Tres problemas en grafos *con costos*

a) Grafos no direccionales:

- encontrar el *árbol de cobertura de costo mínimo*

b) Grafos direccionales:

- encontrar la *ruta más corta desde un vértice a todos los otros*

c) Grafos direccionales:

- encontrar las *rutas más cortas entre todos los pares de vértices*

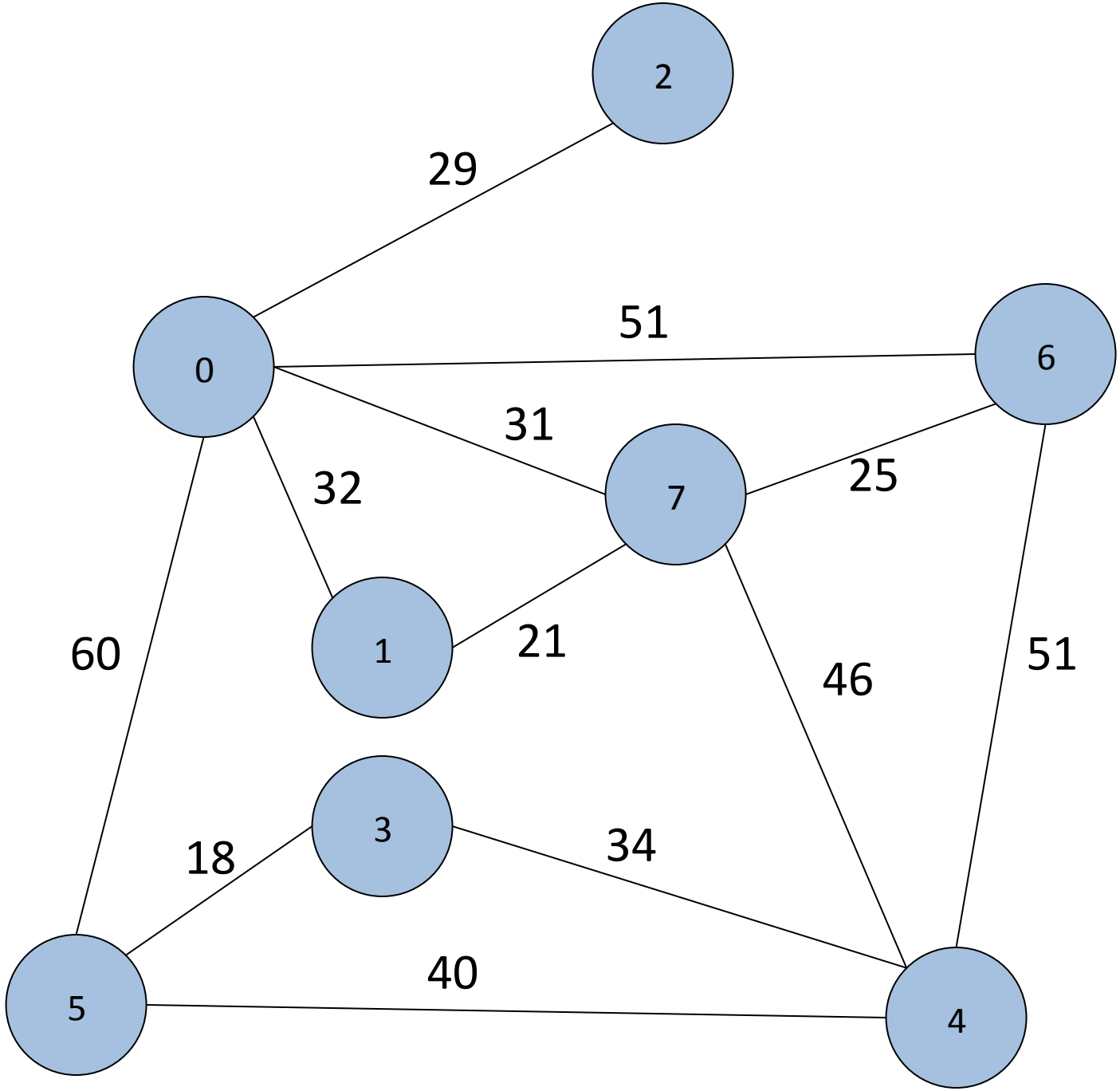
Árboles de extensión de costo mínimo (MST's)

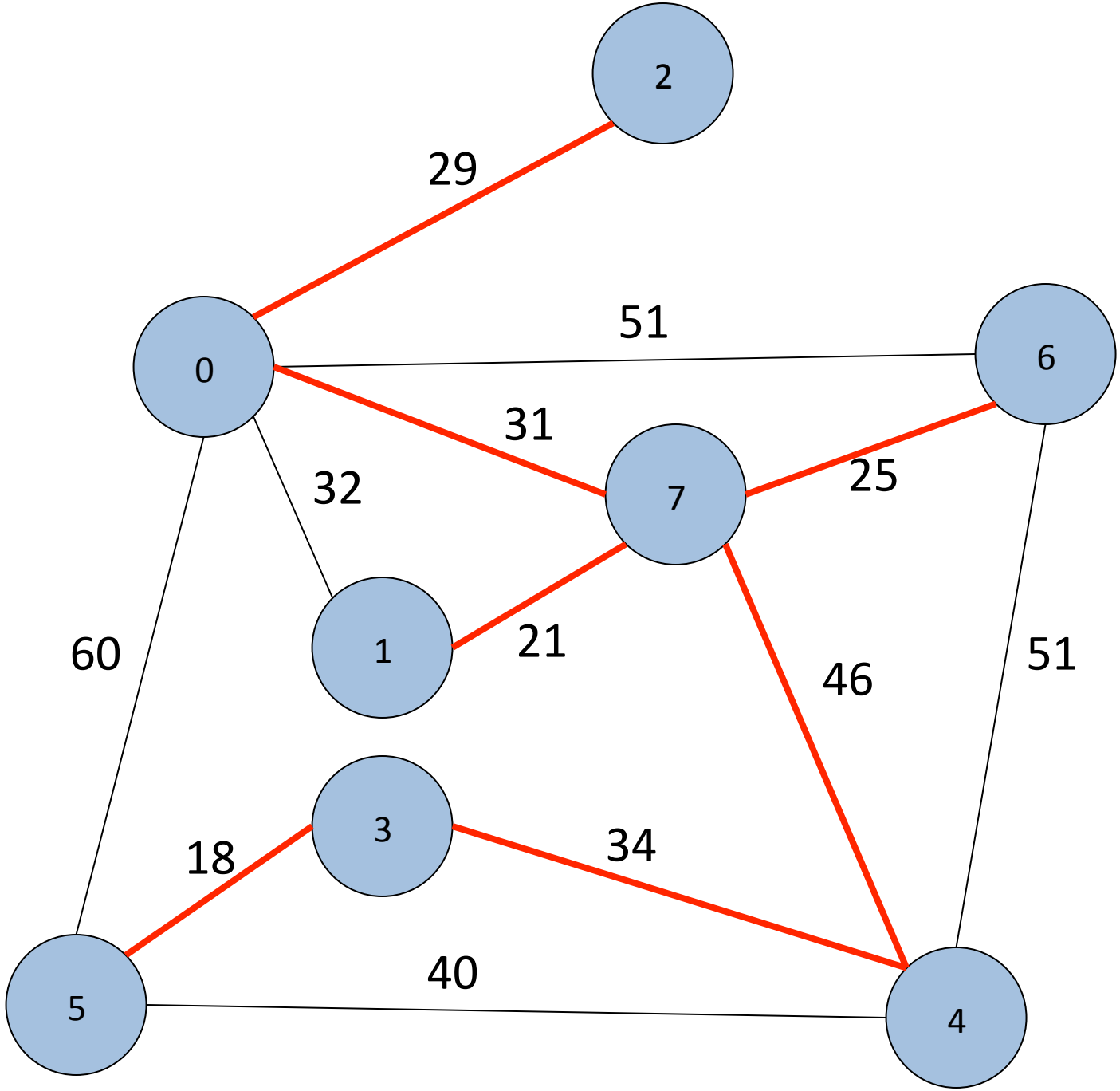
Dado un grafo conectado no direccional $G = (V, E)$ y una función de costo $w : E \rightarrow R$, buscamos un subconjunto acíclico $T \subseteq E$ de aristas que conecte todos los vértices de G y cuyo costo total —la suma de los costos de las aristas de T— sea mínimo:

- el costo total está dado por

  $w(T) = \Sigma(u, v)_{(u, v) \in T} w(u, v)$

- como T es acíclico y conecta todos los vértices, debe formar un árbol — **árbol de extensión de costo mínimo** (MST)


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


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

Los algoritmos construyen el MST de a una arista a la vez, empleando un enfoque codicioso

Mantenemos un conjunto A que es siempre un subconjunto de algún MST

En cada paso, encontramos una arista (u, v) que puede agregarse a A, tal que $A \cup \{(u, v)\}$ es un subconjunto de un MST —(u, v) es una **arista segura** para A

$A = \emptyset$
`while` *A no forma un árbol de extensión*:
    encuentre una arista (u, v) *segura para* A
    $A = A \cup \{(u, v)\}$
`return` A

La parte difícil del algoritmo es encontrar la *arista segura*

Un **corte** (*S*, *V*–*S*) de *G* es una partición de *V*

Una arista **cruza** el corte si uno de sus vértices está en *S* y el otro en *V*–*S*

Un corte **respeta** al conjunto *A* de aristas si ninguna arista de *A* cruza el corte

Una arista que cruza un corte es **liviana** si su costo es el menor de cualquier arista que cruza el corte

¿Cómo encontramos aristas seguras?

Sean:

A un subconjunto de E incluido en algún MST de G

$(S, V–S)$ un corte de G que respeta a A

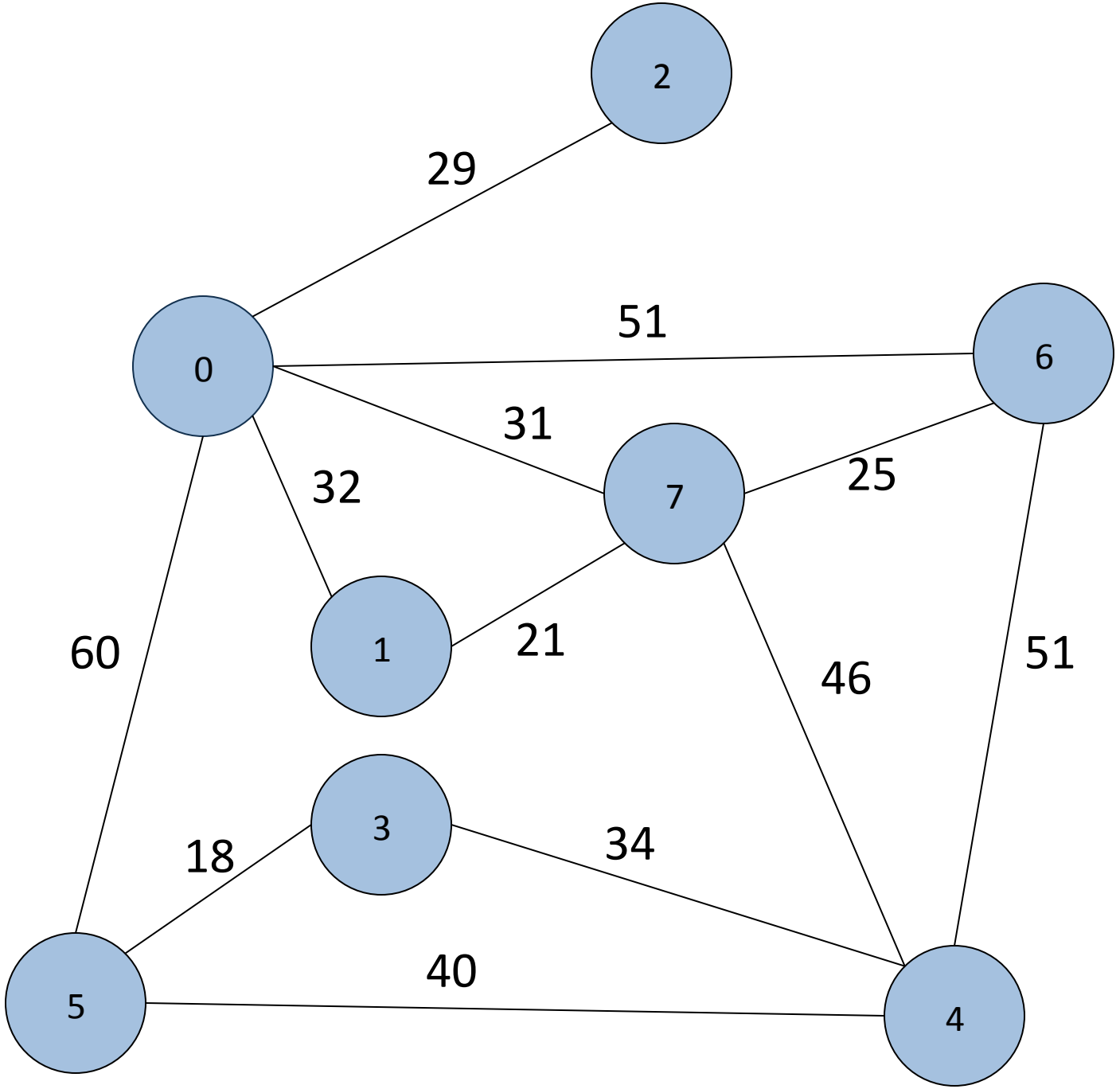
(u, v) una arista liviana que cruza el corte $(S, V–S)$

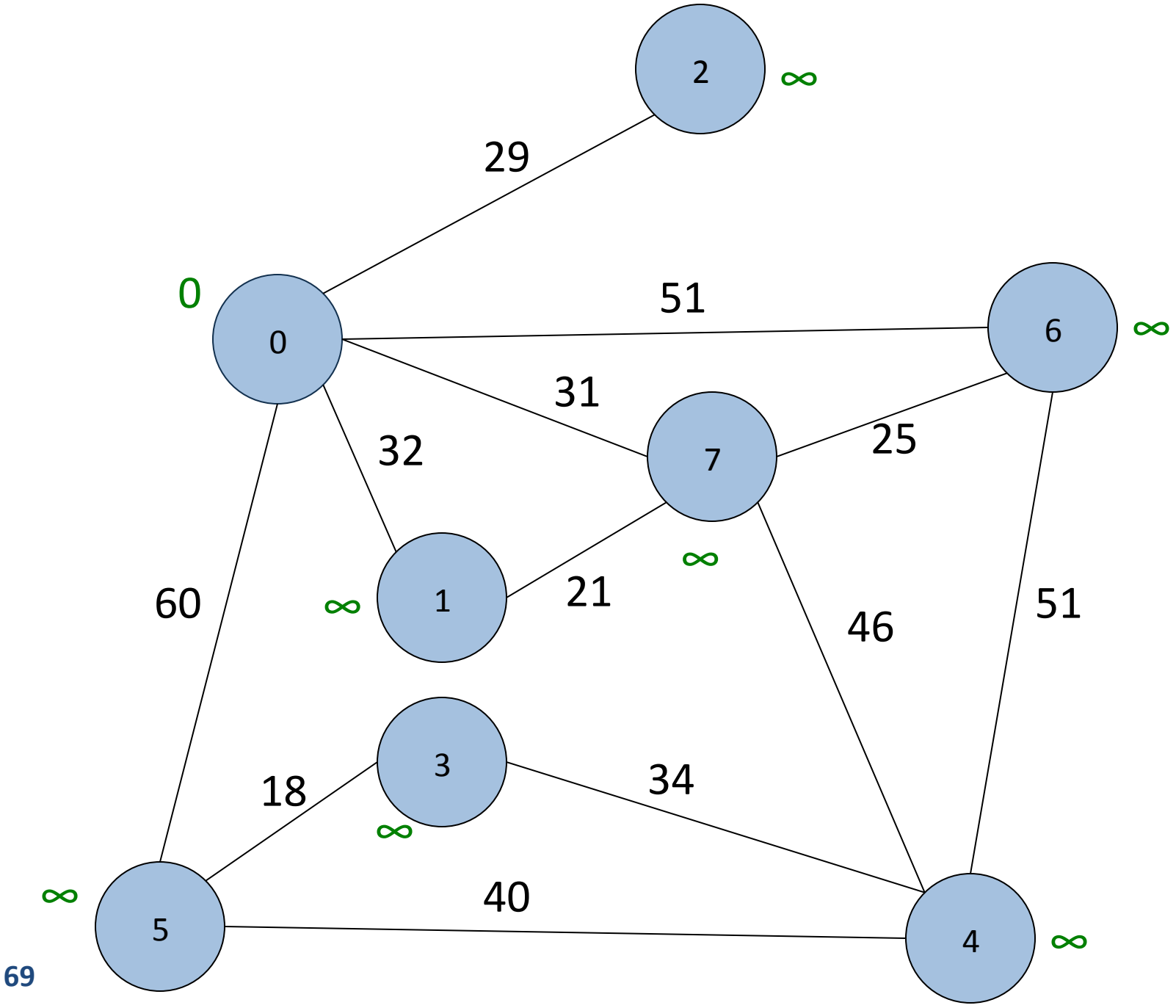
Entonces (u, v) es segura para A

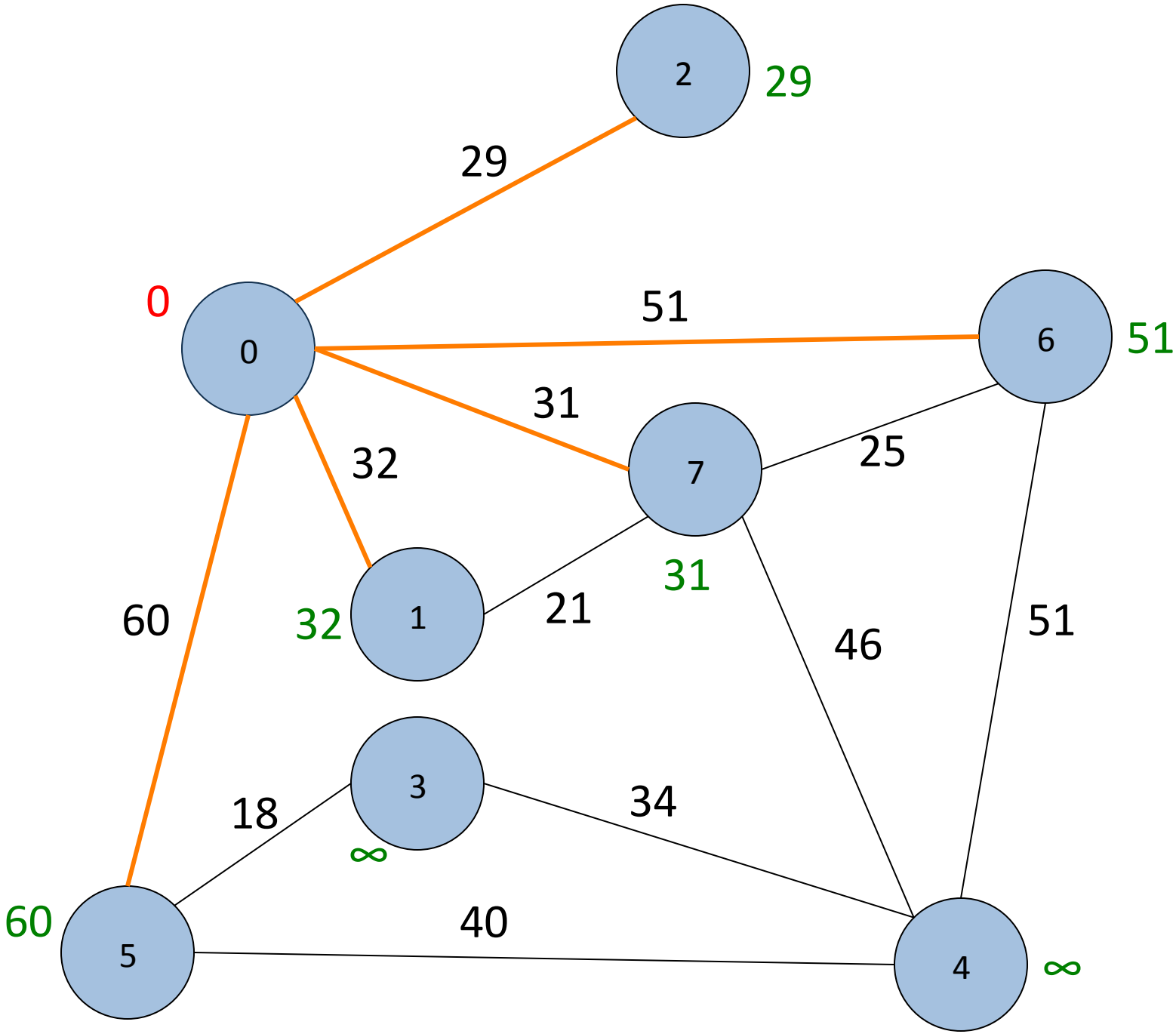
El *algoritmo de Prim* es un caso especial del algoritmo de la diap. #64

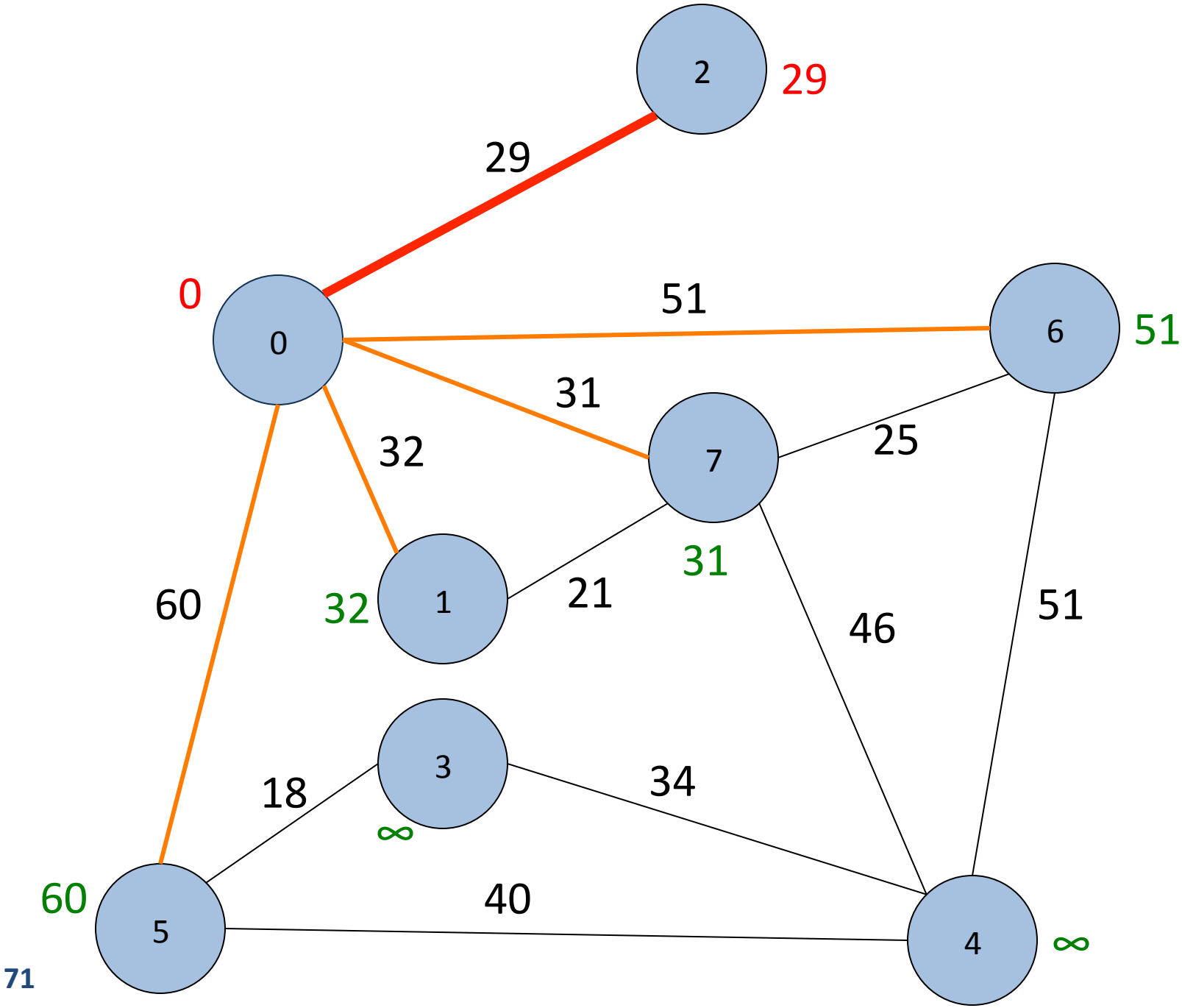
Las aristas en *A* forman un único árbol, que comienza a partir de un vértice arbitrario *r* y crece hasta incluir todos los vértices de *V*

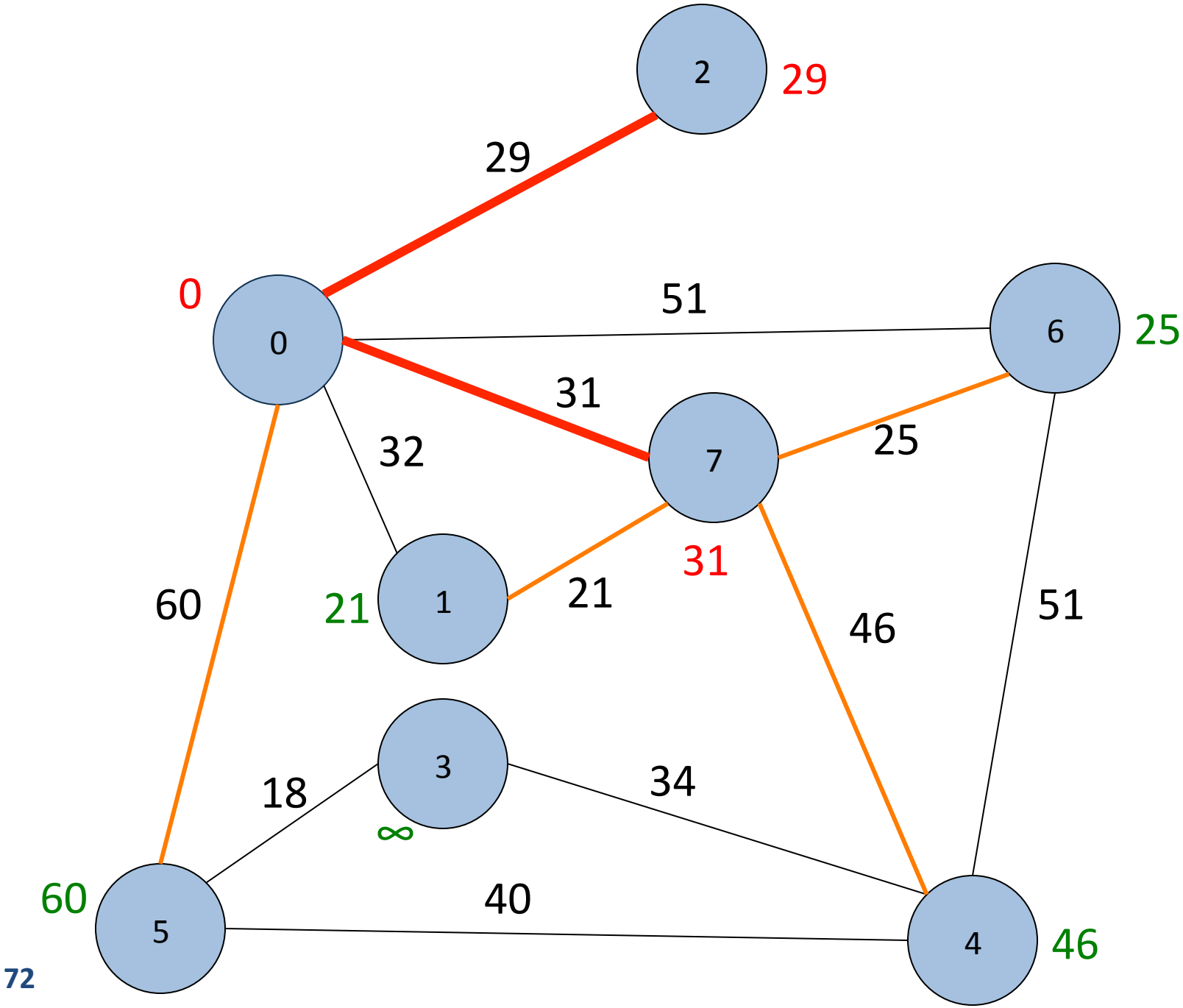
En cada paso, se agrega al árbol una arista liviana que conecta un vértice en *A* con un vértice en *V*–*A*

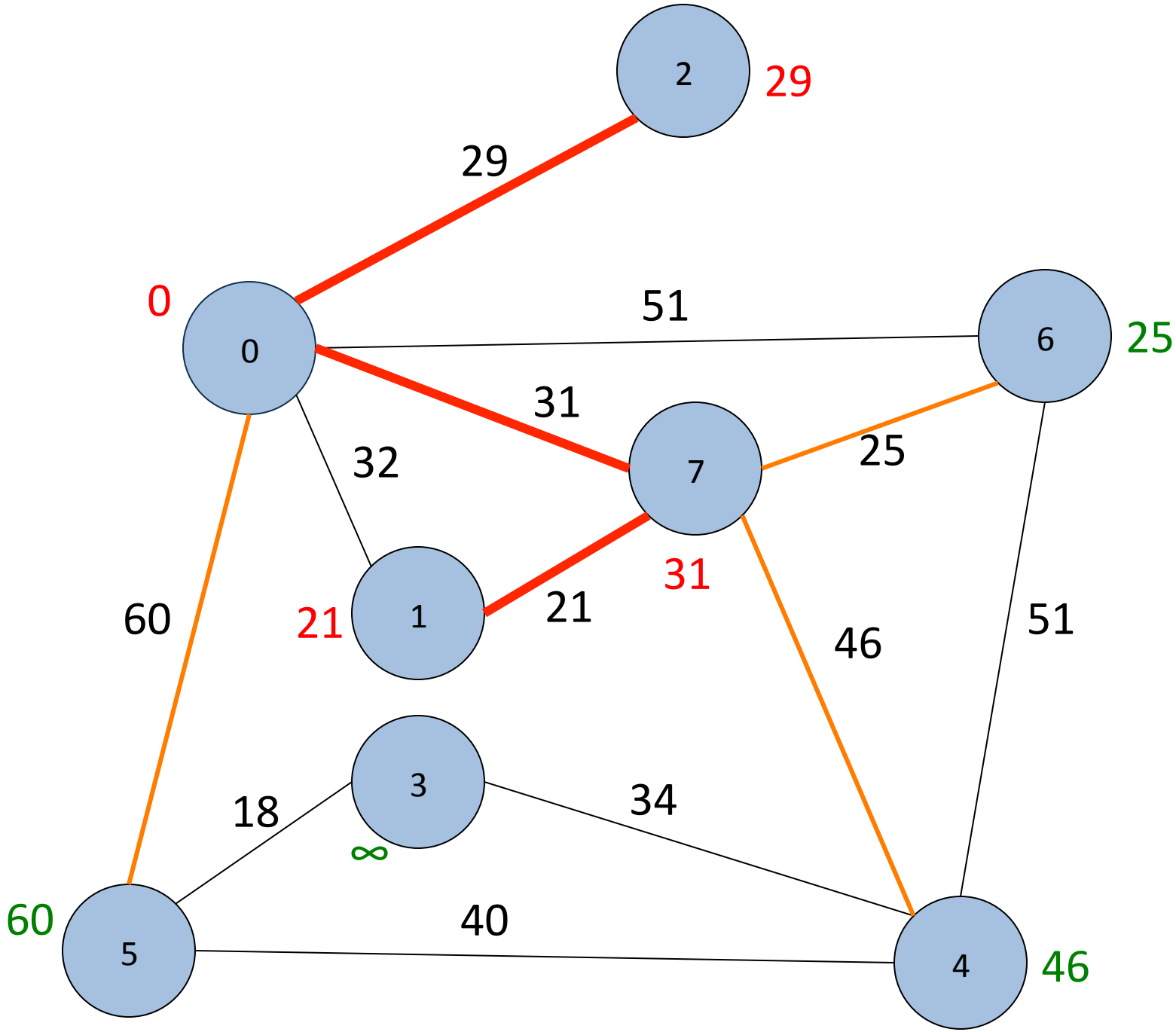

2
29
0
51
6
31
32
7
25
1
21
60
46
51
3
18
34
40
5
4

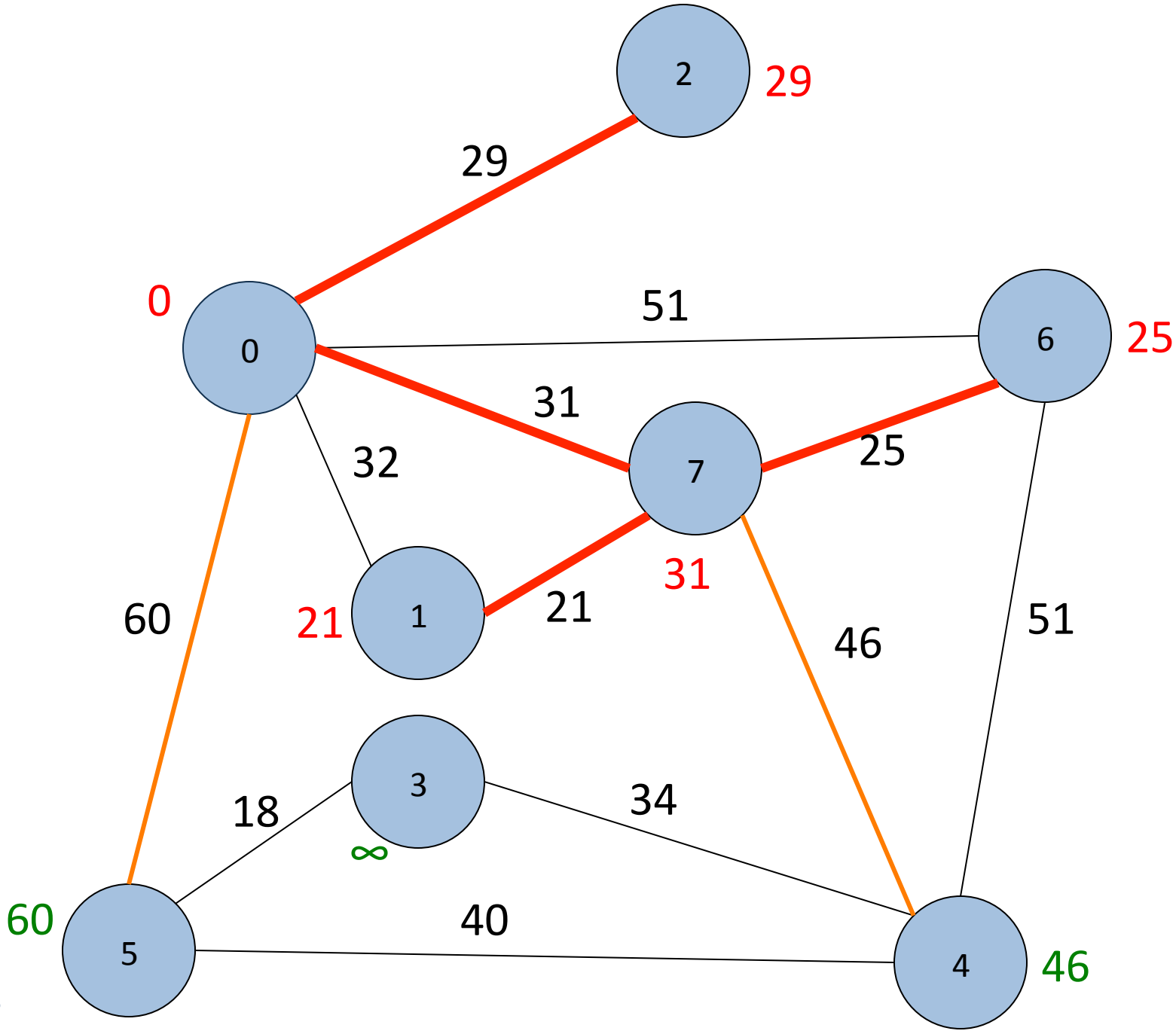

2
∞
29
0
0
51
6
∞
31
32
7
25
∞
60
∞
1
21
46
51
3
18
34
∞
∞
5
40
4
∞

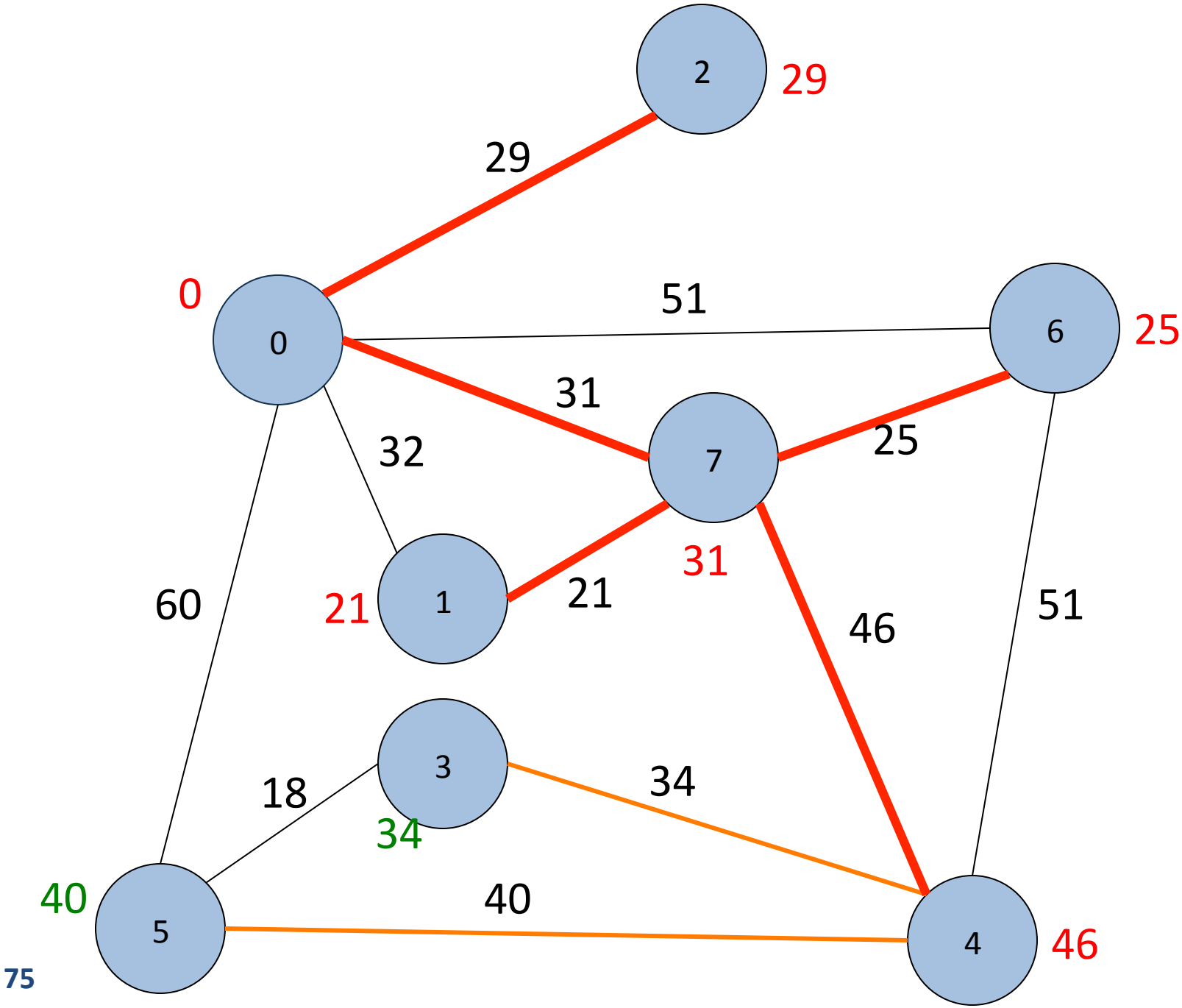

2
29
29
0
0
51
6
51
31
32
7
25
31
60
32
1
21
46
51
3
18
34
∞
60
5
40
4
∞

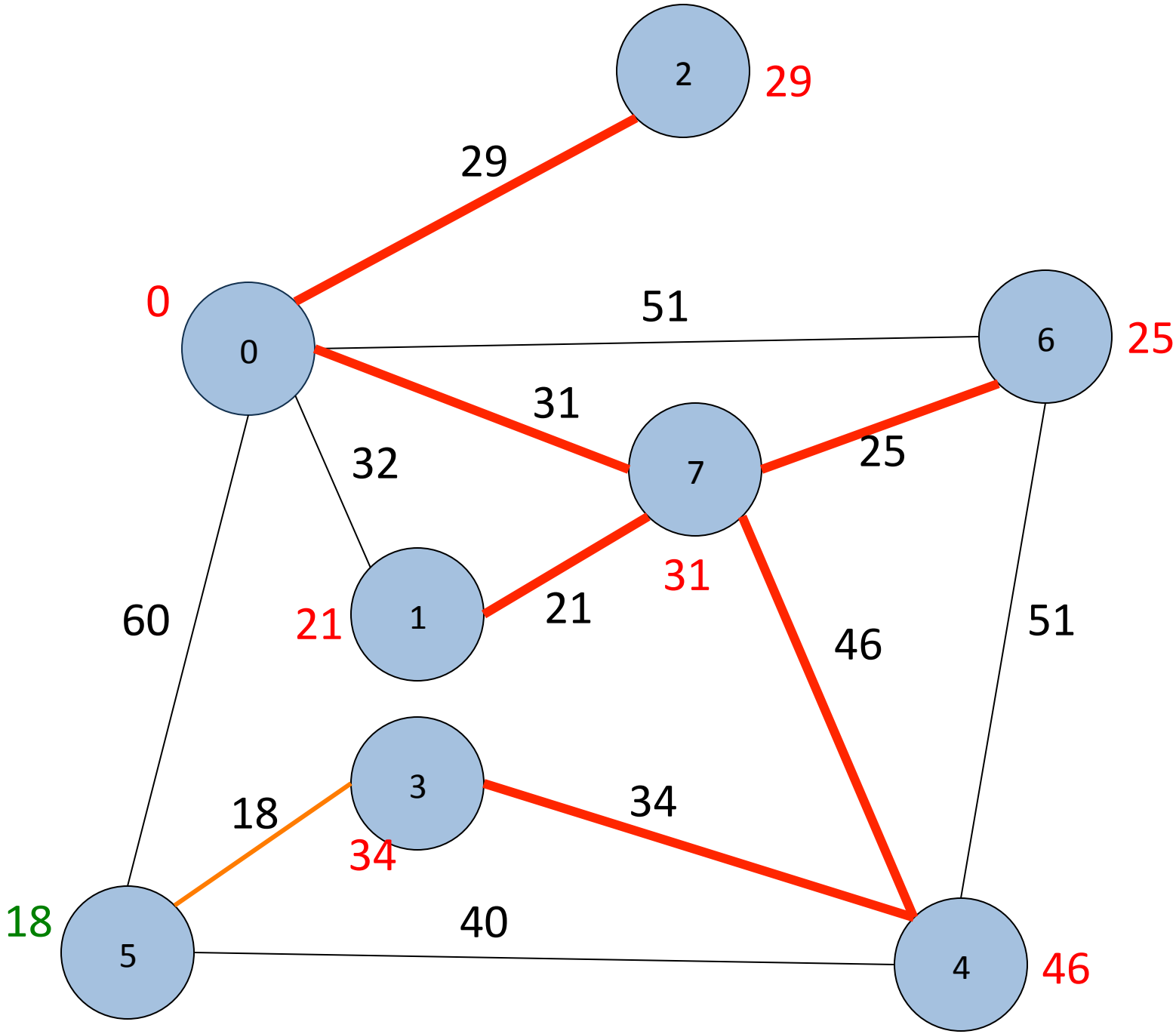

2
29
29
0
0
51
6
51
31
32
7
25
31
60
32
1
21
46
51
3
18
34
∞
60
5
40
4
∞

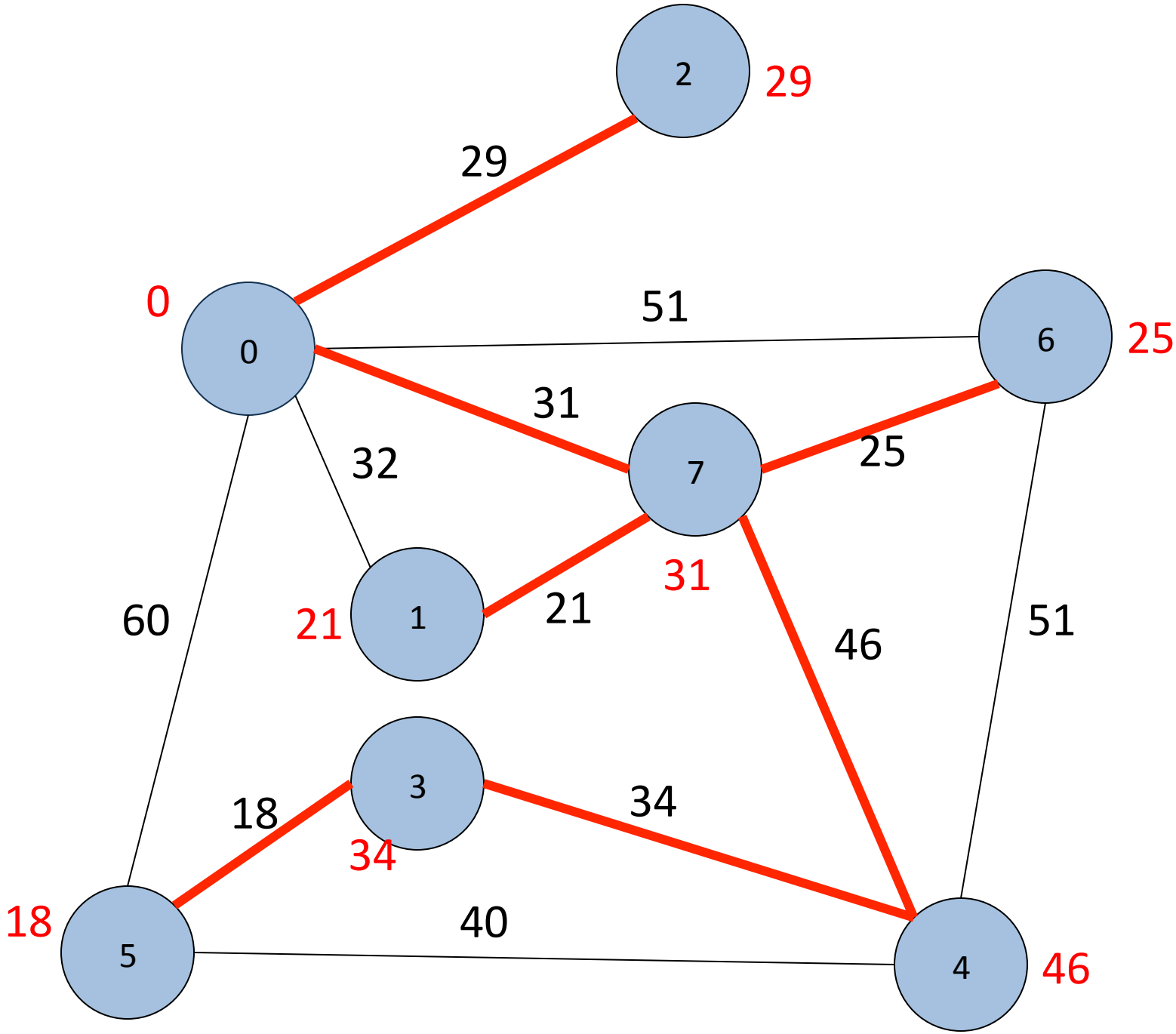

2
29
29
0
0
51
6
25
31
32
7
25
31
60
21
1
21
46
51
3
18
34
∞
60
5
40
4
46


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


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


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


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


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

El algoritmo de Prim


```
prim(r)
    q = Queue(V)
    for each u in q:
        u.key = ∞
    r.key = 0
    π[r] = null
    while !q.empty():
        u = q.xMin()
        for each v in α[u]:
            if v ∈ q ∧ w(u,v) < v.key:
                π[v] = u
                v.key = w(u,v)
```

La clave de `prim()` está en la selección de una nueva arista a ser agregada al árbol *A*

Todos los vértices que no están en el árbol están en una cola *q* priorizada según un atributo *key*

v.key es el mínimo de los costos de cual-quier arista que conecta a *v* con un vértice en el árbol

El conjunto *A* es $\{ (v, \pi(v)) : v \in V - \{r\} - q \}$, en que $\pi(v)$ es el padre de *v* en el árbol

El desempeño de `prim()` depende de cómo implementamos la cola *q*

Si *q* es un heap binario, lo inicializamos en tiempo O(*V*)

El tiempo total para todas las llamadas a `xMin()` en el `while` es O (*V*log*V*) —se hacen |*V*| llamadas y cada una toma tiempo O(log*V*)

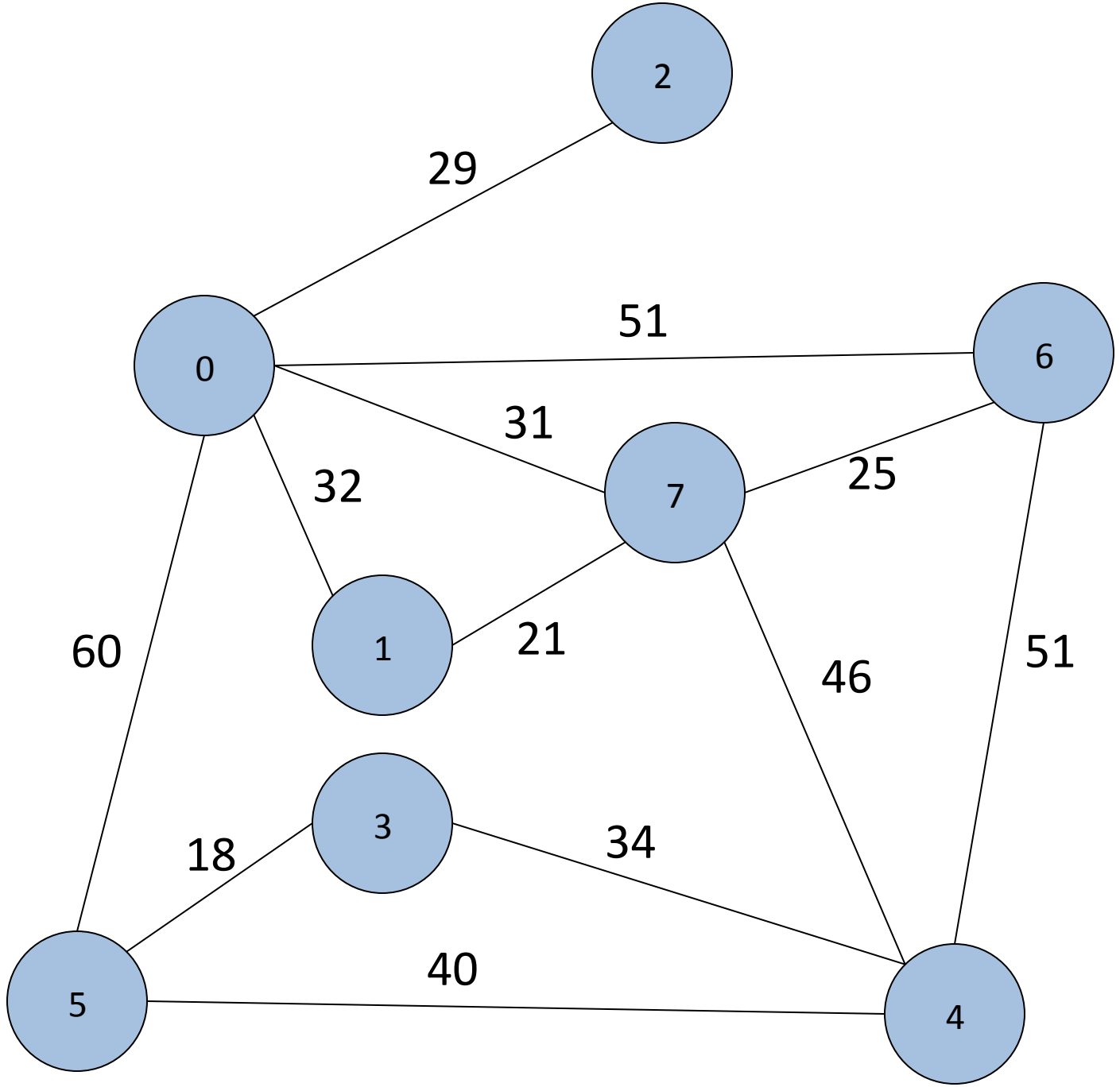
El tiempo para todas las asignaciones al atributo *key* en el `for` es O (*E*log*V*) —cada asignación implica una operación `decreaseKey()` y hay O(*E*) asignaciones

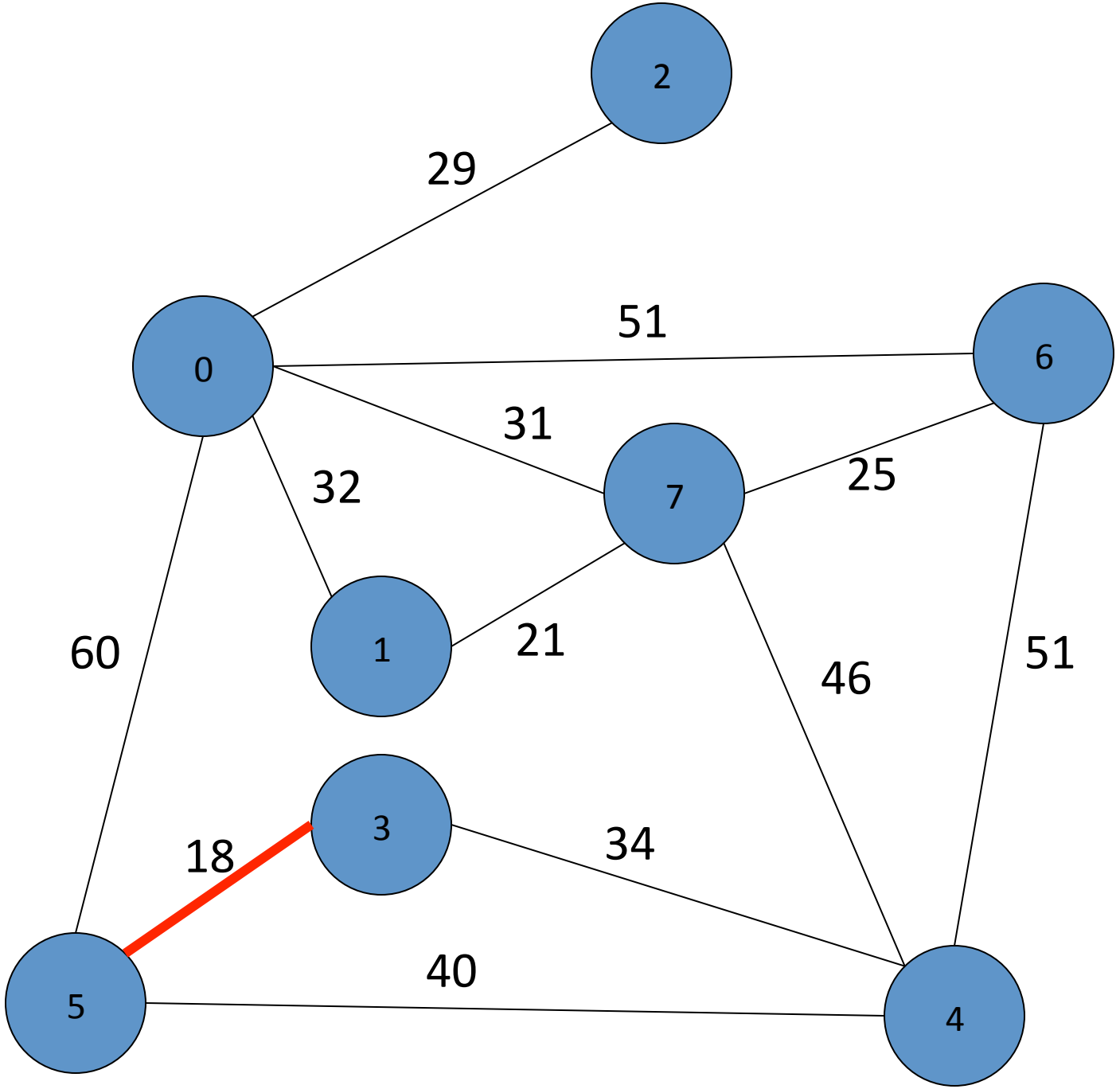
En total, `prim()` toma tiempo O(*V*log*V*+*E*log*V*) = O(*E*log*V*)

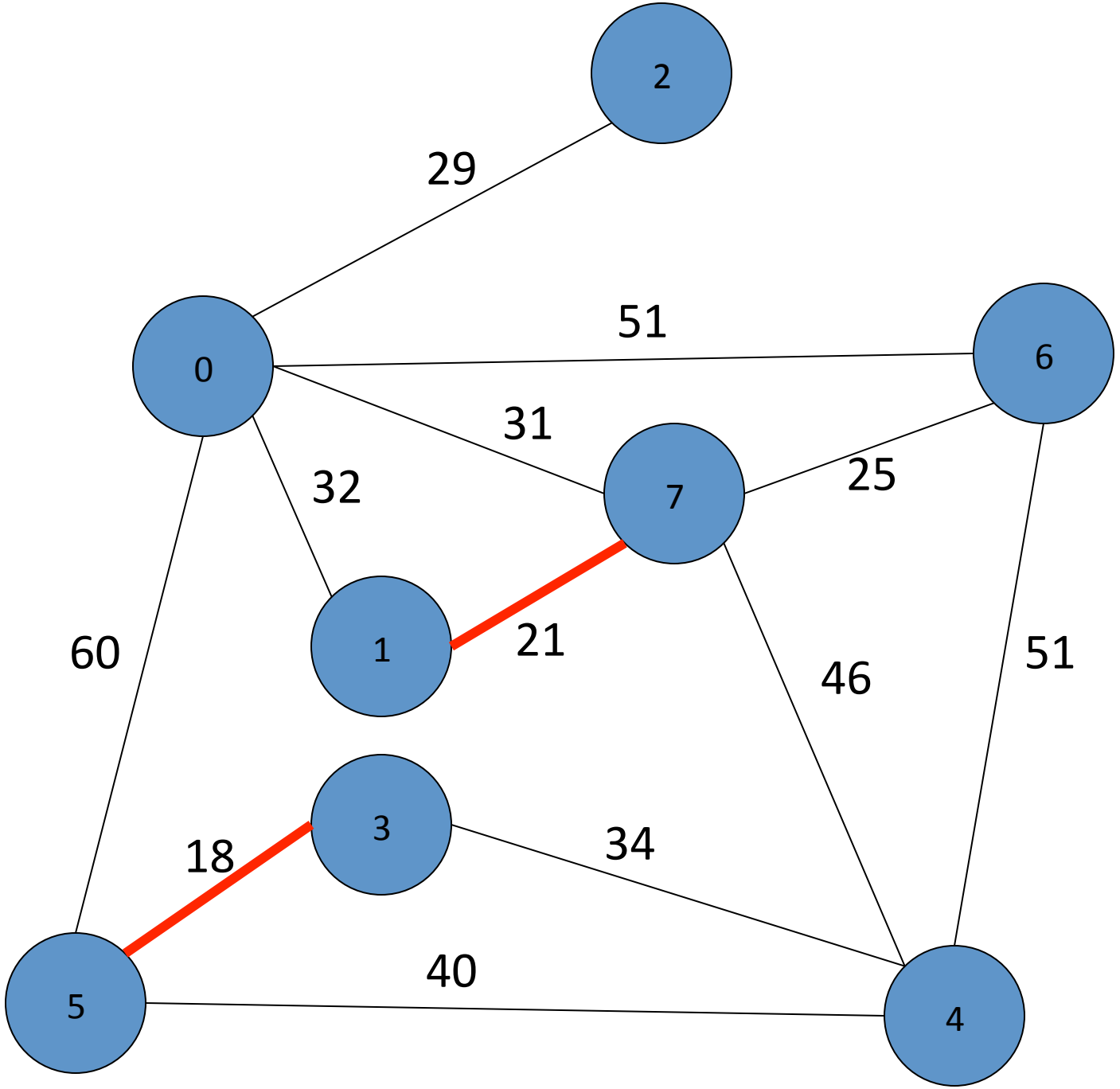
El *algoritmo de Kruskal* es un caso especial del algoritmo de la diap. #64

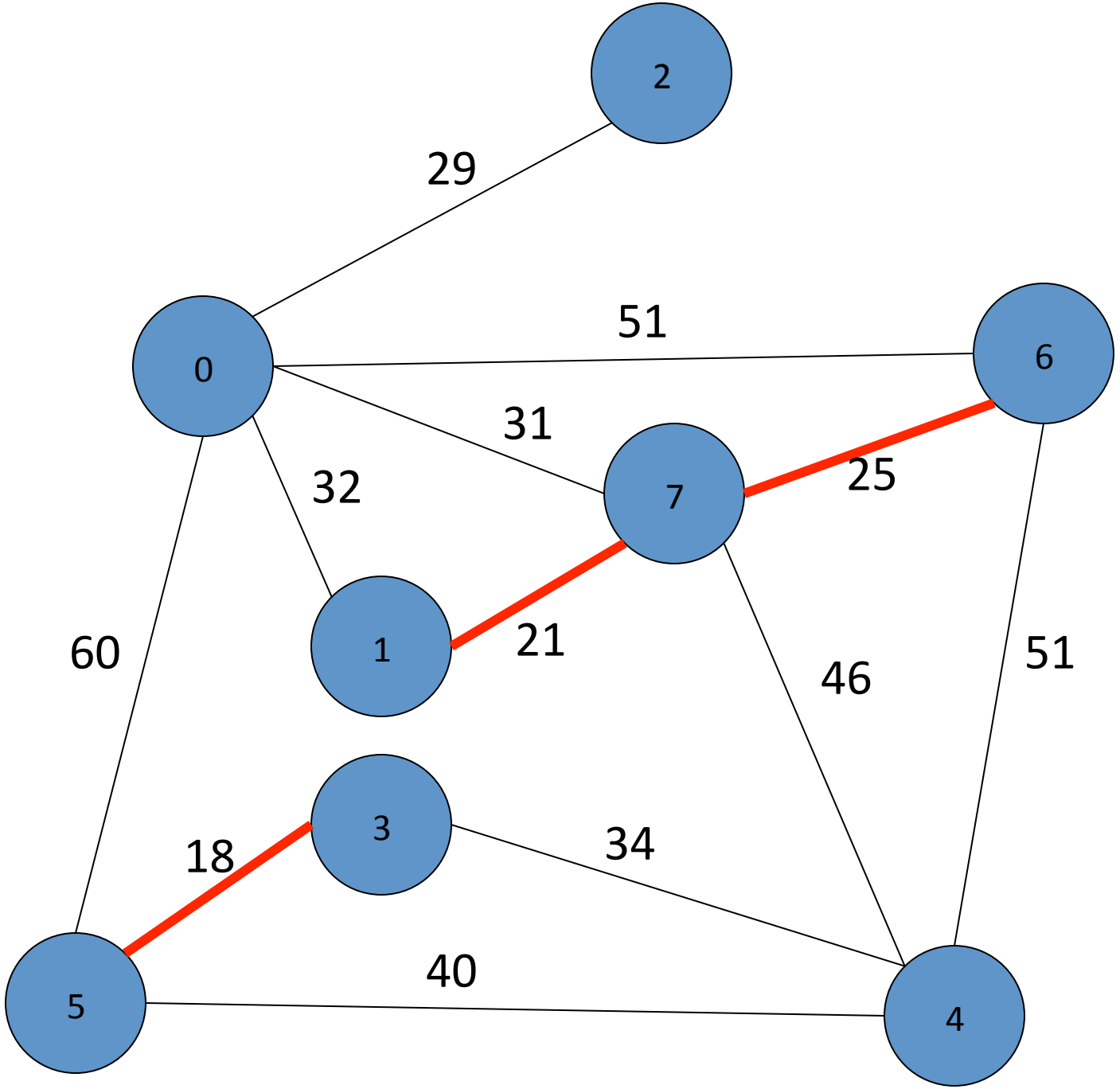
Las aristas en *A* forman un *bosque de árboles*

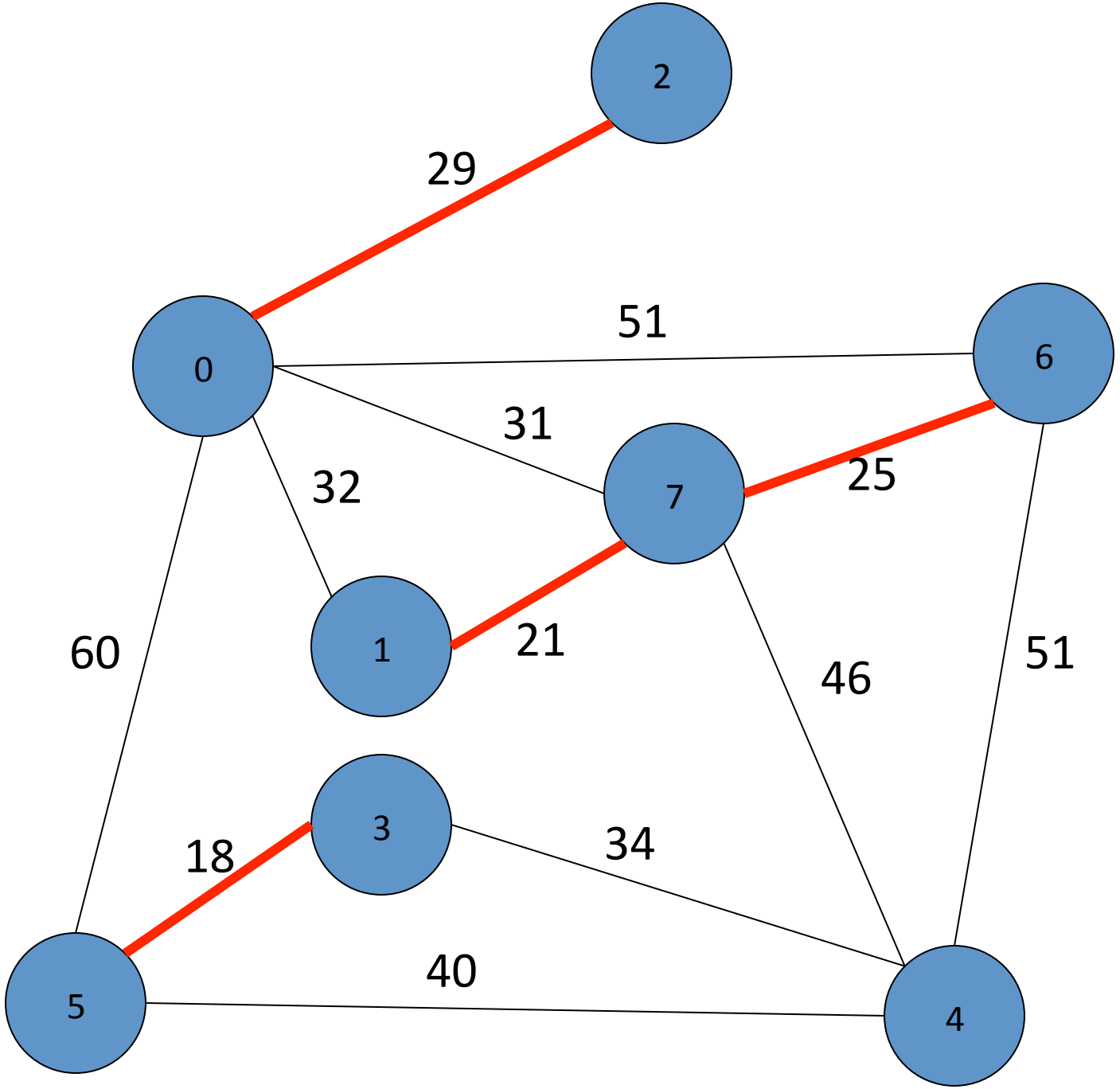
En cada paso, encuentra una arista segura para agregar al bosque: de todas las aristas que conectan dos árboles cualquiera en el bosque, encuentra una de menor costo

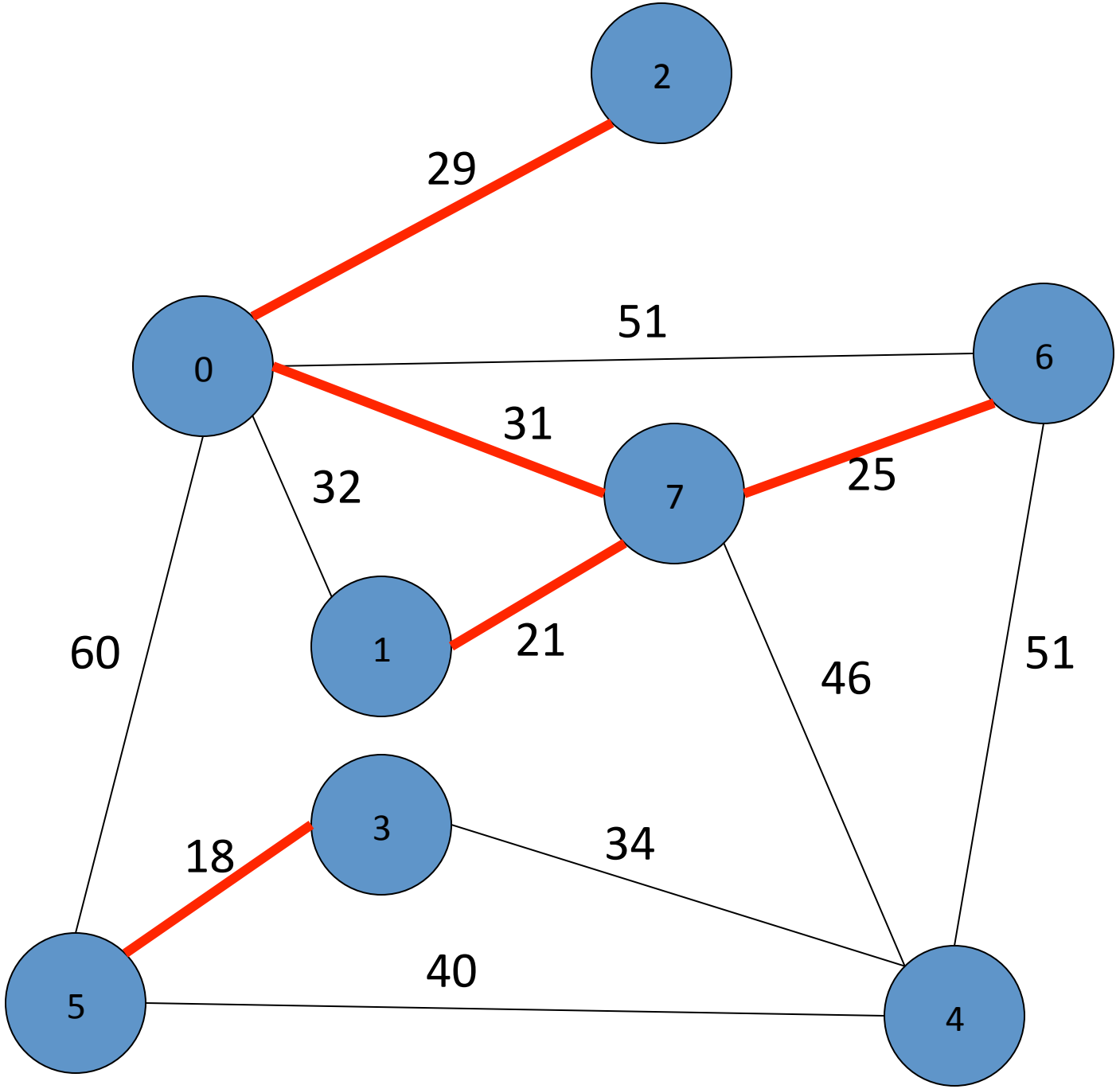

2
29
0
51
6
31
32
7
25
1
21
60
46
51
3
18
34
40
5
4

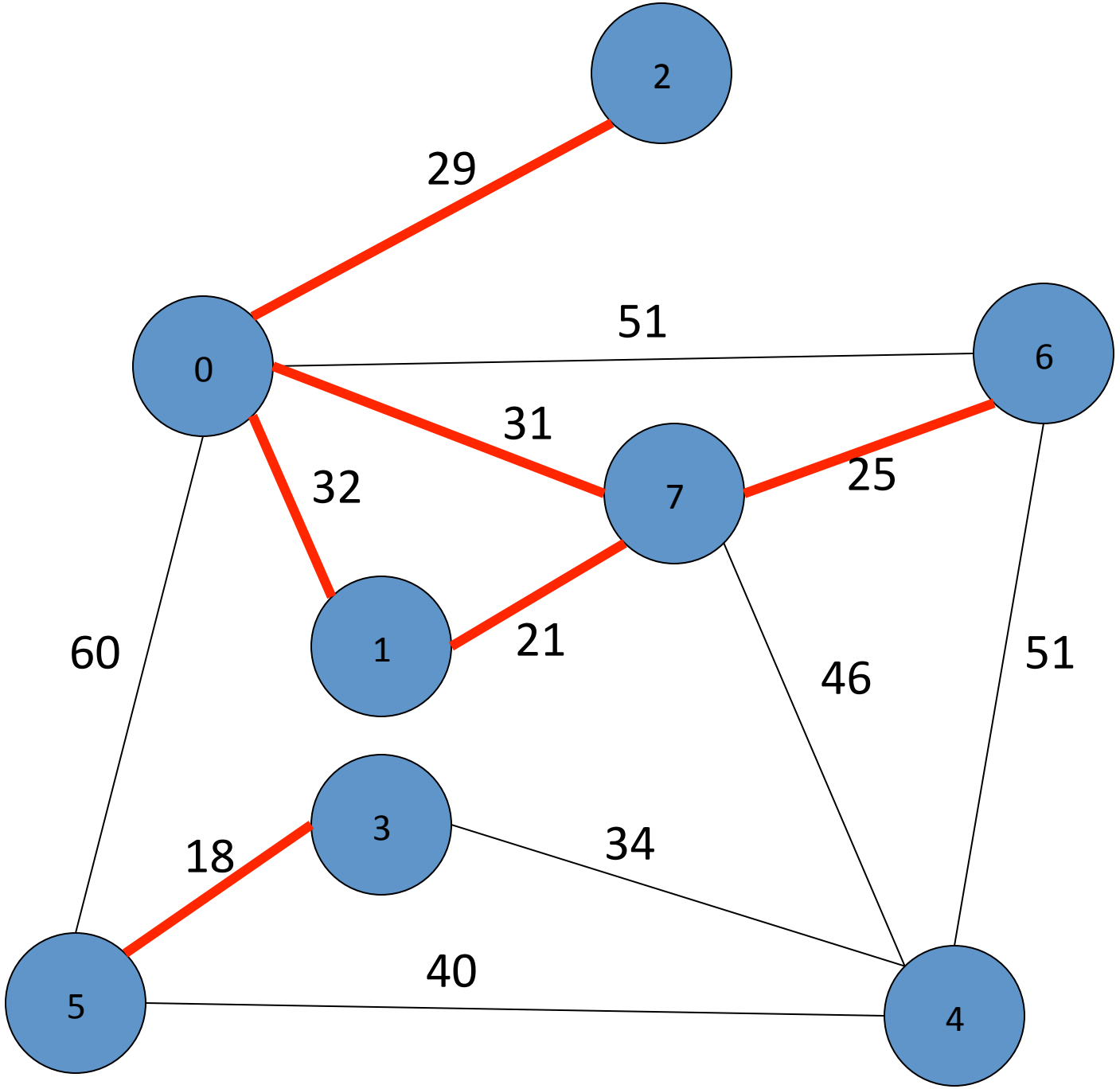

2
29
0
51
6
31
32
7
25
1
21
60
46
51
3
18
34
40
5
4


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

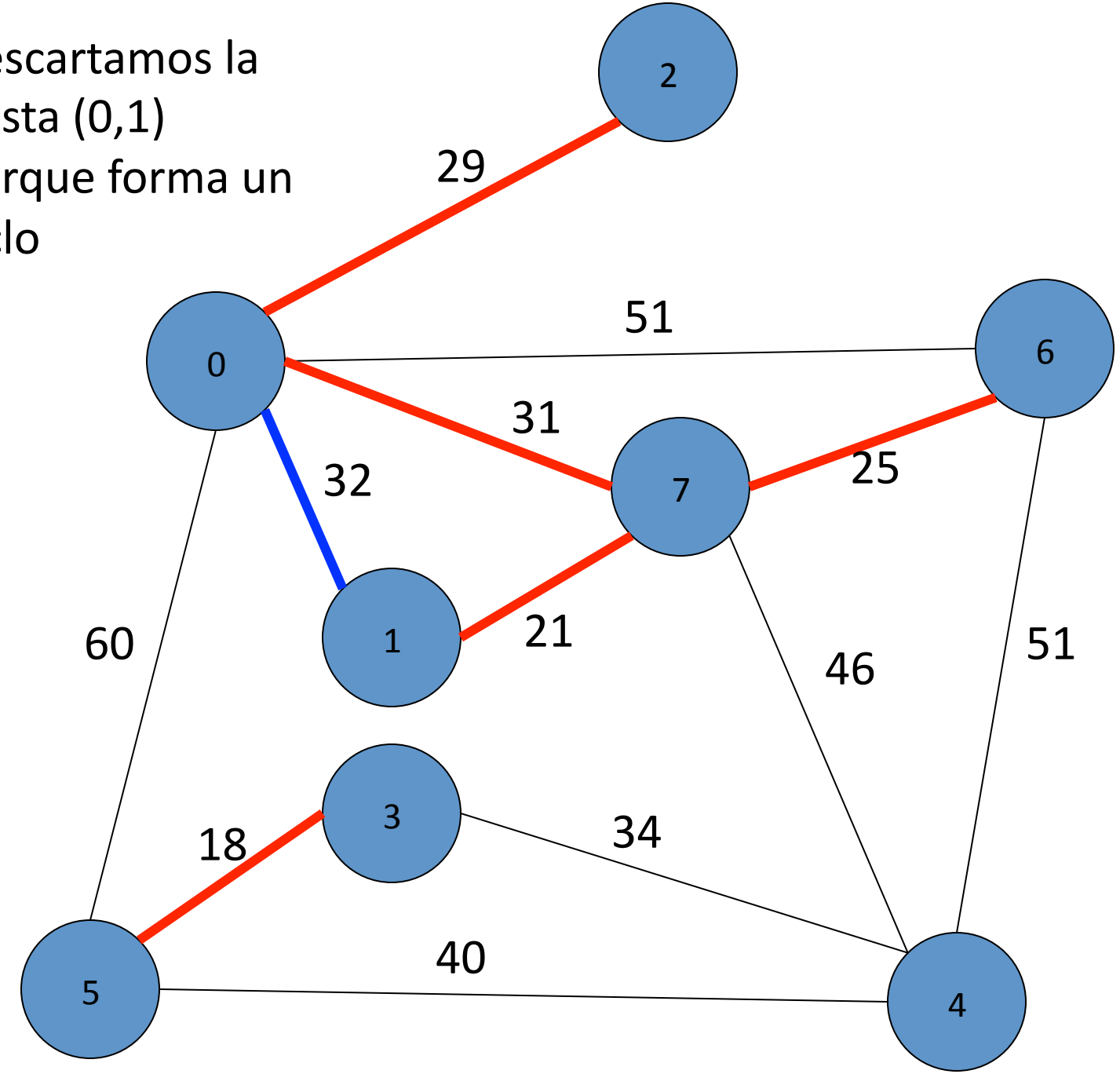

2
29
0
51
6
31
32
7
25
1
21
60
46
51
3
18
34
40
5
4

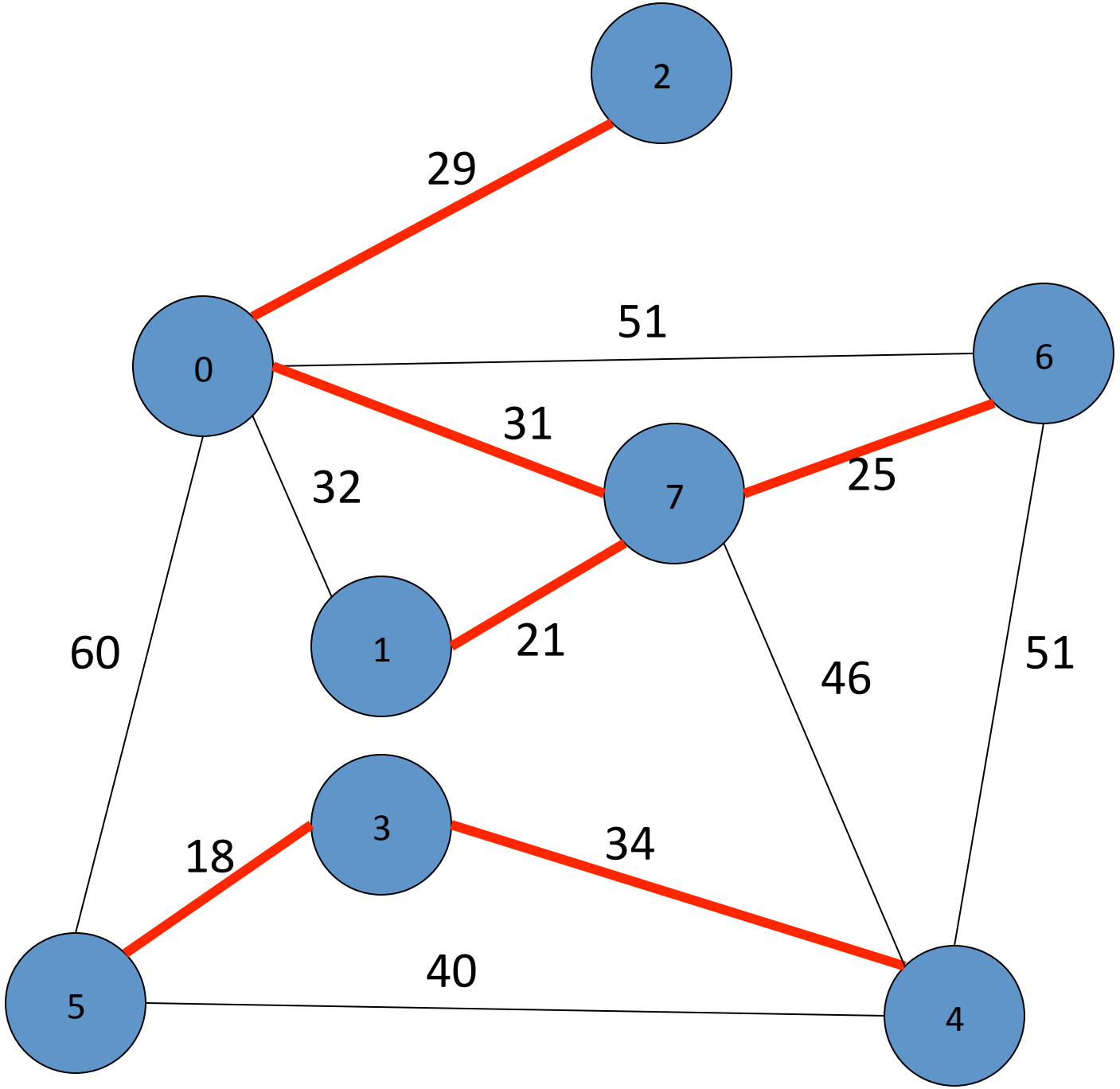

2
29
0
51
6
31
32
7
25
1
21
60
46
51
3
18
34
40
5
4


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

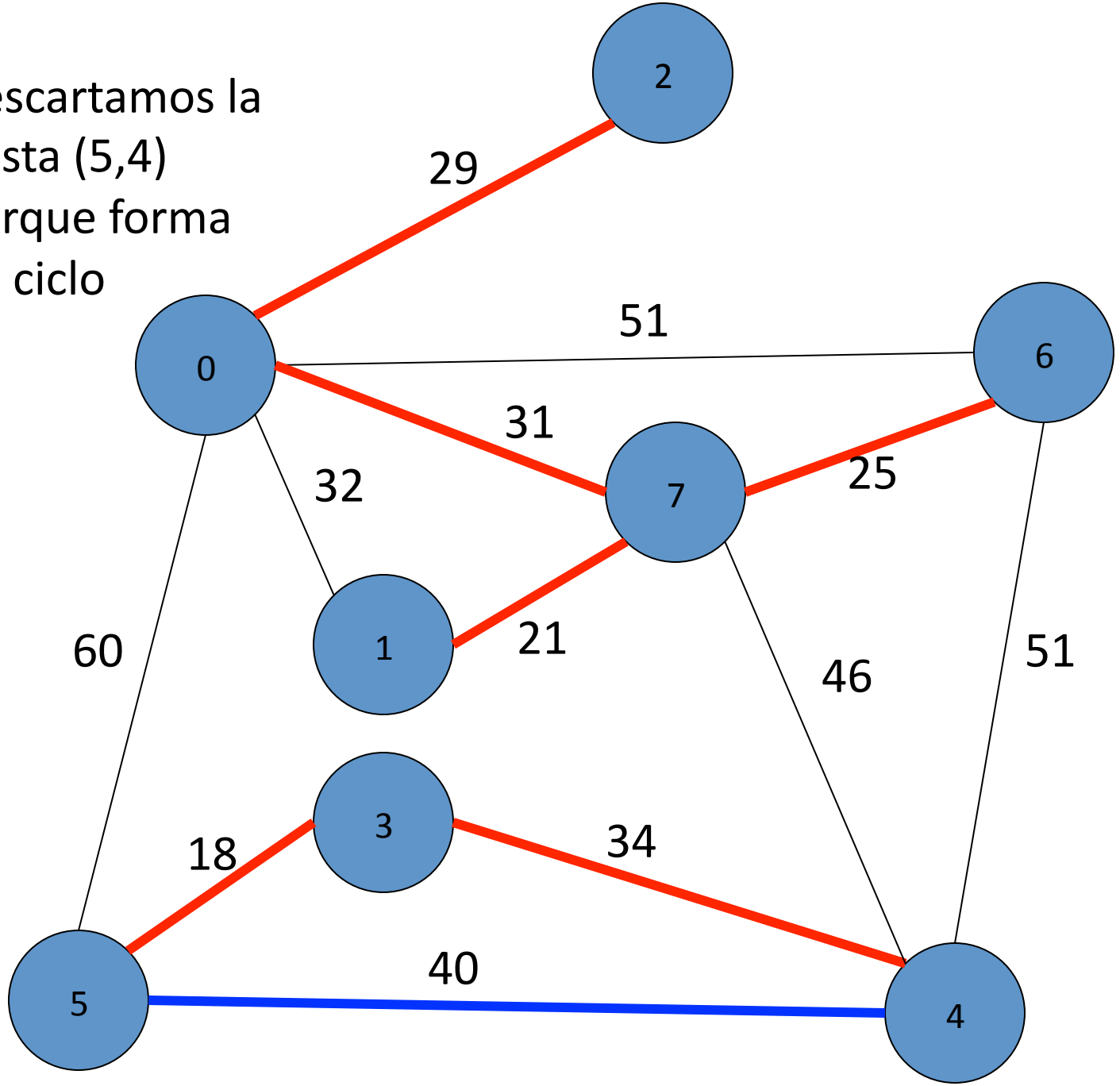

2
29
0
51
6
31
32
7
25
1
21
60
46
51
3
18
34
40
5
4

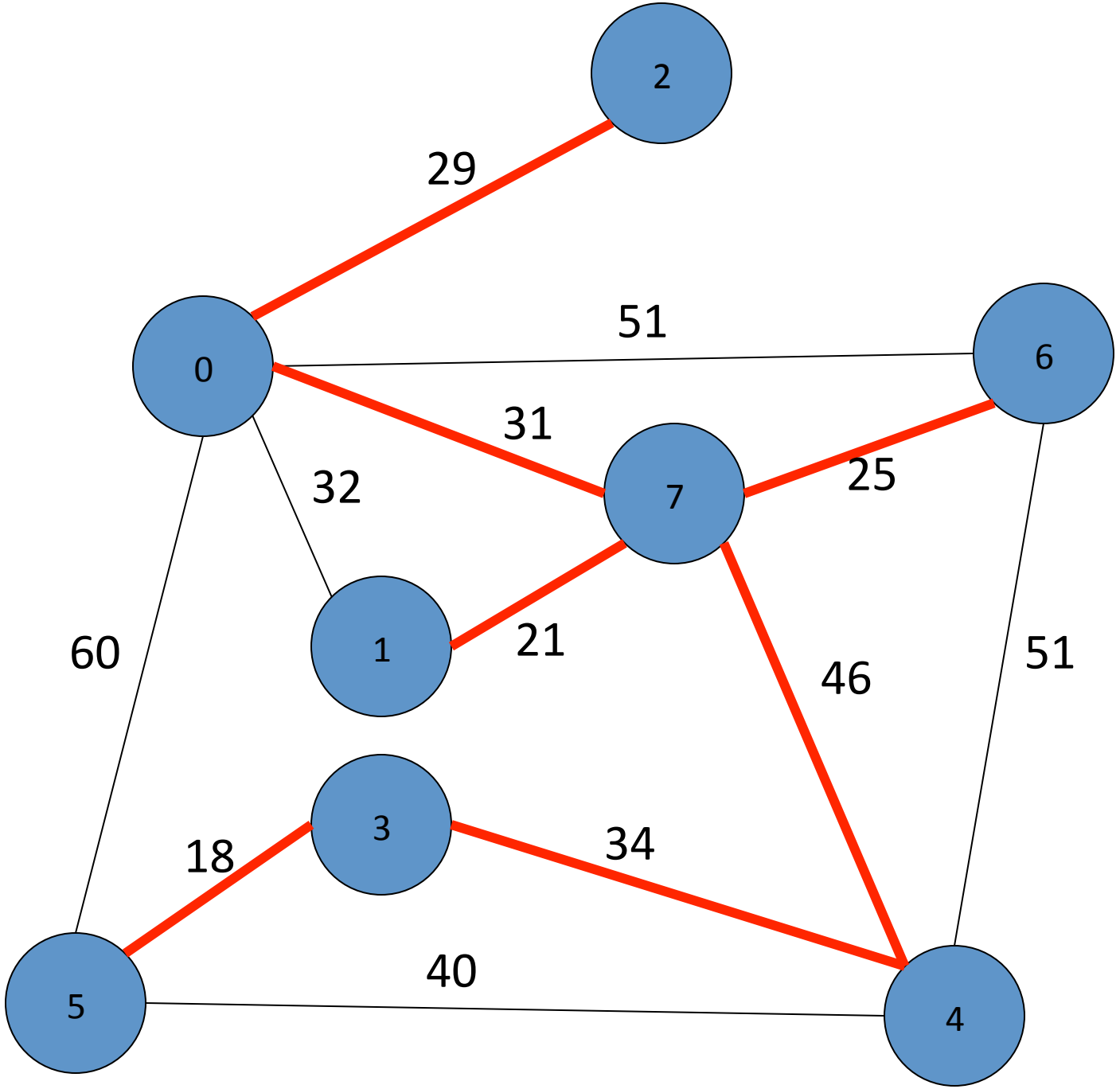
Descartamos la arista (0,1) porque forma un ciclo




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

Descartamos la arista (5,4) porque forma un ciclo




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

El algoritmo de Kruskal


```
kruskal():
    A = set()
    for each v in V:
        makeSet(v)
    ordenar las aristas de E de menor a mayor costo w
    for each (u,v) in E, tomadas en el orden anterior:
        if findSet(u) ≠ findSet(v):
            A = A ∪ {(u,v)}
            union(u,v)
    return A
```

`kruskal()` se basa en manejar los árboles del bosque *A* como *conjuntos disjuntos*

Cada árbol es un conjunto de vértices (y las aristas que los conectan)

`findSet(u)` devuelve el representante del conjunto que contiene al vértice *u*

Para saber si dos vértices *u* y *v* pertenecen al mismo árbol probamos si `findSet(u)` es igual a `findSet(v)`

Para combinar dos árboles en uno solo, usamos `union(u,v)`

El desempeño de `kruskal()` depende de la implementación de los conjuntos disjuntos

Supongamos que usamos representación mediante árboles con unión por rango y compresión de ruta

La inicialización de A toma tiempo O(1), y la ordenación de las aristas, O(Elog$E$)

El ciclo for sobre las aristas realiza O(E) operaciones findSet y union, que junto con las $|V|$ operaciones makeSet toman un tiempo total de O((V + E) $\alpha(V)$) = O($E\alpha(V)$);

Como $\alpha(V)$ = O(log$V$) = O(log$E$), y como $|E| < |V|^2$, `kruskal()` toma tiempo O(Elog$V$)

¿Cómo determinamos las *componentes conectadas* de un grafo no direccional?

Si las aristas son agregadas dinámicamente y queremos conocer las componentes conectadas a medida que cada arista es agregada,

… entonces podemos usar las operaciones para conjuntos disjuntos:

para cada vértice v, makeSet(v)
para cada arista (u, v),
si findSet(u) ≠ findSet(v)
union(u, v)