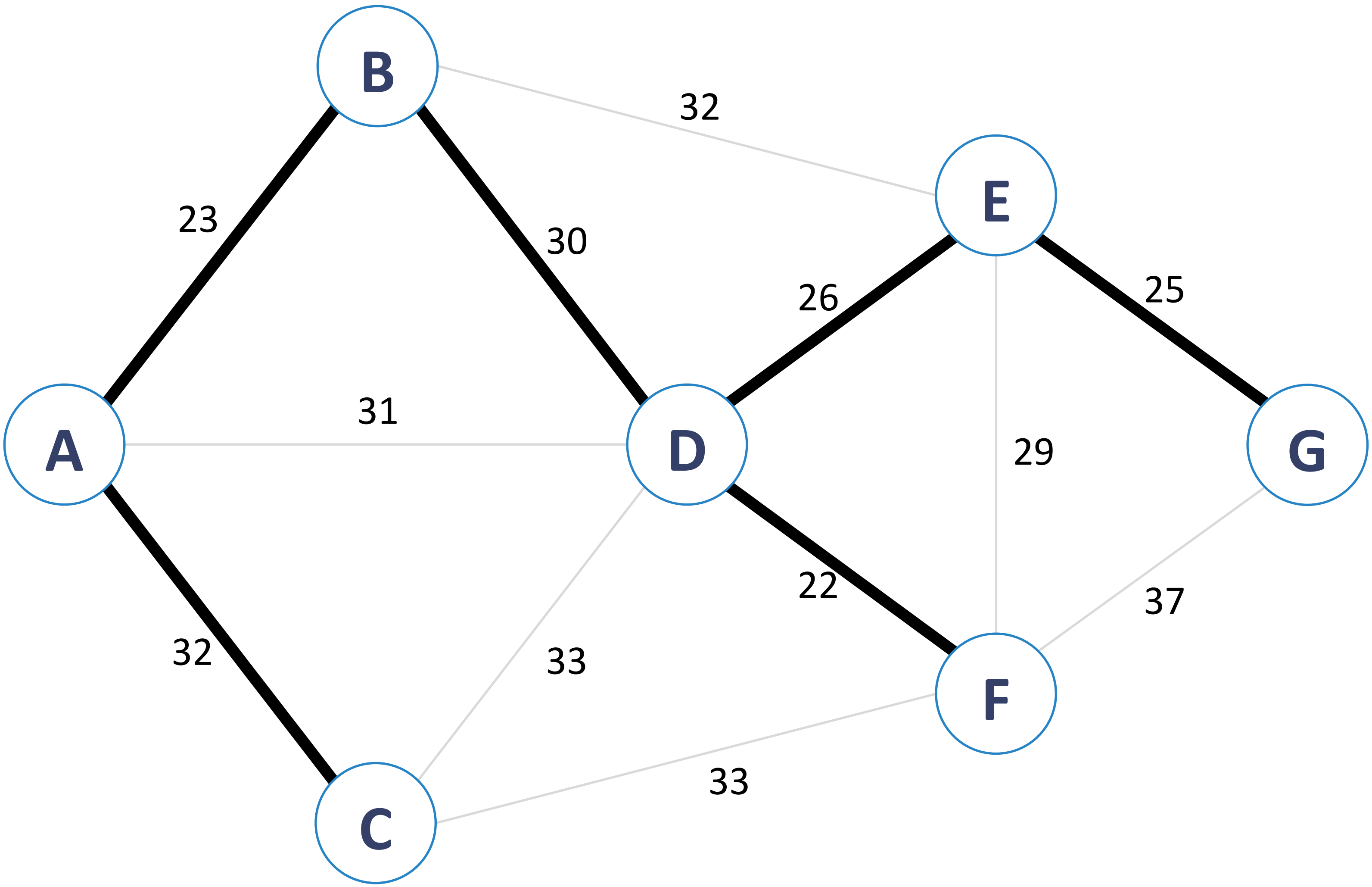
Propiedades del MST

¿Hay alguna arista que **siempre** pertenezca a un **MST**?

¿Se cumple esto recursivamente? ¿En qué casos?

¿Podremos aprovecharlo en un algoritmo **codicioso**?

Ejemplo de un MST para un grafo



El algoritmo de Kruskal

$\boldsymbol{kruskal}\big(\boldsymbol{G}(\boldsymbol{V},\boldsymbol{E})\big)$:

Ordenar $\boldsymbol{E}$ por costo, de menor a mayor

$\boldsymbol{T} \leftarrow \emptyset$

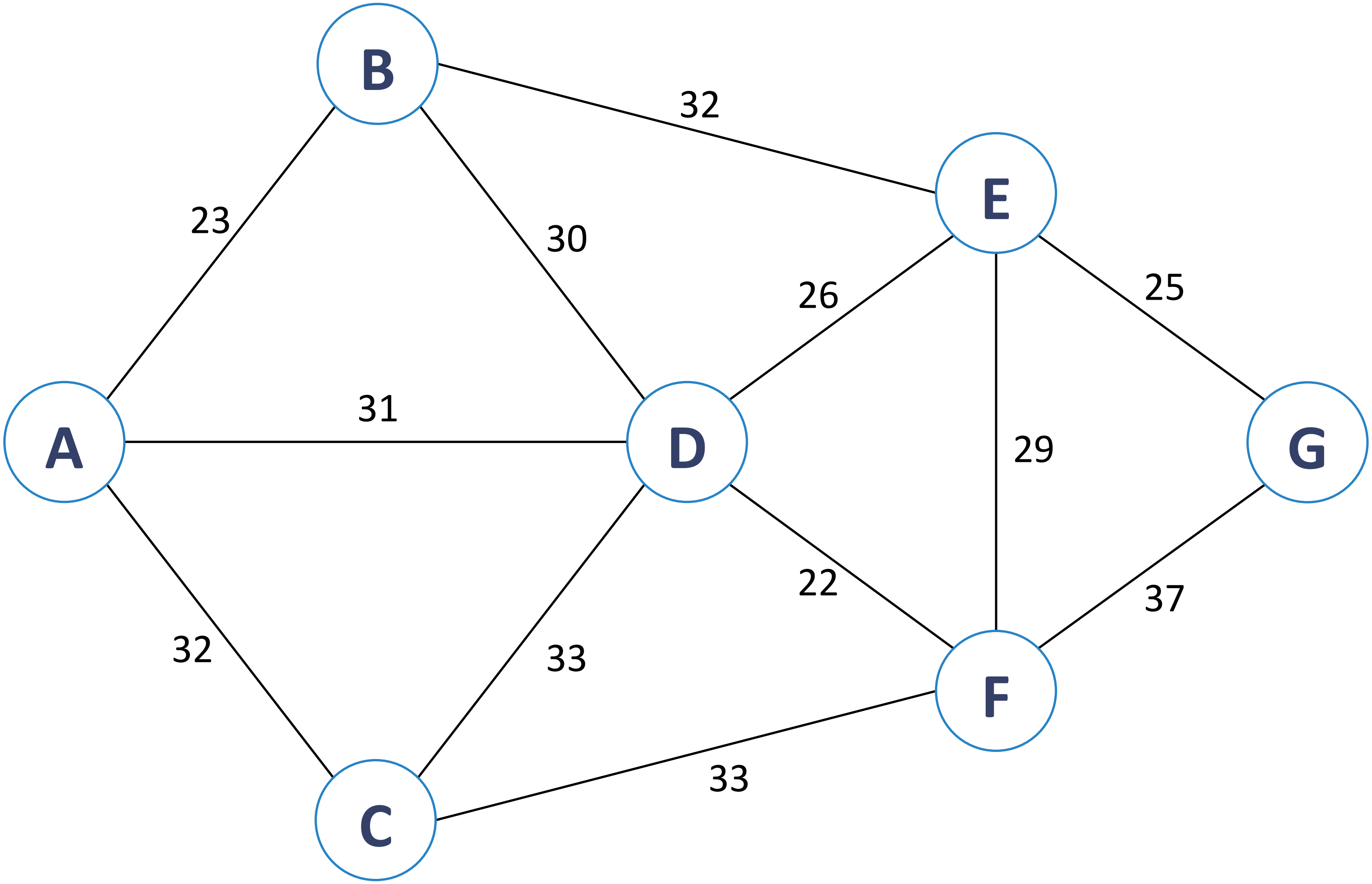
$\boldsymbol{foreach}$ $\boldsymbol{e} \in \boldsymbol{E}$:

$\boldsymbol{if}$ agregar $\boldsymbol{e}$ a $\boldsymbol{T}$ no forma un ciclo:

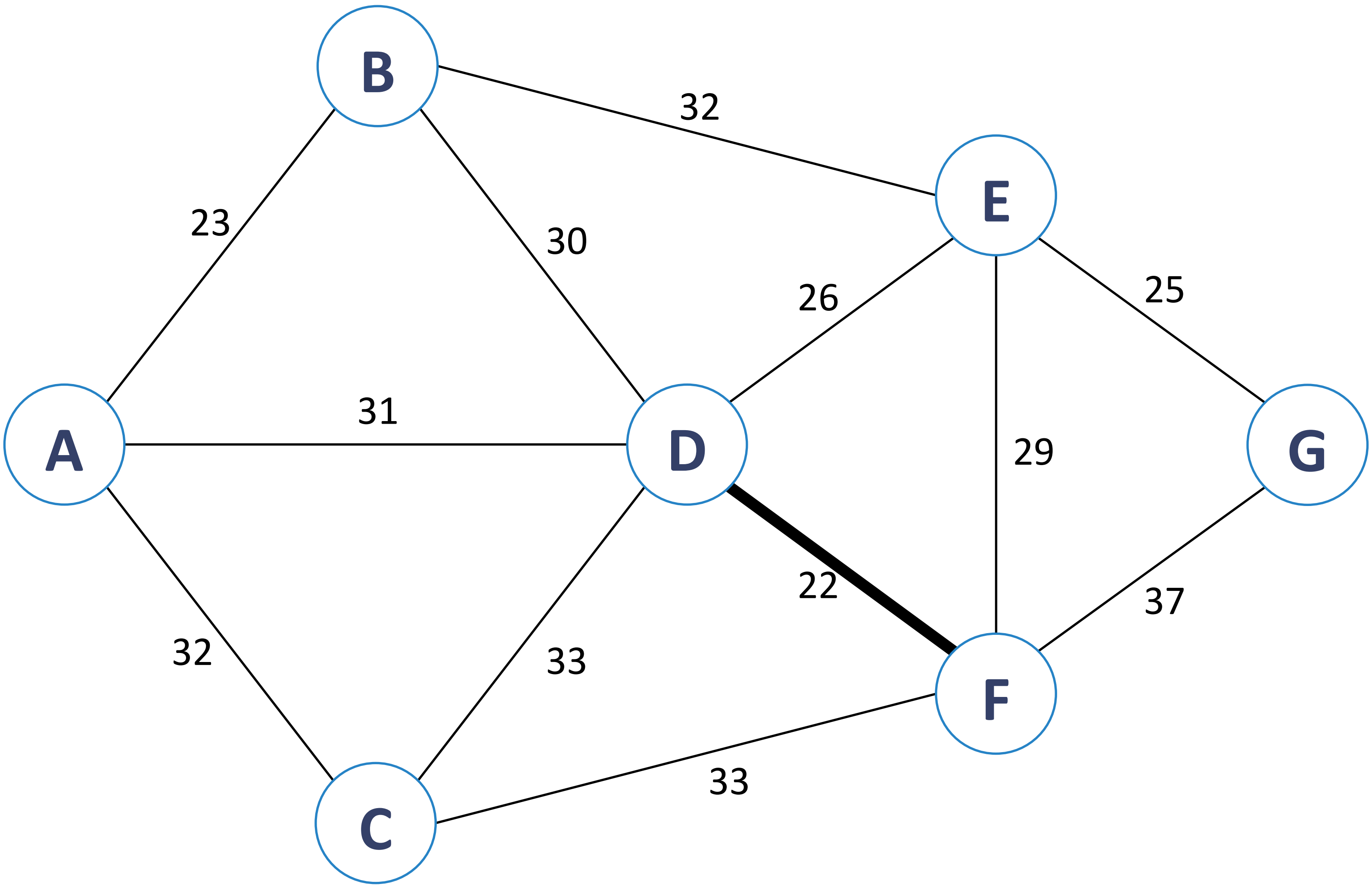
Agregar $\boldsymbol{e}$ a $\boldsymbol{T}$

$\boldsymbol{return}$ $\boldsymbol{T}$

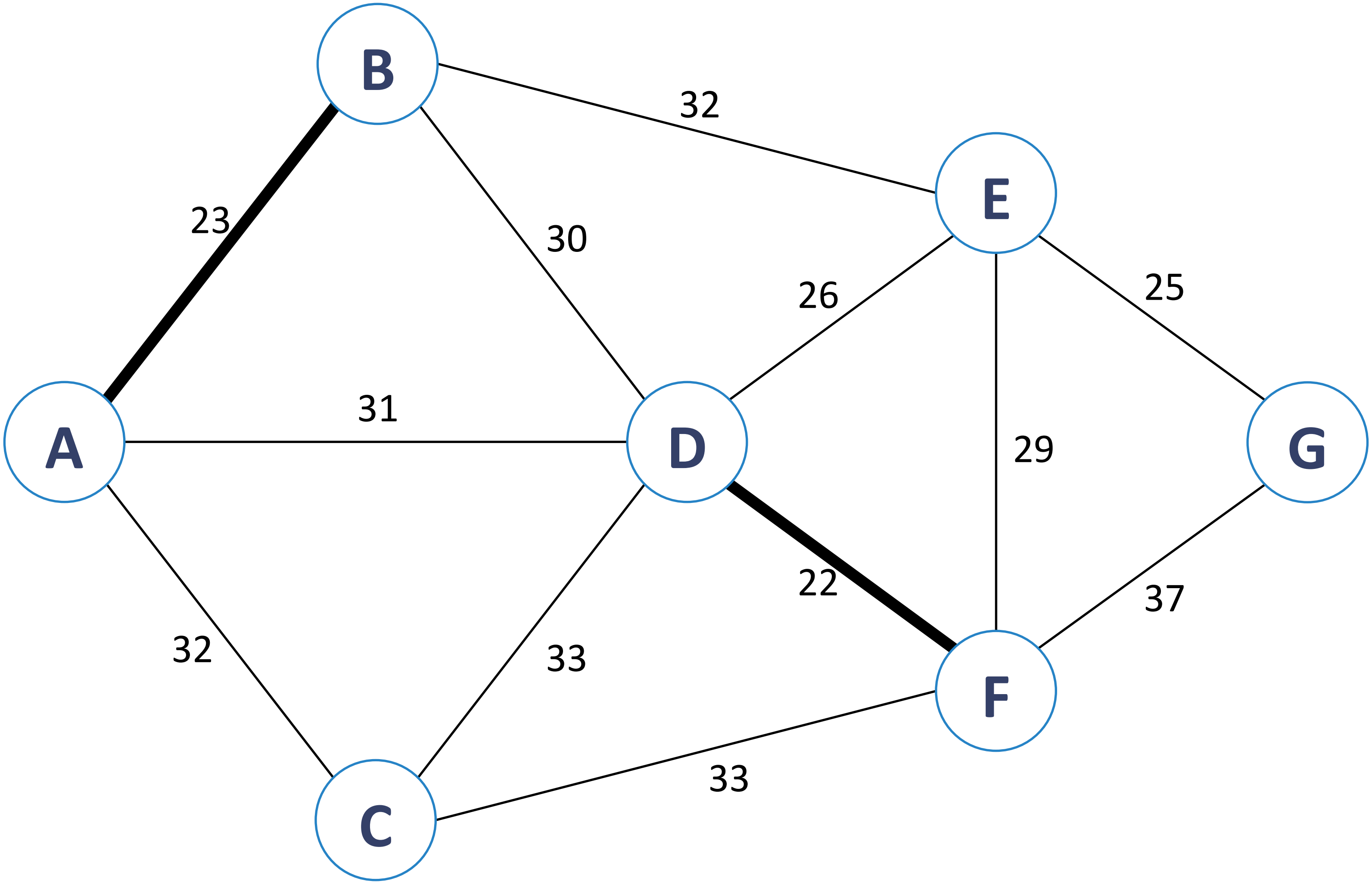
`kruskal` en acción



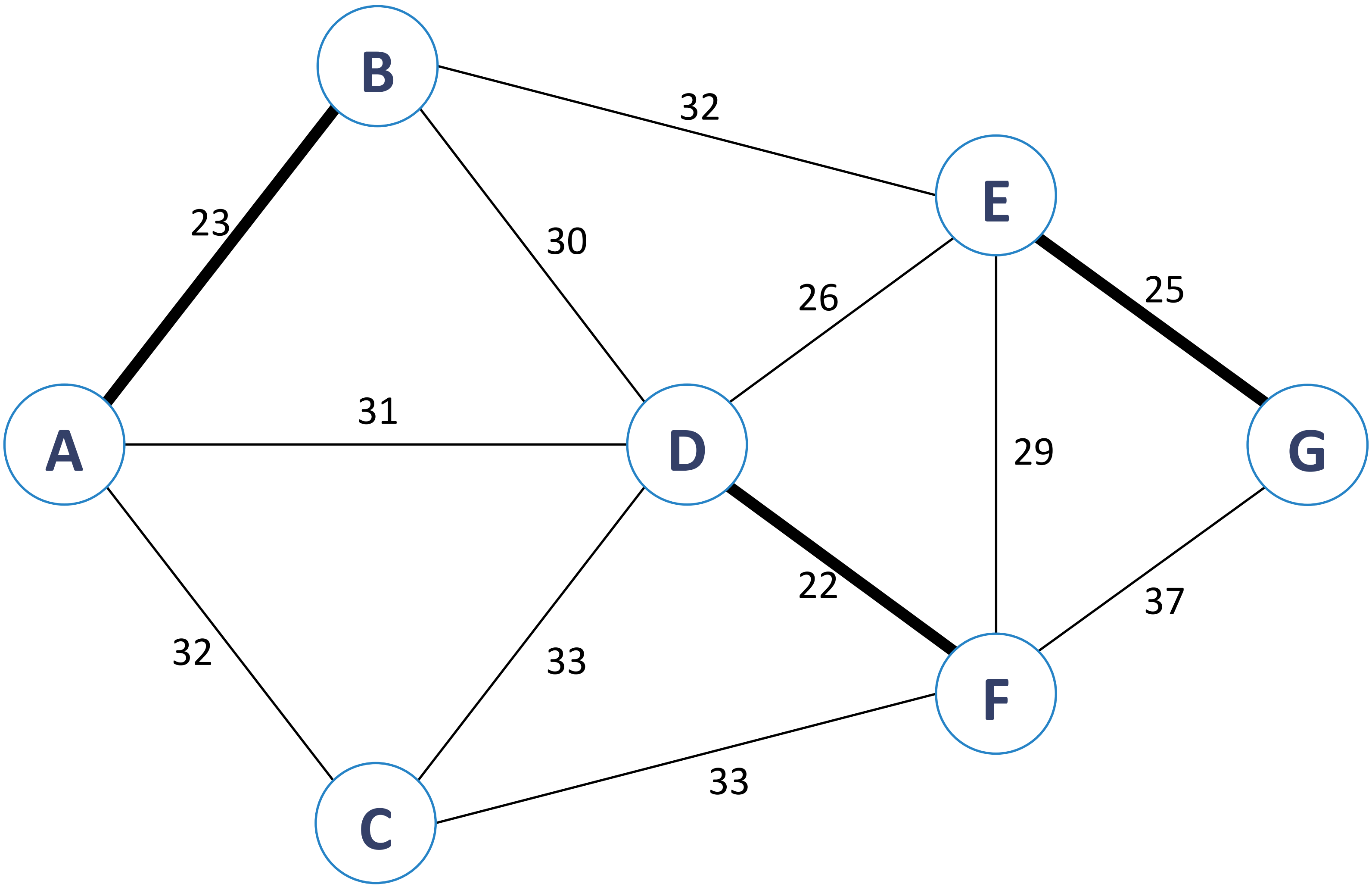
Partimos incluyendo en el MST la arista de menor costo



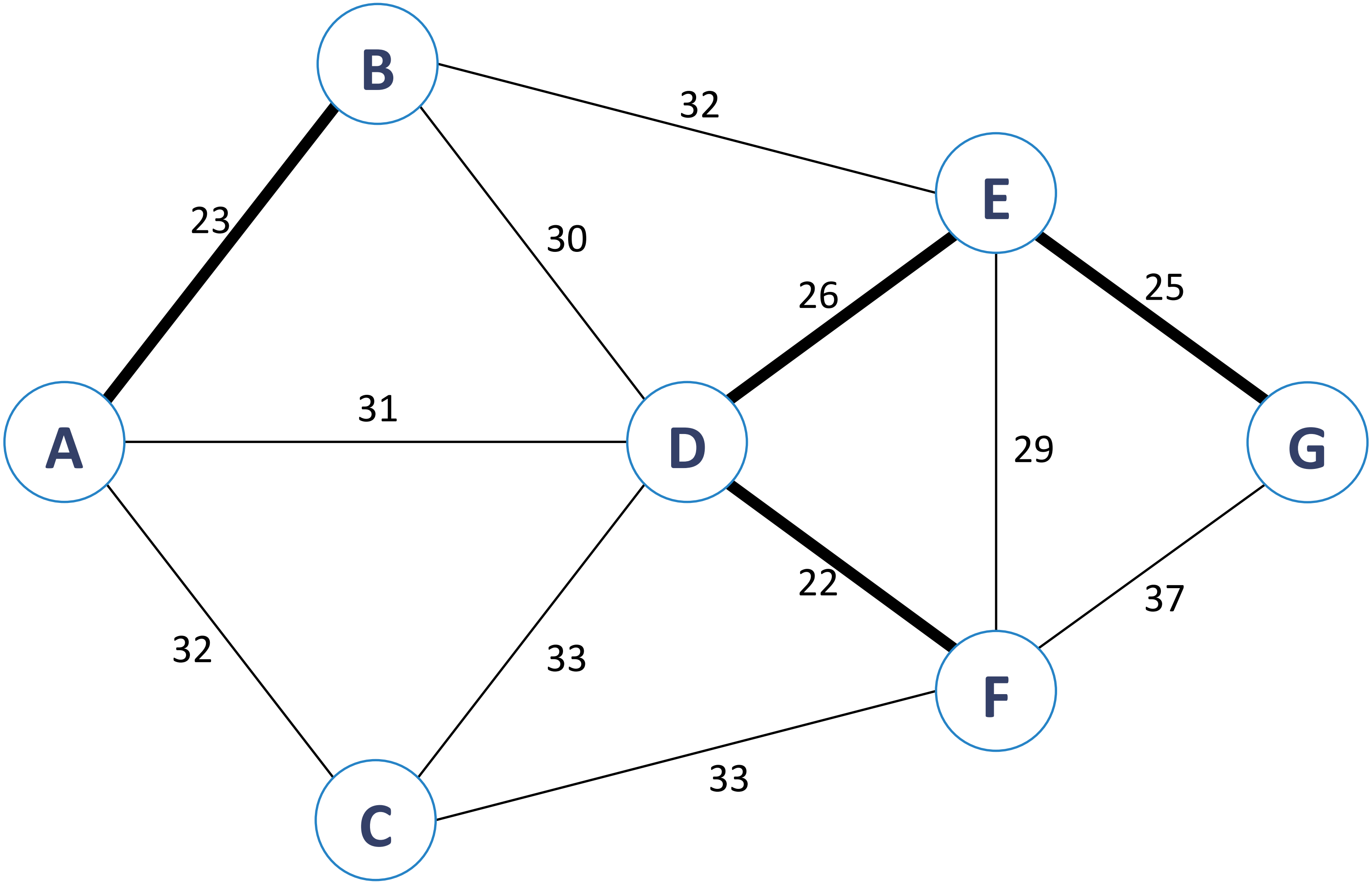
… y seguimos aplicando esta estrategia



...

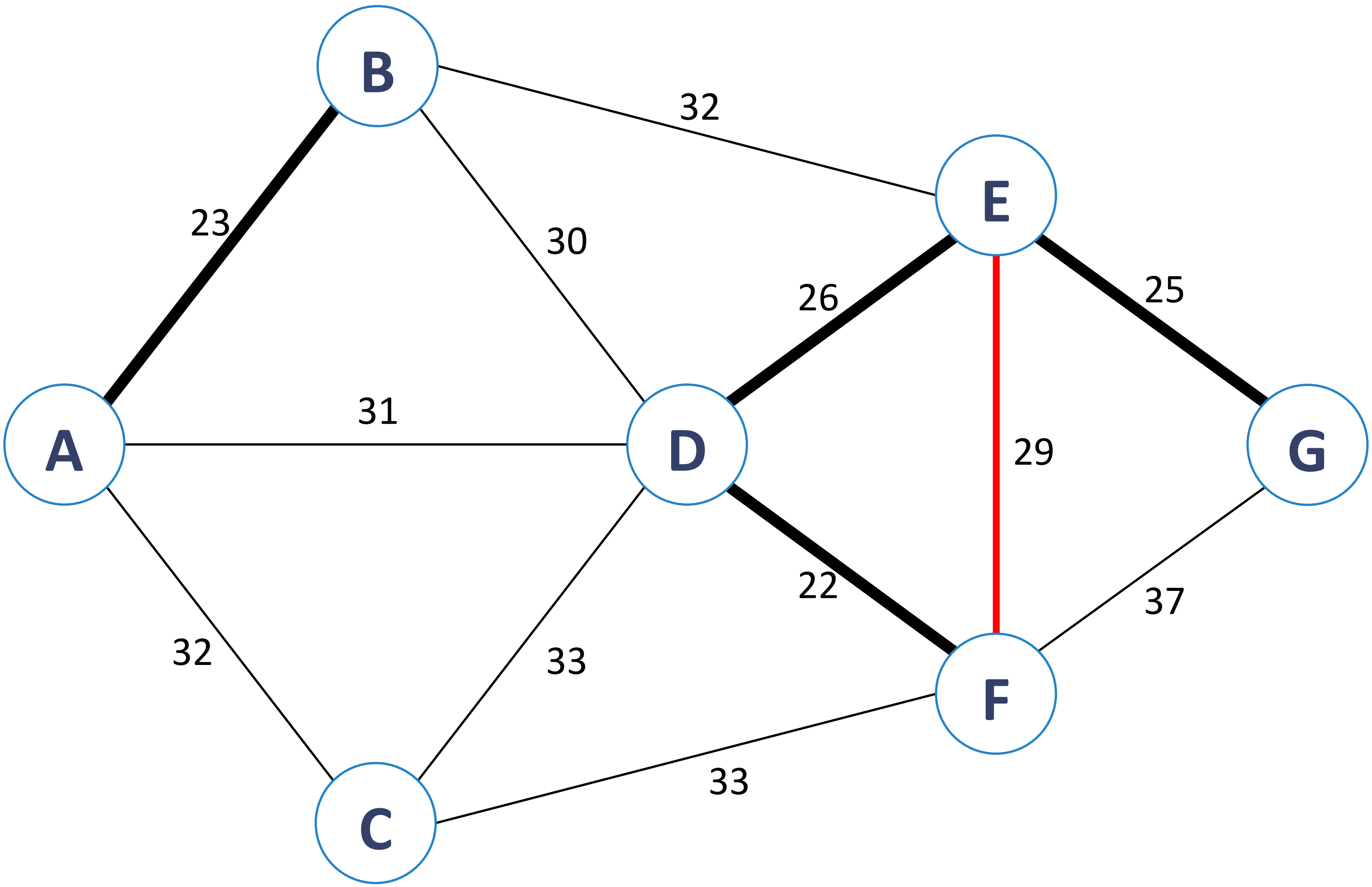

B
32
23
30
E
25
26
A
31
D
29
G
22
37
32
33
F
C
33

...

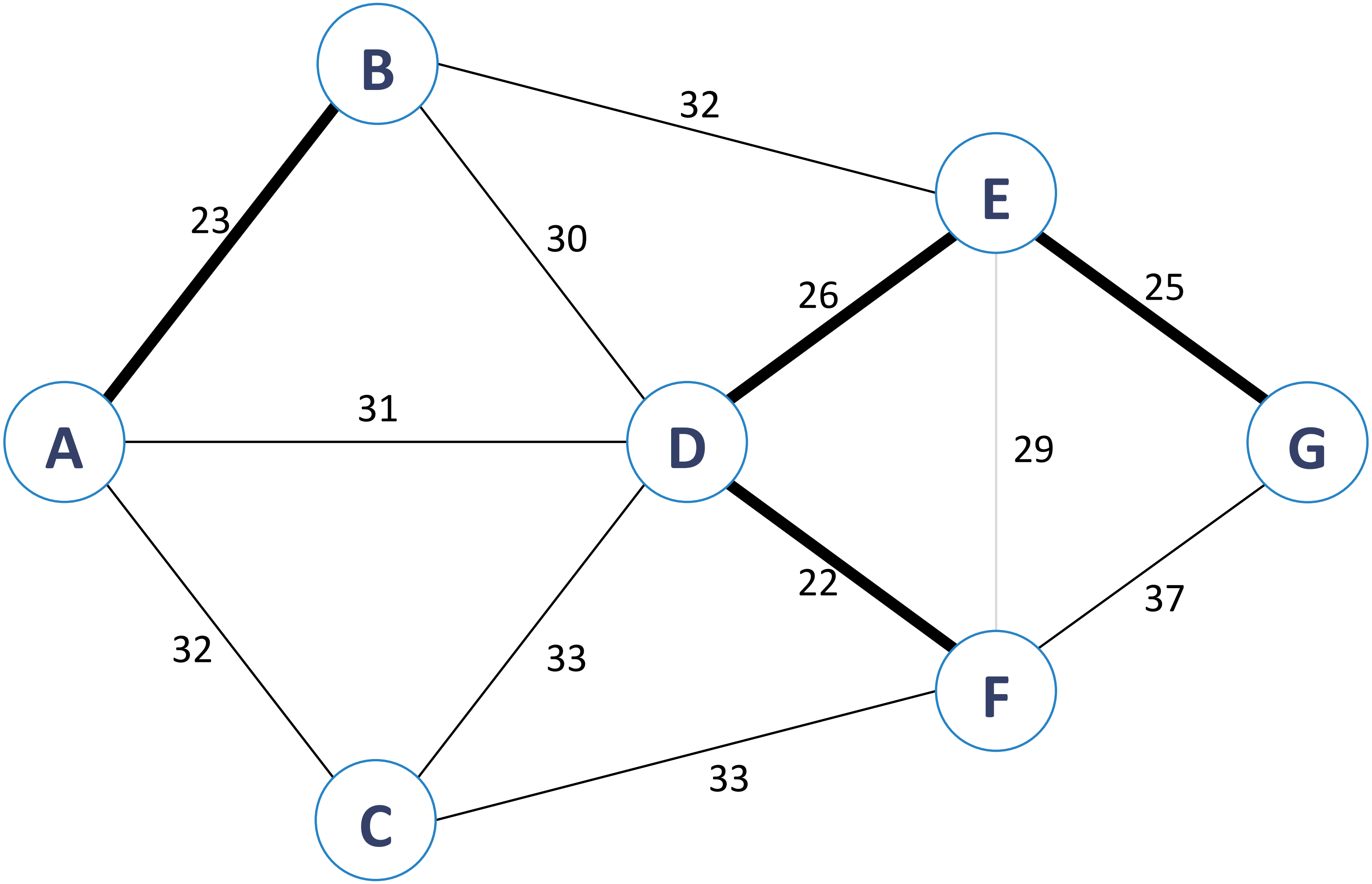


B
32
23
30
E
26
25
31
A
D
29
G
22
37
32
33
F
33
C

… mientras la nueva arista **no forme un ciclo** con las aristas que ya están en el MST



...


B
32
23
30
E
26
25
A
31
D
29
G
22
37
32
33
F
33
C

¿Cómo se define el corte en este caso?

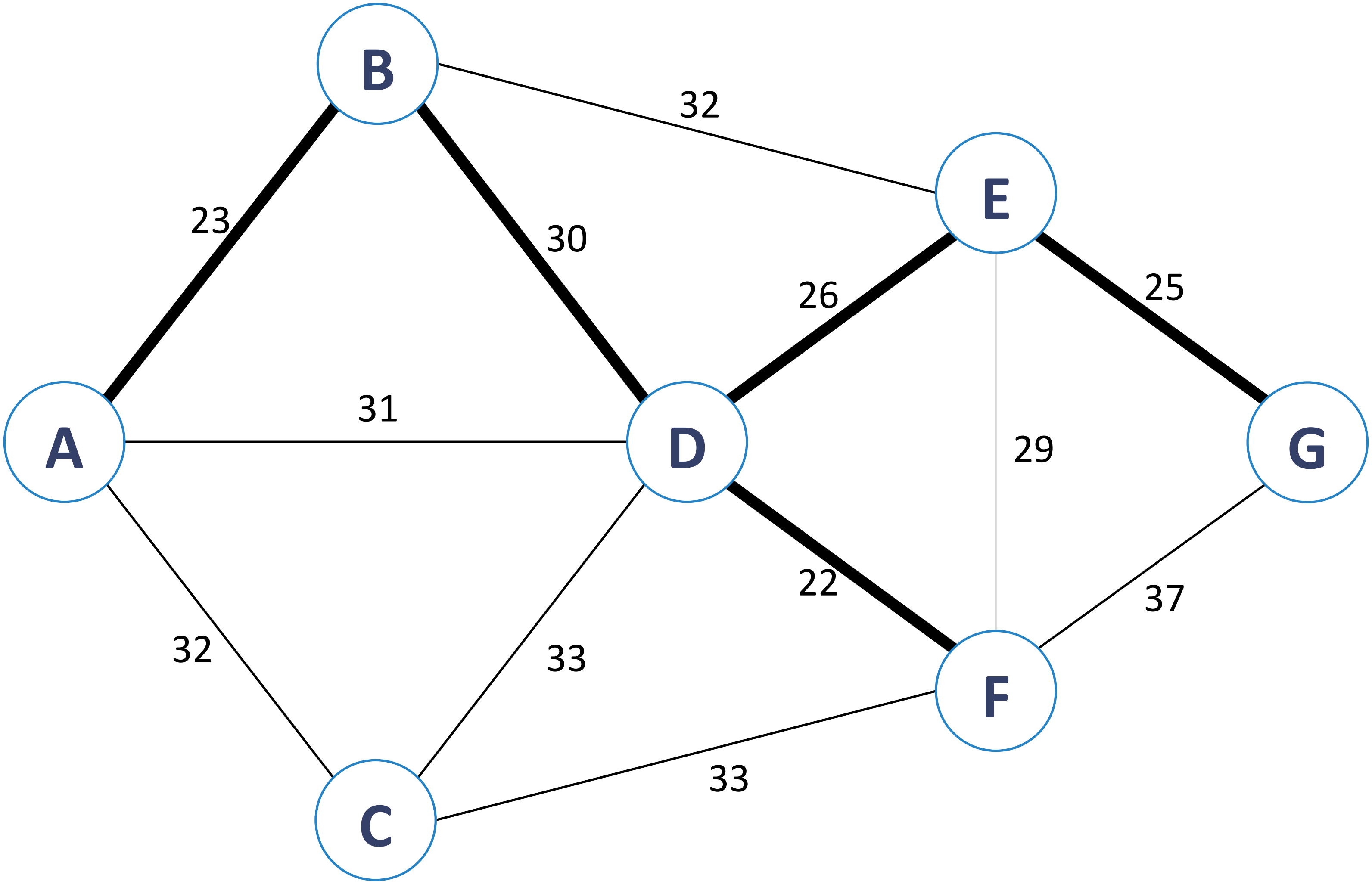
En cada paso, al considerar la próxima arista más liviana,

… el corte queda formado por el conjunto de los nodos conectados mediante aristas negras a uno de los nodos de la arista considerada (y por su complemento):

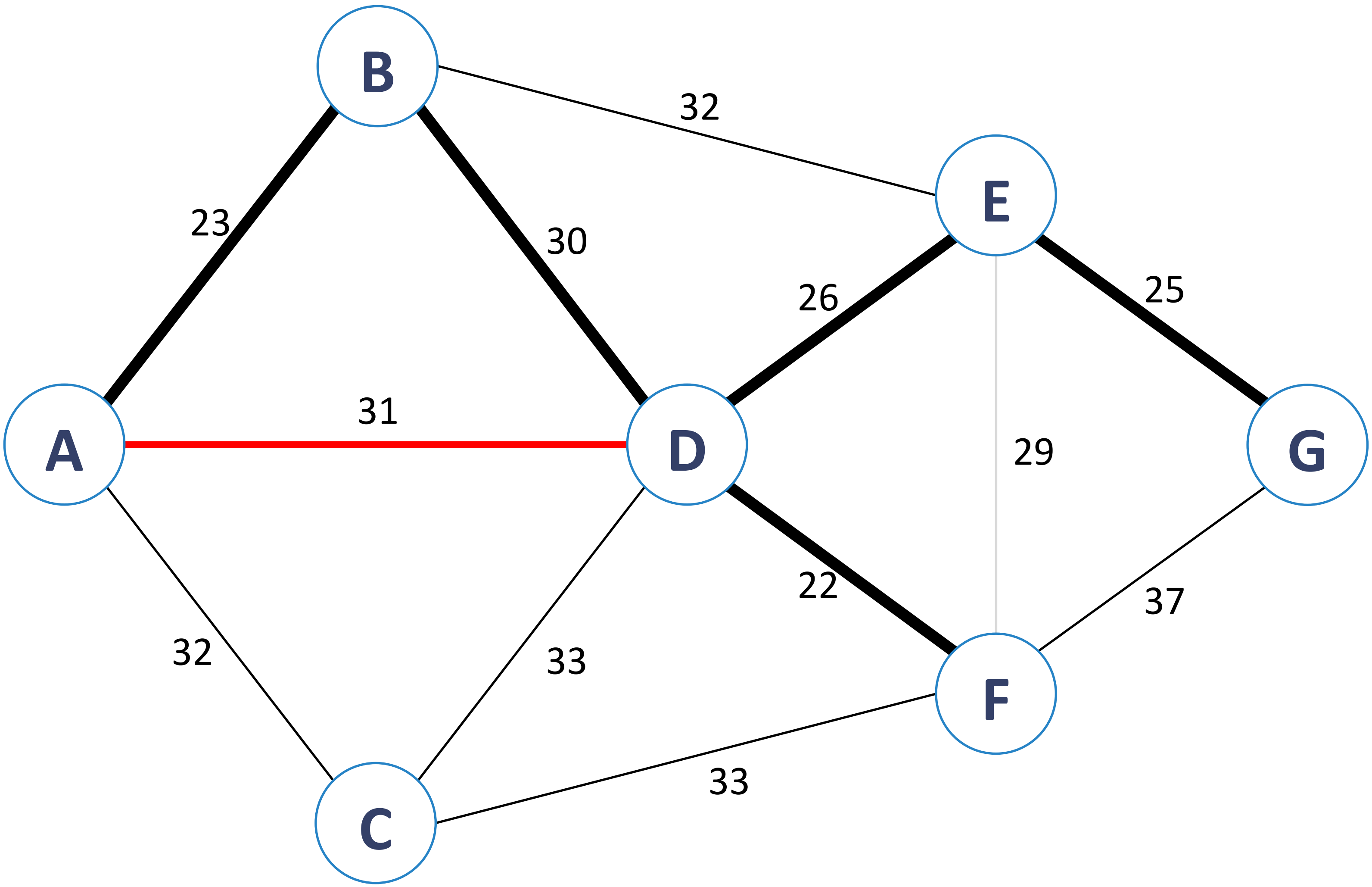
- en el ej., la próxima arista considerada es la arista (B,D) de costo 30
- el corte puede ser ({A,B}, {C,D,E,F,G})

  … o bien ({E,D,F,G}, {A,B,C})
- en ambos casos, la arista (B,D) *cruza el corte* (es decir, no forma un ciclo) y por lo tanto la agregamos a la solución

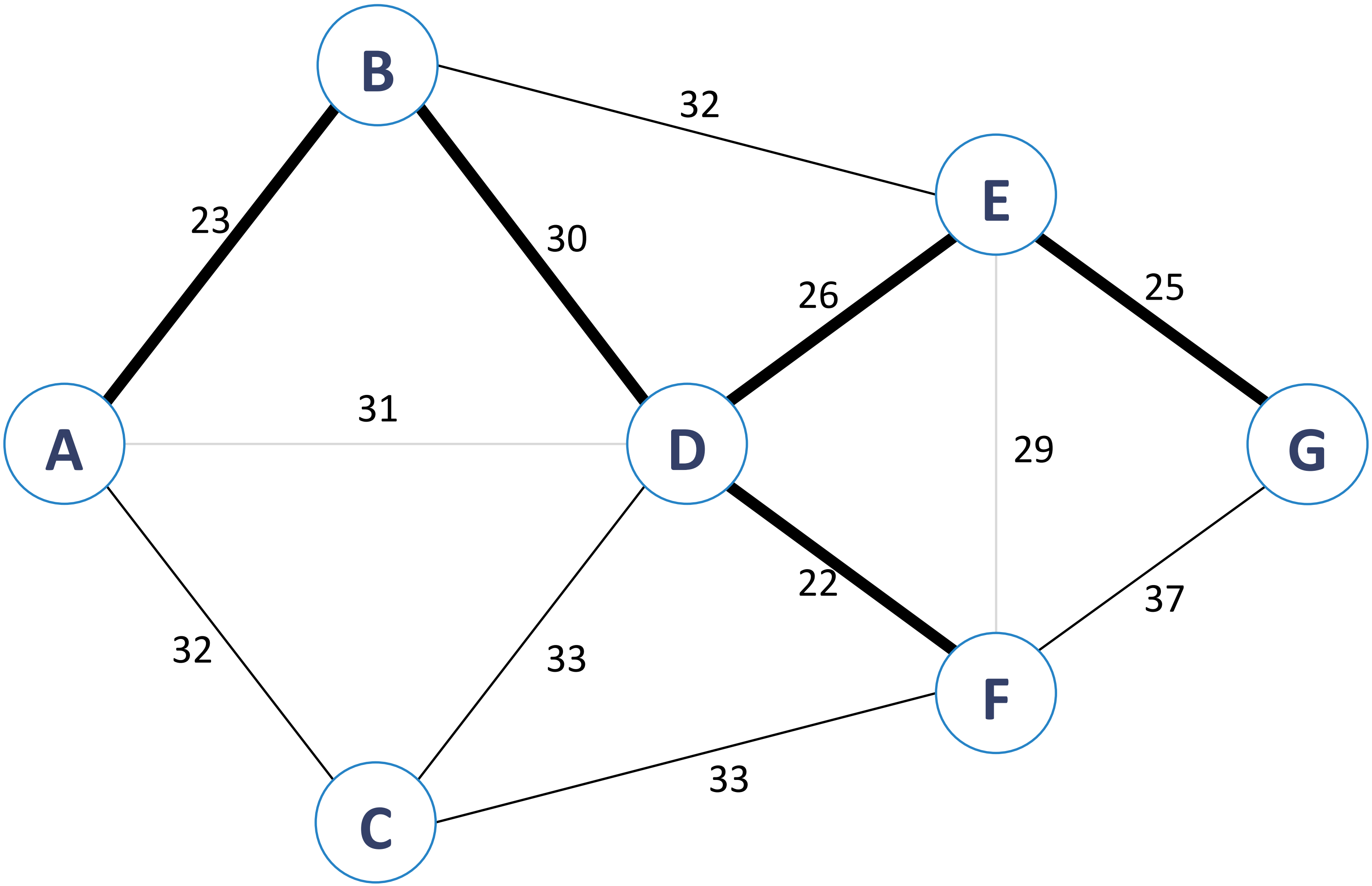
...


B
32
E
23
30
26
25
A
31
D
29
G
22
37
32
33
F
33
C

Si la nueva arista forma un ciclo, significa que *no cruza el corte* y la descartamos

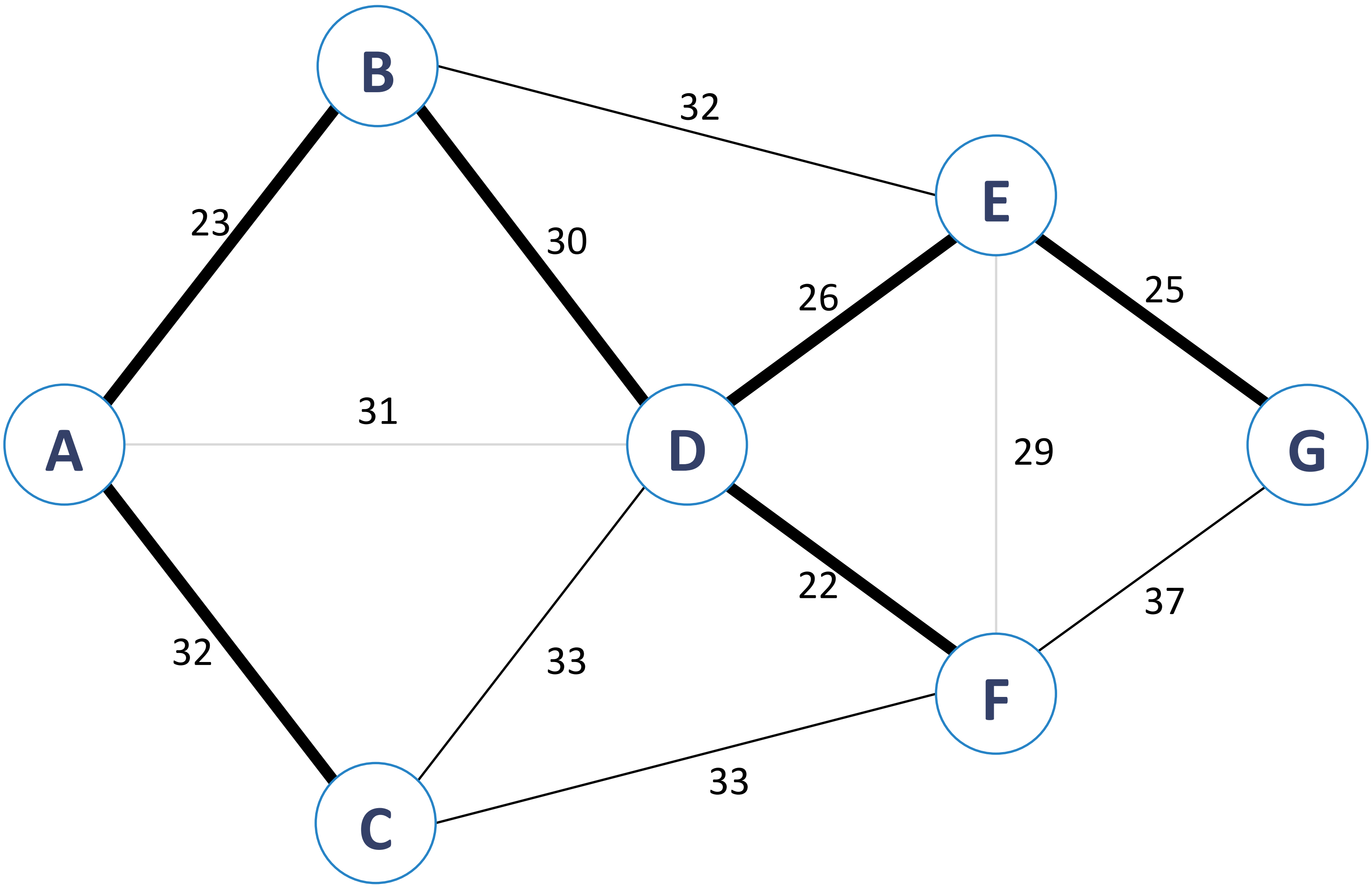


...

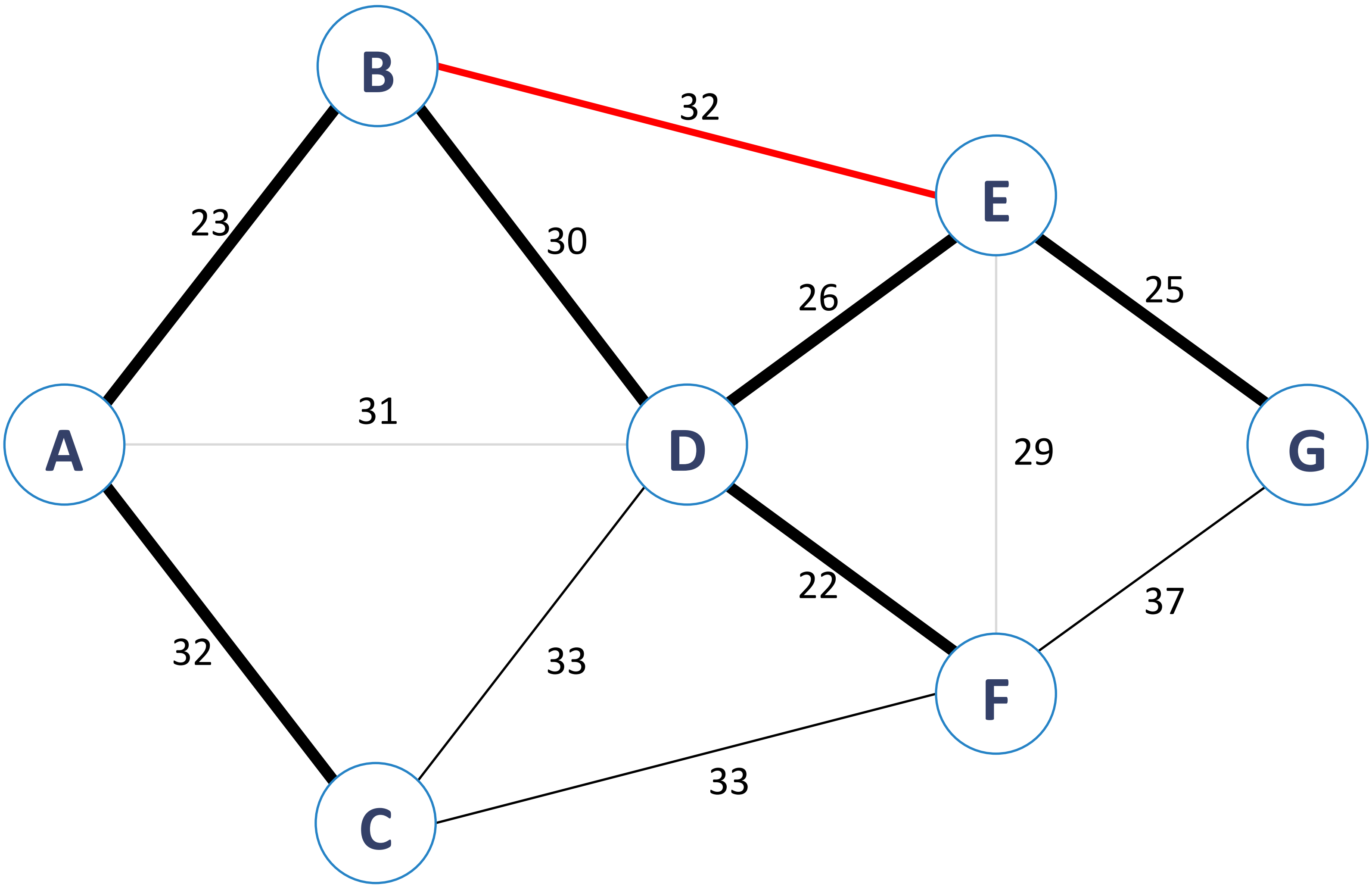


B
32
23
30
E
26
25
A
31
D
29
G
22
37
32
33
F
C
33

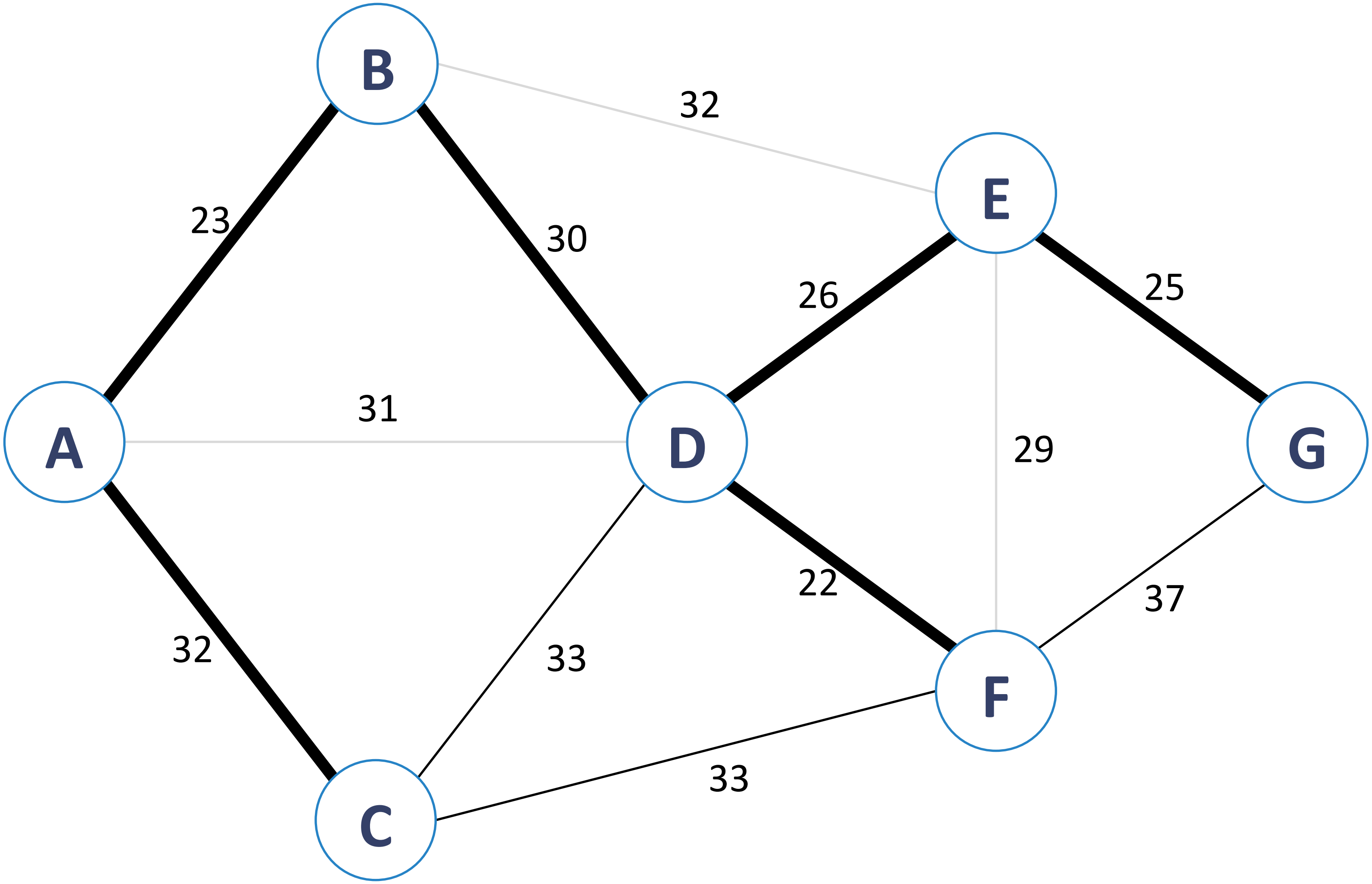
…


A
B
C
D
E
F
G
23
30
32
31
26
25
29
32
33
22
37
33

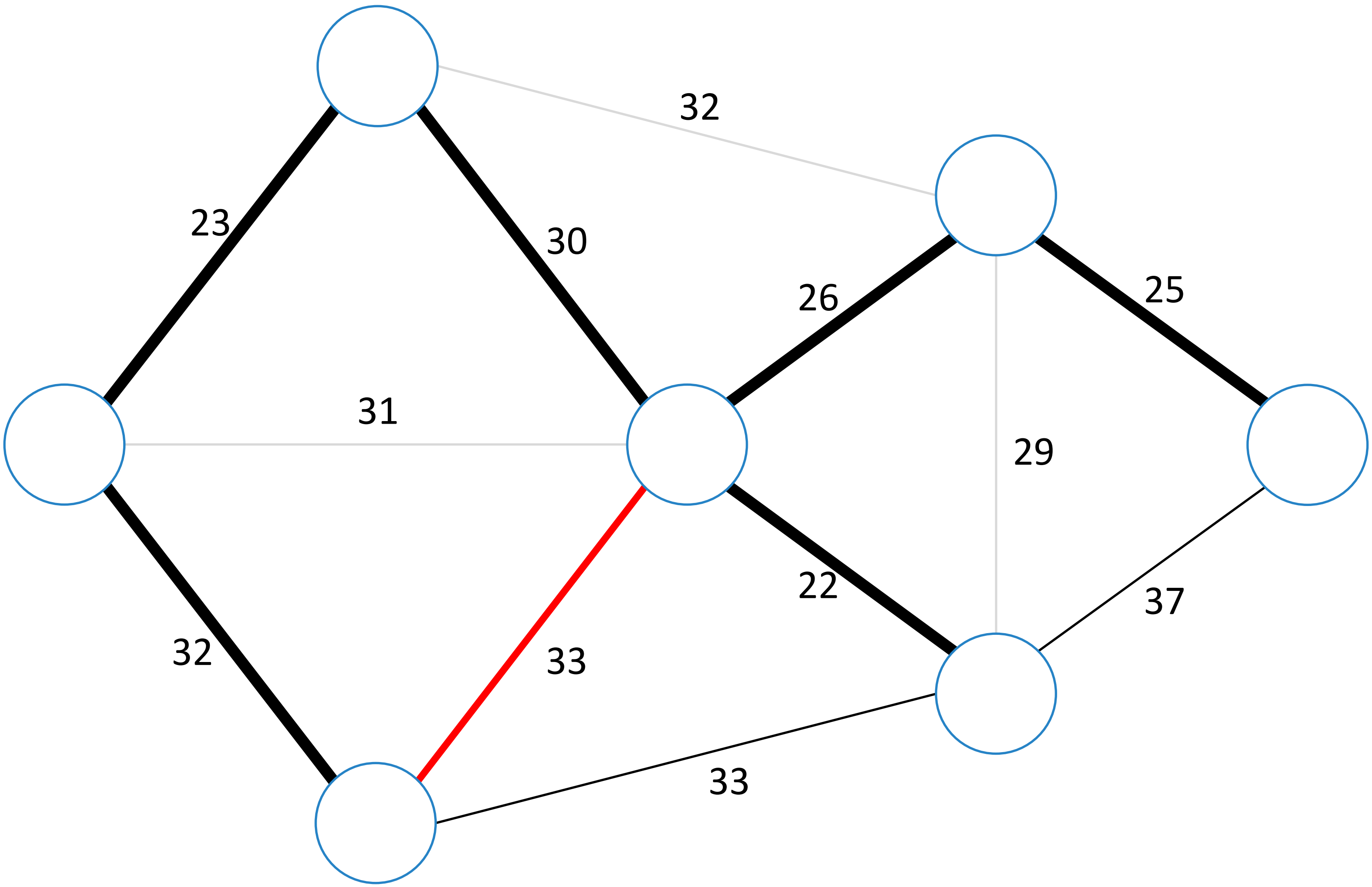
Si el grafo tiene $|V|$ vértices, entonces el MST tiene $|V|-1$ aristas



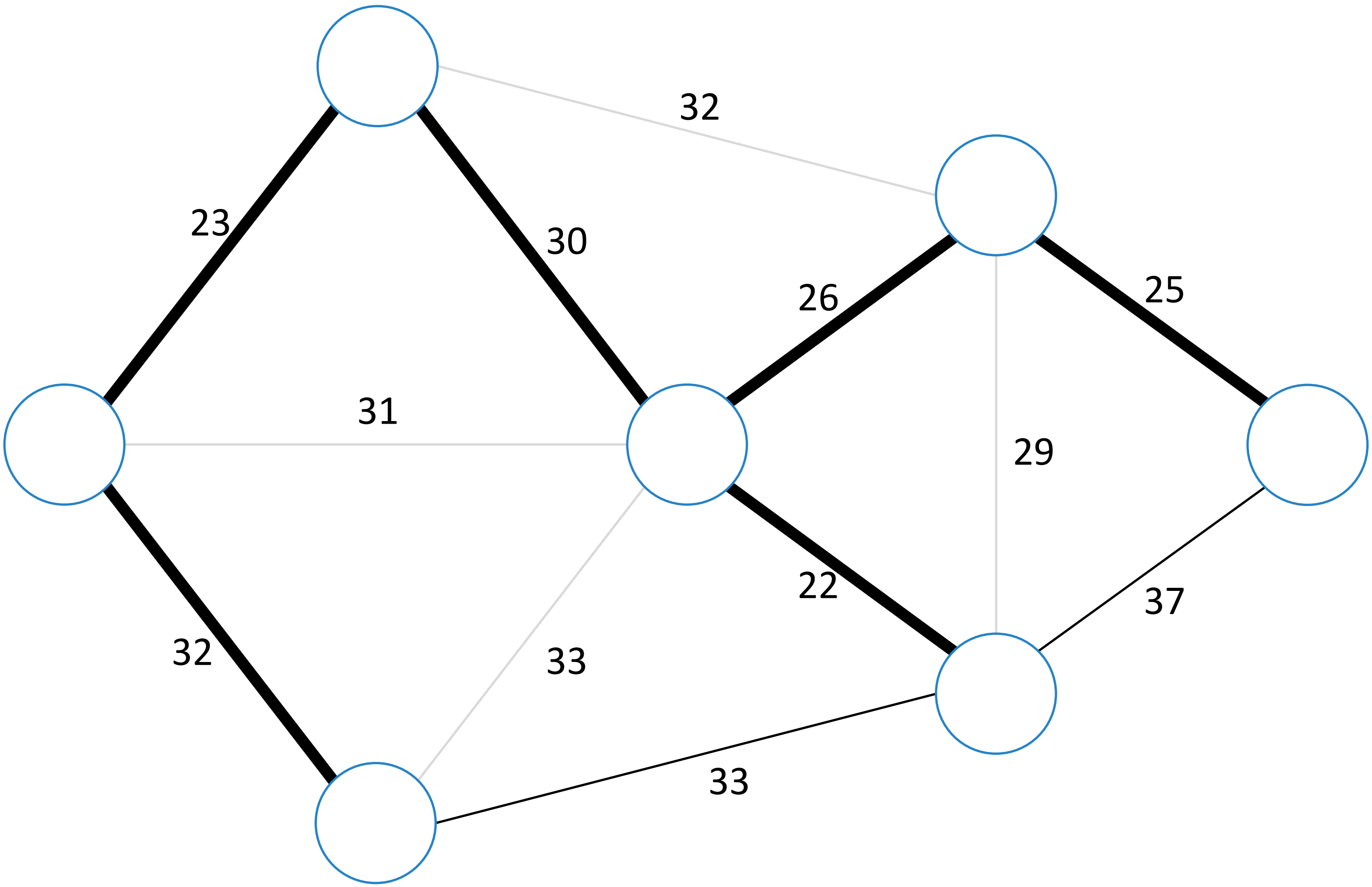
Una vez que el MST tiene $|V|-1$ aristas, cualquier otra arista forma un ciclo



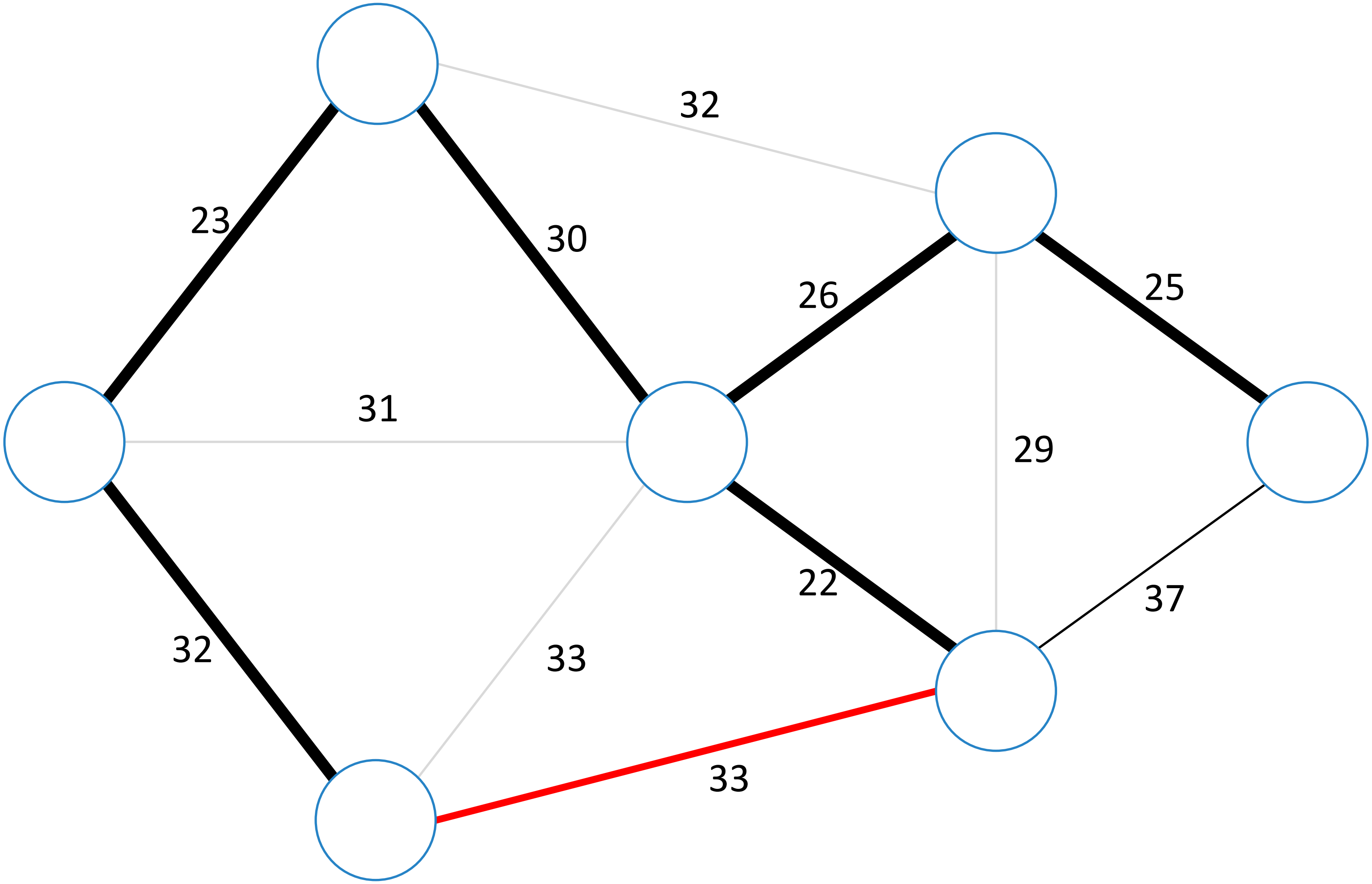
… y en la práctica no es necesario revisar las aristas restantes



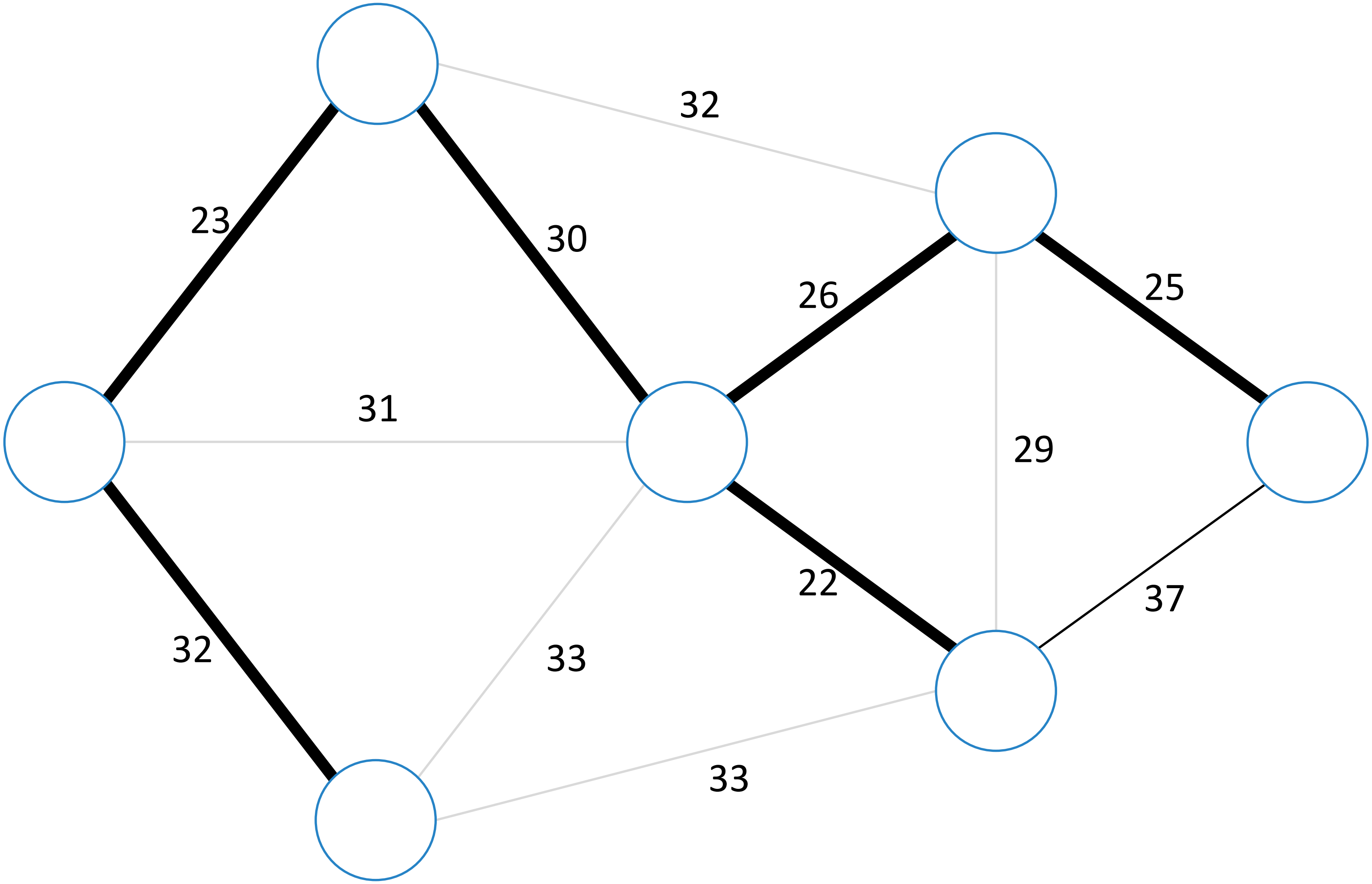
...


32
23
30
26
25
31
29
22
37
32
33
33

...

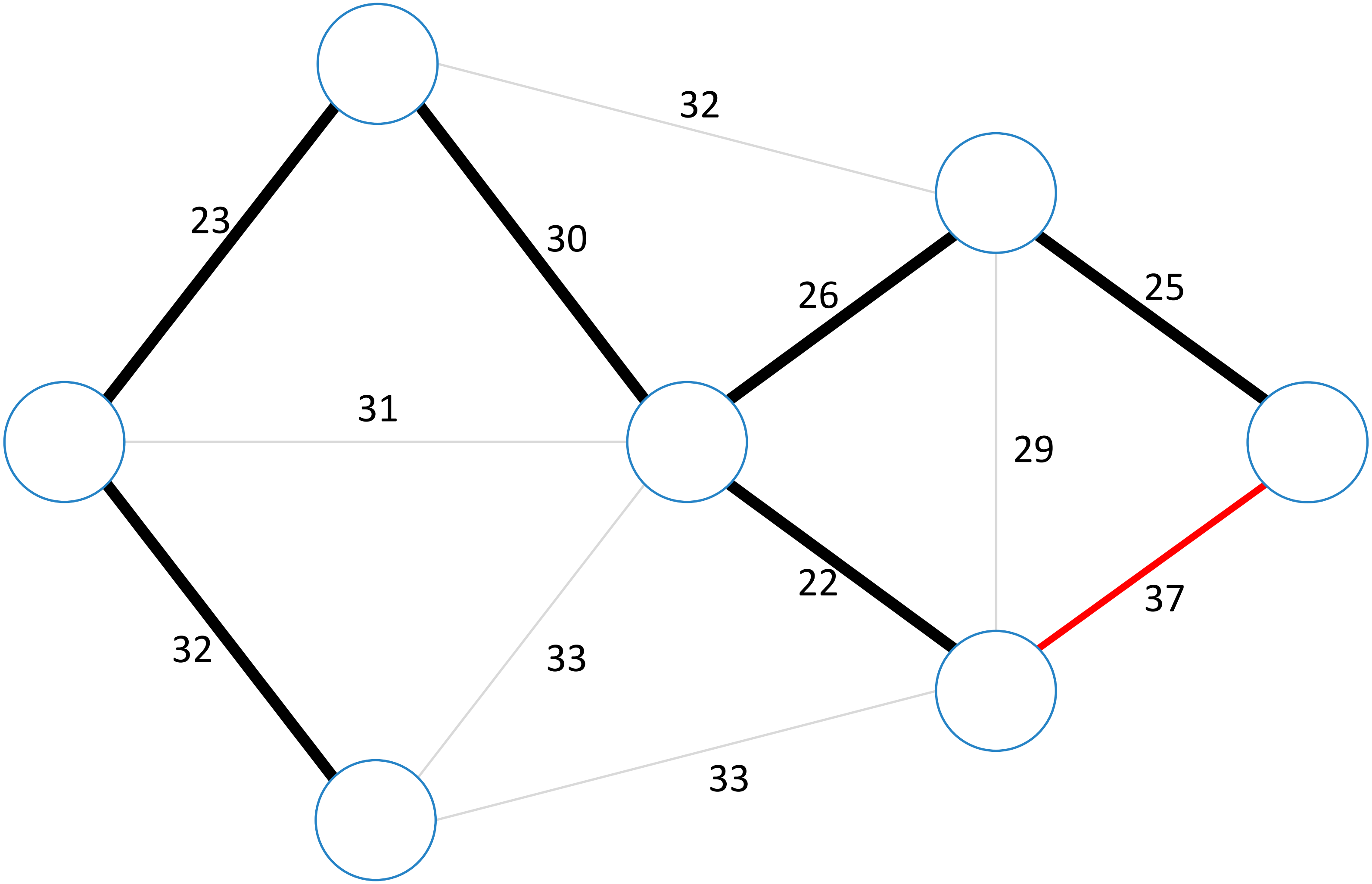

32
23
30
26
25
31
29
22
37
32
33
33

...

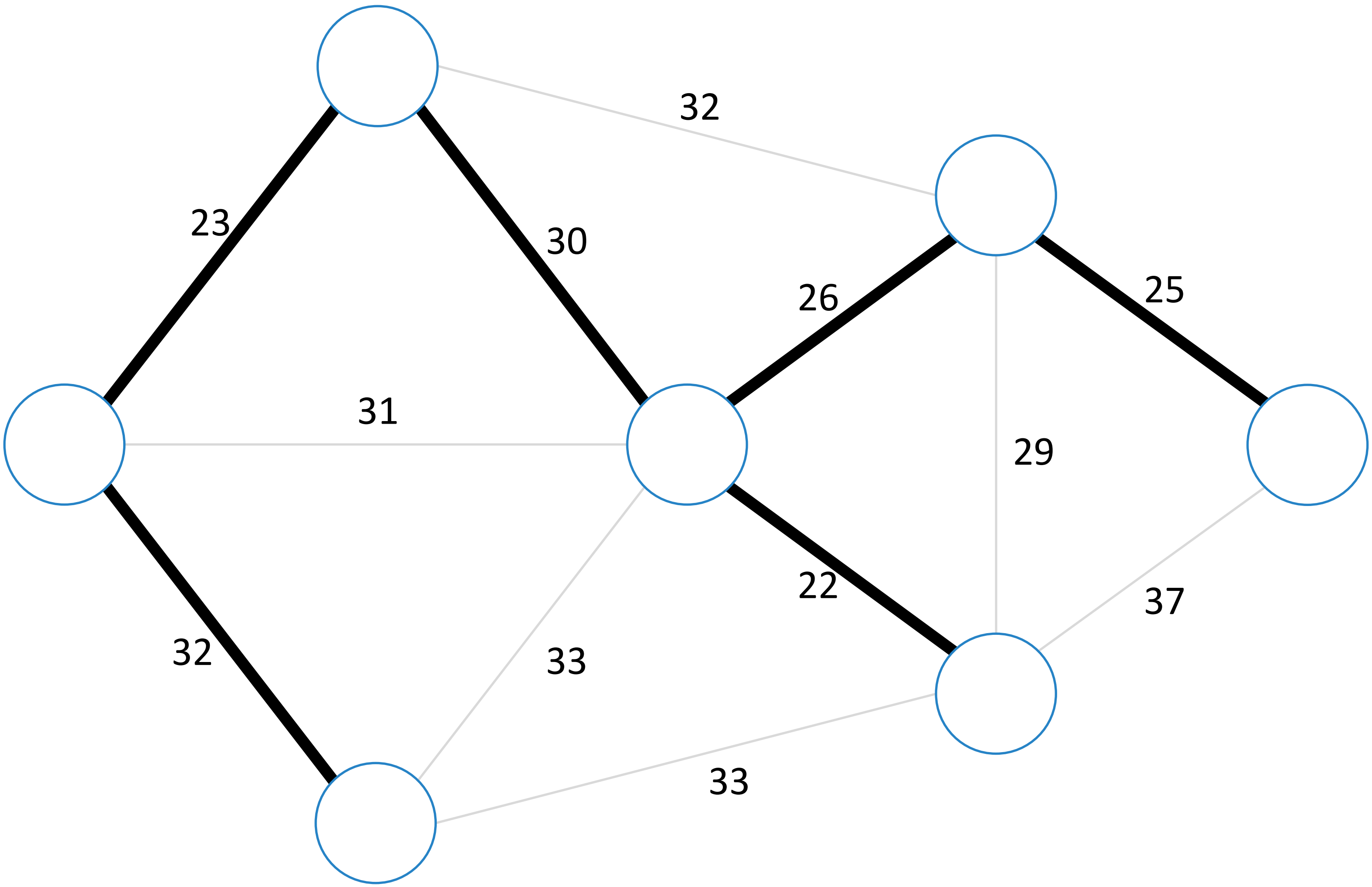


32
23
30
26
25
31
29
22
37
32
33
33

...


32
23
30
26
25
31
29
22
37
32
33
33

...


32
23
30
26
25
31
29
22
37
32
33
33

Corrección de `kruskal`

`kruskal` es otra implementación de la estrategia codiciosa estudiada la clase pasada:

- dado un corte, elegir la arista más liviana que cruza el corte

Por lo tanto, lo que hay que demostrar es que `kruskal` es efectivamante una implementación de esa estrategia

Un “detalle” no menor

$\boldsymbol{kruskal(G(V,E))}$:

Ordenar $\boldsymbol{E}$ por costo, de menor a mayor

$\boldsymbol{T} \leftarrow \emptyset$

$\boldsymbol{foreach}\ \boldsymbol{e} \in \boldsymbol{E}$:

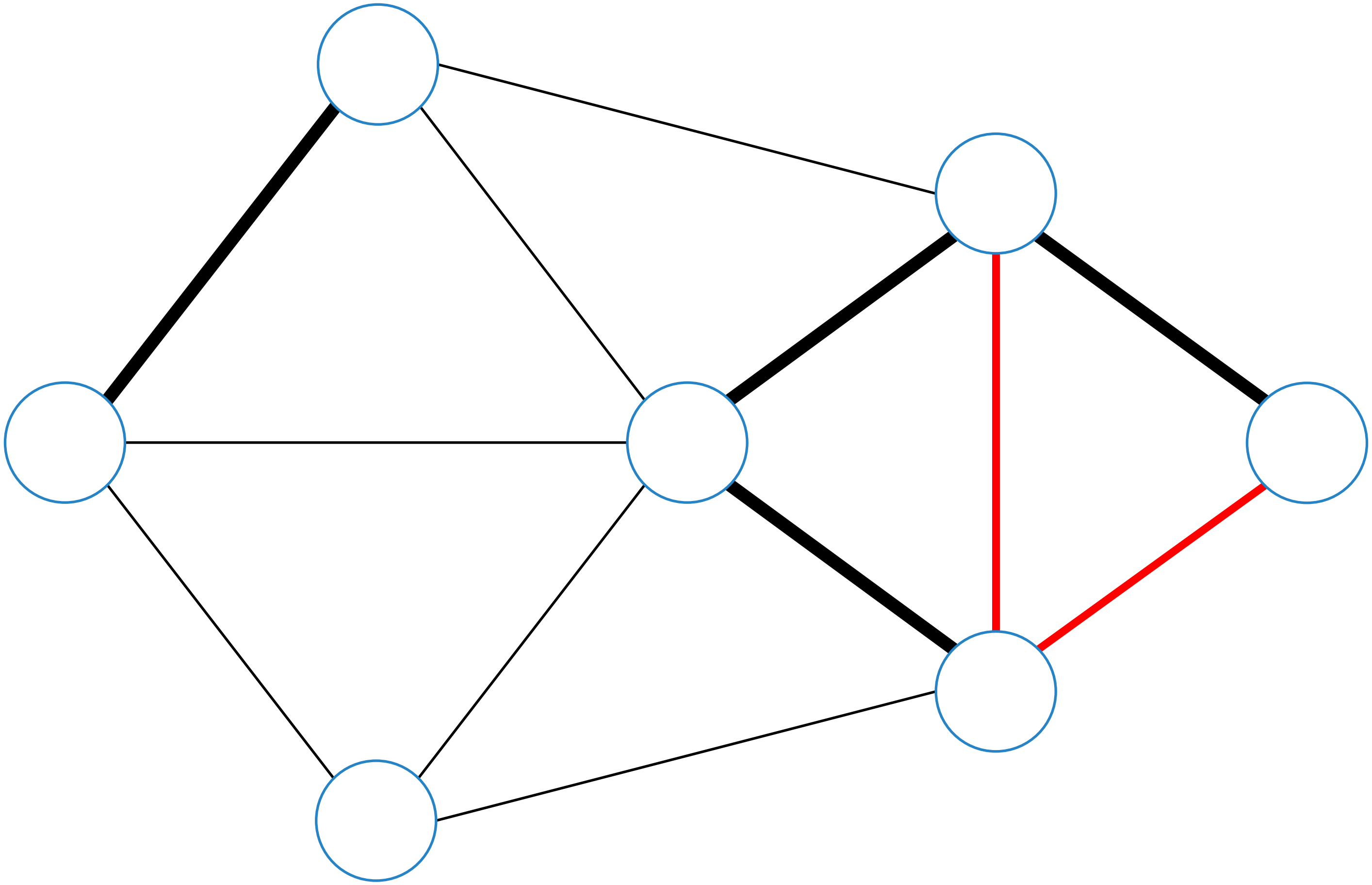
$\boldsymbol{if}$ agregar $\boldsymbol{e}$ a $\boldsymbol{T}$ no forma un ciclo:

Agregar $\boldsymbol{e}$ a $\boldsymbol{T}$

$\boldsymbol{return}\ \boldsymbol{T}$

¿Cómo revisamos esto de manera eficiente?

Observación



Agregar (u, v) forma un ciclo **ssi** u y v están en el mismo **sub-árbol**

Conjuntos disjuntos

Un nodo puede pertenecer a un solo **sub-árbol** del grafo

Los **conjuntos** de nodos de cada sub-árbol son **disjuntos**

¿Cómo podemos modelar esto para aprovecharlo?

`kruskal` con conjuntos disjuntos

$\textbf{\textit{kruskal}}\big(\boldsymbol{G}(\boldsymbol{V},\boldsymbol{E})\big)$:

Ordenar $\boldsymbol{E}$ por costo, de menor a mayor

Considerar cada nodo como formando un conjunto por sí mismo

$\boldsymbol{T} \leftarrow \emptyset$

$\boldsymbol{foreach}$ $(\boldsymbol{u},\boldsymbol{v}) \in \boldsymbol{E}$:

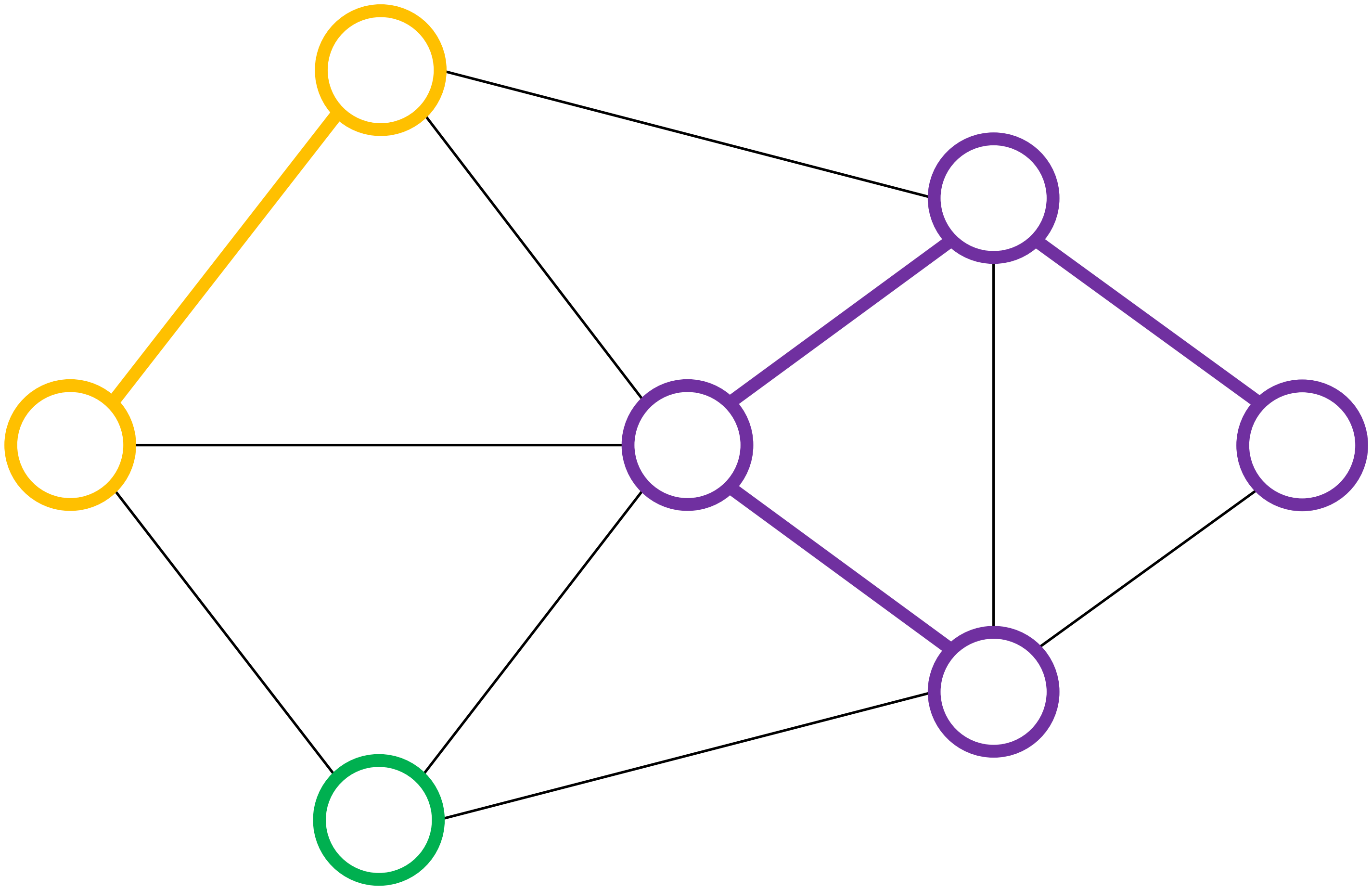
$\boldsymbol{if}$ si $\boldsymbol{u}$ y $\boldsymbol{v}$ no están en el mismo conjunto:

Agregar $(\boldsymbol{u},\boldsymbol{v})$ a $\boldsymbol{T}$

Unir los conjuntos de $\boldsymbol{u}$ y $\boldsymbol{v}$

$\boldsymbol{return}$ $\boldsymbol{T}$

Sub-árboles como conjuntos



Agregar una arista significa unir dos conjuntos

Operaciones necesarias sobre conjuntos disjuntos

Nos interesan dos cosas:

- **Identificar** en qué conjunto está un elemento
- **Unir** dos conjuntos (y que sólo quede esta unión y no los conjuntos originales)

¿Cómo podemos hacer esto de manera eficiente?

Representación

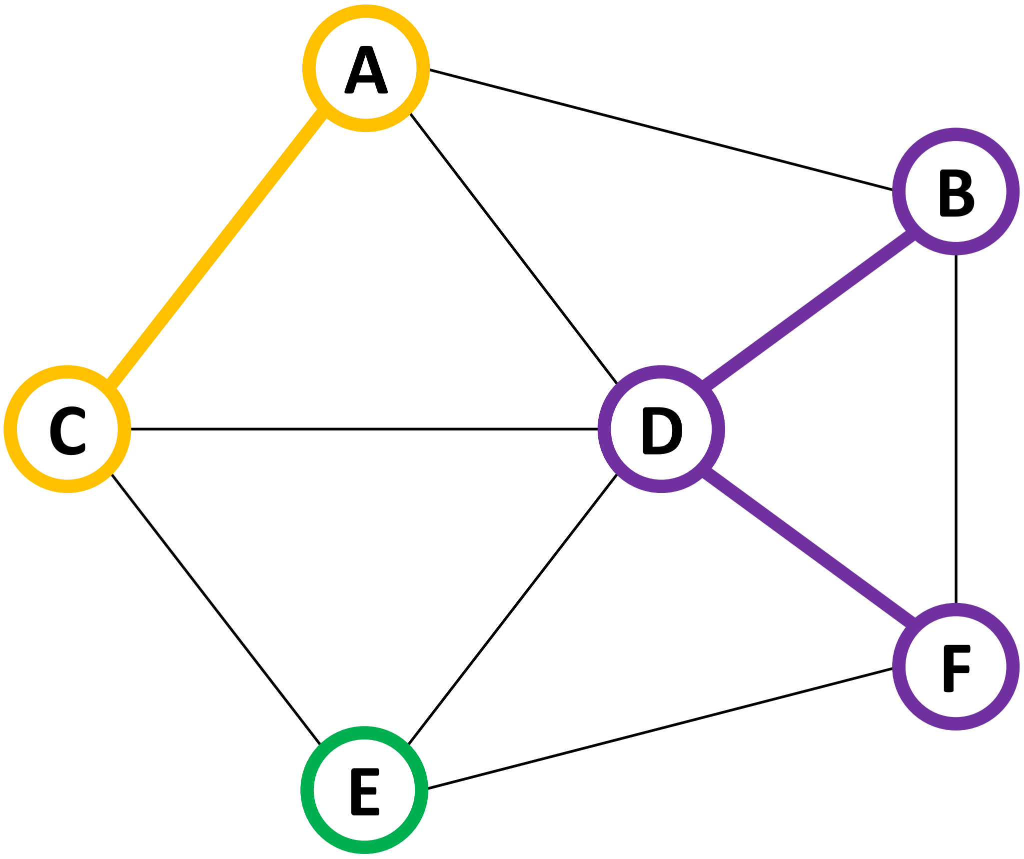
Para cada conjunto, escogemos un **representante** : uno de sus elementos

Cada nodo tiene una **referencia** a su representante, incluyendo el propio representante

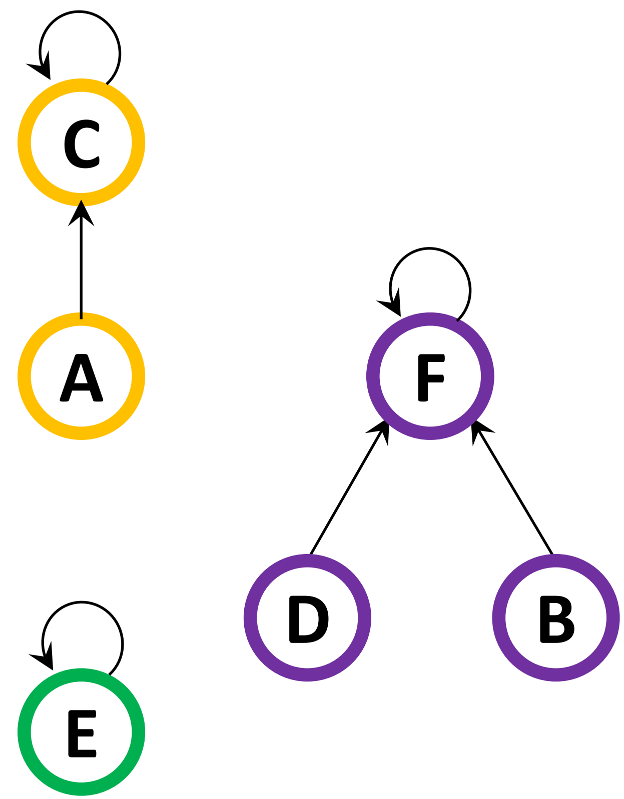
Dos nodos están en el **mismo** conjunto si y sólo si tienen el mismo representante

Conjuntos disjuntos

Sub-árboles



Conjuntos



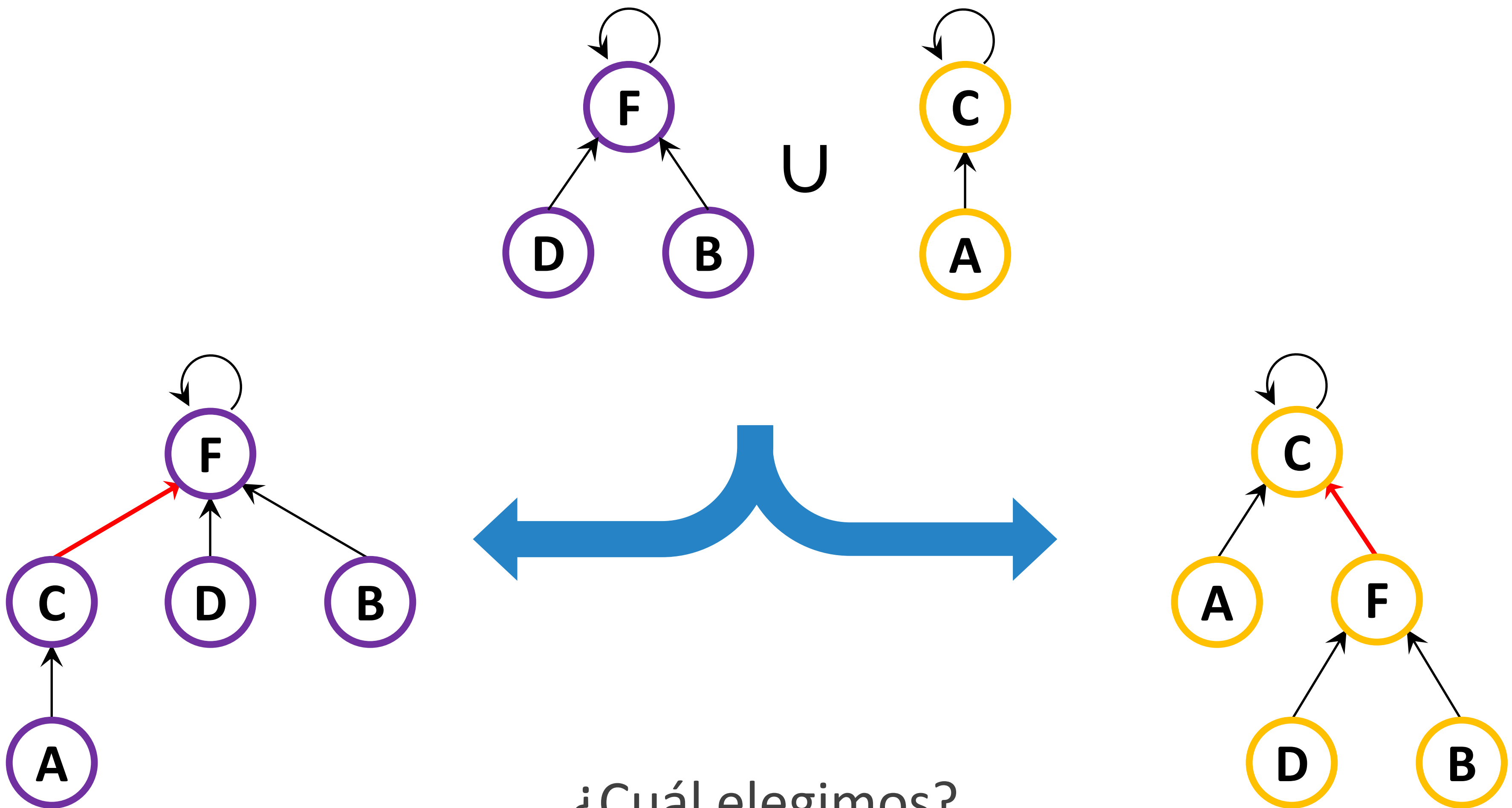
Operaciones sobre conjuntos disjuntos

Definimos 3 funciones para esta estructura:

- $\boldsymbol{make\ set}(\boldsymbol{x})$: inicializa $\boldsymbol{x}$ como su propio representante —cada $\boldsymbol{x}$ está en un conjunto por sí solo inicialmente
- $\boldsymbol{find\ set}(\boldsymbol{x})$: retorna el representante del nodo $\boldsymbol{x}$ —el conjunto al que pertenece $\boldsymbol{x}$
- $\boldsymbol{union}(\boldsymbol{x}, \boldsymbol{y})$: une los conjuntos a los que pertenecen $\boldsymbol{x}$ e $\boldsymbol{y}$ —quedando sólo la unión y desapareciendo los conjuntos originales

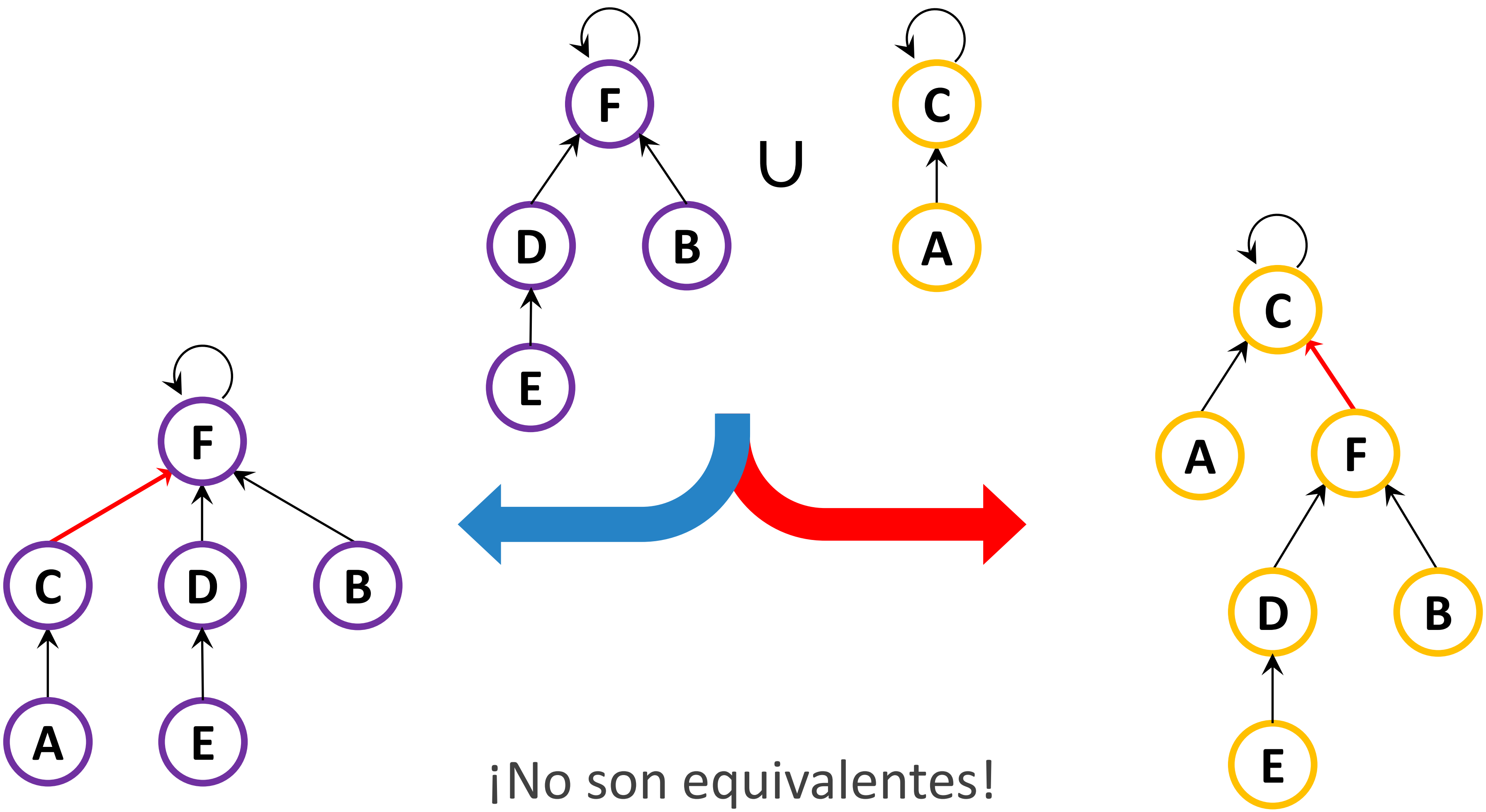
Todas son bastante directas, y pueden implementarse eficientemente; ¿cómo?

Unión



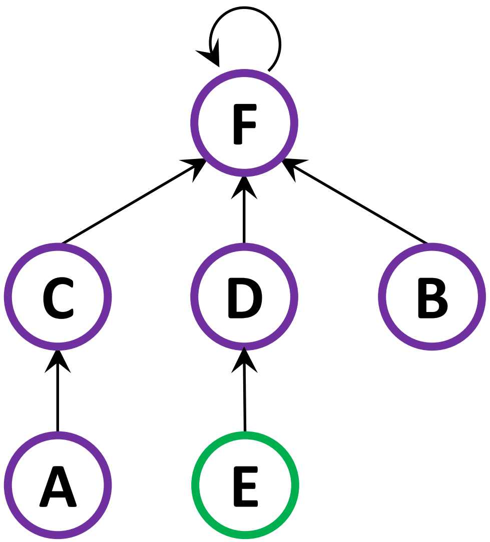
¿Cuál elegimos?

Unión

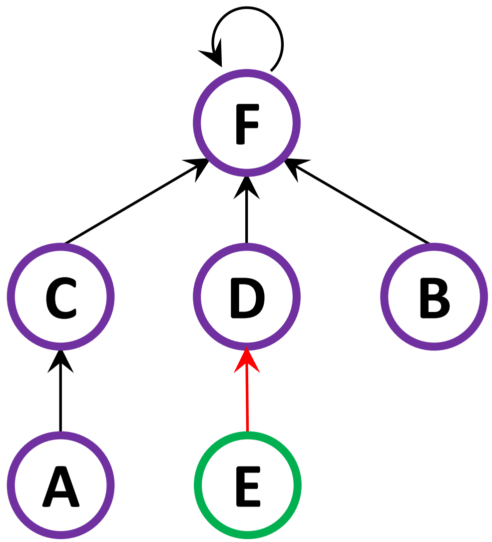


¡No son equivalentes!

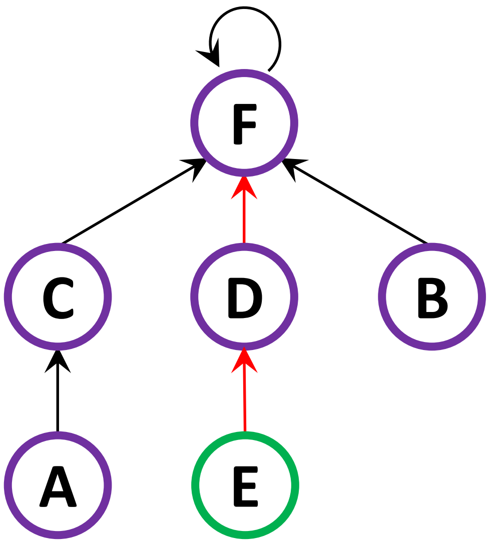
find-set(E) = ...



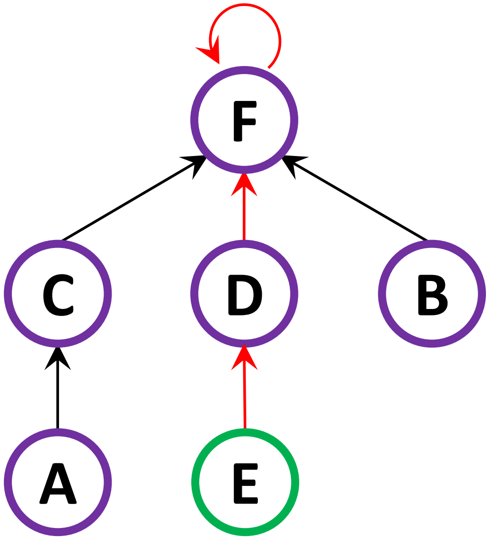
find-set(E) = find-set(D)



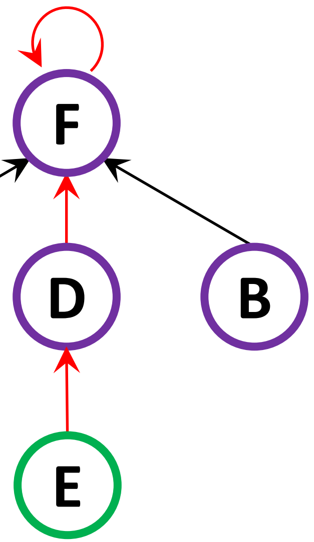
find-set(E) = find-set(F)



find-set(E) = F

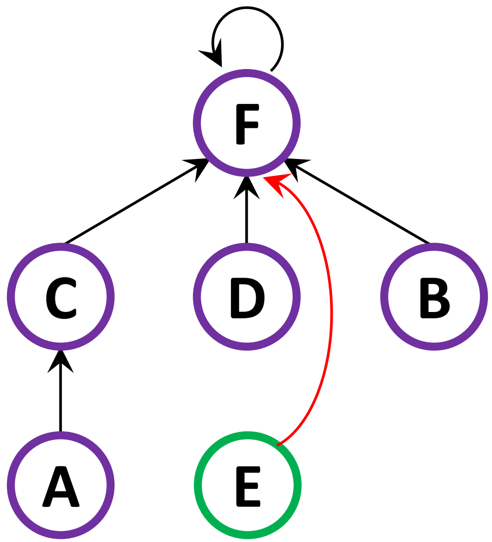


¿Cómo podemos aprovechar esta información una vez que la tenemos?



$$\boldsymbol{find\ set(E)} = \mathrm{F}$$

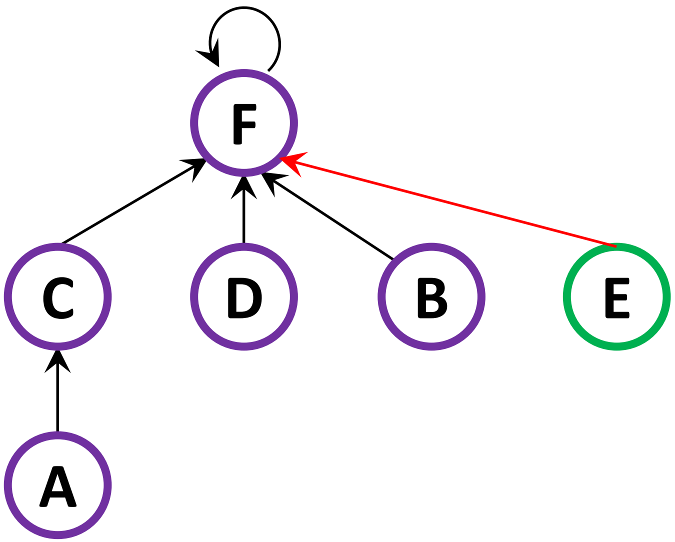
Compresión de caminos



$\boldsymbol{find\ set(E)} =$ F

¡Acortando el camino al representante!

Compresión de caminos



$$\boldsymbol{find\ set}(\boldsymbol{E}) = \text{F}$$

Complejidad de `kruskal`

Si pretendemos operar sobre n conjuntos disjuntos

… ¿cuál es la complejidad de estas operaciones?

… ¿y usando las mejoras?

`kruskal` con conjuntos disjuntos
(como los acabamos de ver)

$\boldsymbol{kruskal}\big(\boldsymbol{G}(\boldsymbol{V},\boldsymbol{E})\big)$:

Ordenar $\boldsymbol{E}$ por costo, de menor a mayor

$\boldsymbol{foreach}\ \boldsymbol{v} \in \boldsymbol{V}$: $\boldsymbol{make\ set}(\boldsymbol{v})$

$\boldsymbol{T} \leftarrow \emptyset$

$\boldsymbol{foreach}\ (\boldsymbol{u},\boldsymbol{v}) \in \boldsymbol{E}$:

$\boldsymbol{if}\ \boldsymbol{find\ set}(\boldsymbol{u}) \neq \boldsymbol{find\ set}(\boldsymbol{v})$:

$\boldsymbol{T} \leftarrow \boldsymbol{T} \cup \{(\boldsymbol{u},\boldsymbol{v})\}$

$\boldsymbol{union}(\boldsymbol{u},\boldsymbol{v})$

$\boldsymbol{return}\ \boldsymbol{T}$

¿Cuál es la complejidad de `kruskal` con cojuntos disjuntos y compresión de caminos?

Primero, hay que ordenar las $|E|$ aristas → O(Elog$E$)

Luego, hay que construir $|V|$ conjuntos (de un elemento cada uno) → O(V)

Durante la ejecución del segundo *loop*, se realizan $|V|-1$ uniones

… y $2|E|$ operaciones *find set*

Cada operación *union* toma O(1) → O(V) para el total de $|V|-1$ operaciones *union*

¿Cuánto toman en total las $2|E|$ operaciones *find set*?

¿Cuánto toman en total las $2|E|$ operaciones *find set*?

La complejidad de una operación *find set* depende de a cuál elemento se aplica

… aunque en el largo plazo todos los árboles podrían terminar teniendo profundidad 1, si hay suficientes operaciones *find set*

Se puede demostrar que el costo promedio de una operación *find set* en un conjunto de *n* elementos es $O(\log^* n)$

… en que log* es el número de veces que $\log_2$ tiene que ser aplicado iterativamente hasta que el resultado sea ≤ 1

P.ej., leyendo de derecha a izquierda

$$0.54 = \log_2(1.45 = \log_2(2.73 = \log_2(6.64 = \log_2(100))))$$

… de modo que $\log^*(100) = 4$

La función log* crece muy lentamente

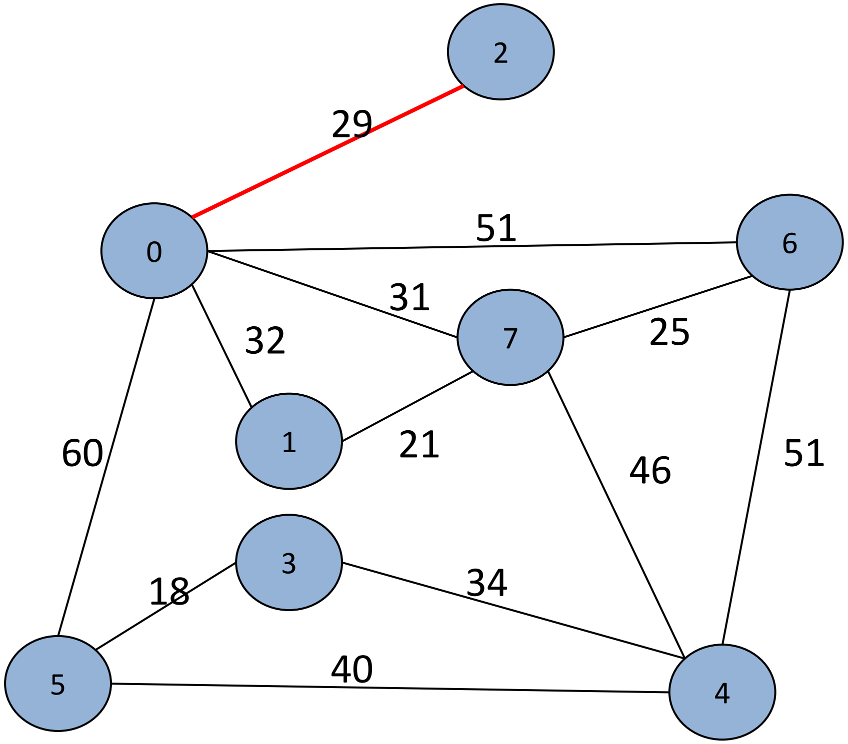
El n más pequeño para el cual log*n es 5 es $n = 2^{16} = 65536$

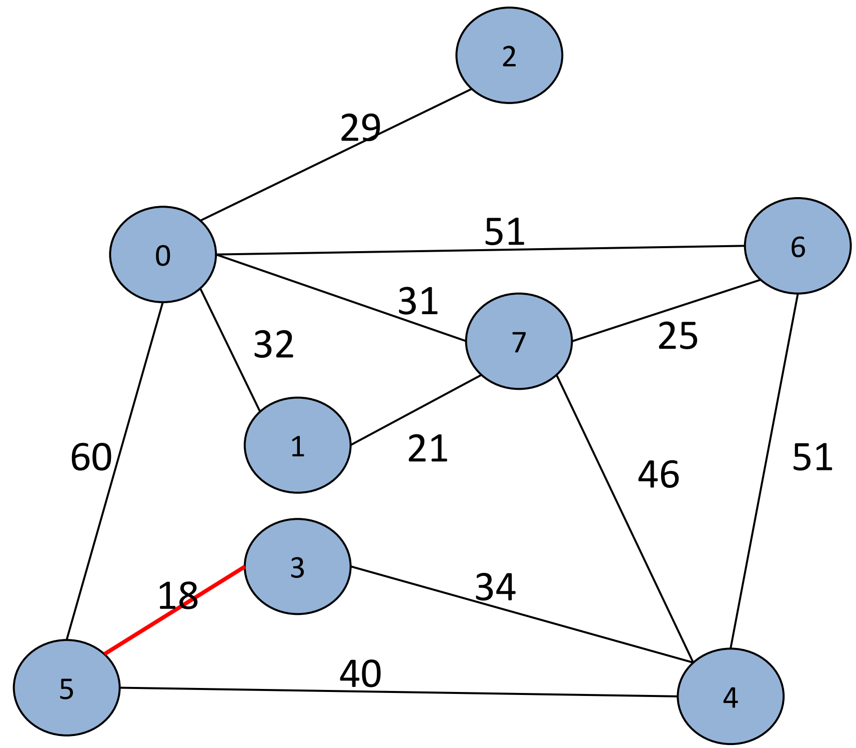
… y va a quedarse en 5 para todos los números razonables (hasta 2^{65536})

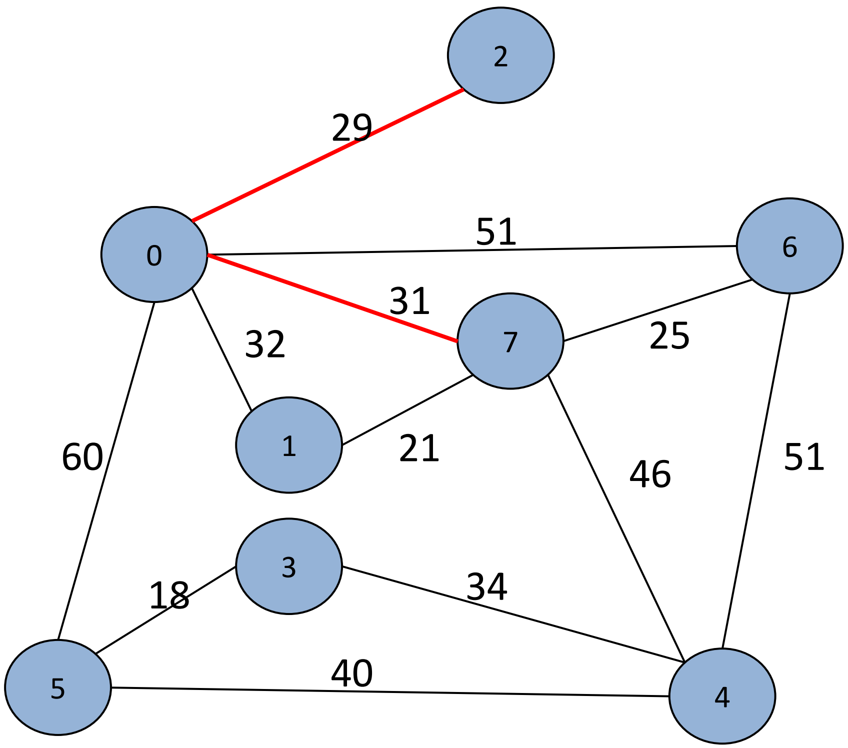
(→ Para cualquier uso práctico, consideramos que log*n es casi constante, aunque teóricamente tiende a ∞)

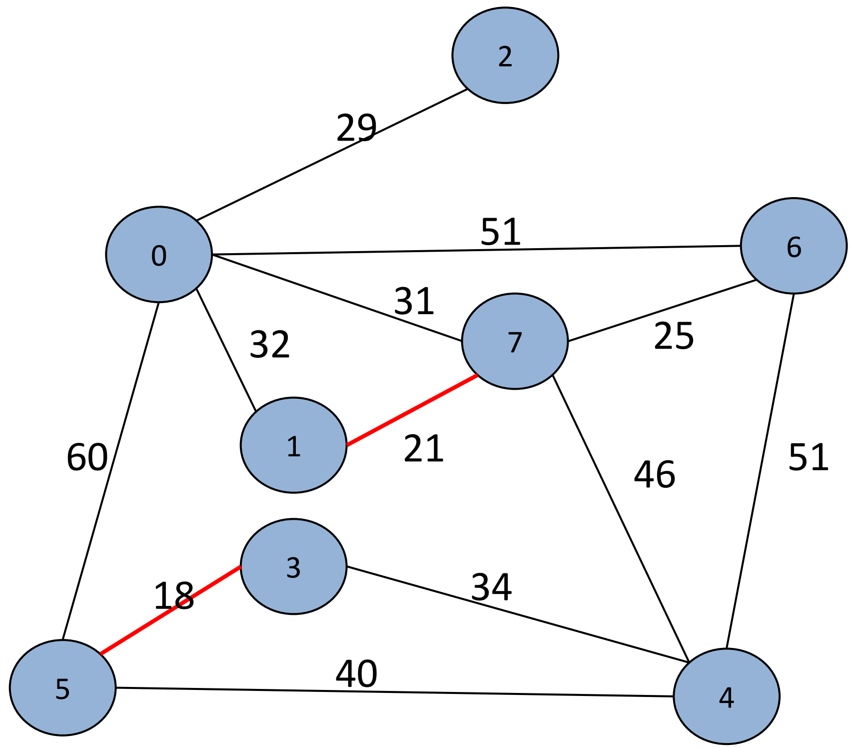
Así, las 2|E| operaciones *find set* toman O(Elog*E)

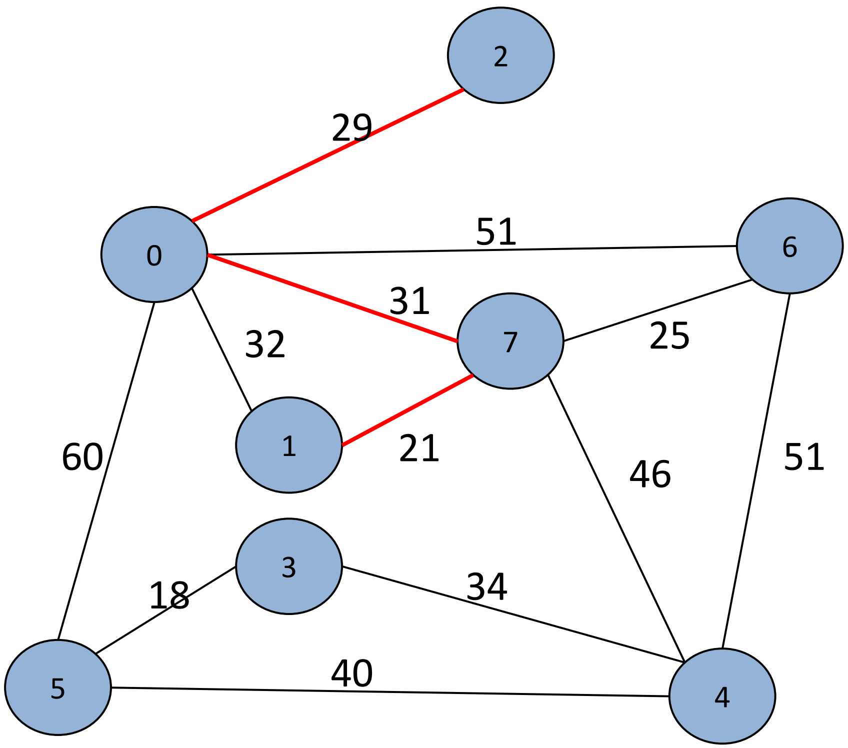
… y la complejidad de *kruskal* es O(Elog$E$) + O(V) + O(Elog*E) = O(Elog$E$) = **O(*E*log*V*)**, ya que |E| = O(V^2)

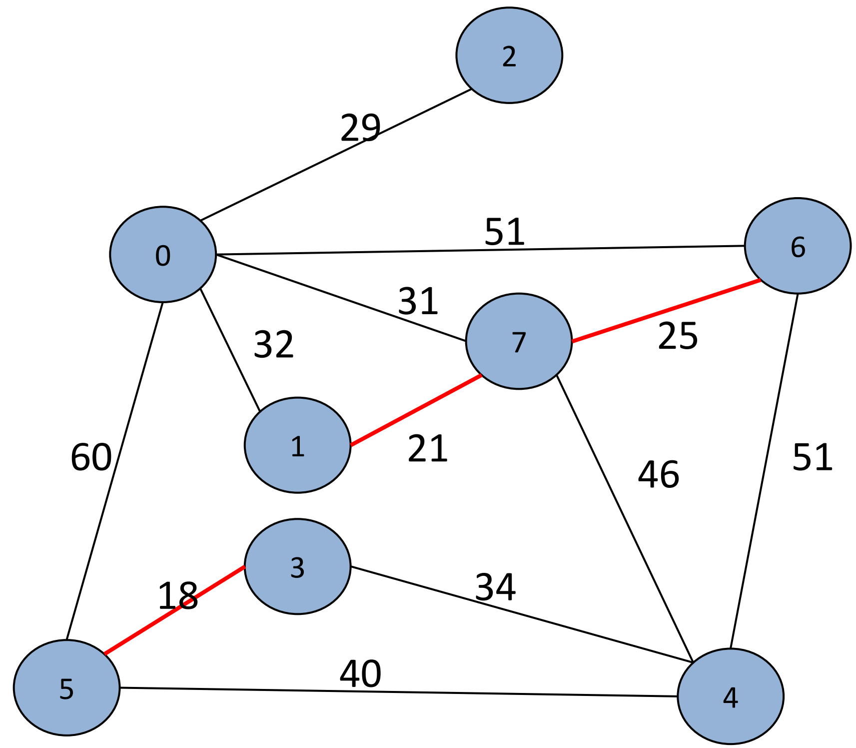

2
29
0
51
6
31
7
25
32
1
21
60
46
51
3
34
18
5
40
4

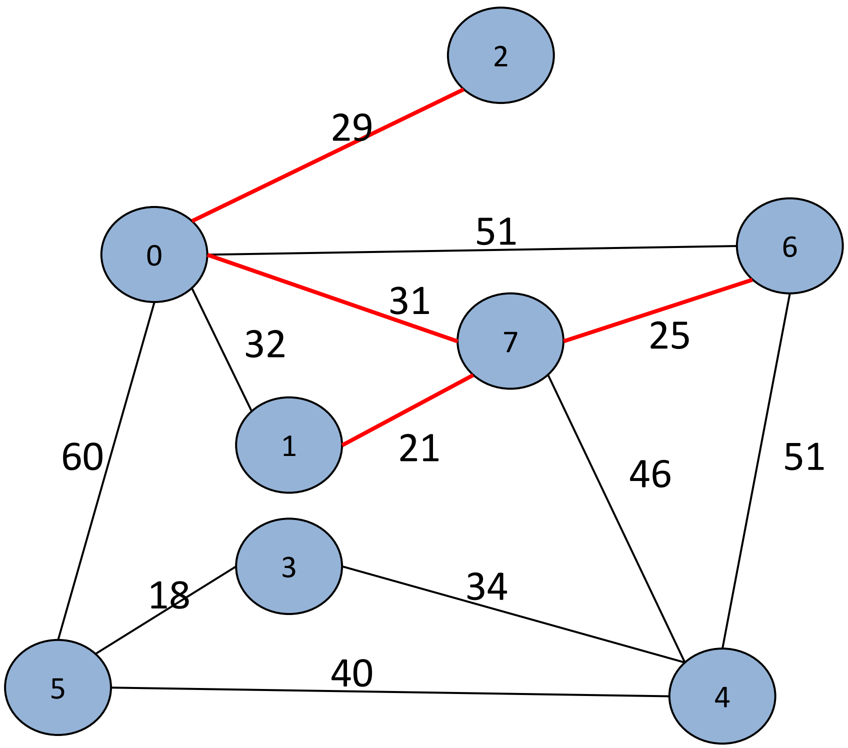

2
29
0
51
6
31
7
25
32
1
21
60
46
51
3
34
18
5
40
4

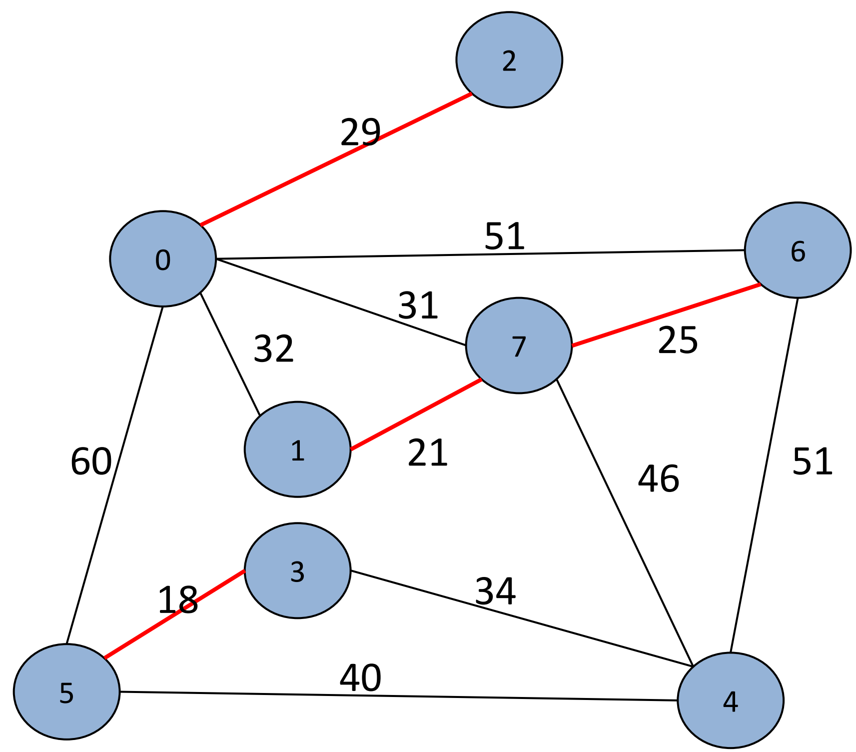

2
29
0
51
6
31
7
32
25
1
21
60
46
51
3
18
34
5
40
4

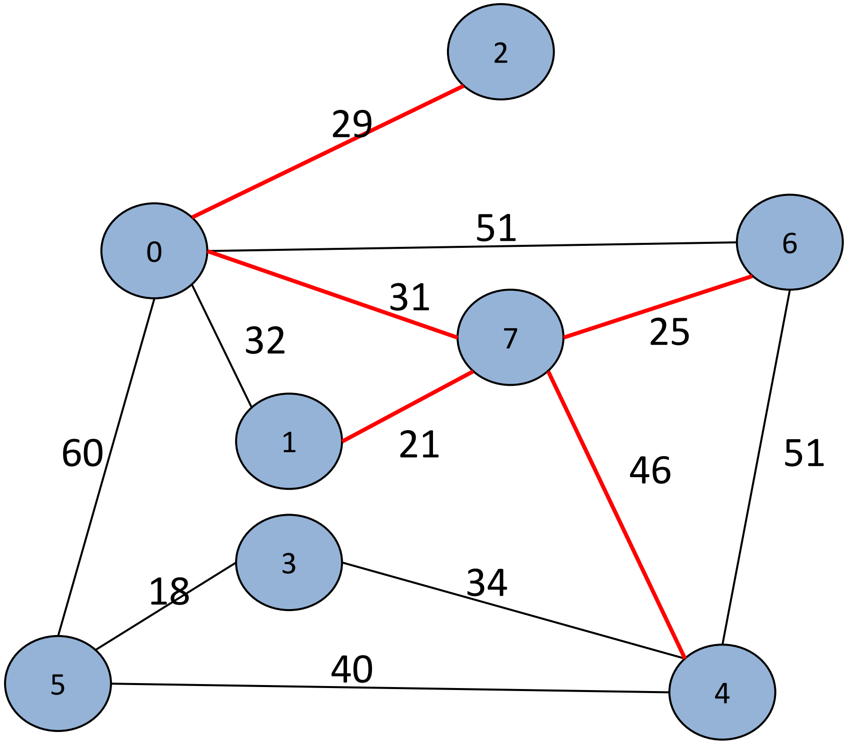

2
29
0
51
6
31
7
32
25
1
21
60
46
51
3
18
34
5
40
4

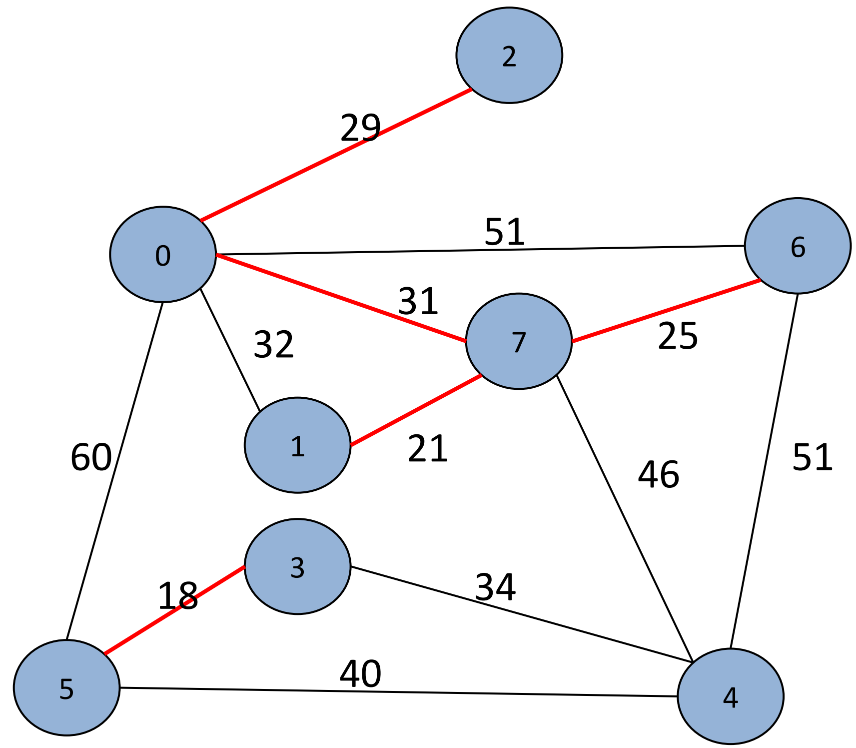

2
29
0
51
6
31
7
32
25
1
21
60
46
51
3
18
34
5
40
4

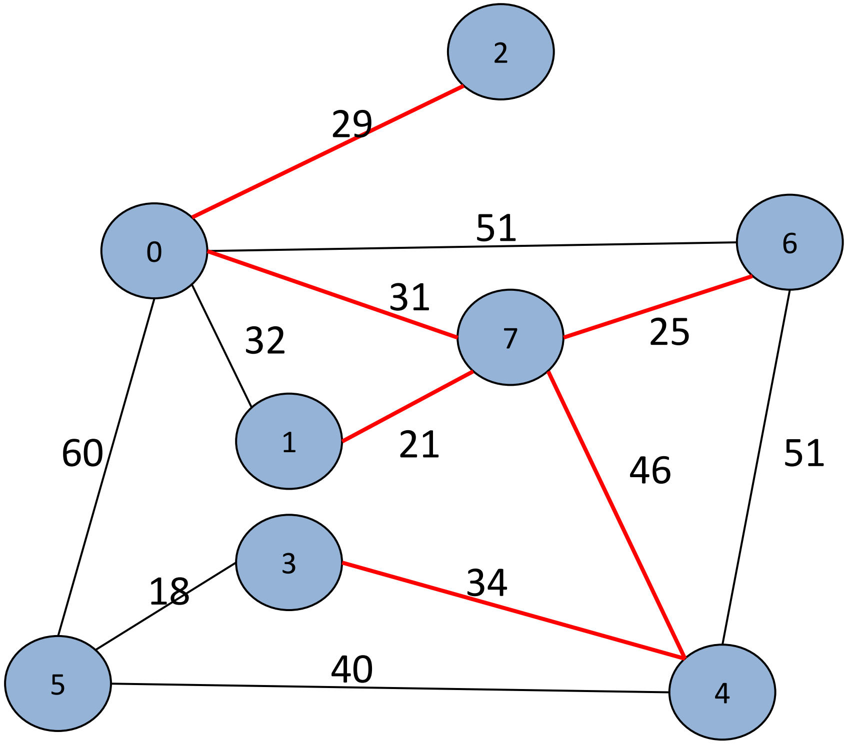

2
29
0
51
6
31
7
32
25
1
21
60
46
51
3
18
34
5
40
4

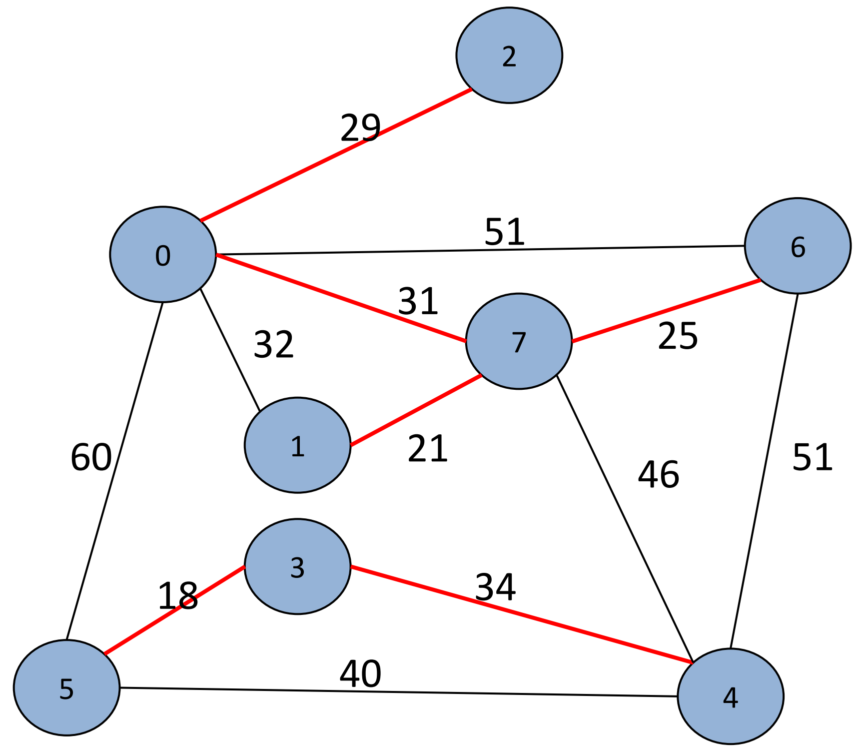

2
29
0
51
6
31
7
25
32
1
21
60
46
51
3
18
34
5
40
4

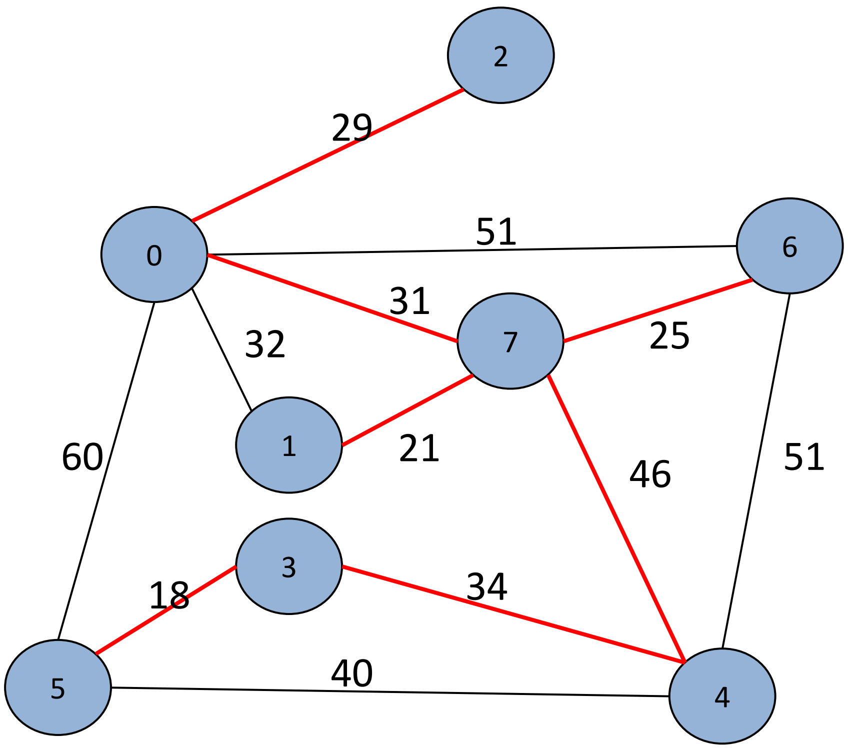

2
29
0
51
6
31
7
25
32
1
21
60
46
51
3
18
34
5
40
4

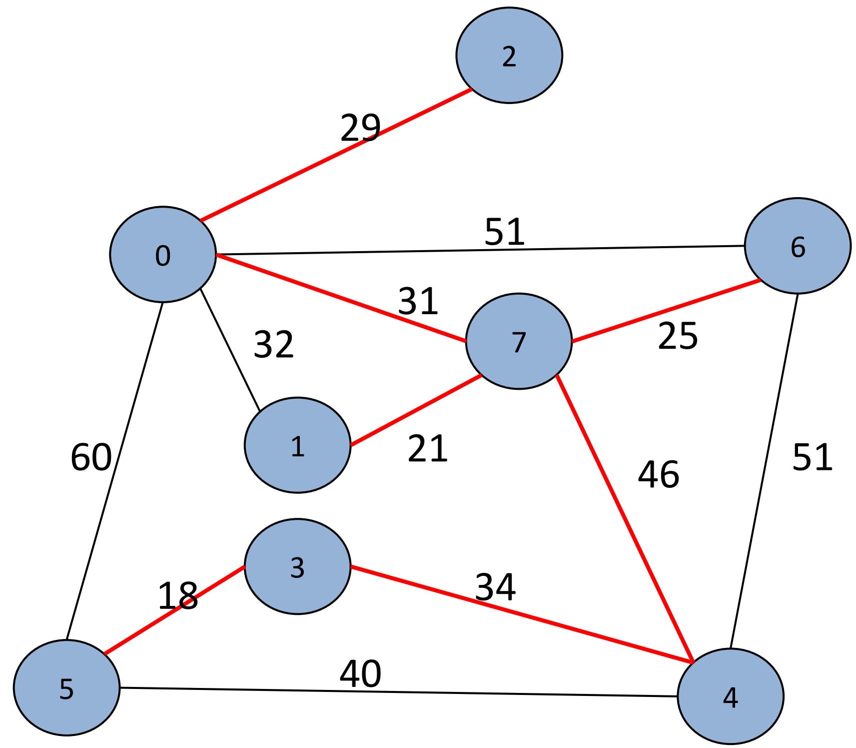

2
29
0
51
6
31
7
32
25
1
21
60
46
51
3
18
34
5
40
4


2
29
0
51
6
31
7
32
25
1
21
60
46
51
3
18
34
5
40
4


2
29
0
51
6
31
7
25
32
1
21
60
46
51
3
18
34
5
40
4


2
29
0
51
6
31
7
25
32
1
21
60
46
51
3
18
34
5
40
4


2
29
0
51
6
31
7
25
32
1
21
60
46
51
3
34
18
5
40
4


2
29
0
51
6
31
7
25
32
1
21
60
46
51
3
34
18
5
40
4