Conferencia No. 4
Estructuras de Datos y Algoritmos II

© Departamento de Programación
Facultad de Matemática y Computación
Universidad de La Habana

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

© Departamento de Programación
Facultad de Matemática y Computación
Universidad de La Habana

Una Aplicación

Diseño de circuitos eléctricos

Problema

Hacer los pines, de varias componentes, eléctricamente equivalentes, uniéndolos por cables

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, $\omega: E \rightarrow \Re$
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 todos los vértices** en V, cuyo ***costo total*** (la suma de todos los *costos* o *pesos* de las aristas) **sea mínimo.** Con todo ello, formar un árbol **T=<V, E'>**

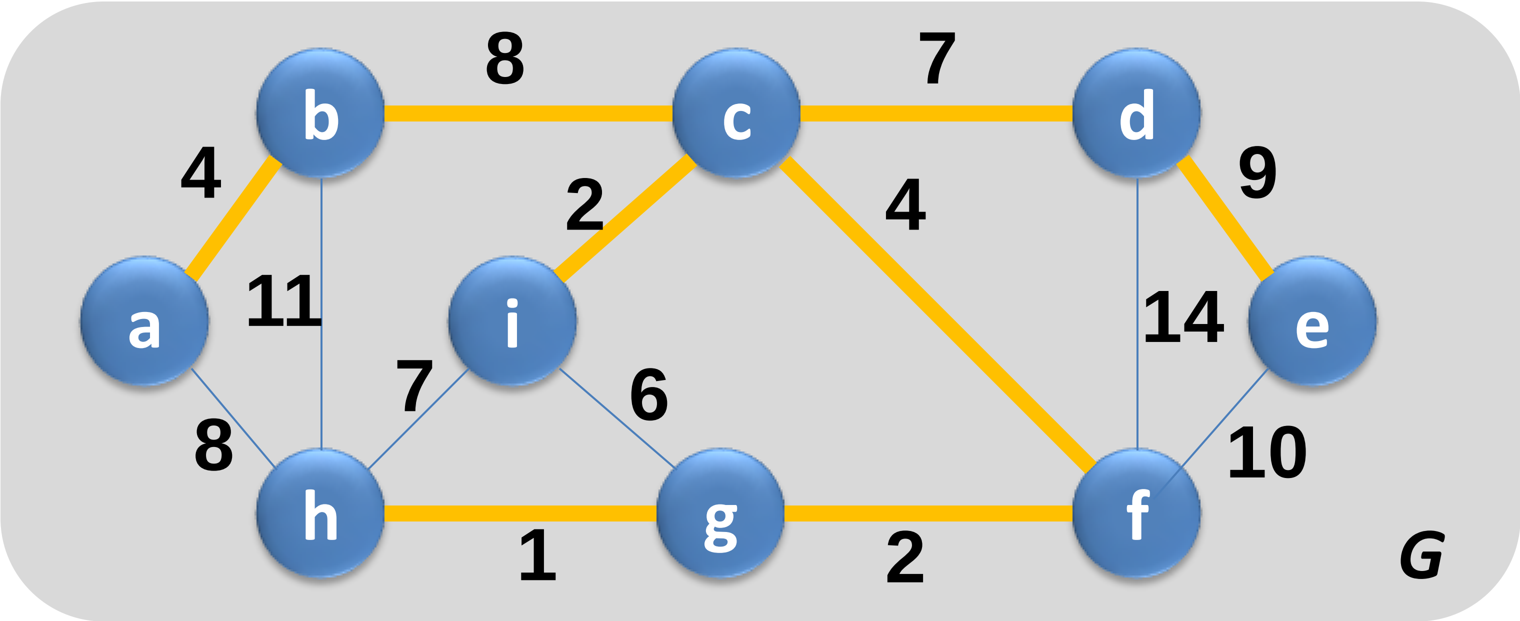
Árbol Abarcador de costo mínimo - *AACM*

- 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 siempre será |V|-1 (porque T es un árbol)**

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, entre varias decisiones posibles, se toma 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 invariante:**

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

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
}
```

- **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:

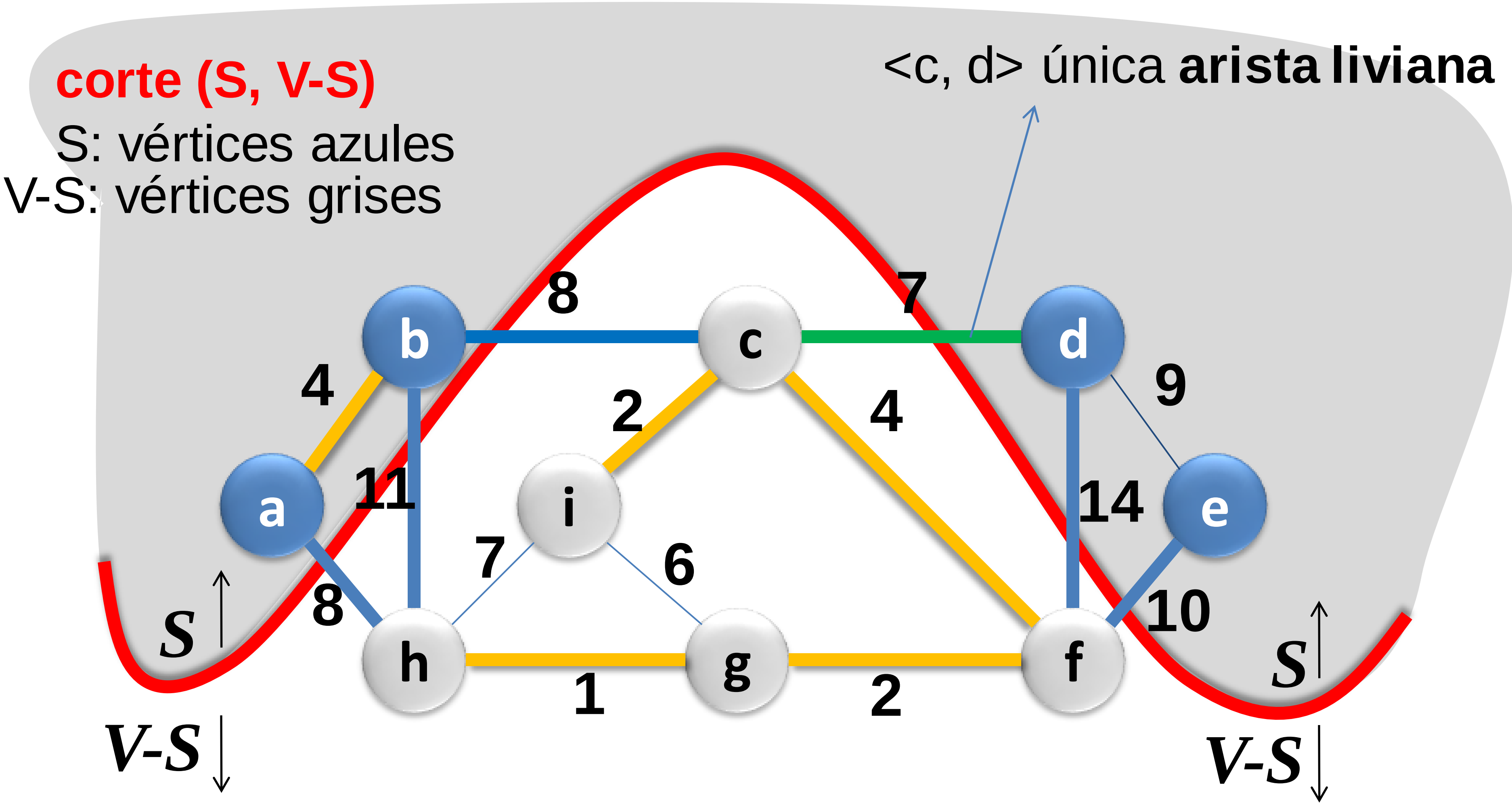
- Un **corte** (S, V-S) de G=(V,E) no dirigido es una partición de los vértices del conjunto V

- 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

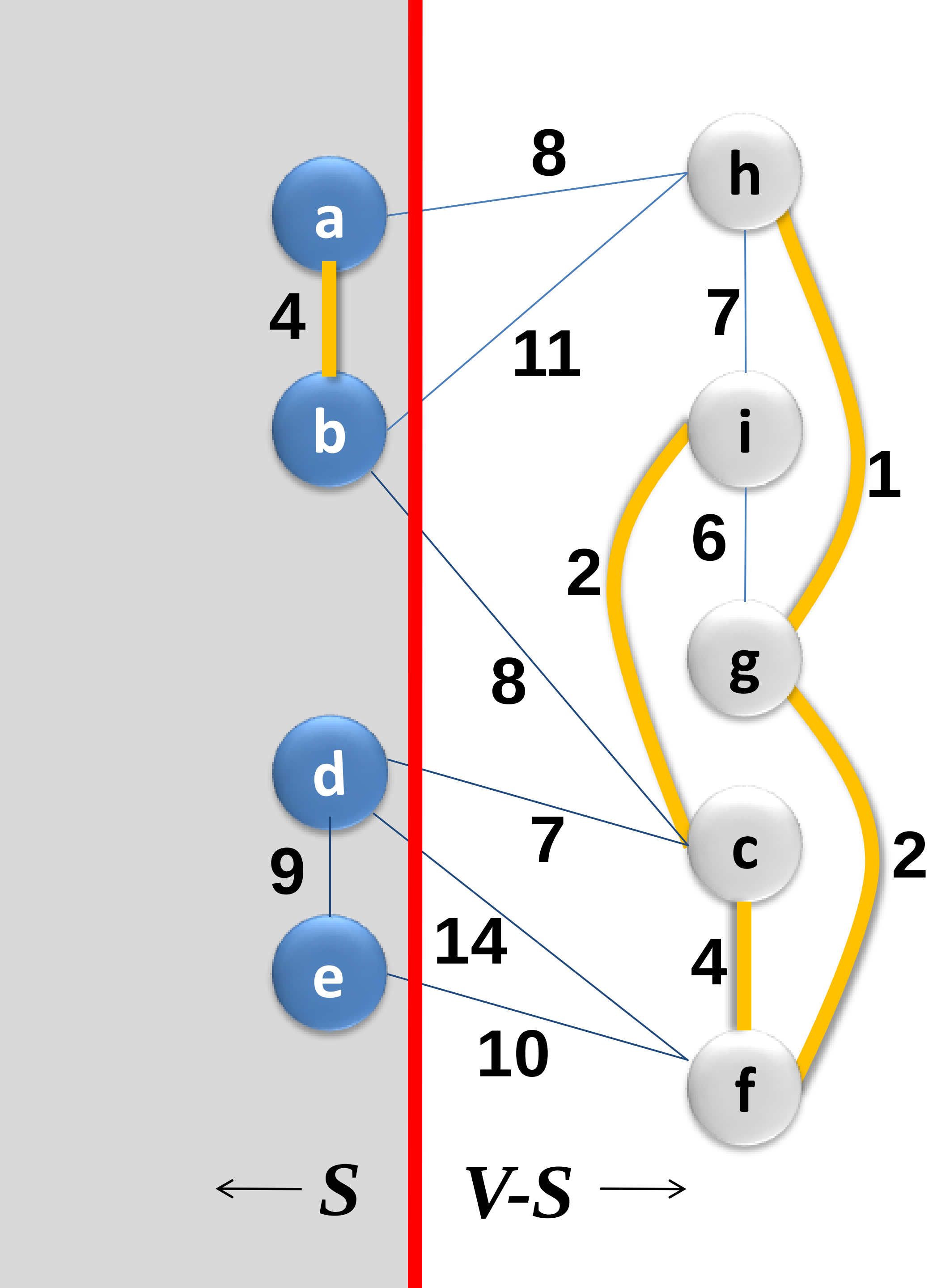
- Un **corte, respeta un conjunto de aristas** A, si NO existen aristas en A que crucen el corte

- 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 la 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> $\in V$ 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*

Teorema 1

Demostración:

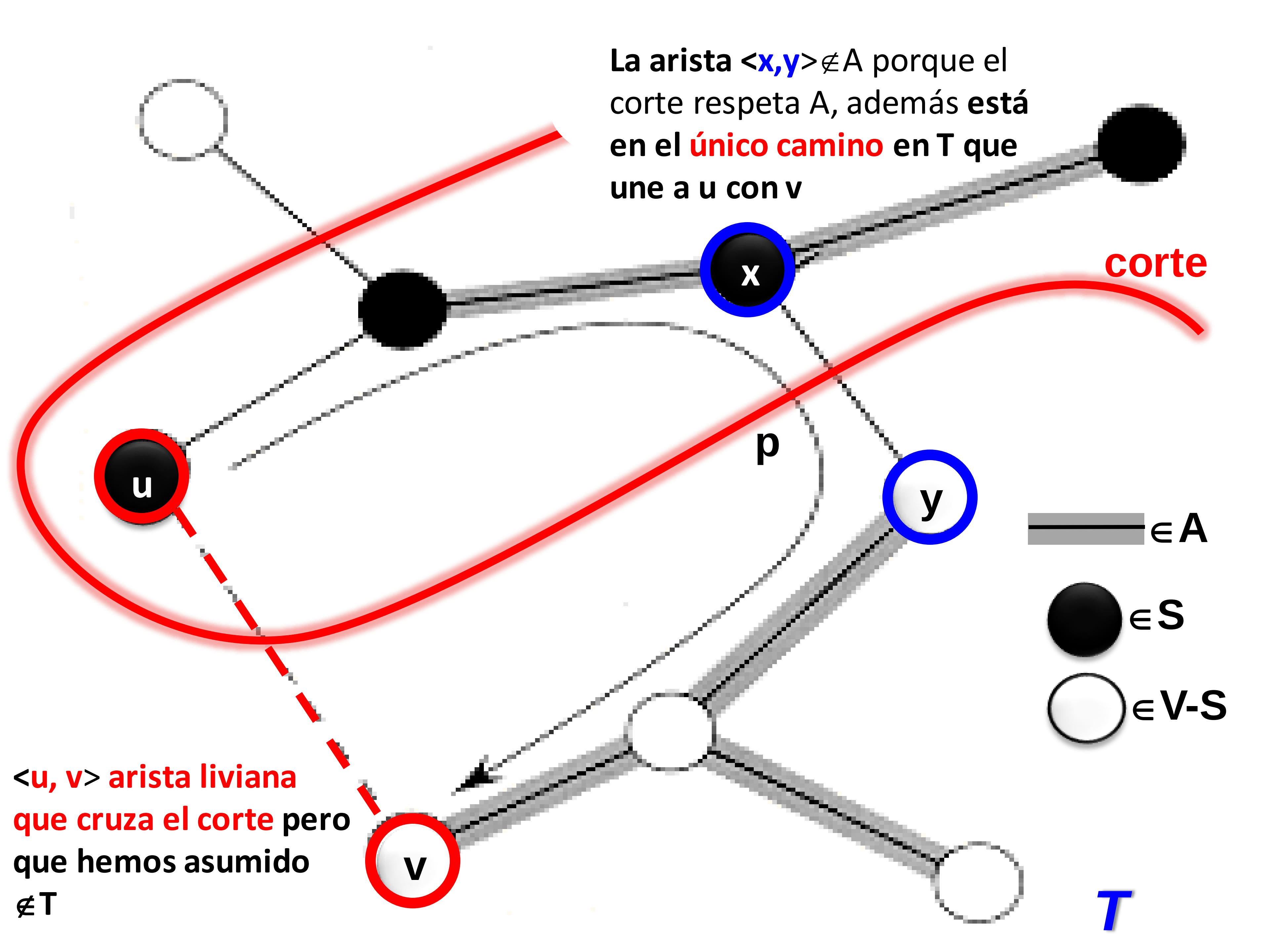
Supongamos que (u, v) no es segura para A $\Rightarrow$

NO EXISTE un *AACM* de G tal que A $\cup$ {(u, v)} $\subseteq$ de dicho árbol

Sea *T* un *AACM* de *G* tal que A $\subseteq$ T

Razonamiento: $<u,v> \notin T$ porque estaríamos contradiciendo lo supuesto

Consideremos que T tiene la estructura siguiente**:**


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
u
p
y
∈A
∈S
∈V-S
<u, v> arista liviana que cruza el corte pero que hemos asumido ∉T
v
T

Teorema 1

Intentemos construir otro *AACM* T' que incluya a A $\cup\{\langle u, v\rangle\}$ y demostrar que $\langle u,v\rangle$ es segura para A

Consideraciones:

- La arista $\langle u, v\rangle$ forma un ciclo con las aristas que están en el camino *p*, de u a v, en T

- A partir de lo anterior, y como *u* y *v* están en conjuntos diferentes del corte (S, V-S), entonces,existe, al menos, otra arista en T, sobre el camino *p*, que también cruza el corte.

- Sea $\langle x, y\rangle$ dicha arista

- $\langle x, y\rangle \notin$ A pues el corte respeta A

Teorema 1

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

Consideraciones:

1. Quitar <x, y> de T $\Rightarrow$ T se desconecta $\Rightarrow$ T deja de ser árbol
2. Poner <u, v> en T $\Rightarrow$ forma un ciclo $\Rightarrow$ T deja de ser árbol

Por tanto, la única forma de mantener un árbol es hacer 1. y 2. a la vez, o sea,

quitar <x, y> de T y poner <u, v> en T

Con ello se forma un árbol que contiene a A $\cup$ {(u, v)} , sea T'

Demostremos entonces que, además, T' es de costo mínimo

Teorema 1

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

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

como $\omega(<u, v>) \leq \omega(<x, y>)$ *pq. $<u, v>$ es liviana*

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

Consideraciones:

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

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

Teorema 1

CONCLUYENDO !!

$\boxed{A \cup \{(u, v)\} \subseteq T' \text{ y } T' \text{ es un AACM}}$

contradicción con lo supuesto

$\Rightarrow$ **<u, v> es segura para A**

El Teorema 1 y el algoritmo genérico

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

- Su funcionamiento garantiza que ***A* es siempre acíclico**, pues, de otra forma, el *AACM* que contiene a *A* tendría un *ciclo* → contradicción pues es un árbol

- **En cualquier momento de la ejecución del algoritmo genérico, el grafo G_A = *(V, A)* constituye un bosque** cuyos árboles serán las componentes conexas de G_A

- **Algunos de estos árboles pueden tener un solo nodo**

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 A $\cup$\{<u, v>\} tiene que ser acíclico

- El ciclo **while** (lines 2–4) del **algoritmo genérico** se ejecuta |V|-1 veces, porque **en cada iteración se determina una de las |V|-1aristas de un AACM**

- Incialmente, cuando A=ϕ , en G_A hay |V| árboles y en cada iteración, se reduce en 1 el número de estos

- Cuando el bosque contiene solo un árbol, concluye la ejecución del algoritmo genérico

La funcionalidad de los algoritmos de Kruskal y PRIM se basa en el siguiente Corolario

Corolario 2

Colorario 2

-Sea $G=(V, E)$ conexo, no dirigido y ponderado
-Sea $A \subseteq E$ incluido en algún *AACM* de G
-Sea $\boldsymbol{C=(Vc, Ec)}$ **comp. conexa** (árbol) en el bosque

$$\boldsymbol{G_A = (V, A)}$$

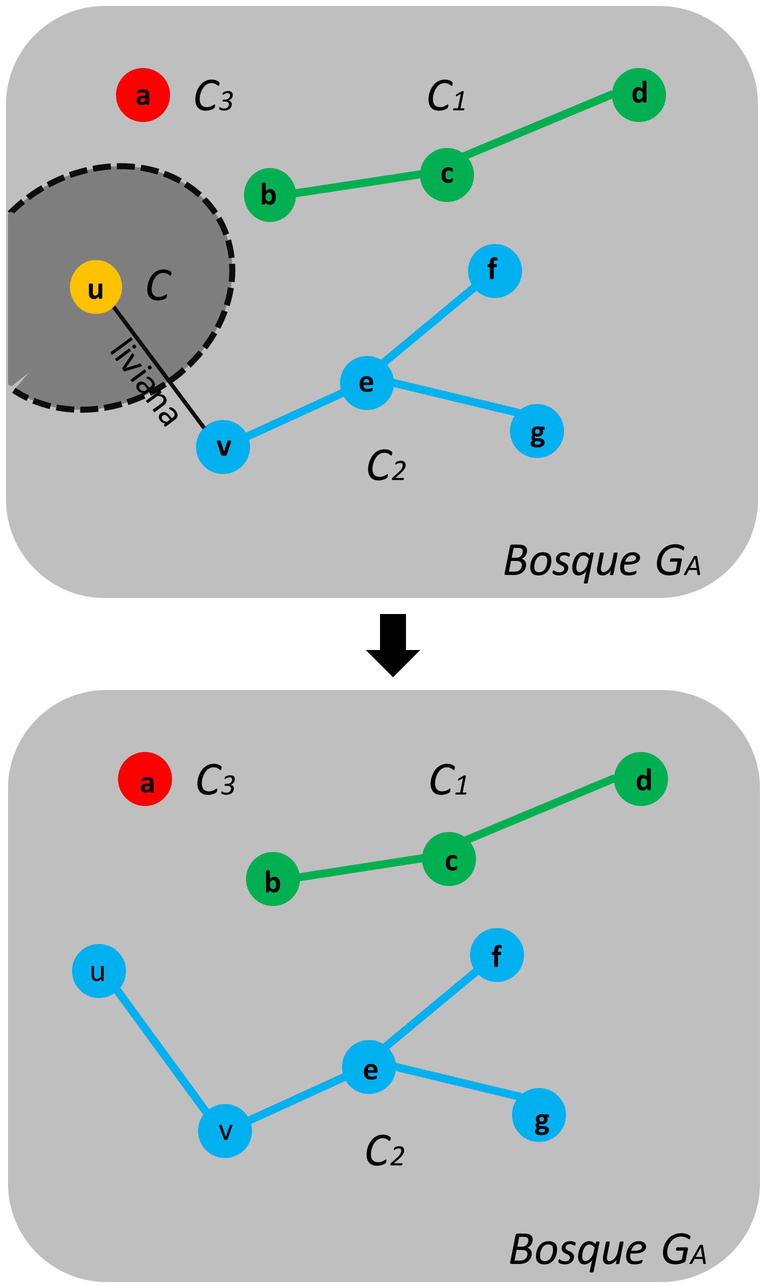
-Si <u, v> es una **arista liviana** que conecta a C con cualquier otra componente en G_A, entonces

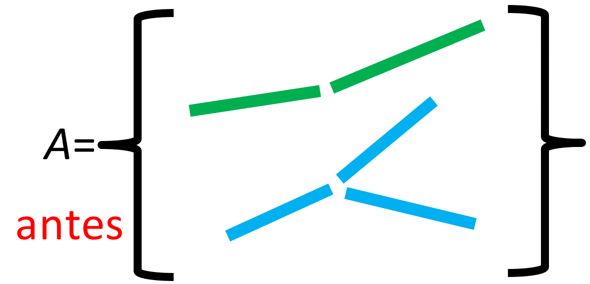
<u, v> **segura** para A

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

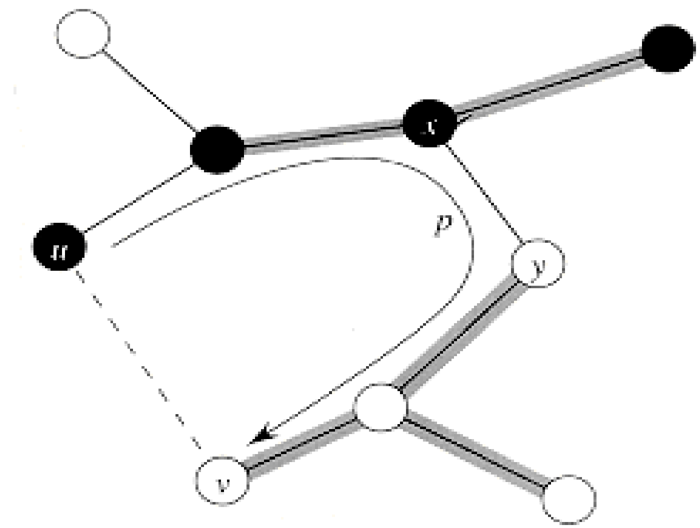
$V=\{a, b, c, d, e\ f, g, u, v\}$

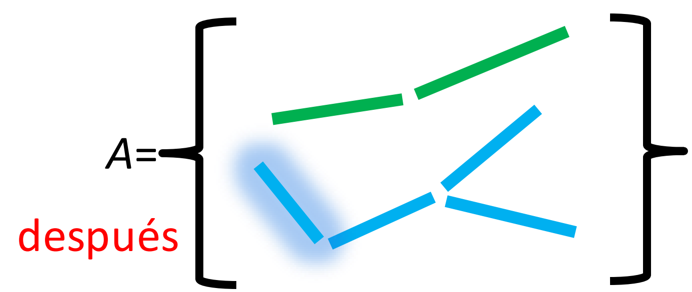
El corte $(V_C, V- V_C)$ respeta A





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





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

Corolario 2

Demostración

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

<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, forman **un solo árbol** dentro de G_A(los restantes árboles de G_A son árboles de un solo vértice)

EN AMBOS CASOS: La arista que se añade a *A* es siempre **la de menor costo (liviana)** entre todas las aristas **que conectan dos árboles (componentes) distintos de** G_A

Algoritmo de Kruskal

Estrategia de funcionamiento: Encuentra la **arista segura** para añadir al árbol creciente, seleccionando, **entre todas las aristas que enlazan árboles distintos** en el bosque G_A, la arista <u, v> **de menor peso** (**liviana**)

Sean C_1 y C_2 dos árboles de G_A. Sea u∈C_1, v∈C_2 y <u, v>∈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,** por el **Corolario 2**, es **segura para A**

Estrategia *glotona* para Kruskal:

En cada paso se añade a *A* la arista de menor peso posible

Algoritmo de Kruskal

La implementación de Kruskal que se ofrece utiliza la estructura de datos: **Conjuntos Disjuntos (estudiada en EDA I)** para representar los árboles del bosque G_A

- Cada conjunto representa los vértices que pertenecen al mismo árbol en el bosque

- **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, de 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 no
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|)
O(|E|log|E|)
O(|V|log|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

$|E| < |V|^2$

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

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

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**

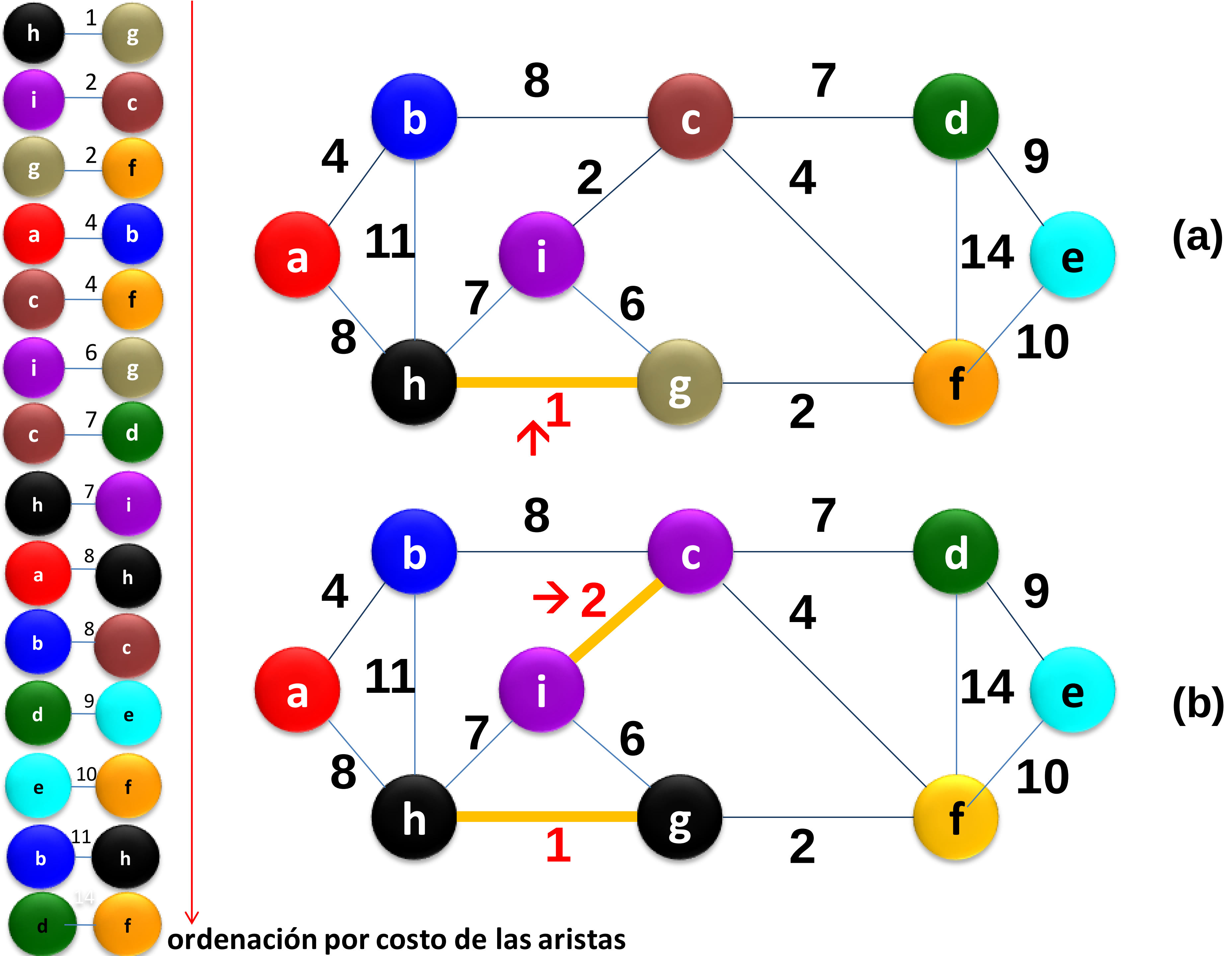
Entonces $|E|$ es $O(|V|^2)$, entonces el costo quedaría

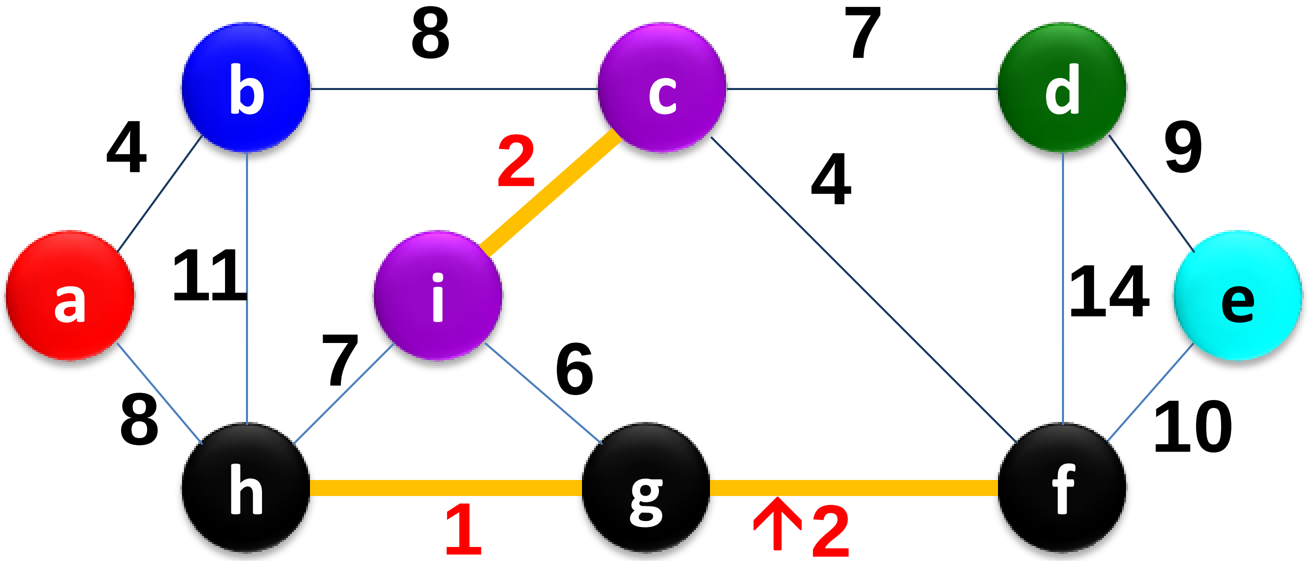
O(|V|² log |V|)

Algoritmo de Kruskal

Observación:

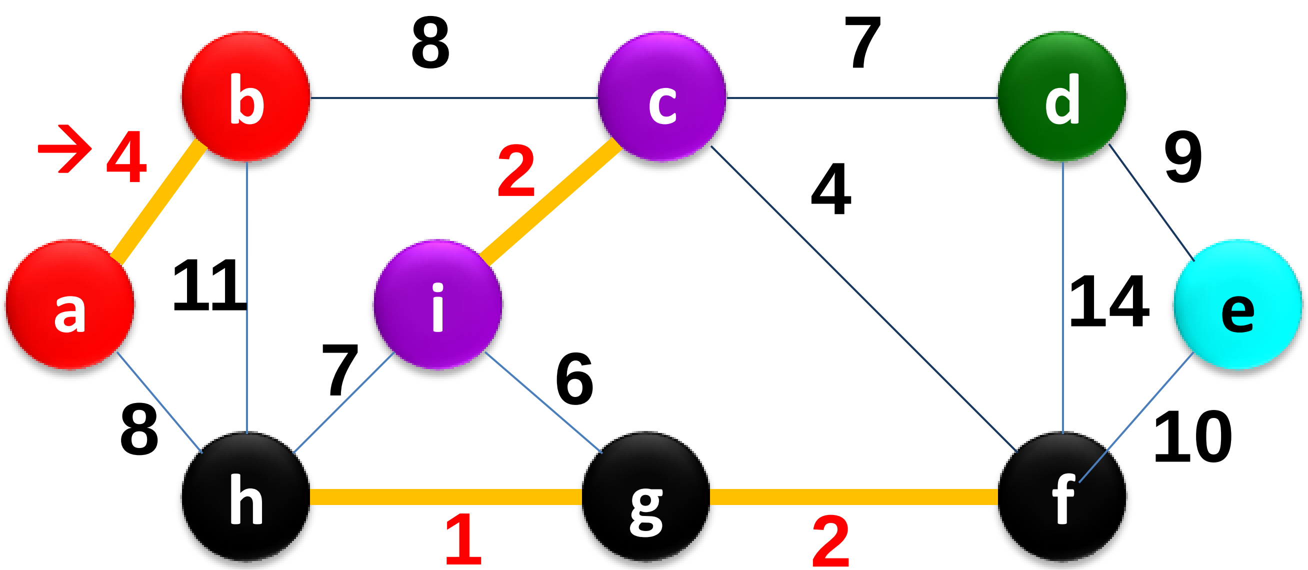
Si G **no fuera conexo, puede aplicarse Kruskal a cada componenete conexa**. Para cada una se obtendrá un *AACM* y finalmente, para G lo que se obtiene un bosque formado por los *AACM* obtenidos para cada una de sus componentes conexas


h 1 g
i 2 c
g 2 f
a 4 b
c 4 f
i 6 g
c 7 d
h 7 i
a 8 h
b 8 c
d 9 e
e 10 f
b 11 h
d 14 f
ordenación por costo de las aristas
a b c d e f g h i
4 8 7 9 2 4 11 14 7 6 8 10 1 2
(a)
a b c d e f g h i
4 8 7 9 2 4 11 14 7 6 8 10 1 2
(b)



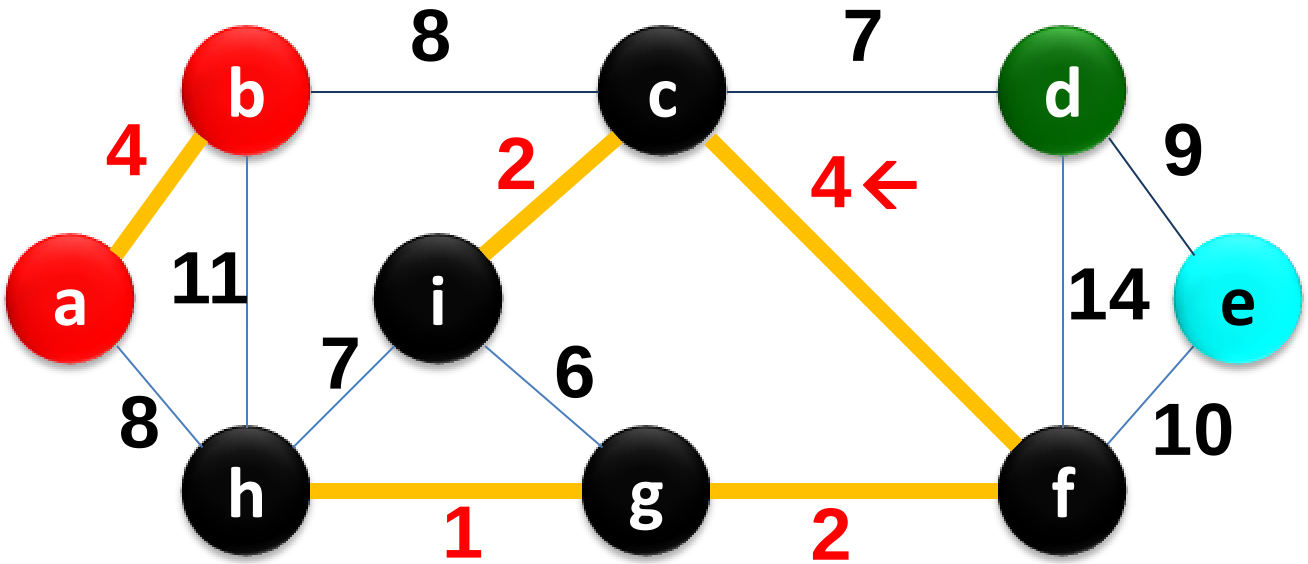
b
8
c
7
d
4
2
4
9
a
11
i
14
e
7
6
8
10
h
g
f
1
↑2

(c)



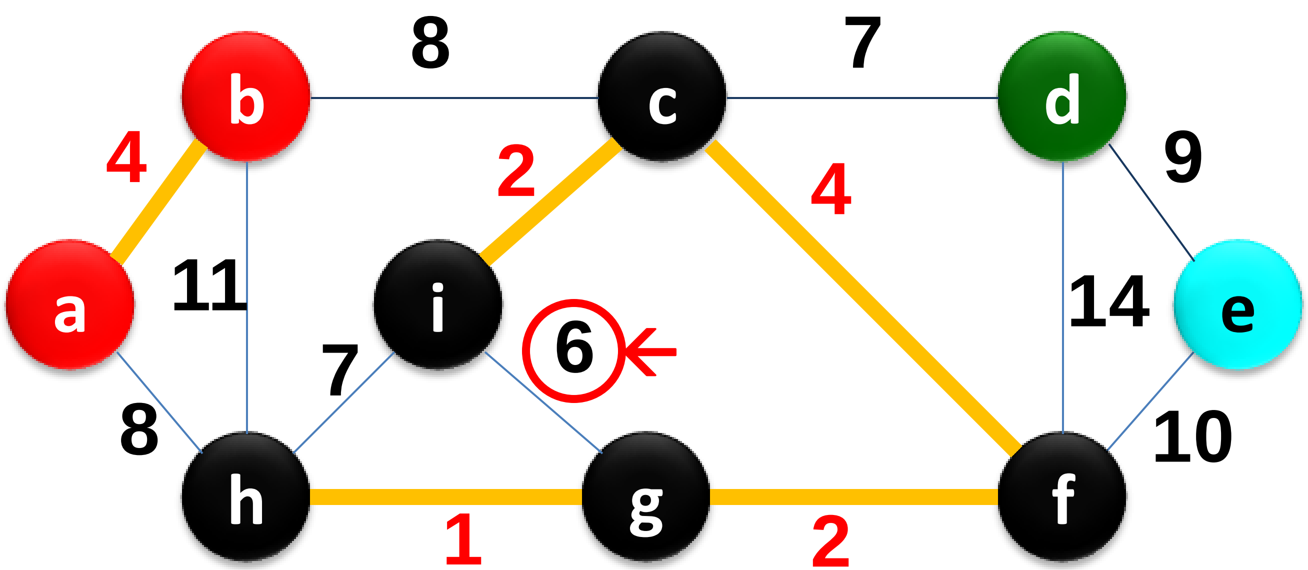
b
8
c
7
d
→ 4
2
4
9
a
11
i
14
e
7
6
8
10
h
g
f
1
2

(d)



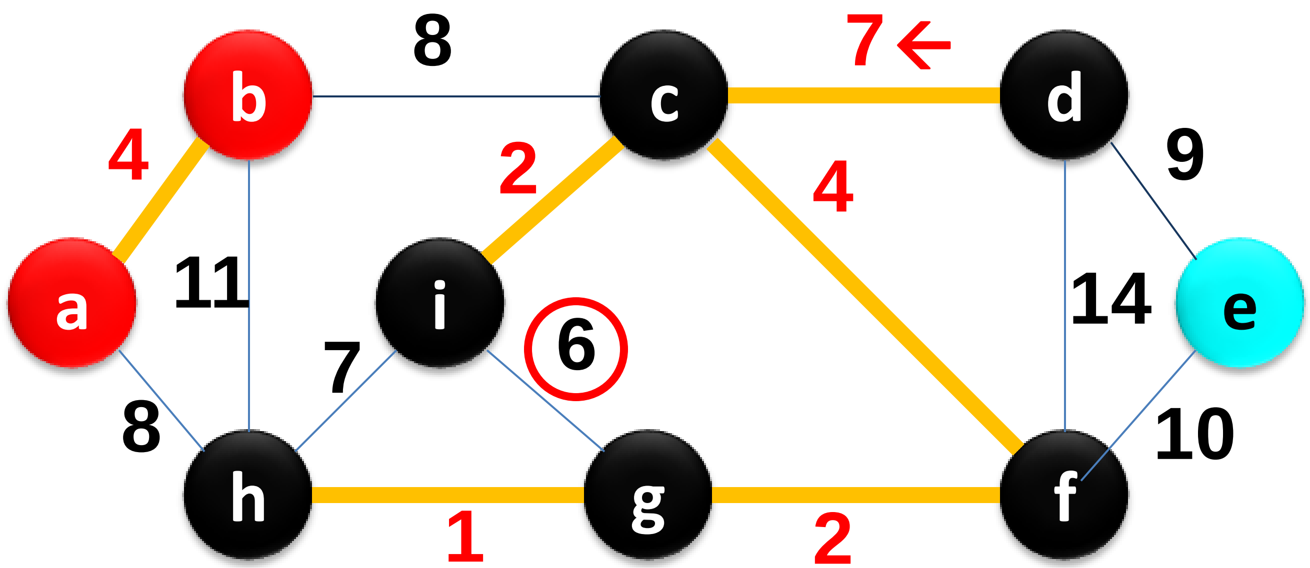
a
b
c
d
e
f
g
h
i
4
8
7
2
4
9
11
14
7
6
8
10
1
2

(e)



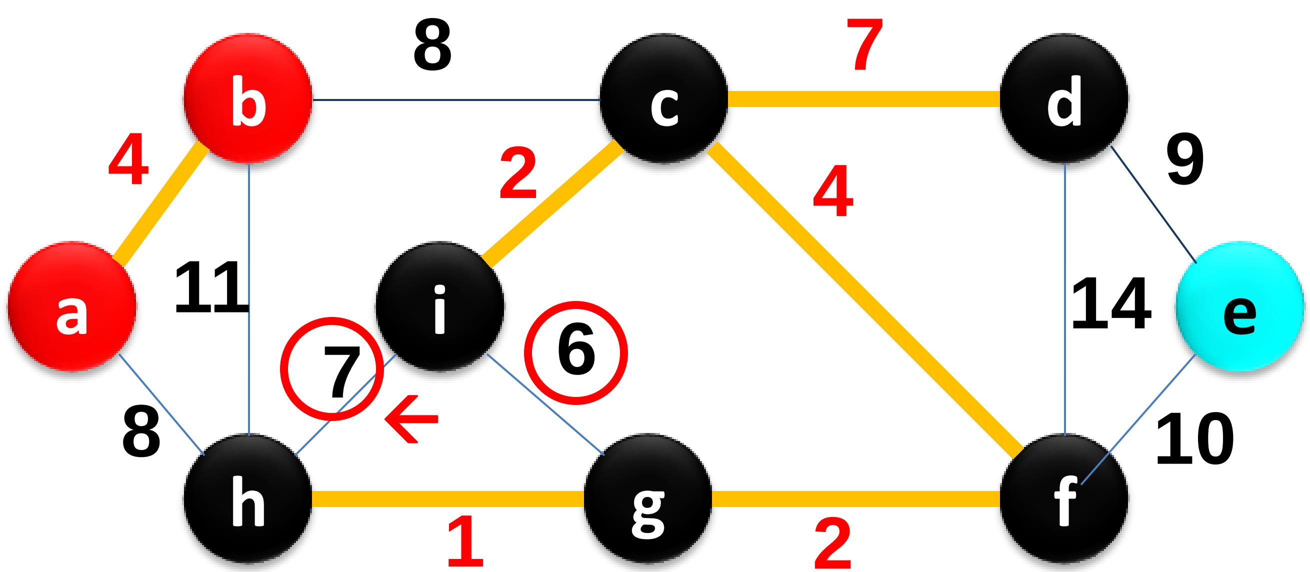
a
b
c
d
e
f
g
h
i
4
8
7
2
4
9
11
14
7
6
8
10
1
2

(f)



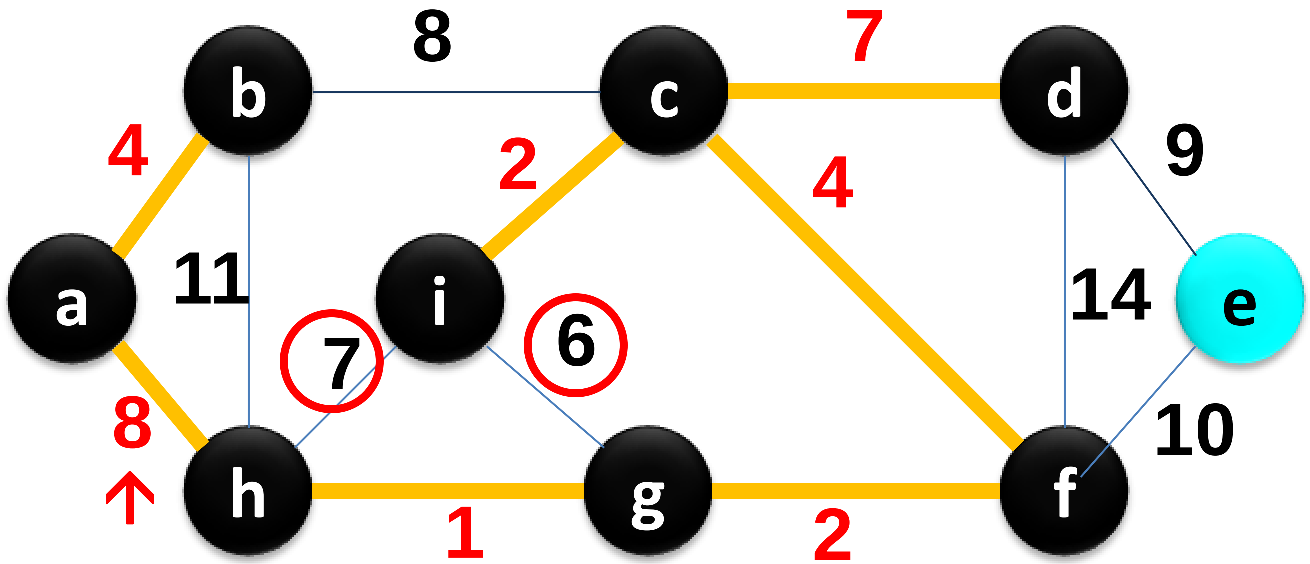
8
7
b
c
d
4
2
4
9
a
11
i
6
14
e
7
8
10
h
g
f
1
2

(g)



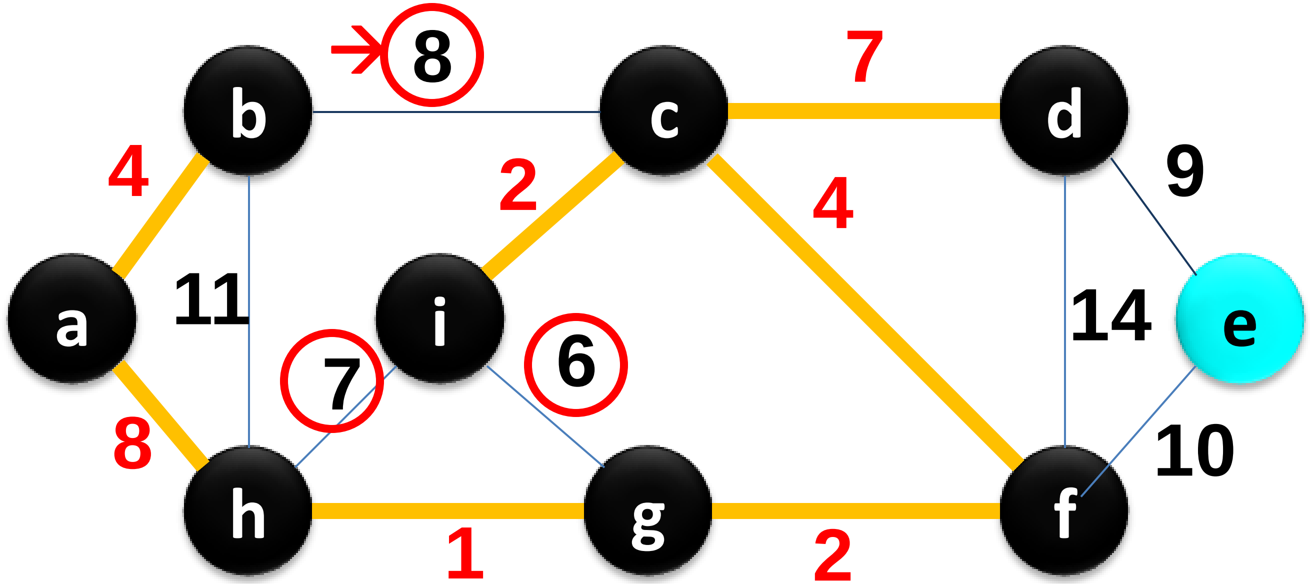
8
7
b
c
d
4
2
4
9
a
11
i
6
14
e
7
8
10
h
g
f
1
2

(h)



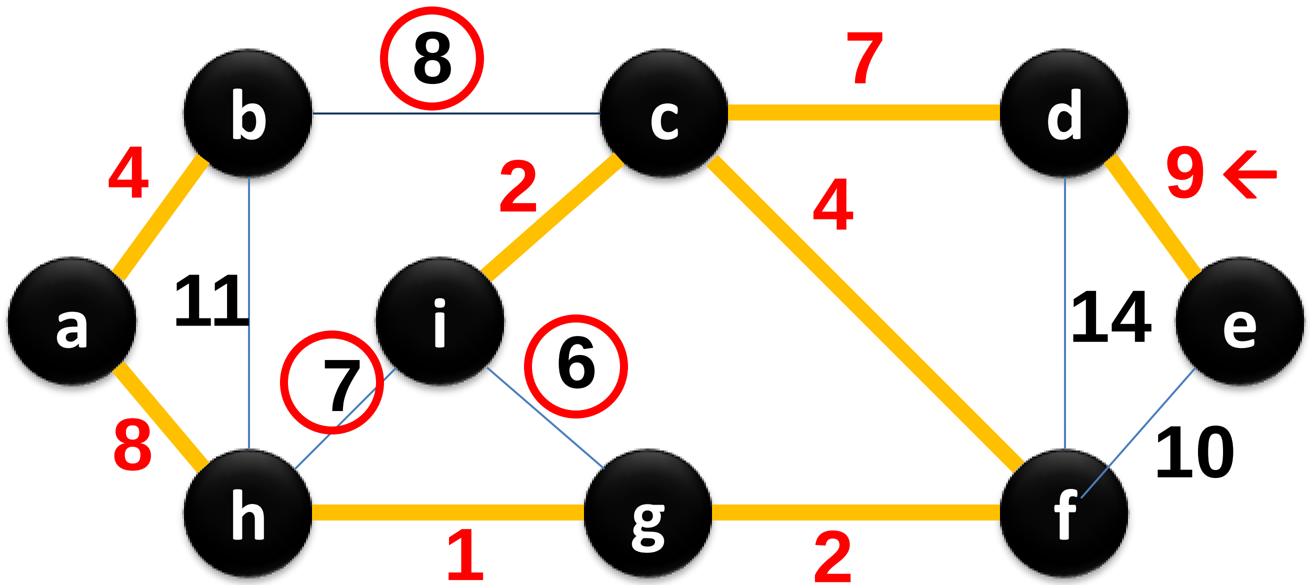
a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
9
14
10
7
6
1
2

(i)



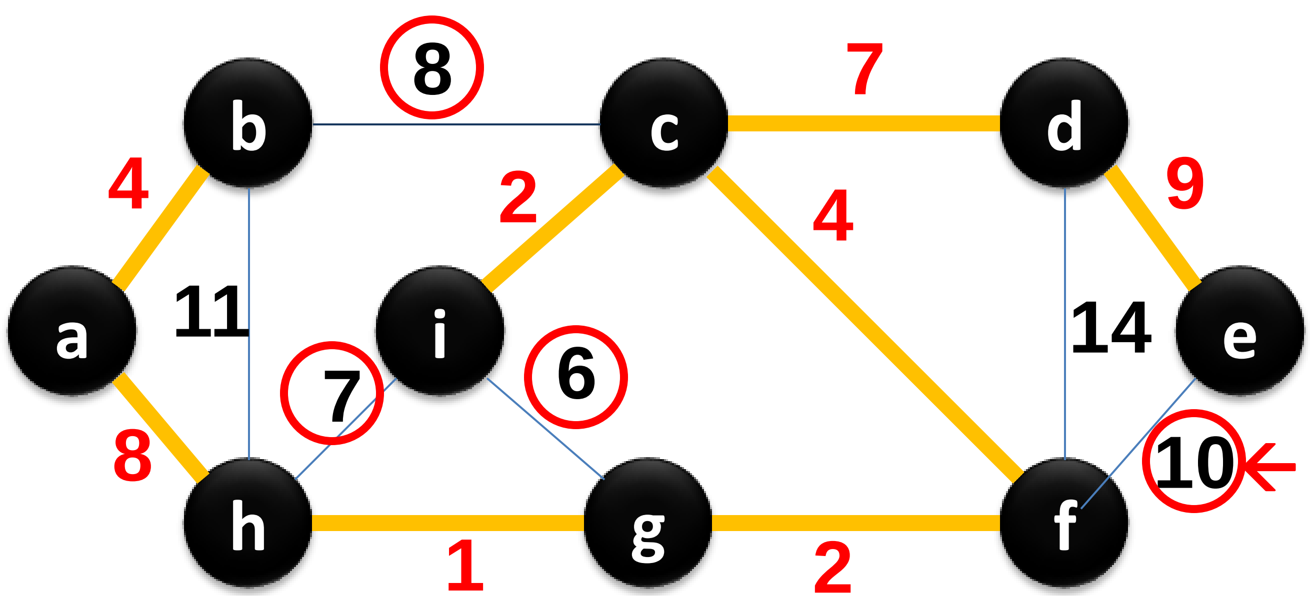
a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
9
14
10
7
6
1
2

(j)



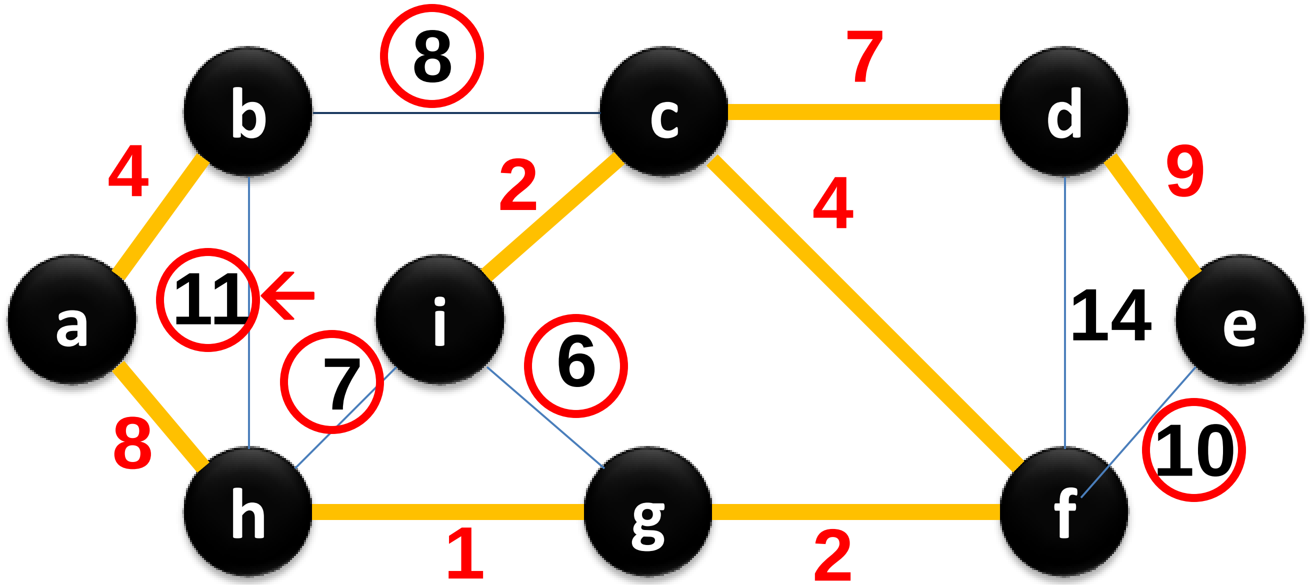
a
b
c
d
e
f
g
h
i
4
8
11
8
7
1
2
6
2
4
7
9
14
10

(k)



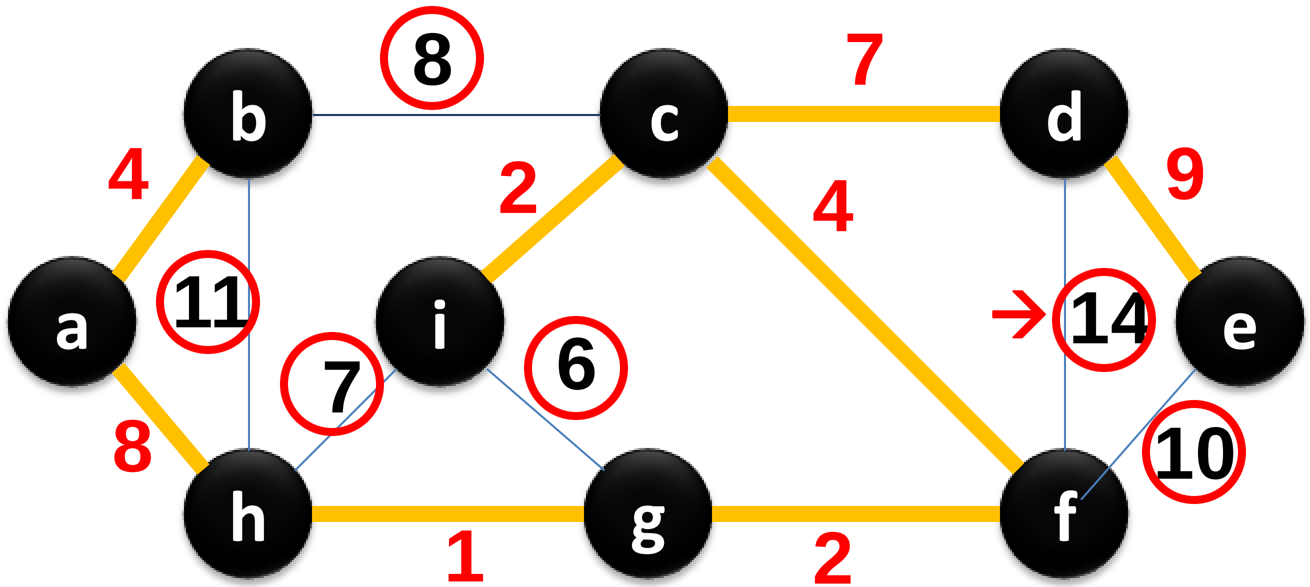
a
b
c
d
e
f
g
h
i
4
8
11
8
7
1
2
6
2
4
7
9
14
10

(l)



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

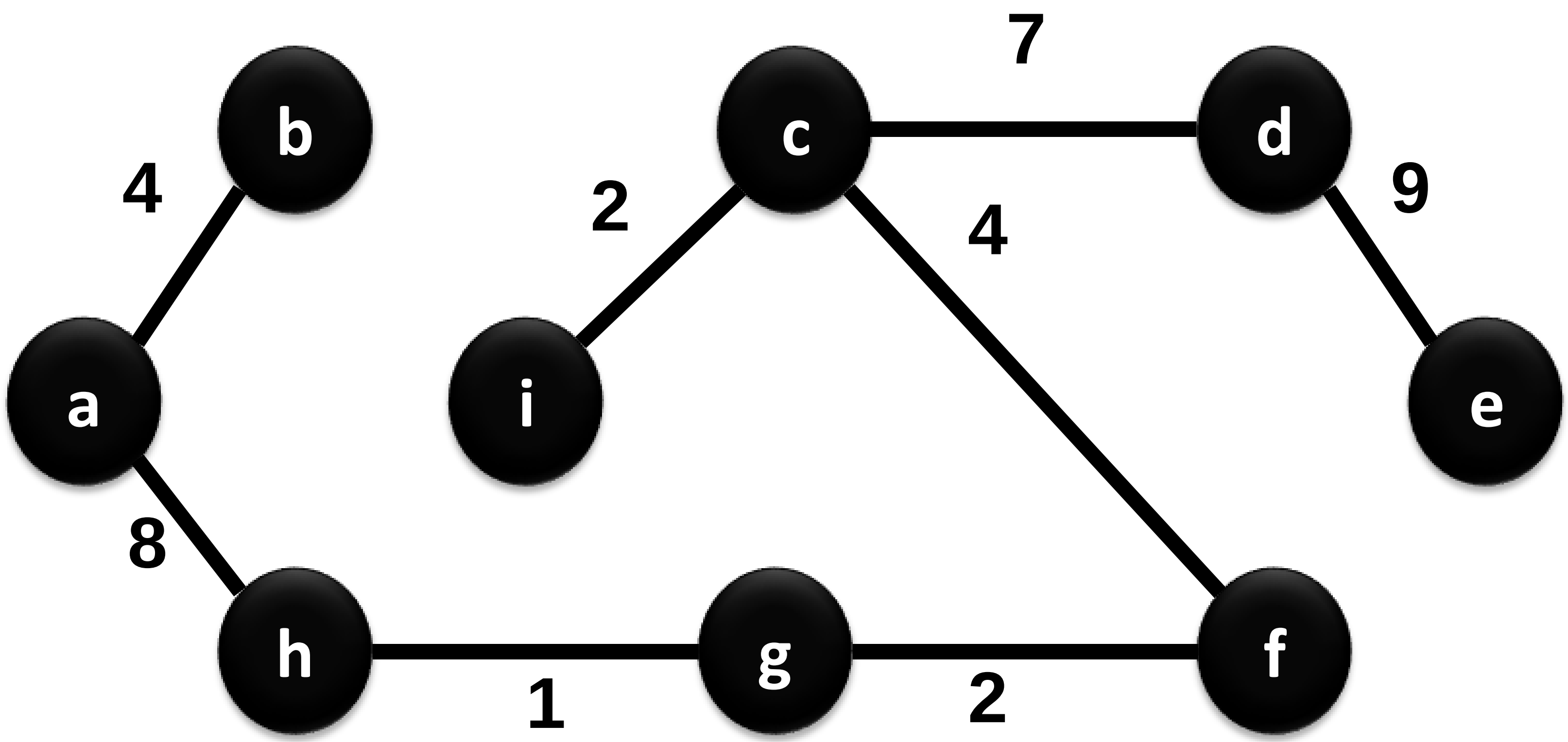
(m)



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

(n)

Resultado de aplicar el Algoritmo de Kruskal



AACM

Algoritmo de PRIM

- **PRIM** es también un caso especial del algoritmo general
- Se caracteriza porque **las aristas en A forman un único árbol dentro del bosque** $\boldsymbol{G_A}$
- Inicialmente, el árbol solo posee una raíz arbitraria *r* y crece hasta que contiene a todos los vértices de V

Estrategia de funcionamiento:

En cada paso, una **arista liviana,** que conecta a A con V-A, se añade al árbol, sabiendo que, por el **Teorema 1**, esta regla siempre añade **aristas seguras** para A, por tanto, cuando el algoritmo termina, las aristas en A forman un ***AACM***

Estrategia glotona para Kruskal:
El árbol siempre se incrementa con una arista que le aporta el menor costo o peso posible

Algoritmo de PRIM

Detalles de implementación:

-Para implementar PRIM eficientemente es necesario hacer fácil la selección de la nueva arista que formará parte de A

-El vértice raíz *r* se da como entrada

- $\forall v \in V$: $v \notin$ al árbol A;

d[v] (su ***distancia* de A**)
$$\begin{cases} \mathit{mín}(w(<v, a>: a \in A) & \text{si } \exists <v, a> \\ \infty & \text{si no existe } <v, a> \end{cases}$$

- Los vértices que aun no forman parte del árbol A se mantienen en una **cola con prioridad** de acuerdo con su valor de *distancia* de A

- π[v] (vértice cercano de v): es el otro extremo (que no es v) de la arista que marca la distancia de v con A

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>
```

Esto se puede hacer con una array de bool para que sea menos costoso

inicialmente:

V-Q = ϕ / 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

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

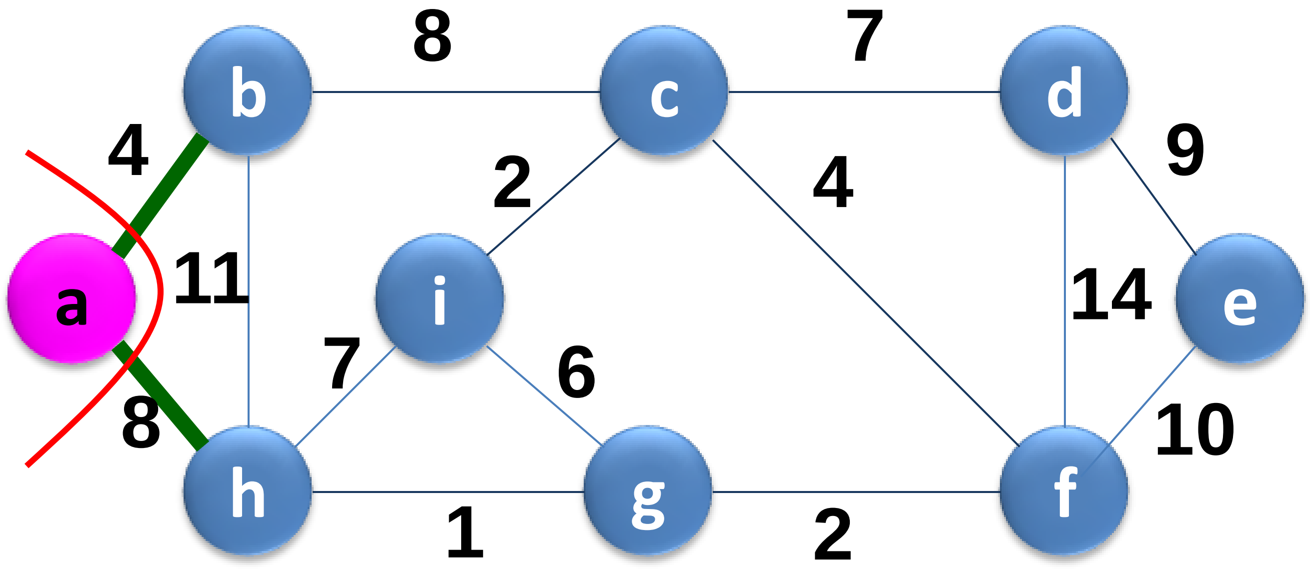
Inicialmente:

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

aristas en A:

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

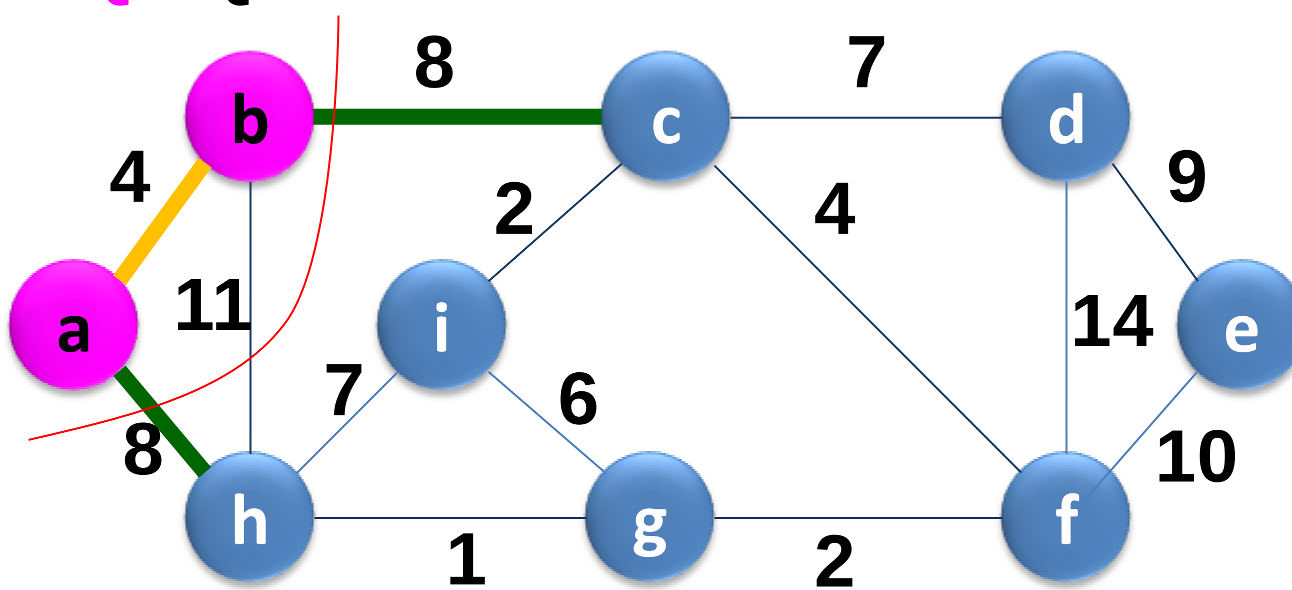
d[a]=0 π[a]=-
d[b]=4 π[b]=a
d[c] =∝ π[c] = null
d[d] =∝ π[d] = null
d[e] =∝ π[e] = null
d[f] =∝ π[f] = null
d[g] =∝ π[g] = null
d[h] =8 π[h] = a
d[i] =∝ π[i] = null



V-Q / Q

(a)

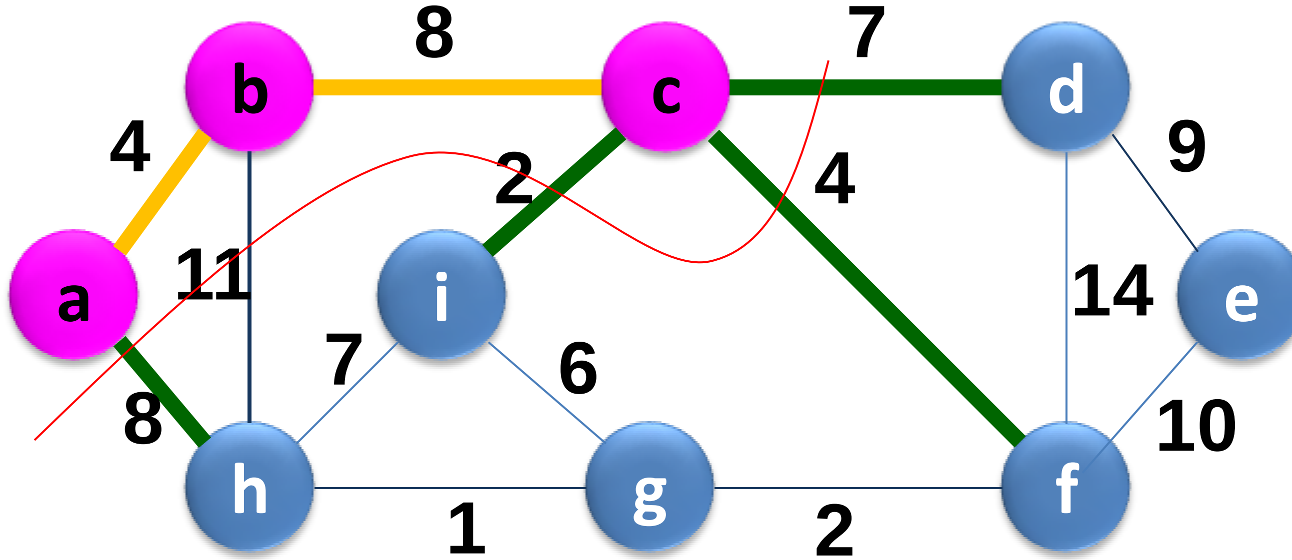
d[a]=0 π[a]=-
d[b]=4 π[b]=a
d[c] =8 π[c] = b
d[d] =∝ π[d] = null
d[e] =∝ π[e] = null
d[f] =∝ π[f] = null
d[g] =∝ π[g] = null
d[h] =8 π[h] = a
d[i] =∝ π[i] = null



V-Q / Q

(b)

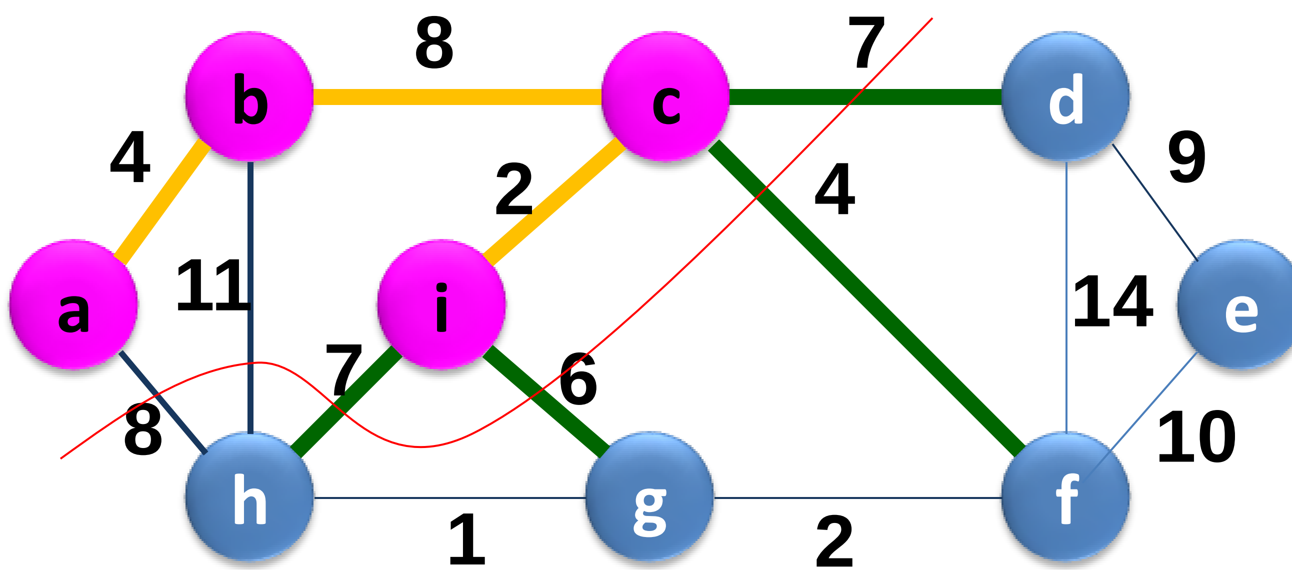
d[a]=0 π[a]=-
d[b]=4 π[b]=a
d[c] =8 π[c] = b
d[d] =7 π[d] = c
d[e] =∝ π[e] = null
d[f] =4 π[f] = c
d[g] =∝ π[g] = null
d[h] =8 π[h] = a
d[i] =2 π[i] = c



V-Q / Q

(c)

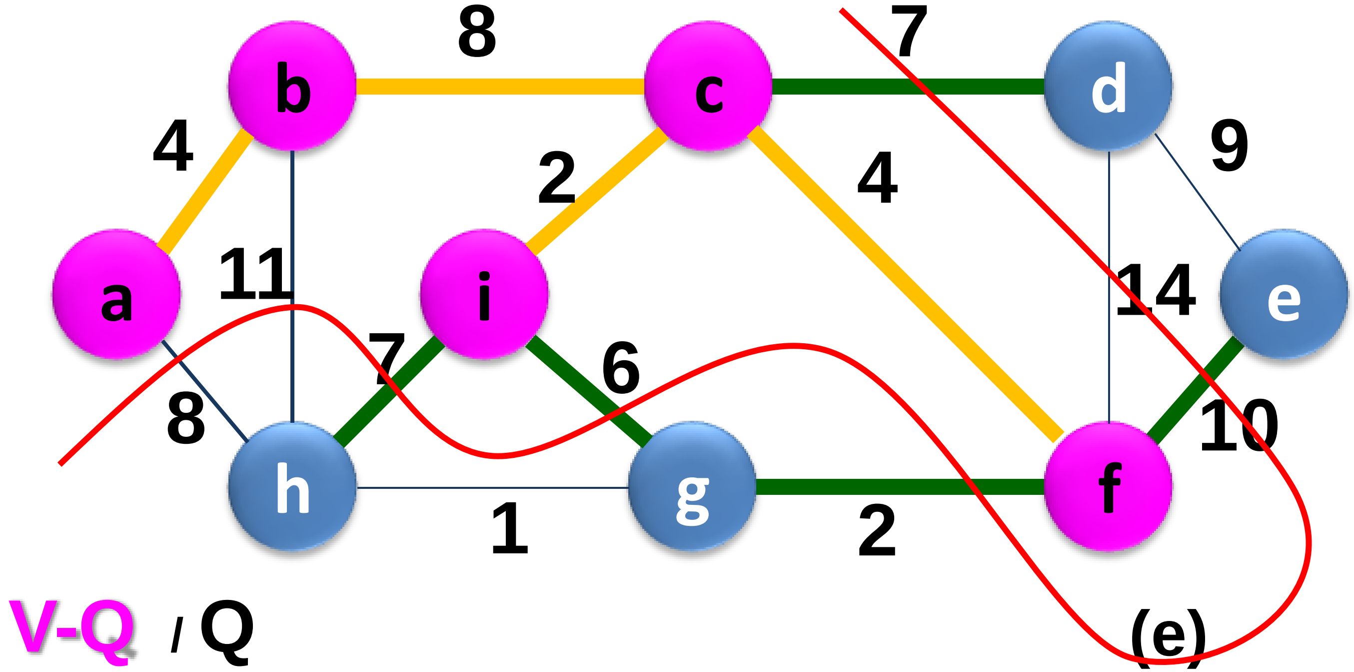
d[a]=0 π[a]=-
d[b]=4 π[b]=a
d[c] =8 π[c] = b
d[d] =7 π[d] = c
d[e] =∝ π[e] = null
d[f] =4 π[f] = c
d[g] =6 π[g] = I
d[h] =7 π[h] = i
d[i] =2 π[i] = c



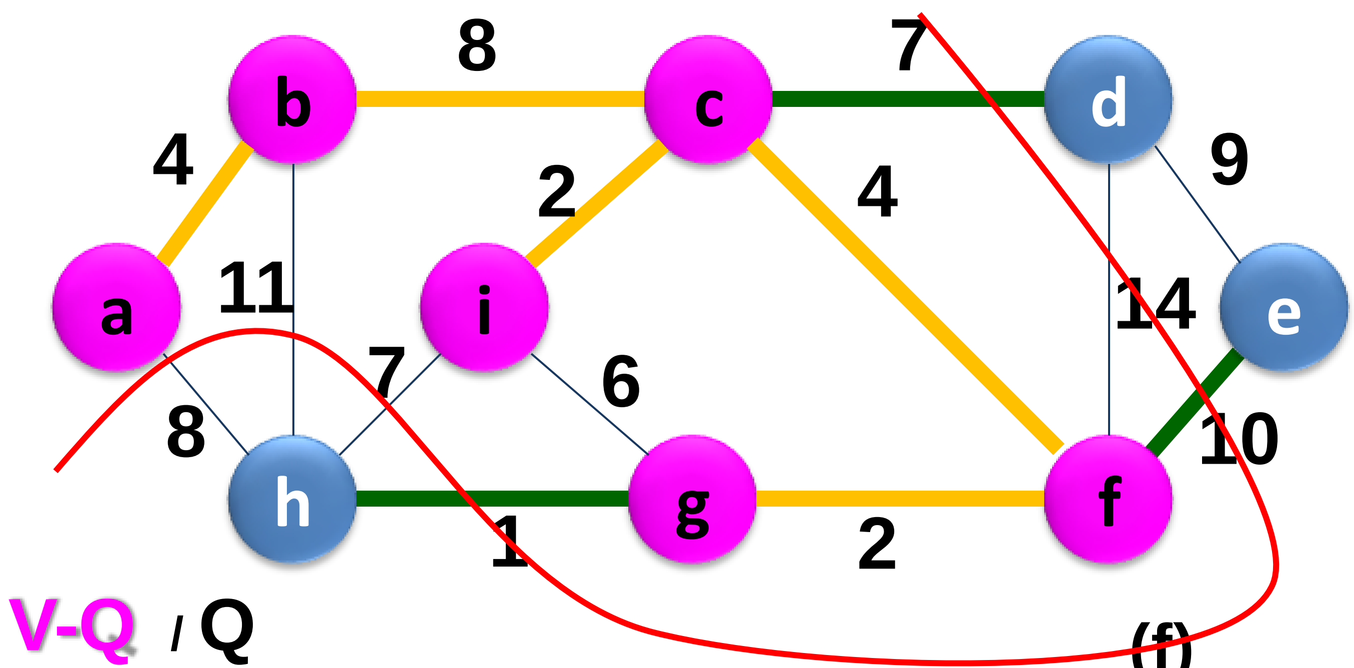
V-Q / Q

(d)

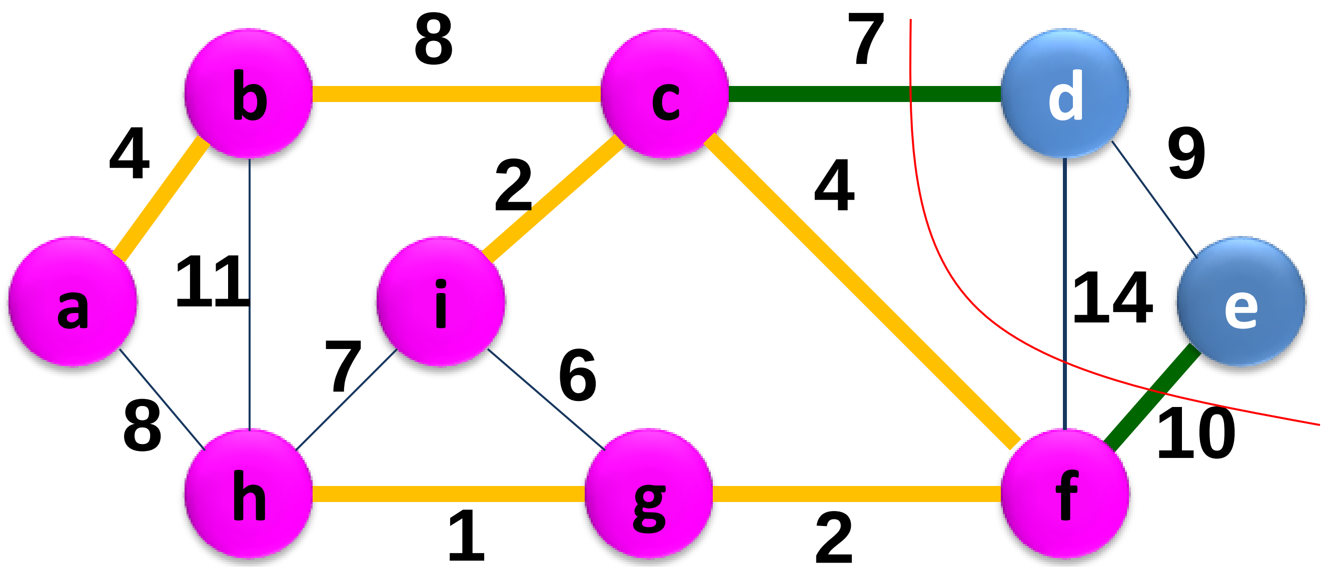
d[a]=0 π[a]=-
d[b]=4 π[b]=a
d[c] =8 π[c] = b
d[d] =7 π[d] = c
d[e] =10 π[e] = f
d[f] =4 π[f] = c
d[g] =2 π[g] = f
d[h] =7 π[h] = i
d[i] =2 π[i] = c



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



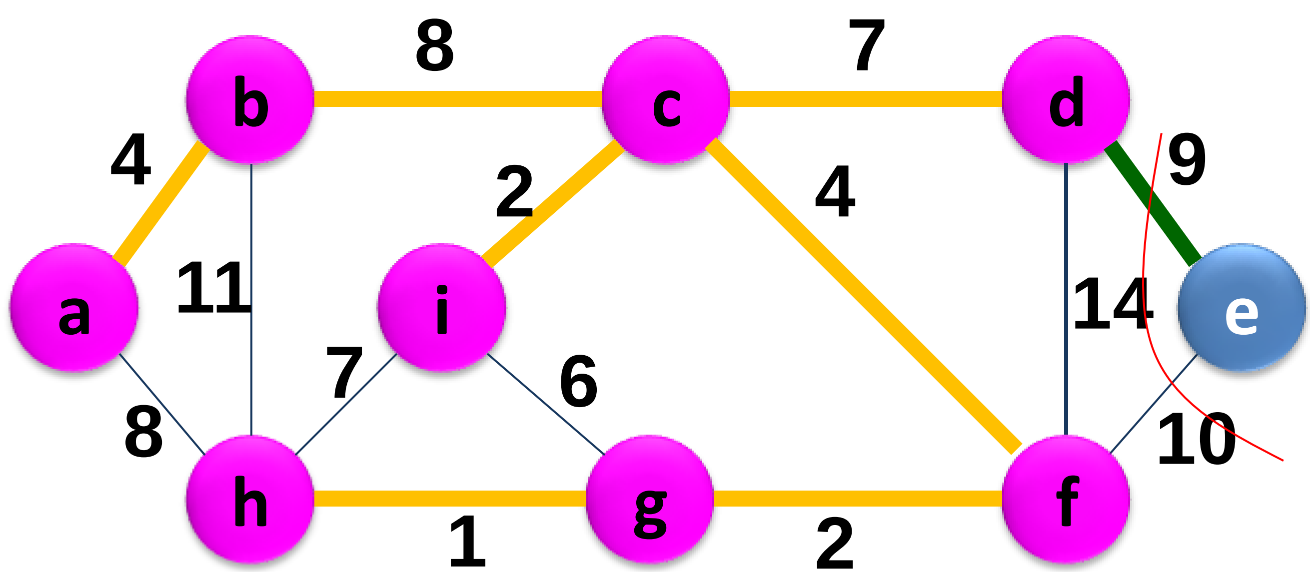
d[a]=0 π[a]=-
d[b]=4 π[b]=a
d[c] =8 π[c] = b
d[d] =7 π[d] = c
d[e] =10 π[e] = f
d[f] =4 π[f] = c
d[g] =2 π[g] = f
d[h] =1 π[h] = g
d[i] =2 π[i] = c



V-Q / Q

(f)

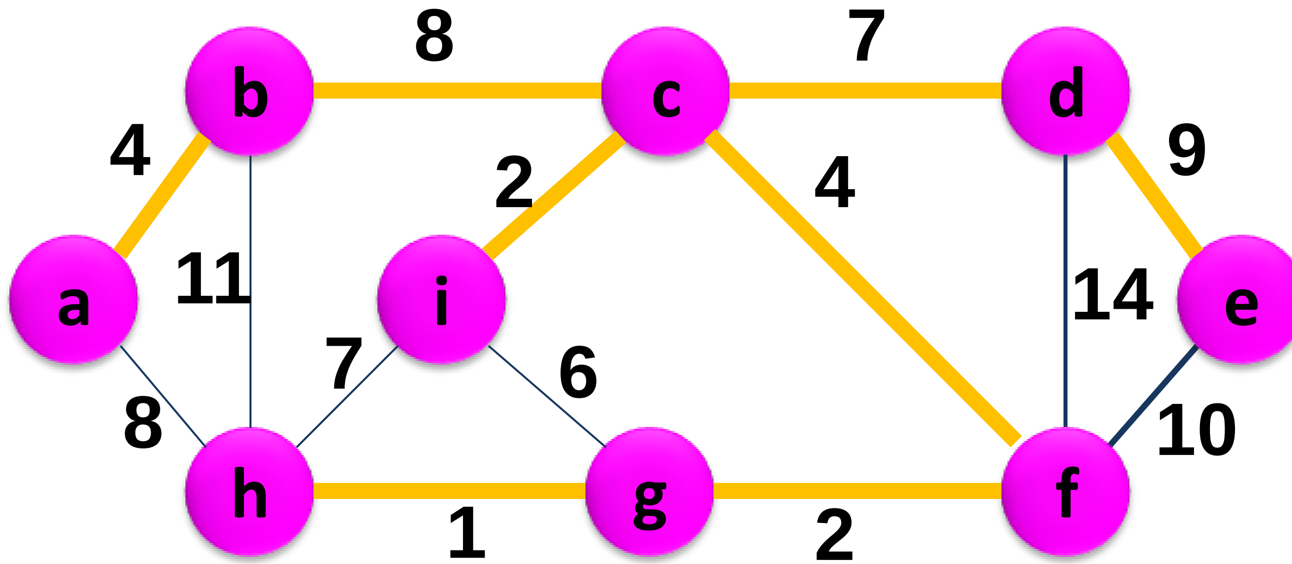
d[a]=0 π[a]=-
d[b]=4 π[b]=a
d[c] =8 π[c] = b
d[d] =7 π[d] = c
d[e] =9 π[e] = d
d[f] =4 π[f] = c
d[g] =2 π[g] = f
d[h] =1 π[h] = g
d[i] =2 π[i] = c



V-Q / Q

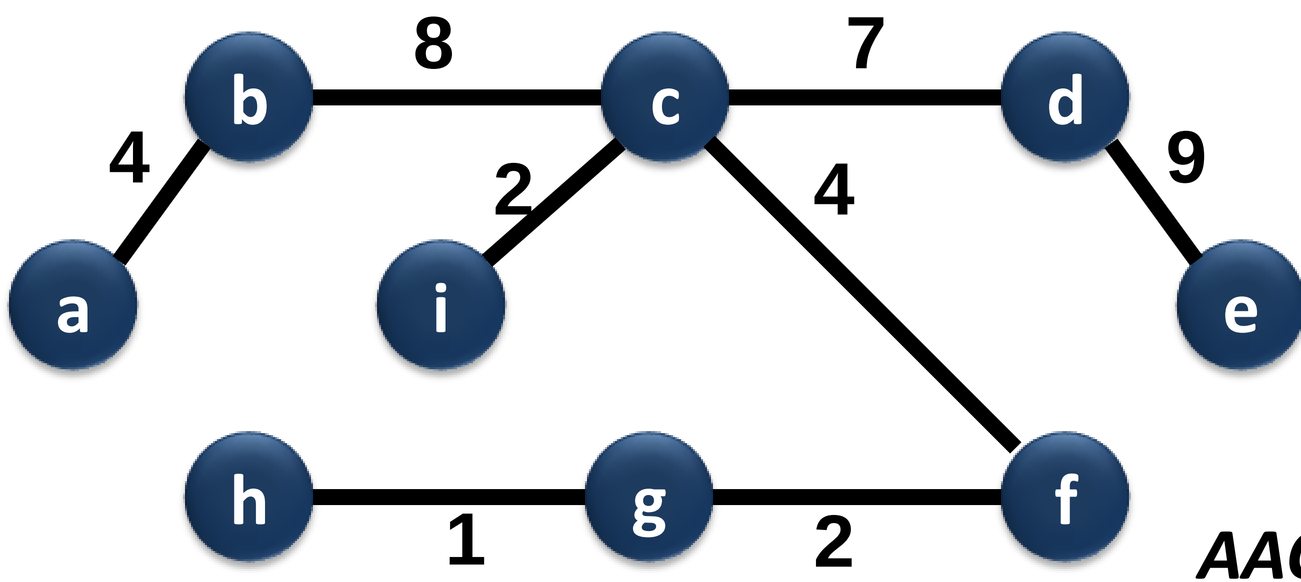
(g)

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(|V|)*

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

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

arreglo booleano → para cada vértice diga si pertenece o no a Q
actualizar arreglo, 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]
6 while Q no esté vacía
7       do u ← EXTRAE_MIN(Q)
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(IVI)

Línea 5: ← Build_Heap: O(IVI)

Línea 6: ← O(IVI)

Línea 7: ← O(log IVI)

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 IVI)

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

Preguntar por la pertenencia a Q puede implementarse en O(1) teniendo un arreglo 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 IVI)

O(IVI log IVI + |E| log IVI)

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

En tal caso, es recomendable usar una versión menos eficiente del algoritmo cuya complejidad temporal, para el caso peor es $O(|V|^2)$

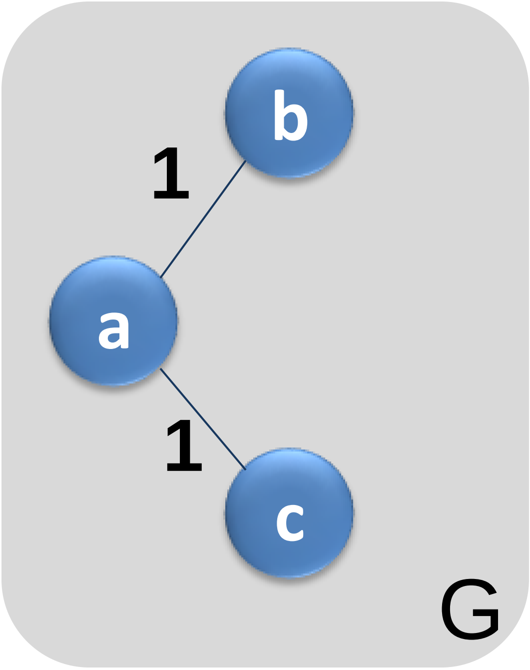
En general, cuando se establecen restricciones para el Grafo se pueden usar distintas implementaciones de la Cola con Prioridad para mejorar la complejidad temporal del algoritmo de PRIM

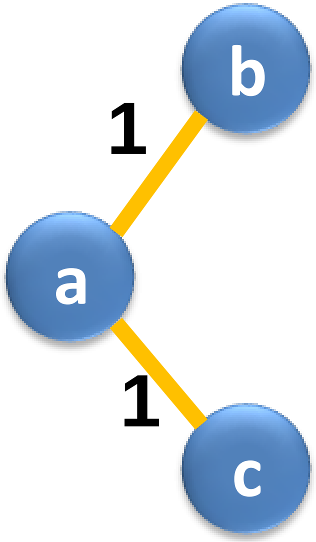
Algoritmo de PRIM

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