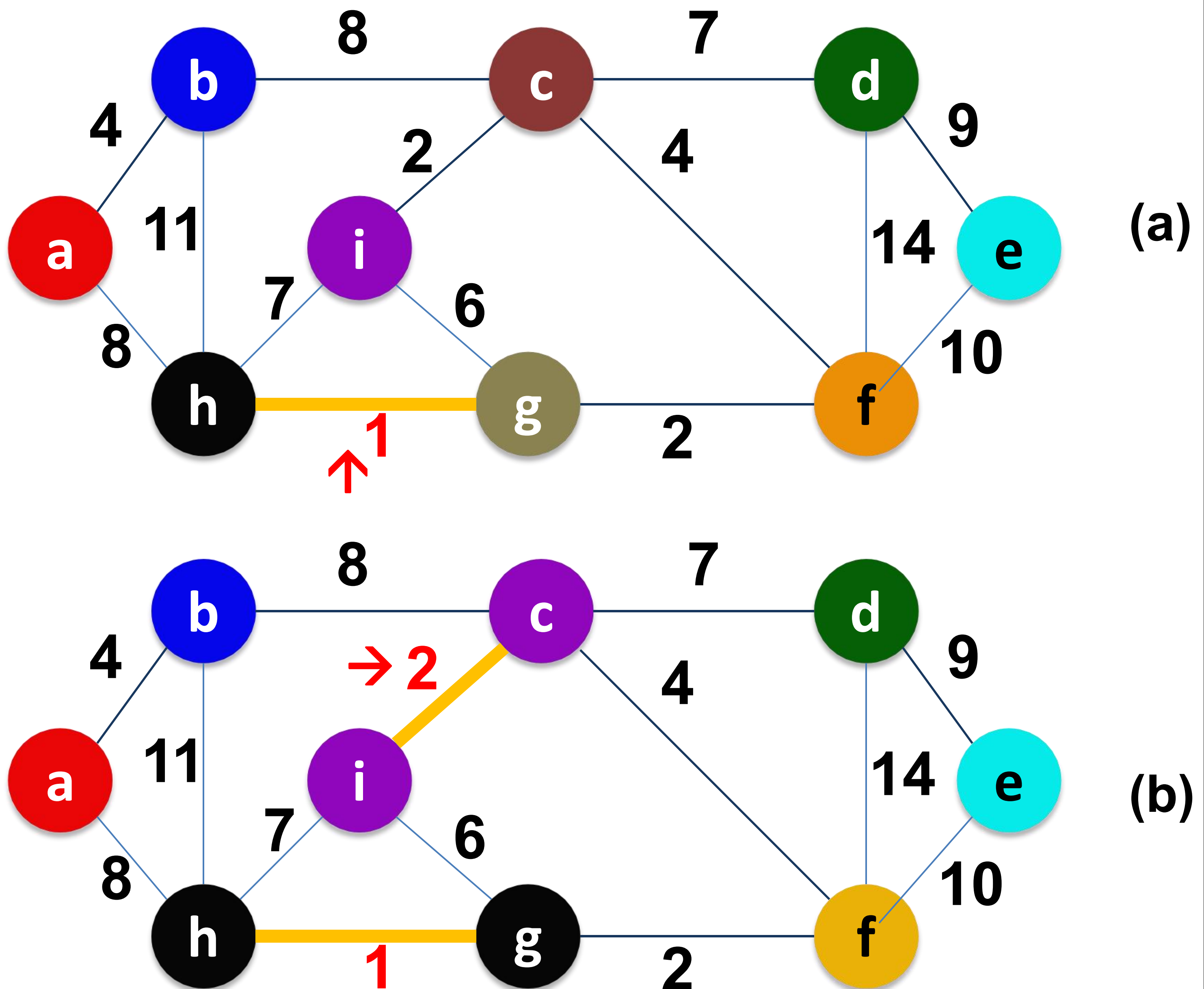
$O(|V|^2 \log |V|)$

Algoritmo de Kruskal

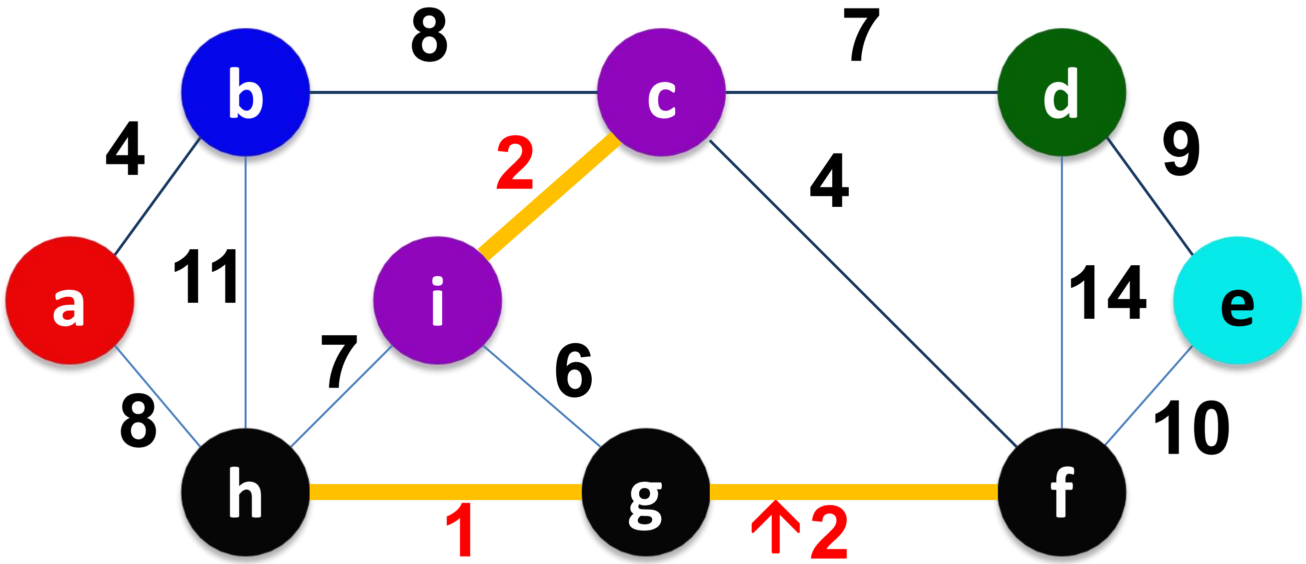
Observación:

Si G **no fuera conexo, puede aplicarse Kruskal a cada componenete conexa**. Para cada una se obtendrá un *AACM* y finalmente, para G lo que se obtiene es un bosque formado por los *AACM* obtenidos para cada una de sus componentes conexas



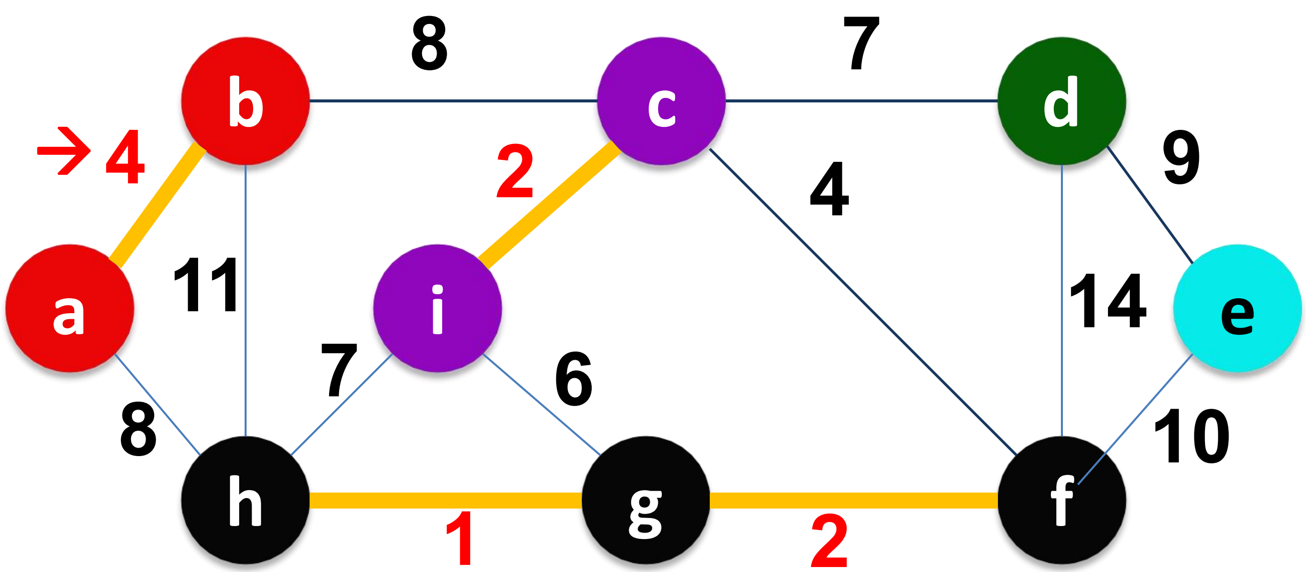


ordenación por costo de las aristas



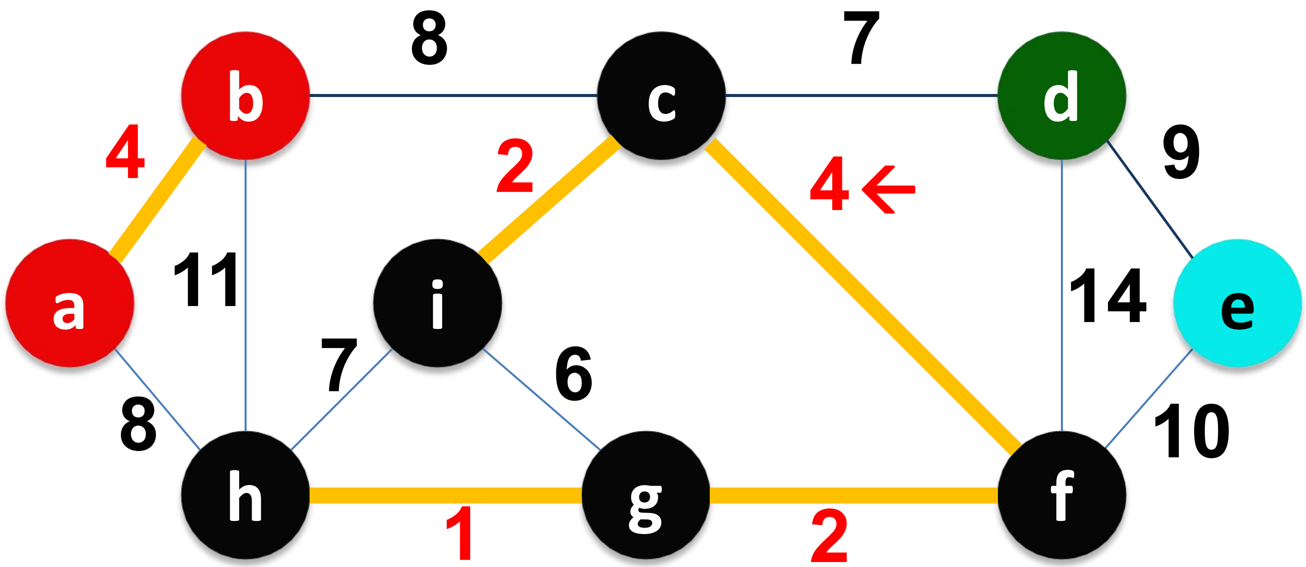
a
b
c
d
e
f
g
h
i
8
7
4
2
4
9
11
7
6
14
8
10
1
↑2

(c)

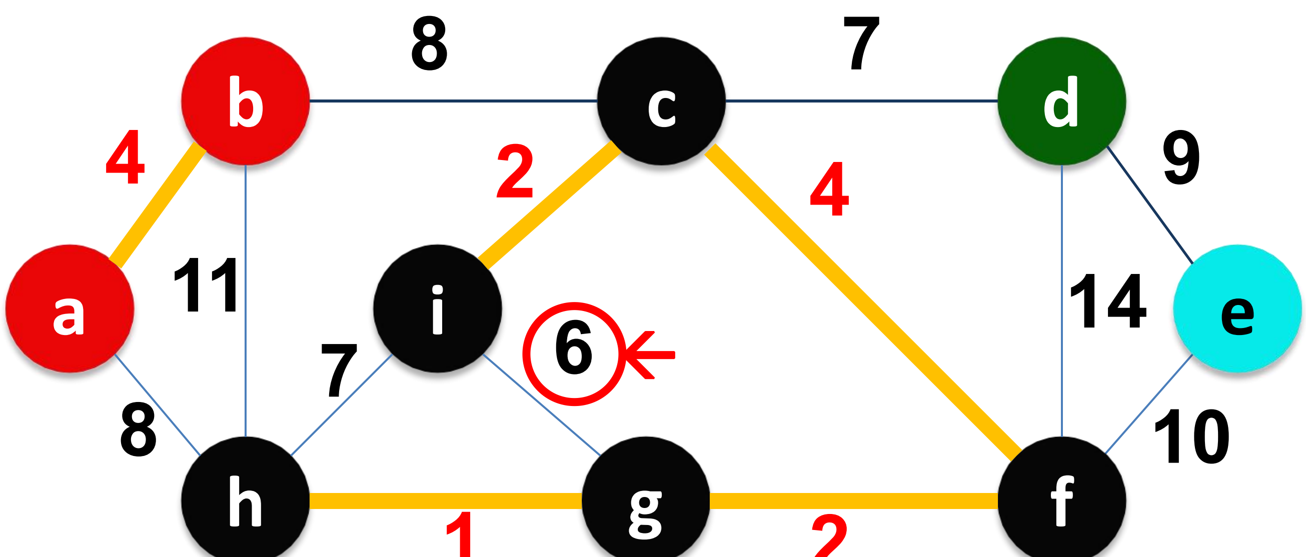


a
b
c
d
e
f
g
h
i
8
7
→ 4
2
4
9
11
7
6
14
8
10
1
2

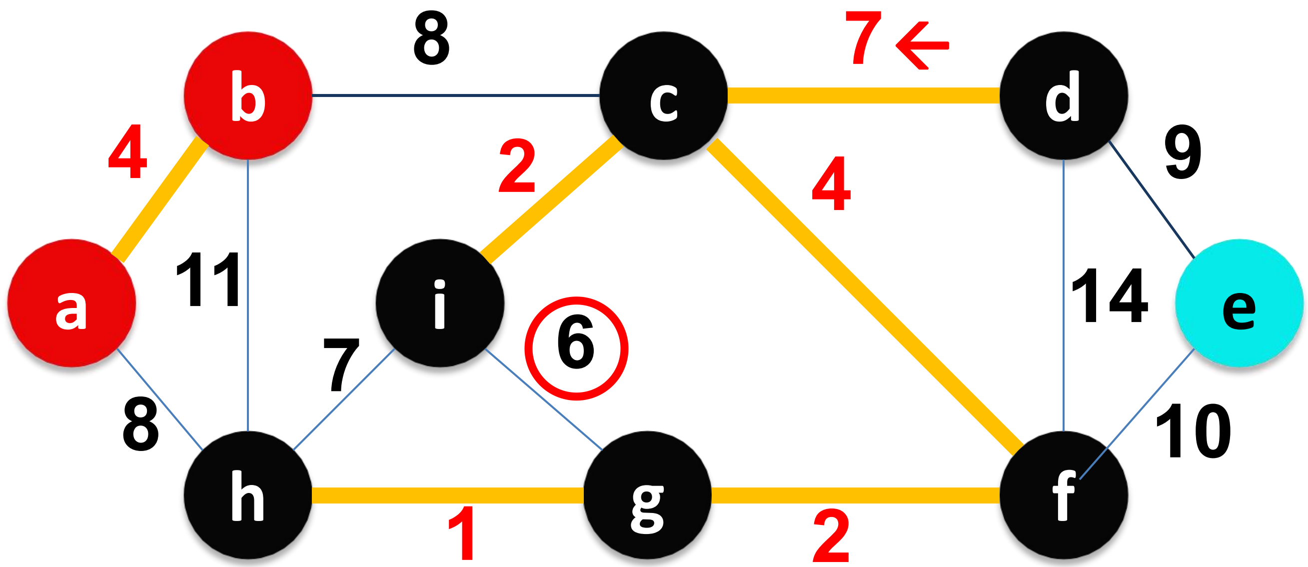
(d)


a
b
c
d
e
f
g
h
i
4
8
7
2
4
9
11
14
7
6
8
10
1
2

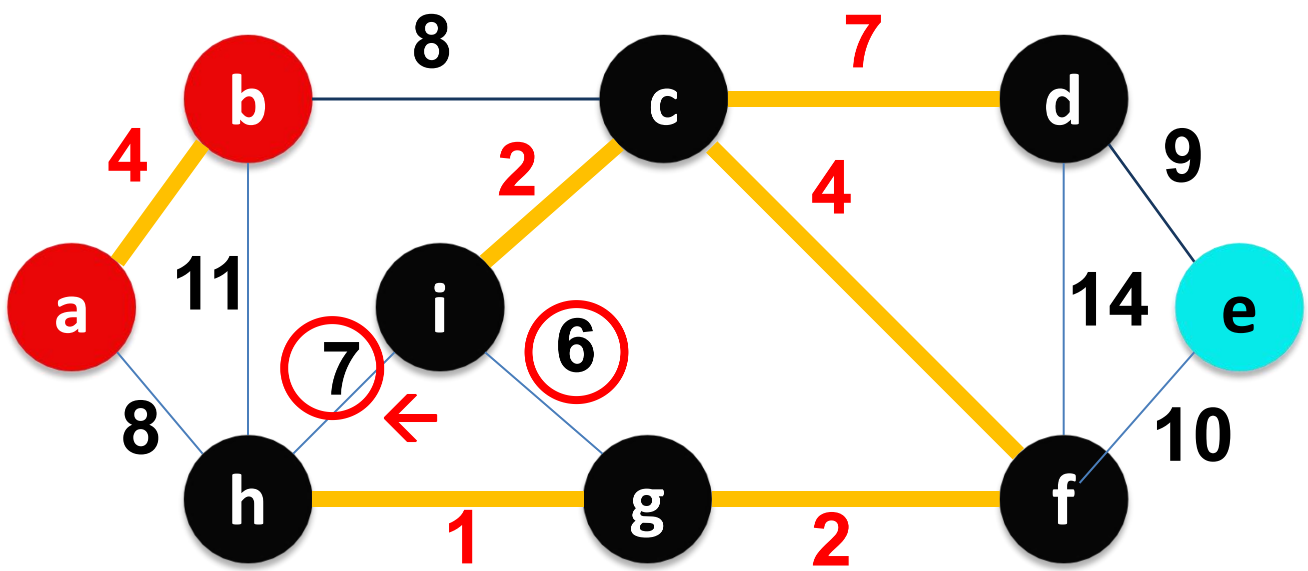
(e)


a
b
c
d
e
f
g
h
i
4
8
7
2
4
9
11
14
7
6
8
10
1
2

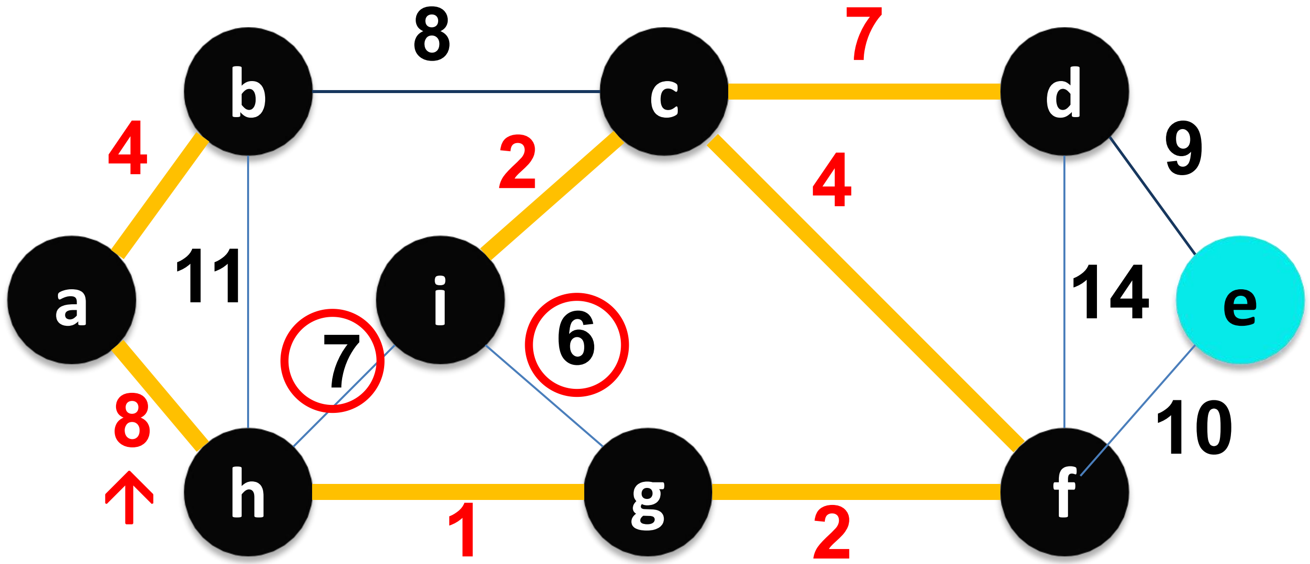
(f)


8
7
b
c
d
4
2
4
9
11
14
a
i
6
e
7
8
10
h
g
f
1
2

(g)

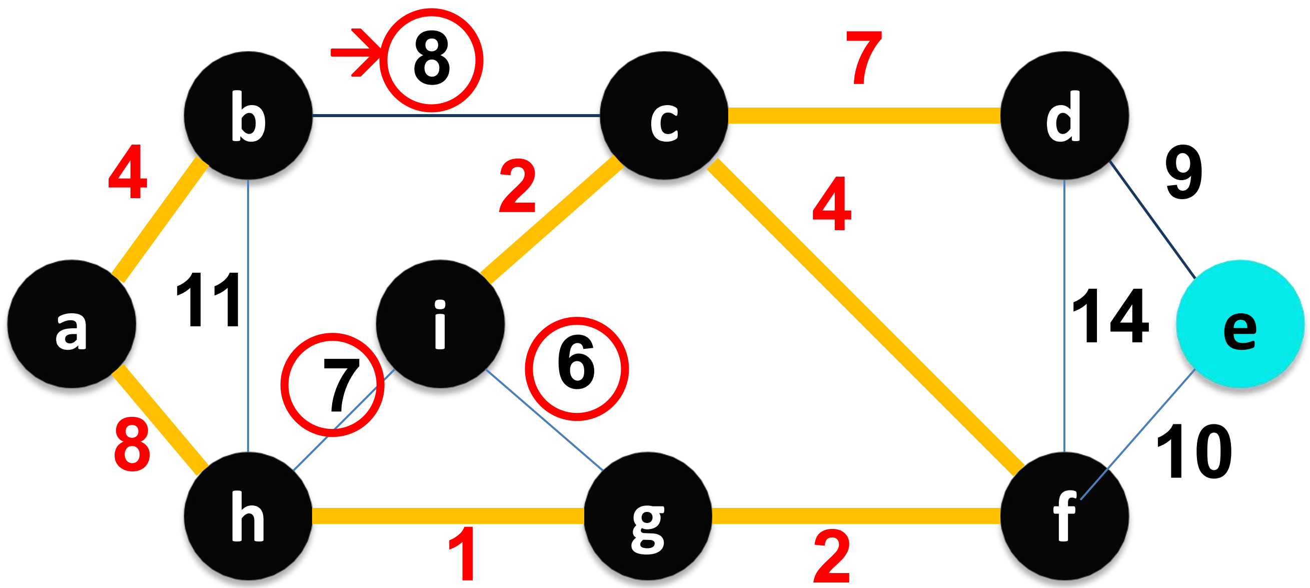

8
7
b
c
d
4
2
4
9
11
14
a
i
7
6
e
8
10
h
g
f
1
2

(h)



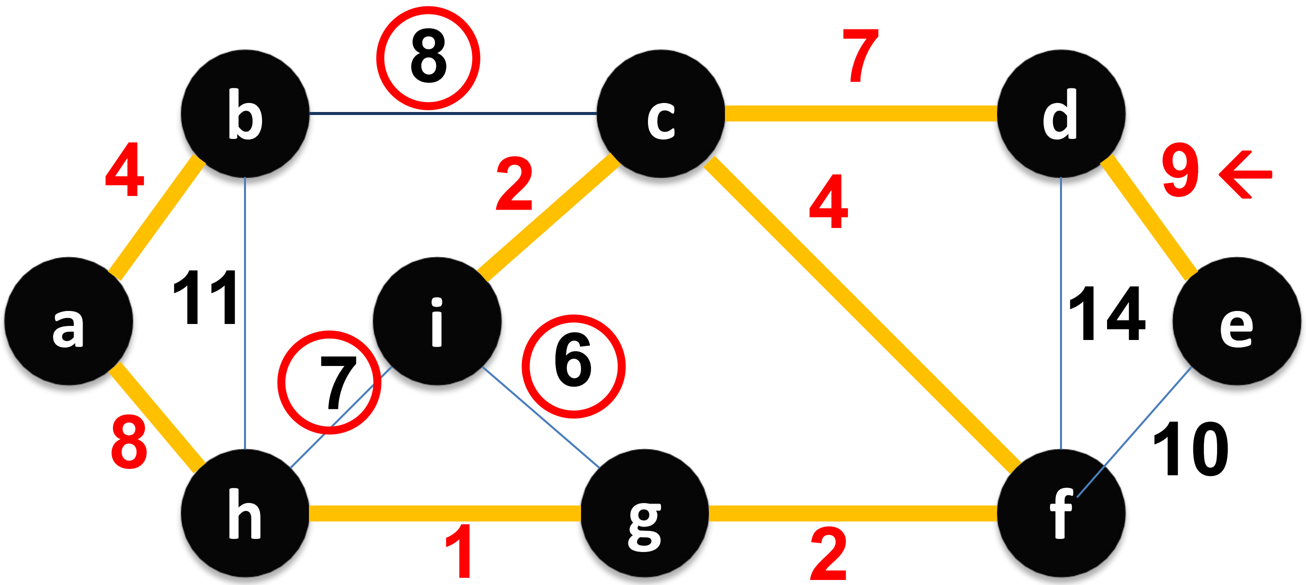
8
7
b
c
d
4
2
4
9
a
11
i
14
e
7
6
8
10
h
g
f
1
2

(i)



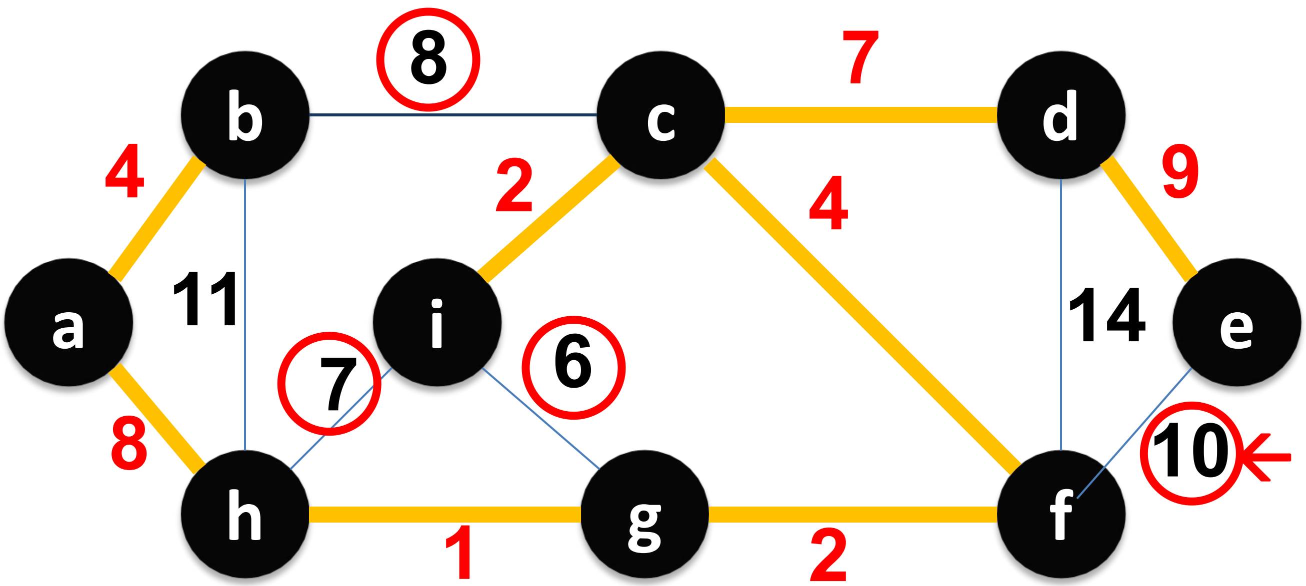
8
7
b
c
d
4
2
4
9
a
11
i
14
e
7
6
8
10
h
g
f
1
2

(j)



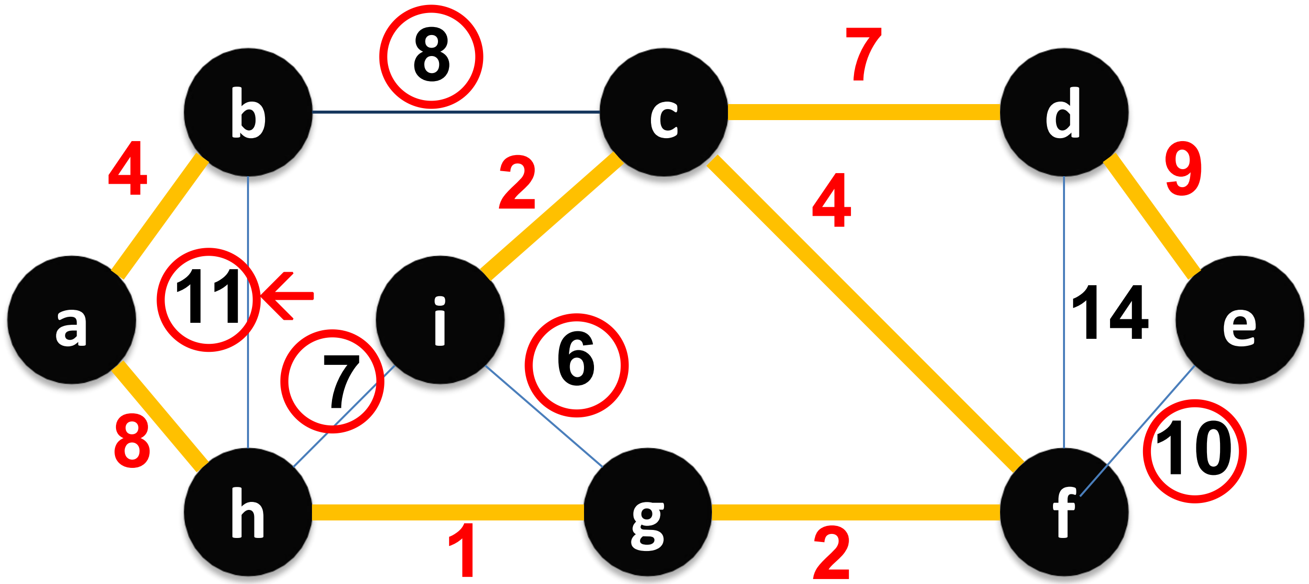
8
7
b
c
d
4
2
4
9
a
11
i
14
e
7
6
8
10
h
g
f
1
2

(k)



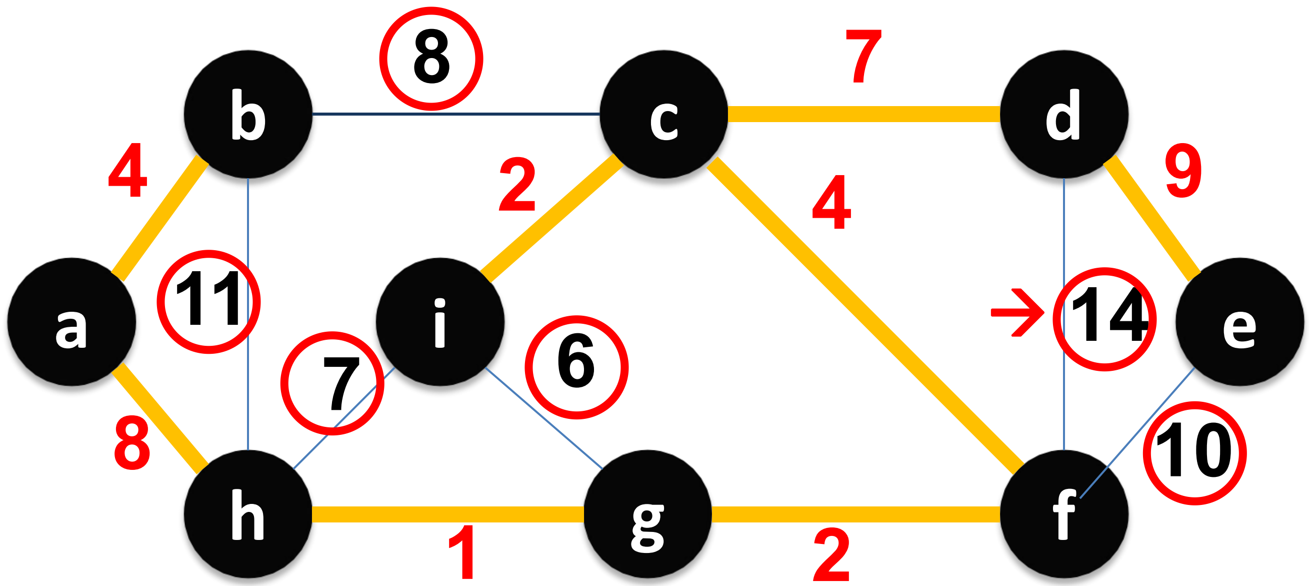
8
7
b
c
d
4
2
4
9
a
11
i
14
e
7
6
8
10
h
g
f
1
2

(l)



a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
7
6
1
2
9
14
10

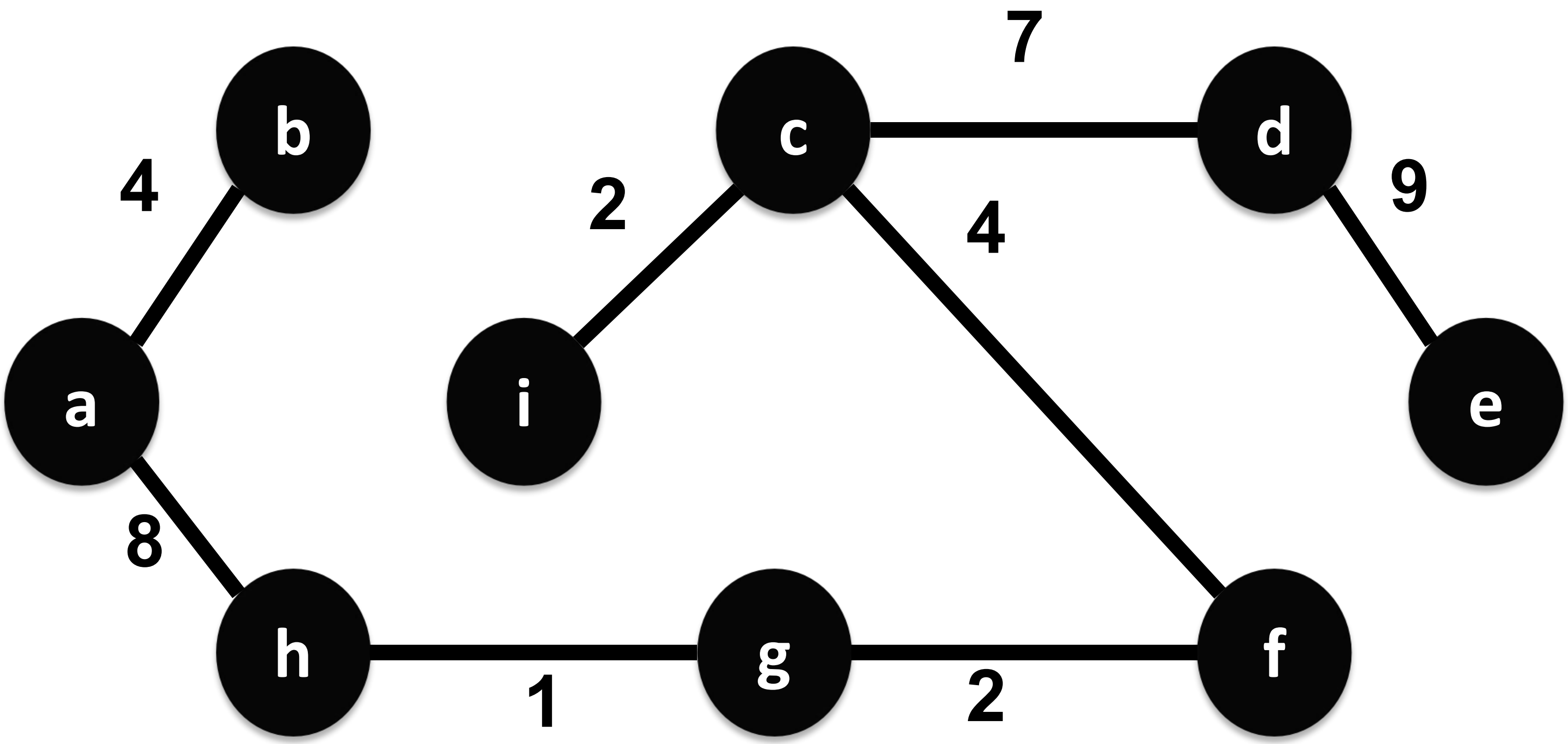
(m)



a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
7
6
1
2
9
14
10

(n)

Resultado de aplicar el Algoritmo de Kruskal



AACM

Algoritmo de PRIM

PRIM es también un caso especial del **algoritmo genérico**

En este caso, las aristas en *A* inducen un grafo $\boxed{\boldsymbol{G_A} = < \boldsymbol{V_A} , A >}$

$\boldsymbol{V_A}$: conjunto de los vértices sobre los que incide alguna arista de A

$\boxed{\boldsymbol{G_A} \text{ es un árbol}}$

Inicialmente, $\boldsymbol{G_A}$ posee un nodo raíz $\boldsymbol{r}$ que se selecciona de forma arbitraria

Durante la ejecución del algoritmo, $\boldsymbol{G_A}$ crece (agregándole en cada pasada una arista más) hasta que llega a contener a todos los vértices de *V*

Algoritmo de PRIM

Estrategia de funcionamiento:

En cada iteración, se considera el corte ($\boldsymbol{V_A}$, *V*- $\boldsymbol{V_A}$), el cual respeta a *A* y se selecciona una **arista liviana para dicho corte, o sea, una arista** que conecta algún vértice de $\boldsymbol{V_A}$ con alguno de *V*- $\boldsymbol{V_A}$, entonces, por el **Teorema 1**, dicha arista es segura para *A*, por tanto, se añade a dicho conjunto

Tras |V|-1 iteraciones, entonces las aristas en *A* forman un ***AACM***

Estrategia glotona para PRIM:

El árbol $\boldsymbol{G_A}$ siempre se incrementa con una arista que le aporta el menor costo posible

Algoritmo de PRIM

Detalles de implementación:

- Para implementar PRIM eficientemente es necesario hacer fácil la selección de la arista liviana que se adiciona al conjunto *A*

- El vértice raíz *r* se da como entrada

- $\forall v \in V$: $v \notin \boldsymbol{V_A}$;

$$d[v] \text{ (su } \textbf{\textit{distancia}} \textbf{ de } \boldsymbol{G_A}) \quad \begin{cases} \mathit{mín}(w(\langle v, a\rangle : a \in \boldsymbol{V_A}) & \text{si } \exists \langle v, a\rangle \\ \infty & \text{si no existe } \langle v, a\rangle \end{cases}$$

- Los vértices que aun no forman parte del árbol $\boldsymbol{G_A}$ se mantienen en una **cola con prioridad** de acuerdo con su valor de *distancia* de $\boldsymbol{G_A}$

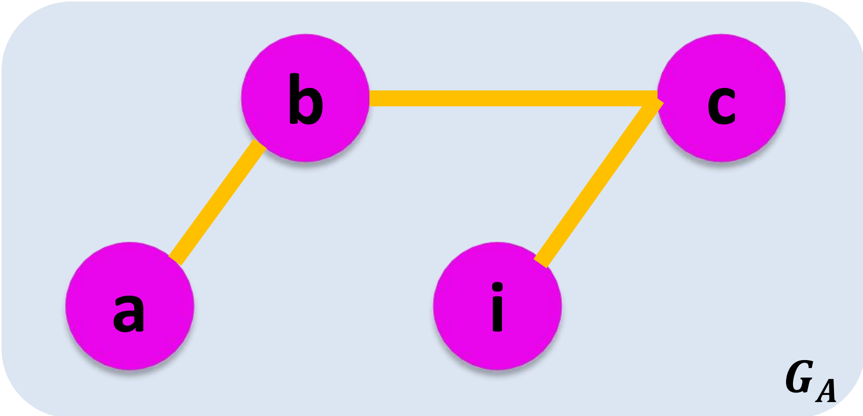
- $\pi[v]$ (vértice cercano de v): es el otro extremo (que no es v) de la arista que marca la distancia de v con $\boldsymbol{G_A}$

A={<a, b>, <b, c>, <i, c>}

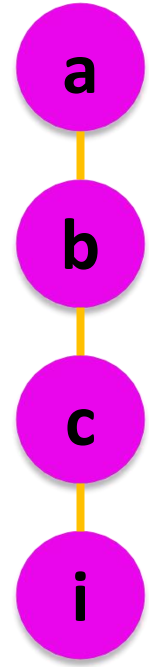
CORTE

V_A ={a, b, i, c}

V - V_A ={h, g, d, f, e}



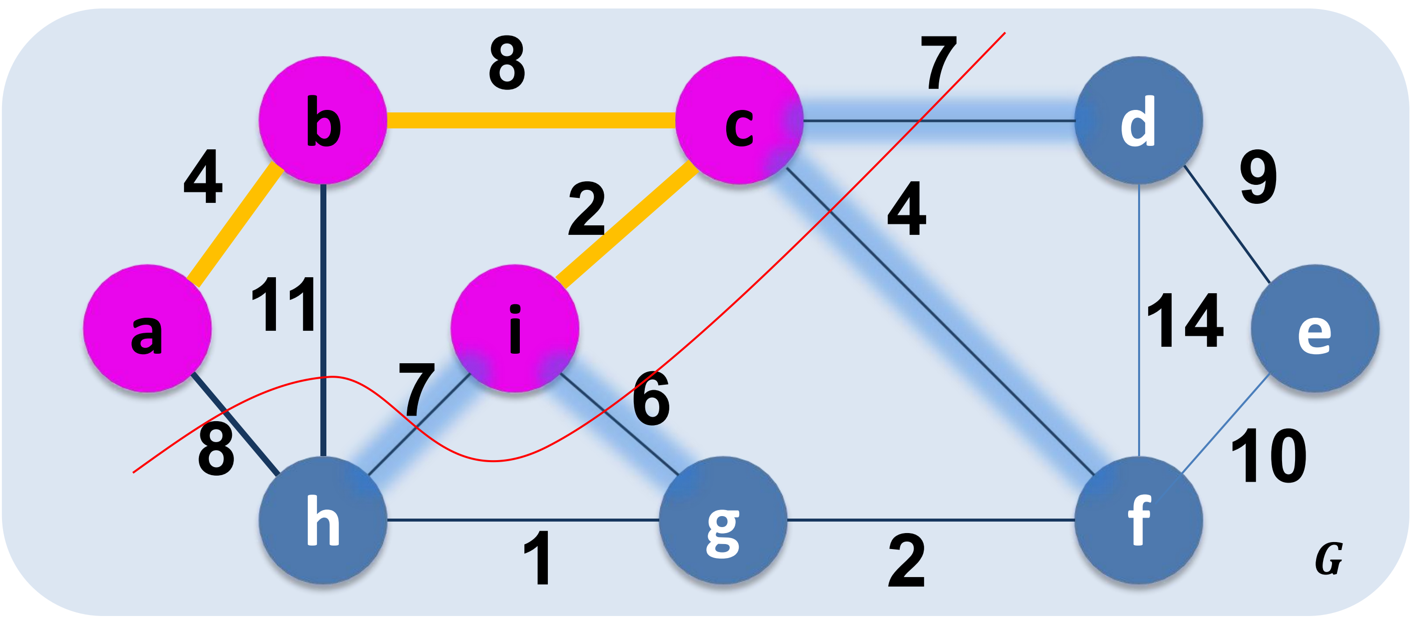
raíz



G_A *es un árbol*

d[h]=7 = mín (8, 7, 11)
d[g]=6
d[d]=7
d[f]=4
d[e]=∞

π[h]=i
π[g]=i
π[d]=c
π[f]=c
π[e]=*null*



Algoritmo de PRIM

```
PRIM(G, costo, r)

for each vértice u∈V[G]
    do distancia[u]←∝
       π[u]←null
distancia[r]← 0
Q ← V[G] //meter en cola todos los vértices
while Q no esté vacía
    do u ← EXTRAE_MIN(Q)
       for each v adyacente a u
           do if v∈Q and costo <u,v> < distancia[v]
                 then π[v]←u
                      distancia[v]←costo <u, v>
```

inicialmente:

$V\text{-}Q = \phi$ / $Q = |V|$

Algoritmo de PRIM

Línea 1- 5: Inicializaciones

- $\forall v \in V$: $v \neq r$, $d[v] = \infty$; $d[r]=0$

-El arreglo π indica cuál fue el vértice que atrajo a cada vértice hacia el AACM. A partir del mismo se generan las aristas del AACM

$\forall v \in V$: $v \neq r$, $\pi[v] = null$; $\pi[r]$= "un valor distintivo"

- Cola con Prioridad: Q ={v: $v \in V$ }

- Inicialmente, V-Q = ϕ,
- Durante el proceso, V-Q = {vértices del AACM en crecimiento}

Algoritmo de PRIM

Línea 6:

Se comienza un proceso iterativo que se ejecuta mientras que la cola Q no se vacíe. Dentro de él:

Línea 7:

- Identificar el vértice $u \in Q$ que es el extremo en Q de la arista liviana que cruza el corte (V-Q, Q) (con excepción de la primera iteración)

Línea 8-11:

- Se actualizan los valores de d[] y π[], que sean necesarios, para cada vértice adyacente a *u* que aún no está en el árbol

INICIALIZACIONES

$d[a]=0$	$\pi[a]=-$
$d[b]=\propto$	$\pi[b]=null$
$d[c]=\propto$	$\pi[c] = null$
$d[d]=\propto$	$\pi[d] = null$
$d[e]=\propto$	$\pi[e] = null$
$d[f]=\propto$	$\pi[f] = null$
$d[g]=\propto$	$\pi[g] = null$
$d[h]=\propto$	$\pi[h] = null$
$d[i]=\propto$	$\pi[i] = null$

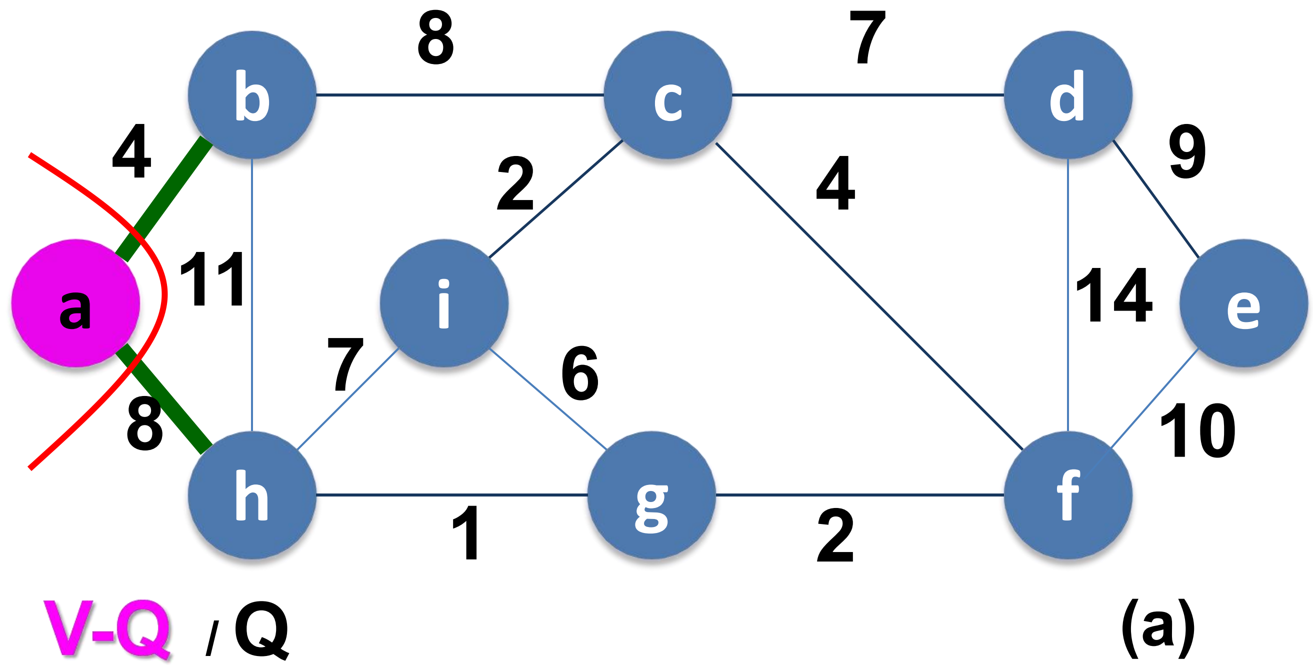
Inicialmente:

Q = {a, b, c, d, e, f, g, h, I}; V-Q = ϕ

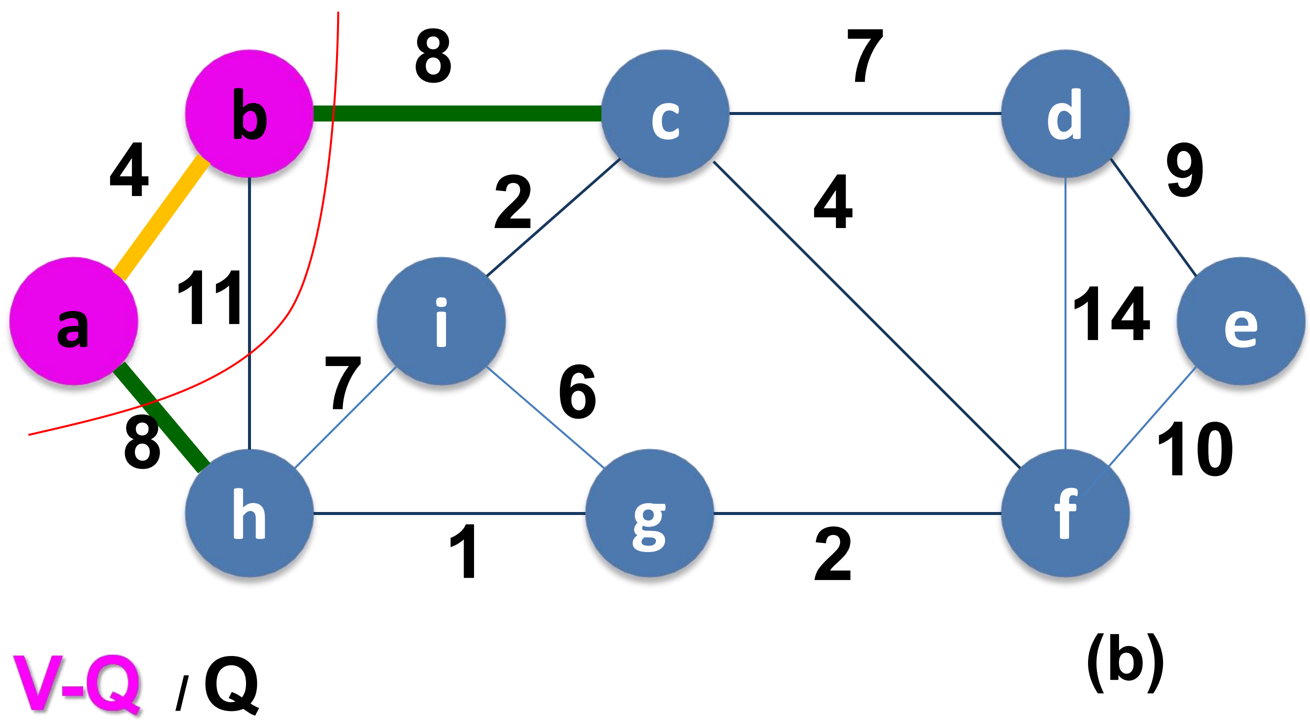
aristas en A:

aristas que cruzan el corte (Q, V-Q):

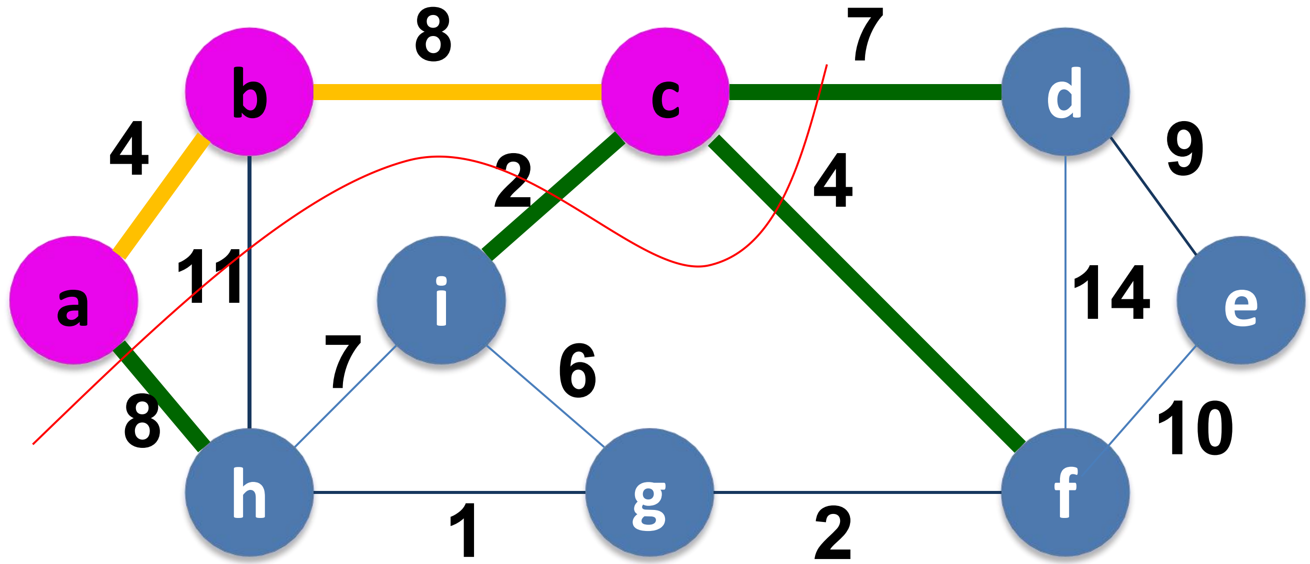
d[a]=0 π[a]=-
d[b]=4 π[b]=a
d[c] =∝ π[c] = null
d[d] =∝ π[d] = null
d[e] =∝ π[e] = null
d[f] =∝ π[f] = null
d[g] =∝ π[g] = null
d[h] =8 π[h] = a
d[i] =∝ π[i] = null



d[a]=0 π[a]=-
d[b]=4 π[b]=a
d[c] =8 π[c] = b
d[d] =∝ π[d] = null
d[e] =∝ π[e] = null
d[f] =∝ π[f] = null
d[g] =∝ π[g] = null
d[h] =8 π[h] = a
d[i] =∝ π[i] = null



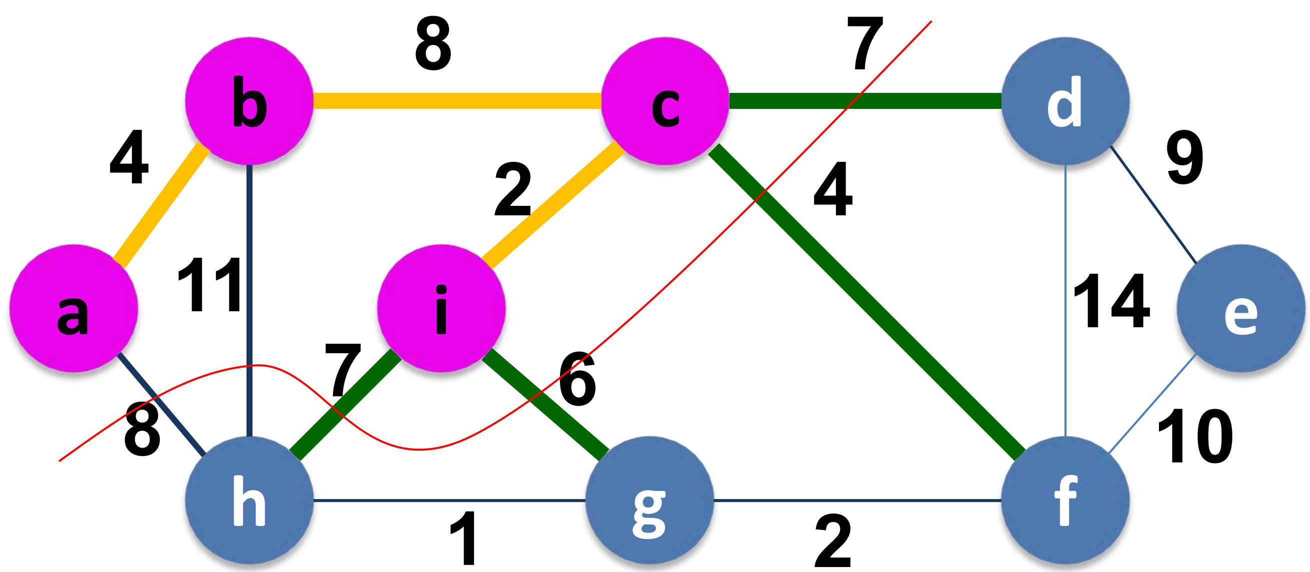
d[a]=0 π[a]=-
d[b]=4 π[b]=a
d[c] =8 π[c] = b
d[d] =7 π[d] = c
d[e] =∝ π[e] = null
d[f] =4 π[f] = c
d[g] =∝ π[g] = null
d[h] =8 π[h] = a
d[i] =2 π[i] = c



V-Q / Q

(c)

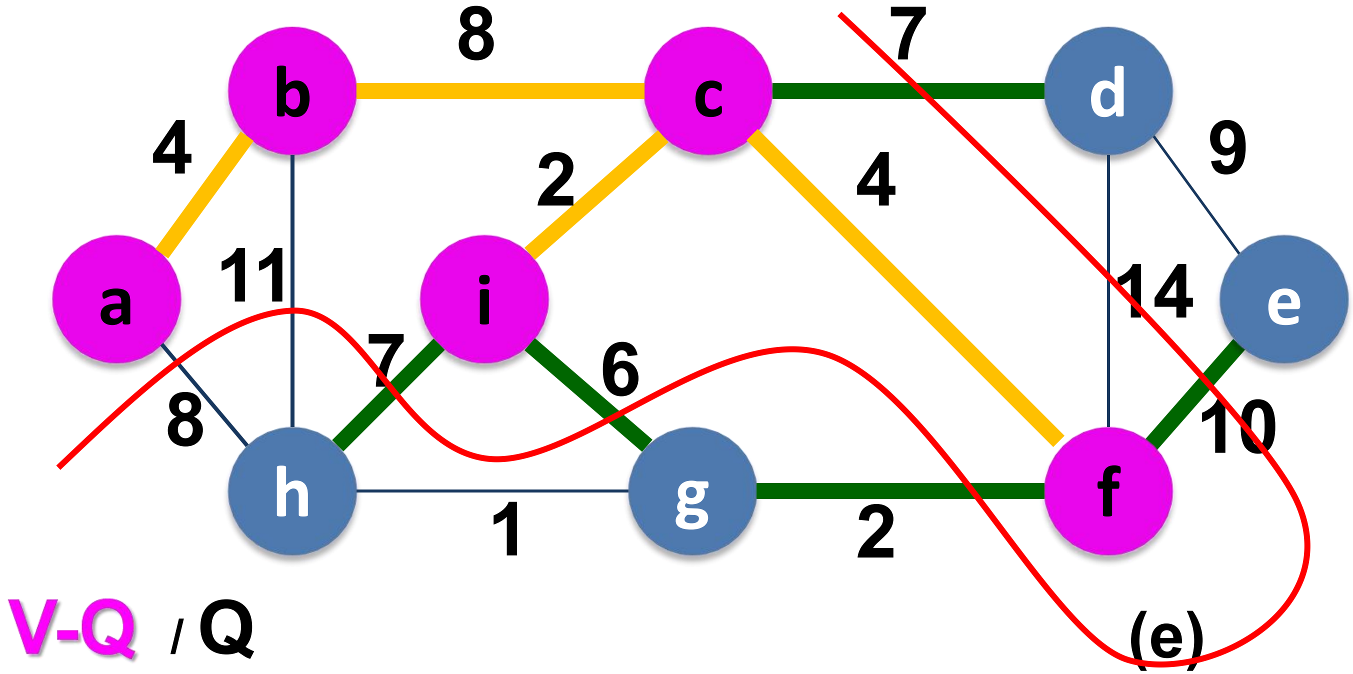
d[a]=0 π[a]=-
d[b]=4 π[b]=a
d[c] =8 π[c] = b
d[d] =7 π[d] = c
d[e] =∝ π[e] = null
d[f] =4 π[f] = c
d[g] =6 π[g] = I
d[h] =7 π[h] = i
d[i] =2 π[i] = c



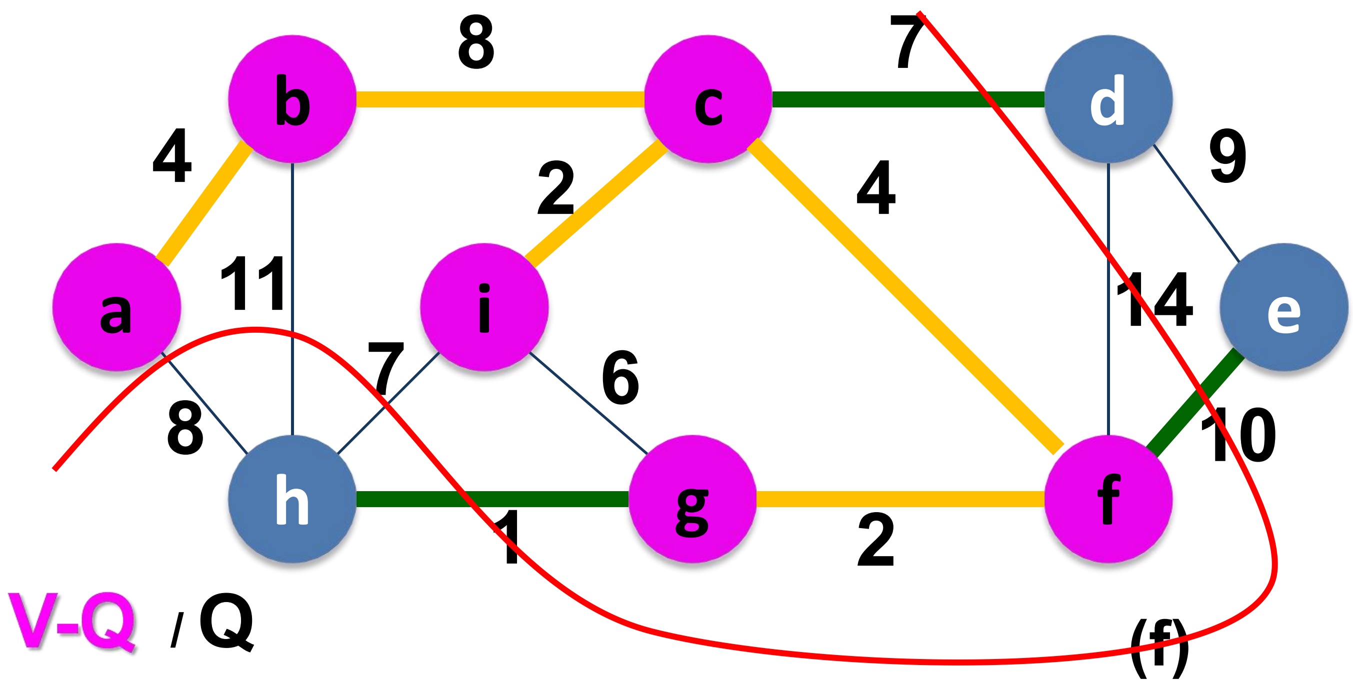
V-Q / Q

(d)

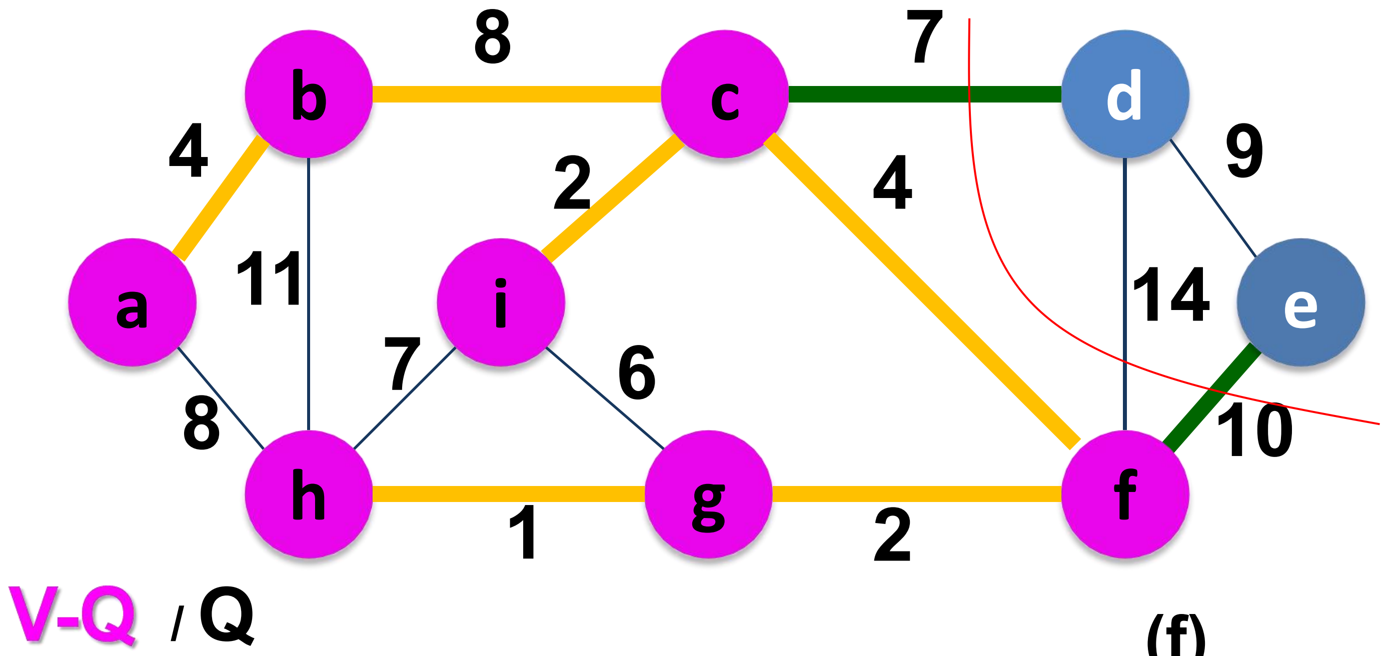
d[a]=0 π[a]=-
d[b]=4 π[b]=a
d[c] =8 π[c] = b
d[d] =7 π[d] = c
d[e] =10 π[e] = f
d[f] =4 π[f] = c
d[g] =2 π[g] = f
d[h] =7 π[h] = i
d[i] =2 π[i] = c



d[a]=0 π[a]=-
d[b]=4 π[b]=a
d[c] =8 π[c] = b
d[d] =7 π[d] = c
d[e] =10 π[e] = f
d[f] =4 π[f] = c
d[g] =2 π[g] = f
d[h] =1 π[h] = g
d[i] =2 π[i] = c

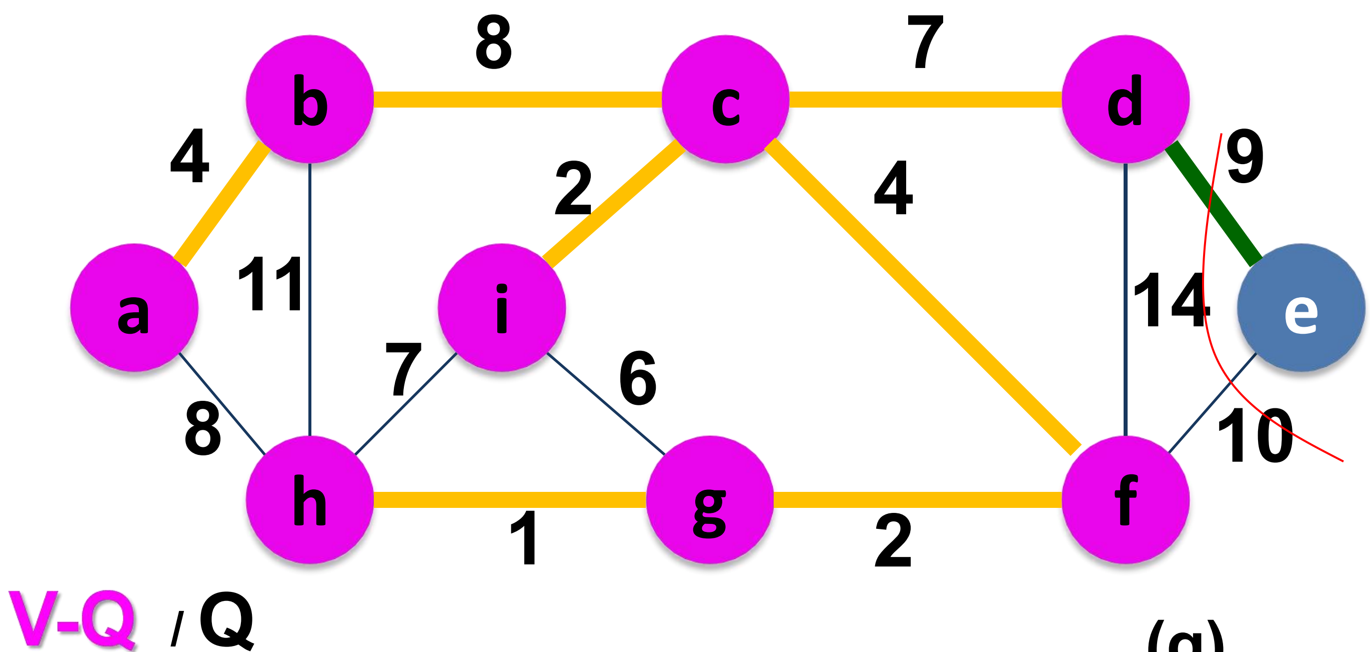


d[a]=0 π[a]=-
d[b]=4 π[b]=a
d[c] =8 π[c] = b
d[d] =7 π[d] = c
d[e] =10 π[e] = f
d[f] =4 π[f] = c
d[g] =2 π[g] = f
d[h] =1 π[h] = g
d[i] =2 π[i] = c



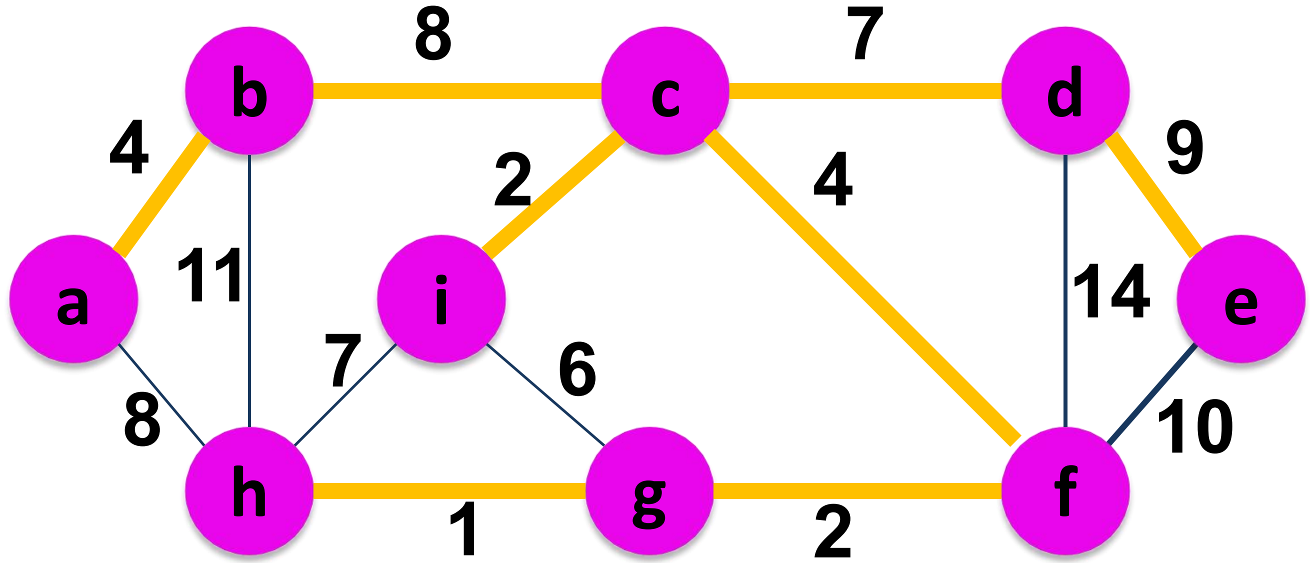
(f)

d[a]=0 π[a]=-
d[b]=4 π[b]=a
d[c] =8 π[c] = b
d[d] =7 π[d] = c
d[e] =9 π[e] = d
d[f] =4 π[f] = c
d[g] =2 π[g] = f
d[h] =1 π[h] = g
d[i] =2 π[i] = c



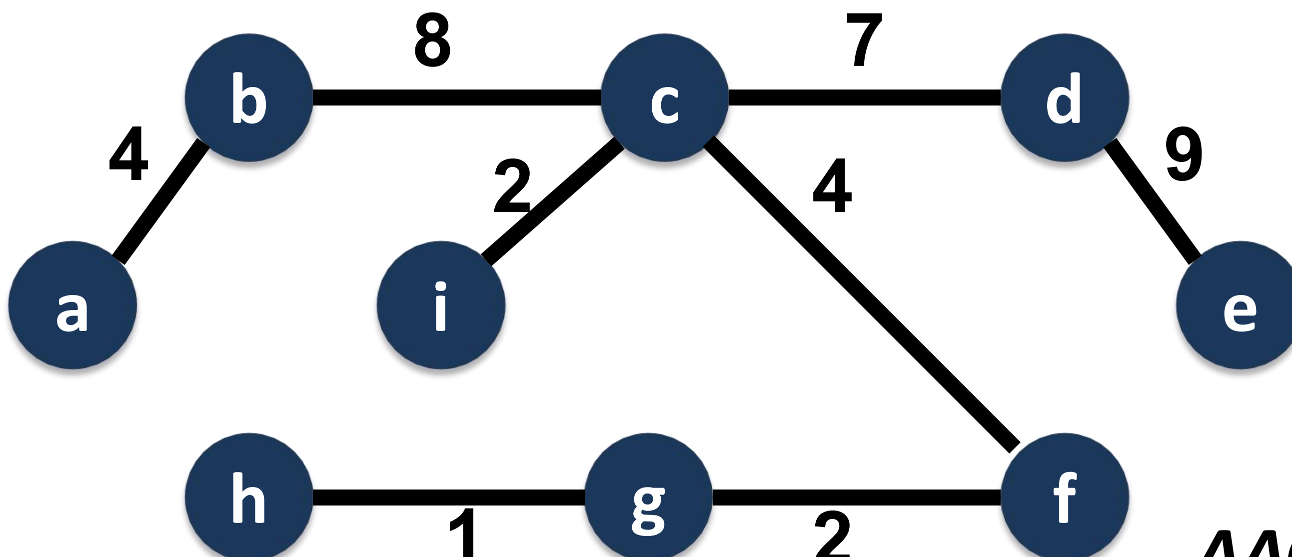
(g)

d[a]=0	π[a]=-
d[b]=4	π[b]=a
d[c] =8	π[c] = b
d[d] =7	π[d] = c
d[e] =9	π[e] = d
d[f] =4	π[f] = c
d[g] =2	π[g] = f
d[h] =1	π[h] = g
d[i] =2	π[i] = c



V-Q / Q=ϕ

(h)



AACM

Algoritmo de PRIM – Complejidad Temporal

El comportamiento de PRIM depende de la forma en que se implemente la **cola con prioridad**: un **HEAP binario**

- Inicializar el HEAP: **BuildHEAP** es *O(IVI)*

- **ciclo 6-11** se ejecuta IVI veces y como EXTRAER-MIN es *O(log IVI),* entonces, el tiempo total de todas las llamadas a EXTRAE-MIN es *O(IVI log IVI)*

- **ciclo 8 – 11** O(IEI): la suma de las longitudes de todas las listas de adyacencia es 2IEI
  Preguntar por la pertenencia a Q puede ser O(1):

array booleano → para cada vértice diga si pertenece o no a Q.
Actualizar *array*, cada vez que se elimina un vértice de Q.

Algoritmo de PRIM – Complejidad Temporal

- La asignación en la **línea 11** implica variación de la **prioridad** de un elemento en Q, lo cual, en un HEAP Binario, puede implementarse en $O(\log |V|)$

Por tanto, Tiempo TOTAL del algoritmo es

$O(|V| \log |V| + |E| \log |V|) = O(|E| \log |V|)$

cuando cantidad de aristas > cantidad de vértices

Si G conexo, $|V|-1 \leq |E| < |V|^2$

Algoritmo de PRIM – Complejidad Temporal

```
PRIM(G, costo, r)

1  for each vértice u∈V[G]
2       do d[u]←∝
3            π[u]←null
4  d[r]← 0
5  Q ← V[G]← Build_Heap: O(|V|)
6  while Q no esté vacía ←O(|V|)
7       do u ← EXTRAE_MIN(Q)←O(log |V|)
8            for each v adyacente a u
9                 do if v∈Q and costo(u,v)<d[v]
10                     then π[v]←u
11                          d[v]←costo <u,v>
```

Líneas 1–4: **O(|V|)**

El orden de complejidad temporal del `for` COMPLETO (costo amortizado) es |E| ya que la cantidad total de aristas a analizar es 2|E|, o sea O(|E|) para una representación del Grafo por listas de adyacencia.

Líneas 9–11: **O(log |V|)**

L-11 ⇒variación de la prioridad de v∈Q, la cual, se actualiza en O(log |V|)

Preguntar por la pertenencia a Q puede implementarse en O(1) teniendo un *array* booleano que para cada vértice diga si pertenece o no a Q. Este arreglo se actualiza cada vez que se remueve un vértice de Q.

O(|E| log |V|)

O(|V| log |V| + |E| log |V|)

Algoritmo de PRIM – Complejidad Temporal

PRIM es $O(|V| \log |V| + |E| \log |V|) \Rightarrow O(|E| \log |V|)$

Observación:

Cuando el **grafo es denso (una restricción sobre el grafo)** y por tanto, $|E|$ *es* $O(|V|^2)$, entonces el orden del algoritmo es $\mathbf{O(|V|^2 \log |V|)}$

En tal caso, es recomendable usar **OTRA versión,** menos eficiente, del algoritmo cuya complejidad temporal, para el caso peor es $O(|V|^2)$ ***(ejercicio de clase práctica)***

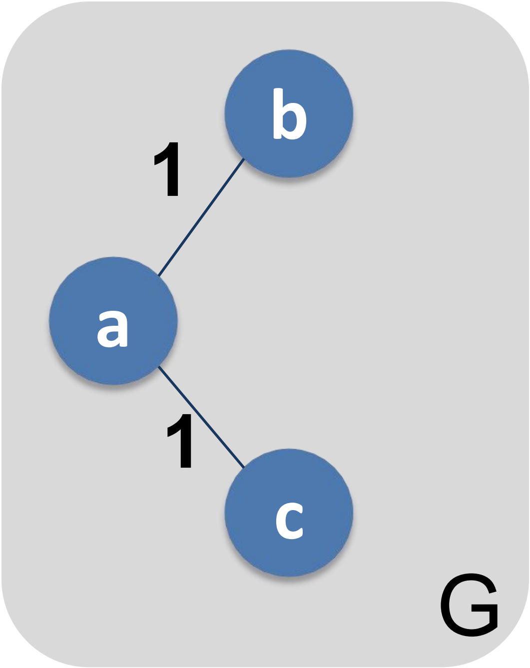
En general, cuando se establecen restricciones para el Grafo se pueden usar distintas implementaciones de la Cola con Prioridad para mejorar la complejidad temporal del algoritmo de PRIM

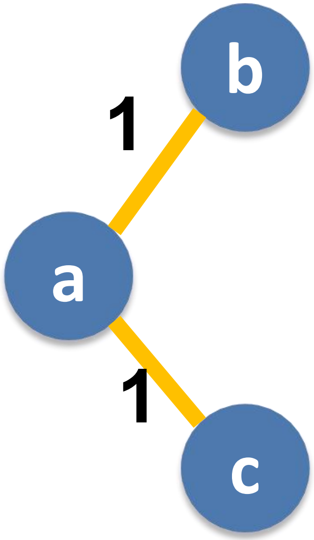
Algoritmo de Kruskal y PRIM

Observación:

No siempre cuando tengo la opción de escoger entre dos **aristas livianas** que tienen igual costo se generan AACM DIFERENTES !!

Contraejemplo:





UNICO
AACM para G