Algoritmos y Estructuras de Datos II

Algoritmos voraces

15 de mayo de 2013

Clase de hoy

1. Repaso
   - Algoritmos voraces
   - Problema de la moneda
   - Problema de la mochila
   - Árboles generadores de costo mínimo
2. El problema union-find
   - Primer intento
   - Segundo intento
   - Tercer intento
   - Último intento

Repaso

- cómo vs. qué
- 3 partes
  1. análisis de algoritmos
  2. tipos de datos
     - tipos concretos
     - tipos abstractos
     - implementaciones
     - árboles binarios: de búsqueda (ABBs) y heaps.
  3. técnicas de resolución de problemas
     - divide y vencerás
     - algoritmos voraces (moneda, mochila, Prim, Kruskal, Dijkstra)
     - backtracking
     - programación dinámica
     - recorrida de grafos

Clase de hoy

Algoritmos Voraces (o Glotones, Golosos) (Greedy)

- Es la técnica más sencilla de resolución de problemas.
- Normalmente se trata de algoritmos que resuelven problemas de **optimización**.
- Los algoritmos voraces intentan construir la solución óptima buscada **paso a paso**,
- **eligiendo** en cada paso
- la **componente** de la solución
- que **parece** más apropiada.

Características

- Nunca revisan una **elección** ya realizada,
- confían en haber elegido bien las componentes anteriores.
- Por ello, lamentablemente, no siempre funcionan,
- pero cuando funcionan son muy eficientes.
- Existen varios problemas interesantes que admiten soluciones voraces,
- que como acabamos de decir, resultan muy eficientes.

Problemas con solución voraz

- Problema de la moneda.
- Problema de la mochila.
- Problema del camino de costo mínimo en un grafo.
- Problema del árbol generador de costo mínimo en un grafo.

Forma general

fun greedy(C) **ret** S
{C: conjunto de candidatos, S: solución a construir}
S:= {}
do S no es solución $\rightarrow$ c:= seleccionar de C
C:= C-{c}
if S$\cup${c} es factible $\rightarrow$ S:= S$\cup${c} **fi**
od
end fun

Lo más importante es el criterio de selección.

Clase de hoy

Problema de la moneda

Se tienen infinitas monedas de c/u de las denominaciones $d_1, \ldots d_n$ y un monto a pagar m y se desea hallar el menor número de monedas necesarias para pagar dicho monto de manera exacta.

{Pre: d[1] $\geq$ d[2] $\geq \ldots \geq$ d[n] }
fun cambio(d:**array**[1..n] **of nat**, m: monto) **ret** S: **array**[1..n] **of nat**
    for i:= 1 **to** n **do**
        S[i]:= m div d[i]
        m:= m mod d[i]
    od
end fun

Sobre este algoritmo

- El orden del algoritmo es n, es decir, el número de denominaciones.
- Si el arreglo de denominaciones no está ordenado requiere $n \log n$ ordenarlo y luego n más el algoritmo, en total es $n \log n$.
- No siempre funciona, depende del conjunto de denominaciones.
- Para un conjunto razonable, funciona.

Conjunto de denominaciones para el que no funciona

- Sean 4, 3 y 1 las denominaciones y sea 6 el monto a pagar.
- El algoritmo voraz intenta pagar con una moneda de denominación 4, queda un saldo de 2 que solamente puede pagarse con 2 monedas de 1, en total, 3 monedas.
- Pero hay una solución mejor: dos monedas de 3.
- De todas formas, el algoritmo anda bien para todas las denominaciones de uso habitual.

Clase de hoy

Problema de la mochila

- Tenemos una mochila de capacidad W.
- Tenemos n objetos de valor $v_1, v_2, \ldots, v_n$ y peso $w_1, w_2, \ldots, w_n$.
- Se quiere encontrar la mejor selección de objetos para llevar en la mochila.
- Por mejor selección se entiende aquélla que totaliza el mayor valor posible sin que su peso exceda la capacidad W de la mochila.
- Para que el problema no sea trivial, asumimos que la suma de los pesos de los n objetos excede la capacidad de la mochila, obligándonos entonces a seleccionar cuáles cargar en ella.
- Se permite fraccionar objetos.

Criterio de selección

- Debemos asegurarnos de que cada kg utilizado de la mochila sea aprovechado de la mejor manera posible: que cada kg colocado en la mochila valga lo más posible.
- Criterio: elegir el de mayor valor relativo (cociente entre el valor y el peso): dicho cociente expresa el valor promedio de cada kg de ese objeto.

Algoritmo voraz

```
fun mochila(v: array[1..n] of valor, w: array[1..n] of peso, W: peso)
                                                ret s: array[1..n] of real
    var weight: peso; c: nat
    for i:= 1 to n do s[i]:= 0 od
    weight:= 0
    do weight ≠ W → c:= tal que s[c] = 0 ∧ v[c]/w[c] máximo
                    if weight + w[c] ≤ W → s[c]:= 1
                                          weight:= weight + w[c]
                      weight + w[c] > W → s[c]:= (W-weight)/w[c]
                                          weight:= W
                    fi
    od
end fun
```

Sobre este algoritmo

- El orden del algoritmo es n, es decir, el número de objetos.
- Si los objetos no están ordenado según su cociente valor/peso, requiere $n \log n$ ordenarlo y luego n más el algoritmo, en total es $n \log n$.
- Si los objetos en total exceden muy largamente la capacidad de la mochila, en vez de ordenar puede convenir utilizar una cola de prioridades, en cuyo caso el orden es n.
- funciona siempre que esté permitido fraccionarse objetos.

Algoritmo voraz si los objetos están ordenados decrecientemente según el cociente v[i]/w[i]

```
fun mochila(v: array[1..n] of valor, w: array[1..n] of peso, W: peso)
                                                  ret s: array[1..n] of real
    var weight: peso; c: nat
    for i:= 1 to n do s[i]:= 0 od
    weight:= 0
    c:= 1
    do weight + w[c] ≤ W → s[c]:= 1
                           weight:= weight + w[c]
                           c:= c+1
    od
    s[c]:= (W-weight)/w[c]
end fun
```

Clase de hoy

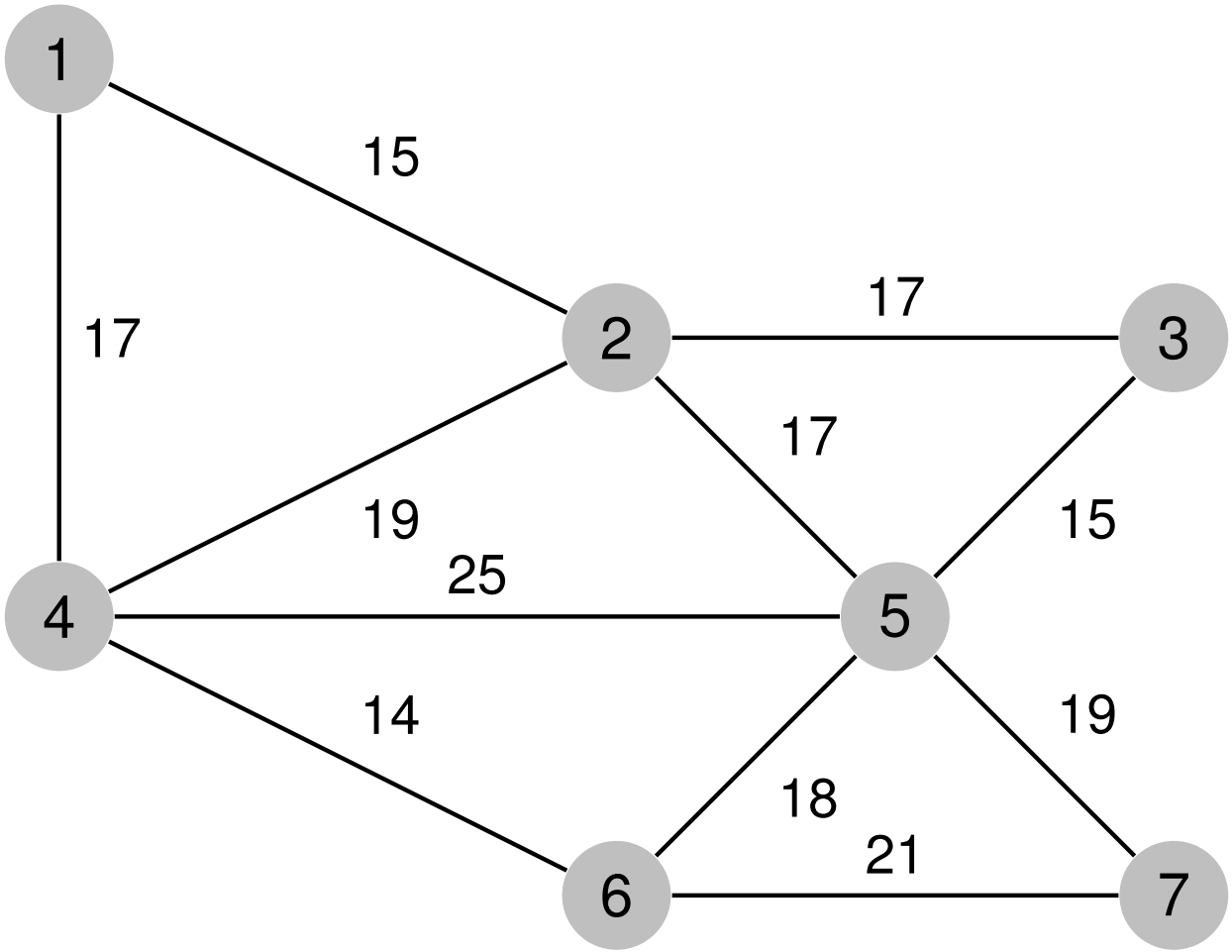
Árbol generador de costo mínimo

- Sea $G = (V, A)$ un grafo conexo no dirigido con un costo no negativo asociado a cada arista.
- Se dice que $T \subseteq A$ es un árbol generador (intuitivamente, un tendido) si el grafo (V, T) es conexo y no contiene ciclos.
- Su costo es la suma de los costos de sus aristas.
- Se busca T tal que su costo sea mínimo.

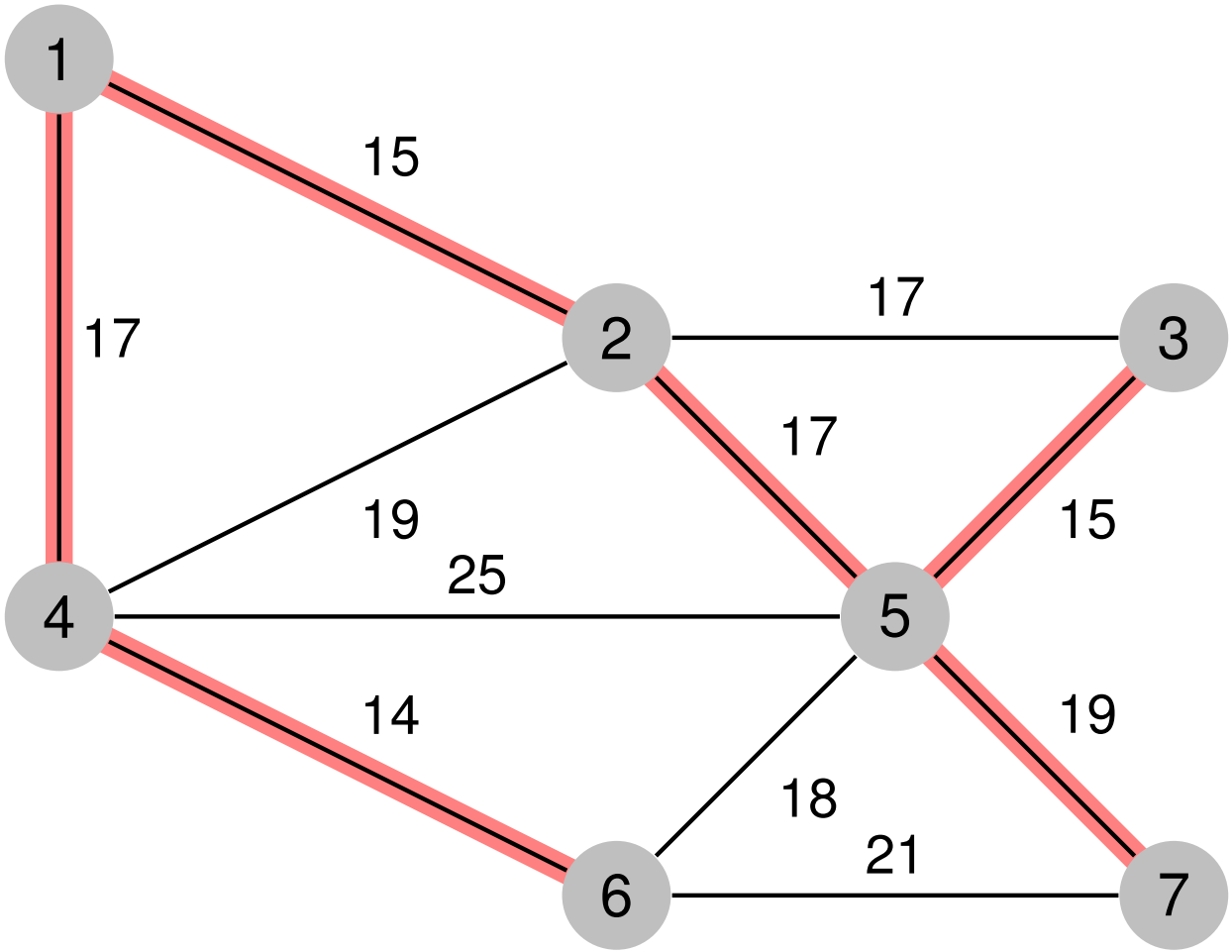
Árbol generador de costo mínimo

- El problema de encontrar un árbol generador de costo mínimo tiene numerosas aplicaciones en la vida real.
- Cada vez que se quiera realizar un tendido (eléctrico, telefónico, etc) se quieren unir distintas localidades de modo que requiera el menor costo en instalaciones (por ejemplo, cables) posible.
- Se trata de realizar el tendido siguiendo la traza de un árbol generador de costo mínimo.

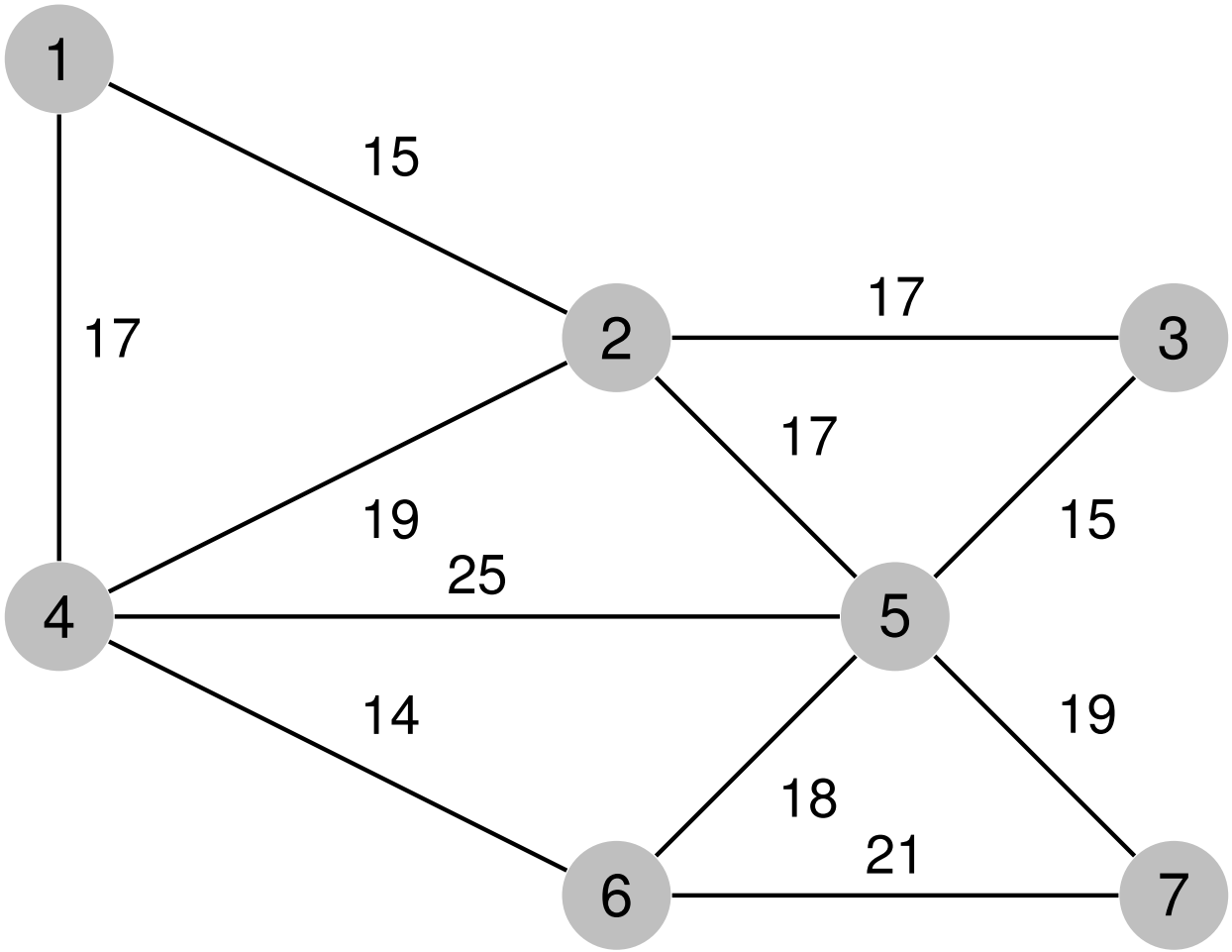
Ejemplo



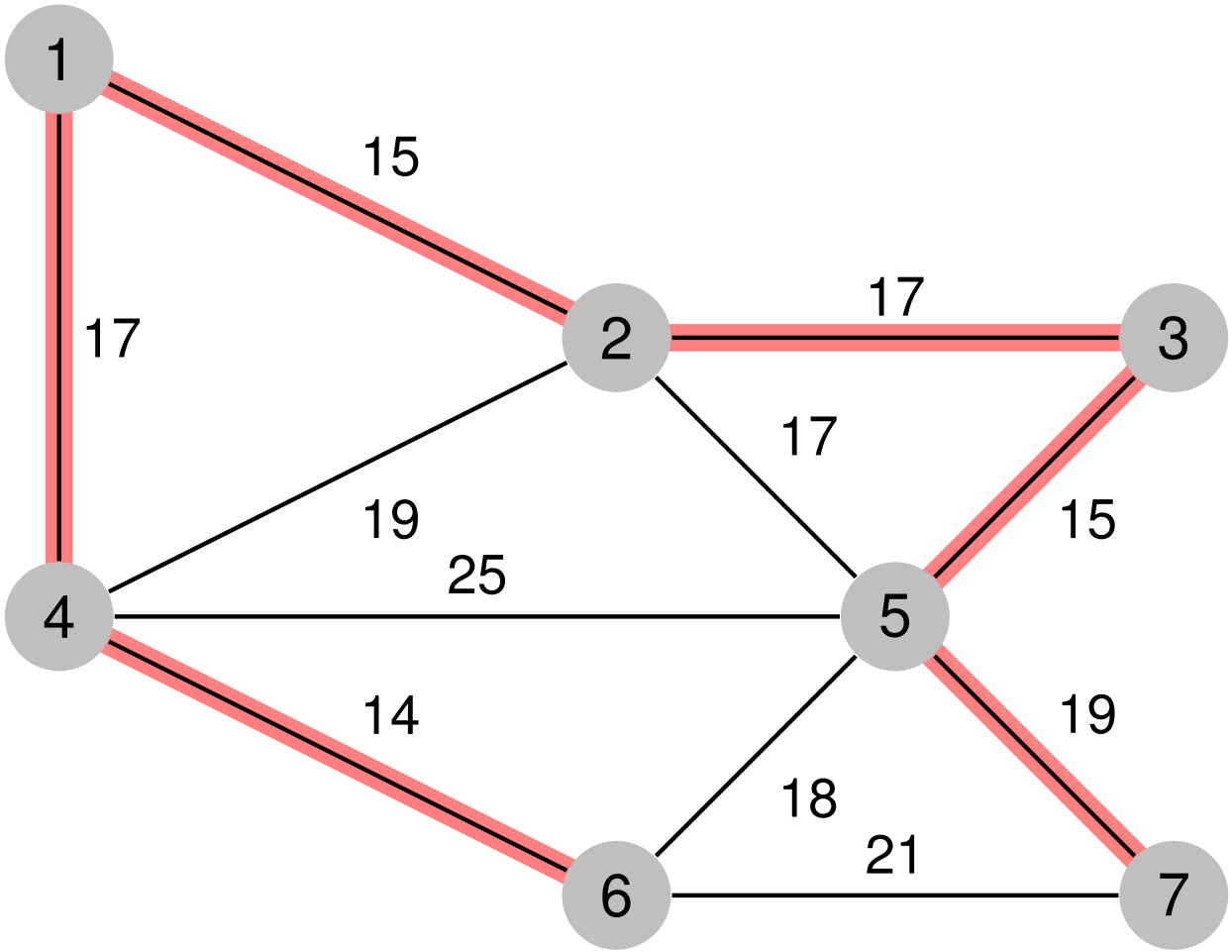
Ejemplo



Ejemplo



Ejemplo

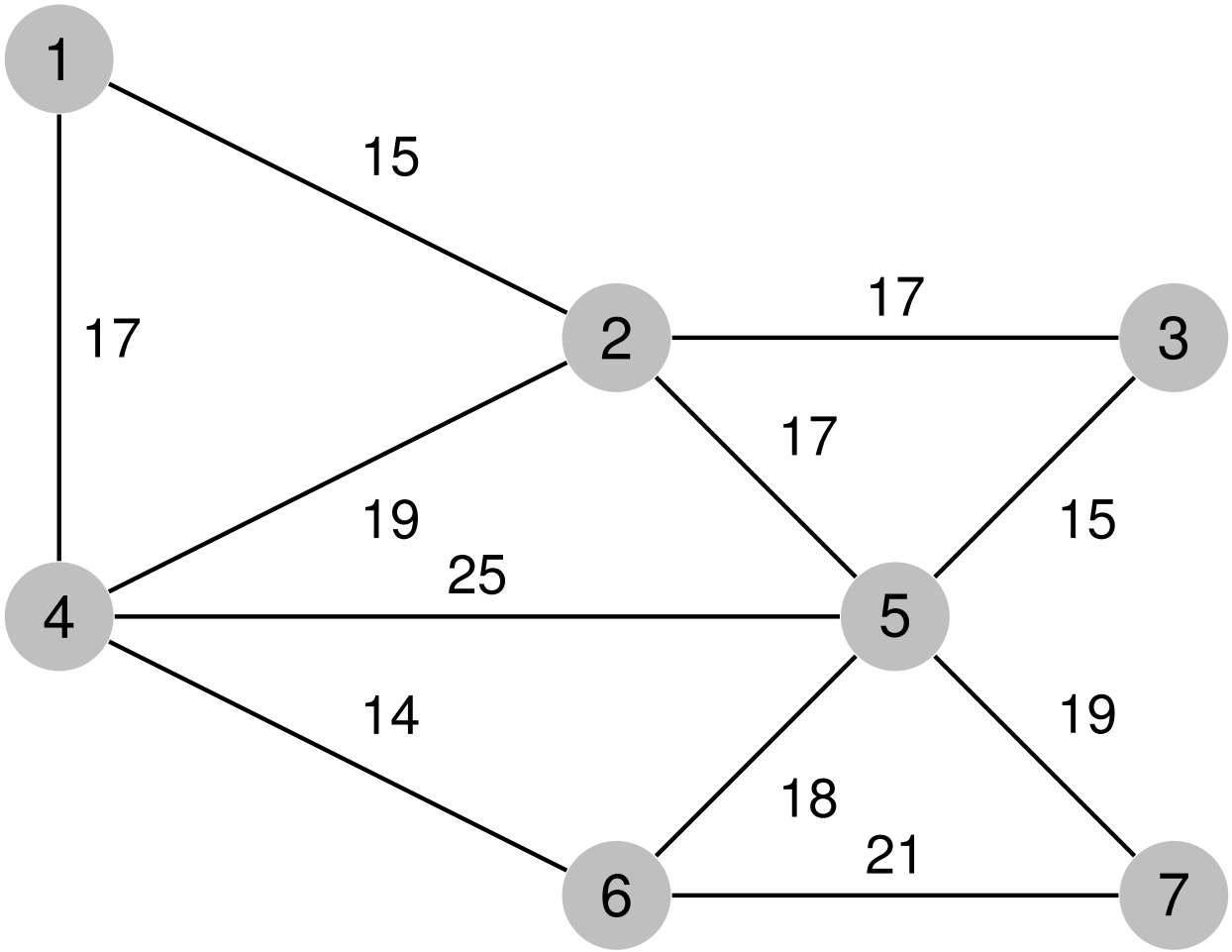


Dos estrategias

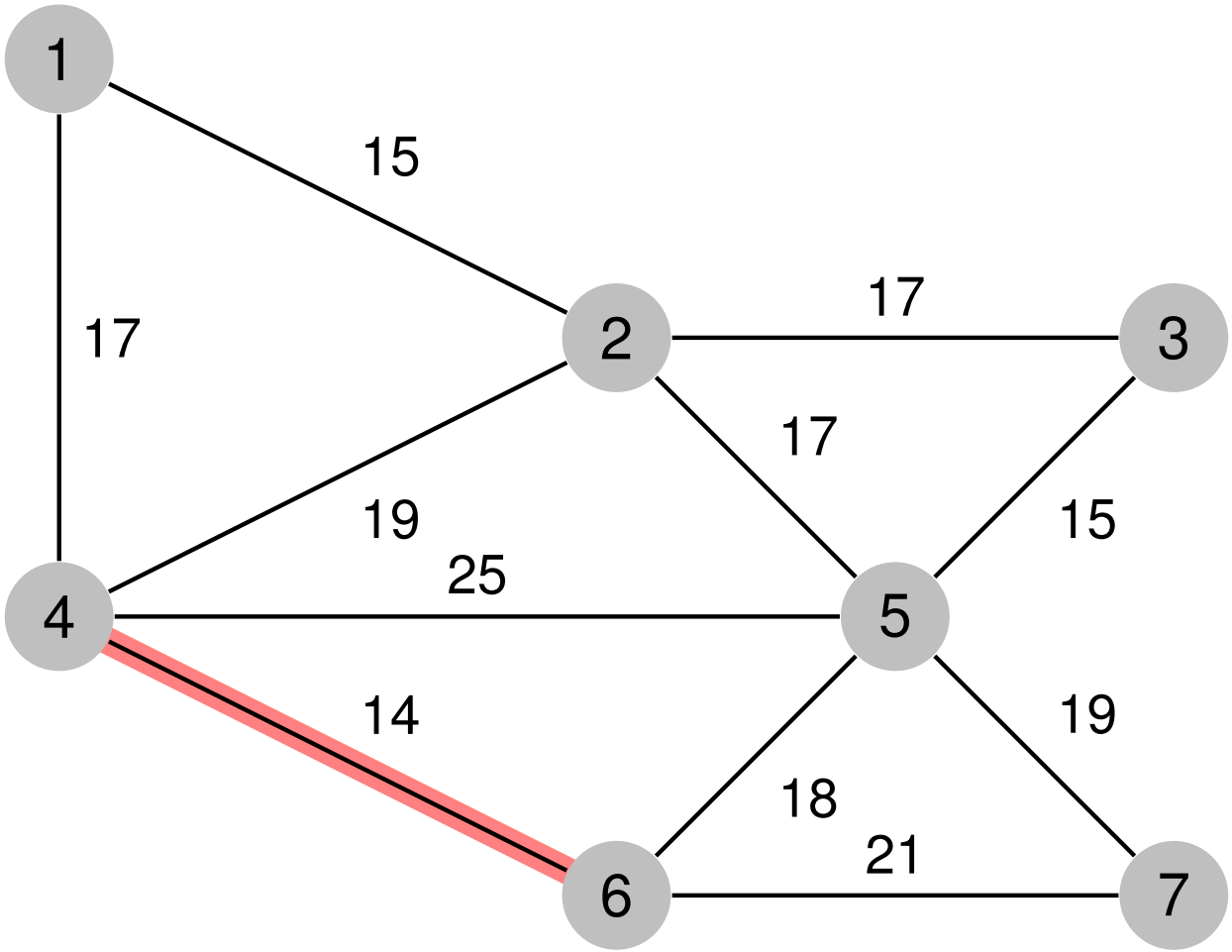
Hay dos grandes ideas de cómo resolverlo:

- La de Prim: se parte desde un vértice origen y se va extendiendo el tendido a partir de ahí:
  - en cada paso se une el tendido ya existente con alguno de los vértices aún no alcanzados, seleccionando la arista de menor costo capaz de incorporar un nuevo vértice
- La de Kruskal: se divide el grafo en distintas componentes (originariamente una por cada vértice) y se van uniendo componentes,
  - en cada paso se selecciona la arista de menor costo capaz de unir componentes.

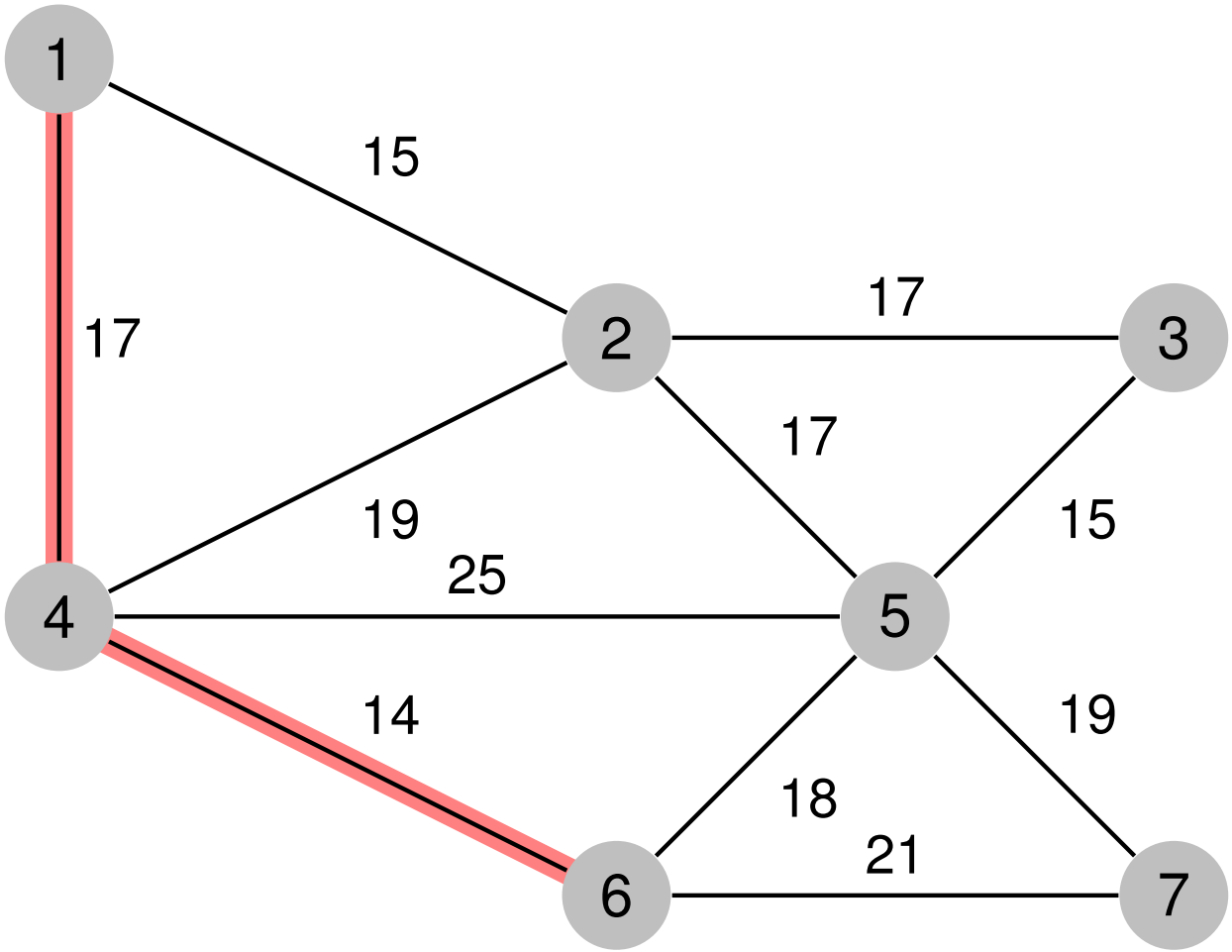
Algoritmo de Prim



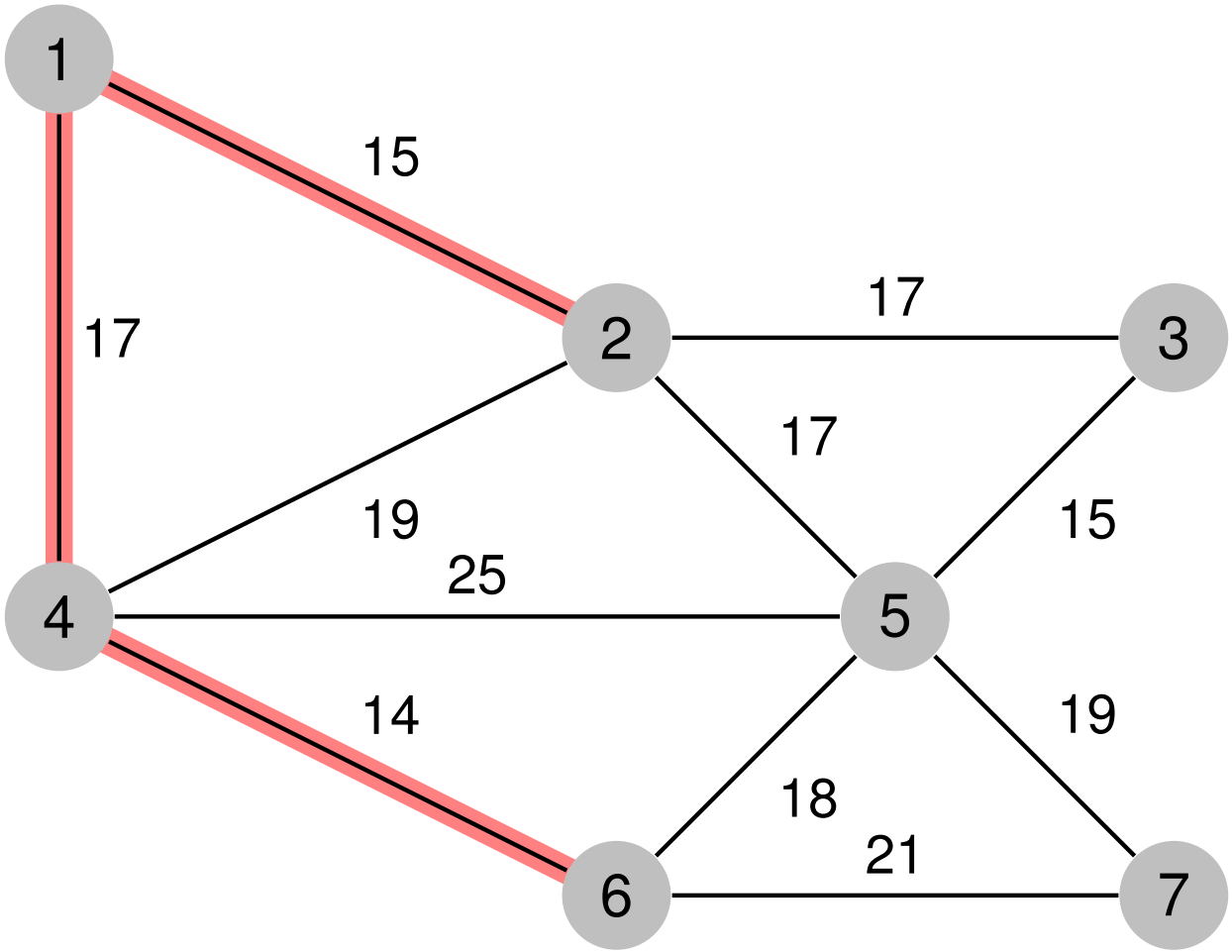
Algoritmo de Prim



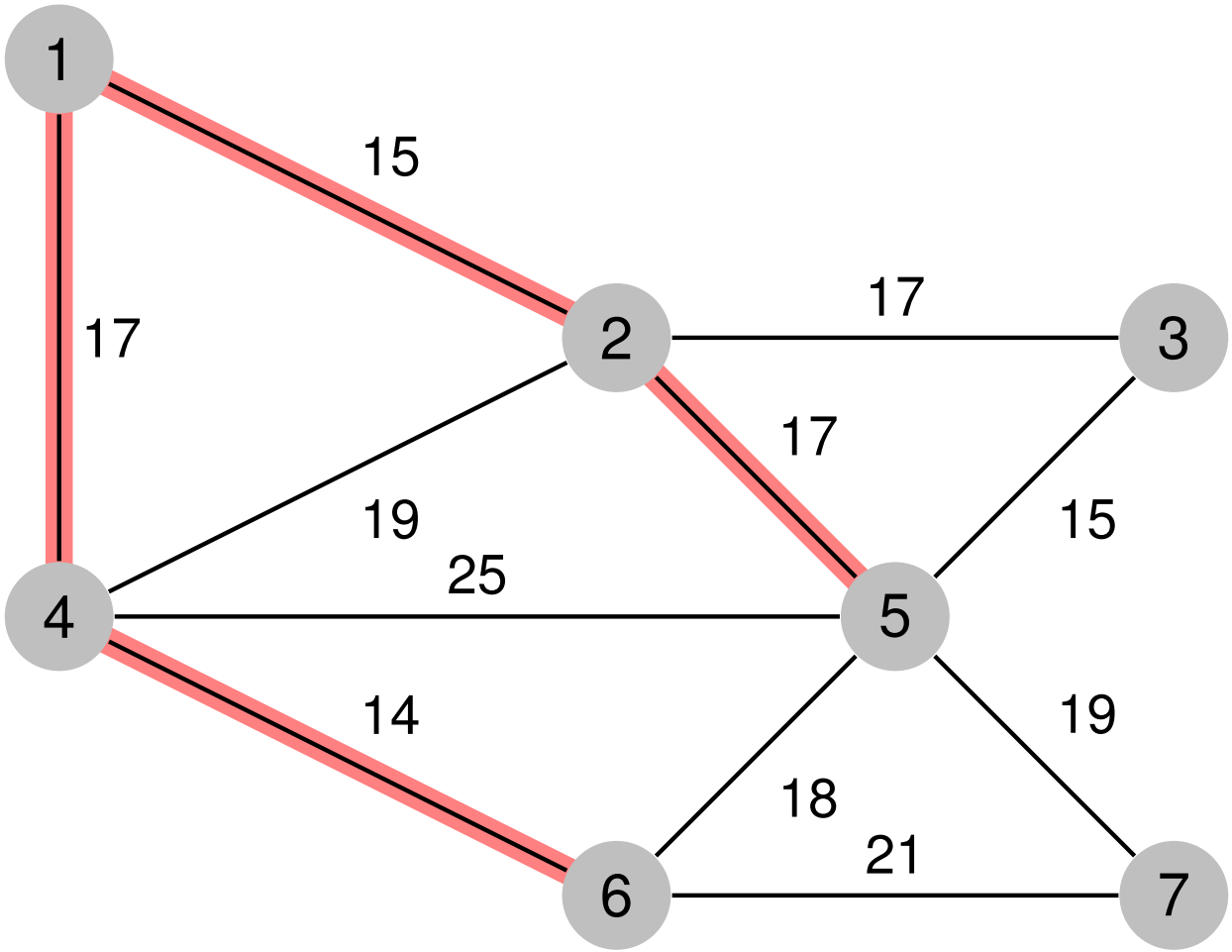
Algoritmo de Prim



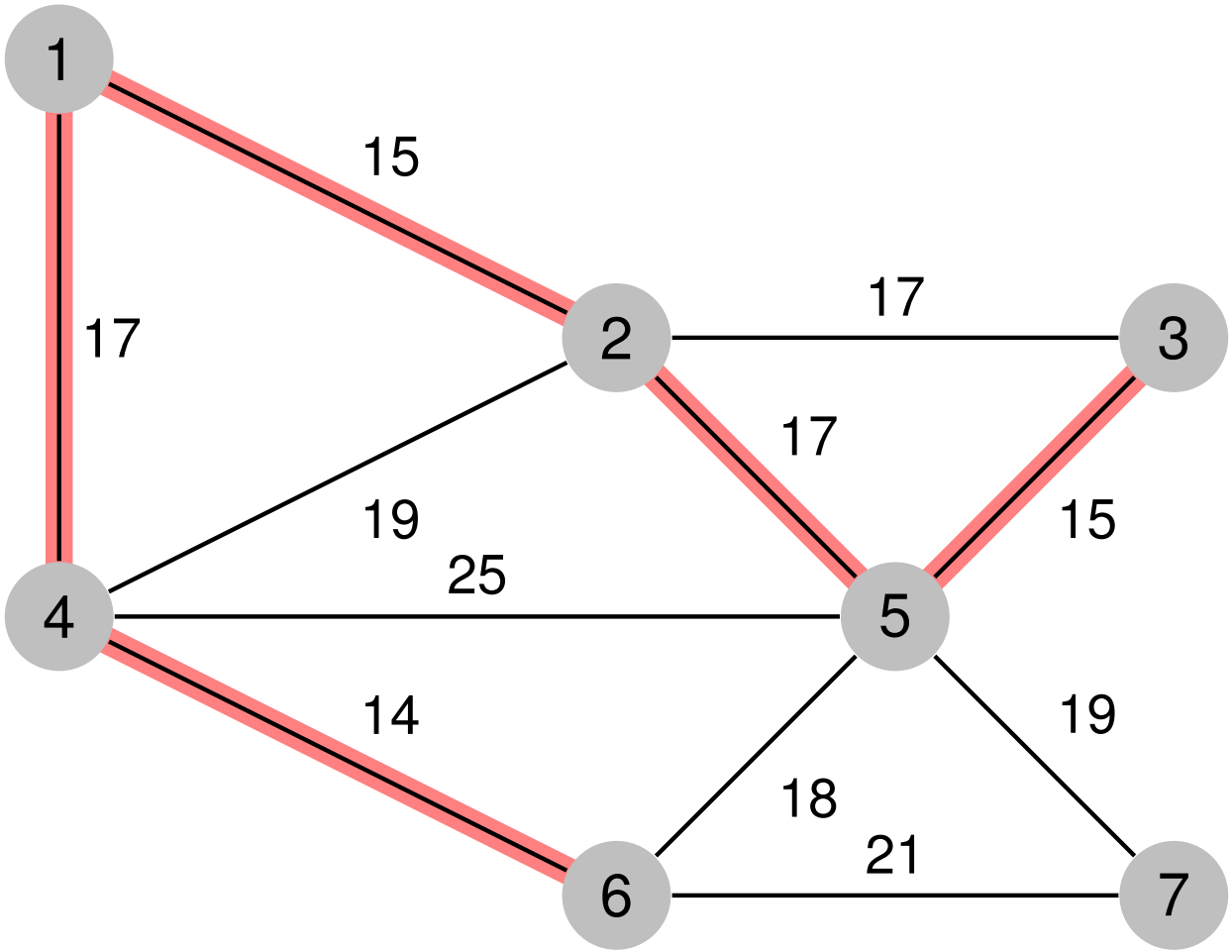
Algoritmo de Prim



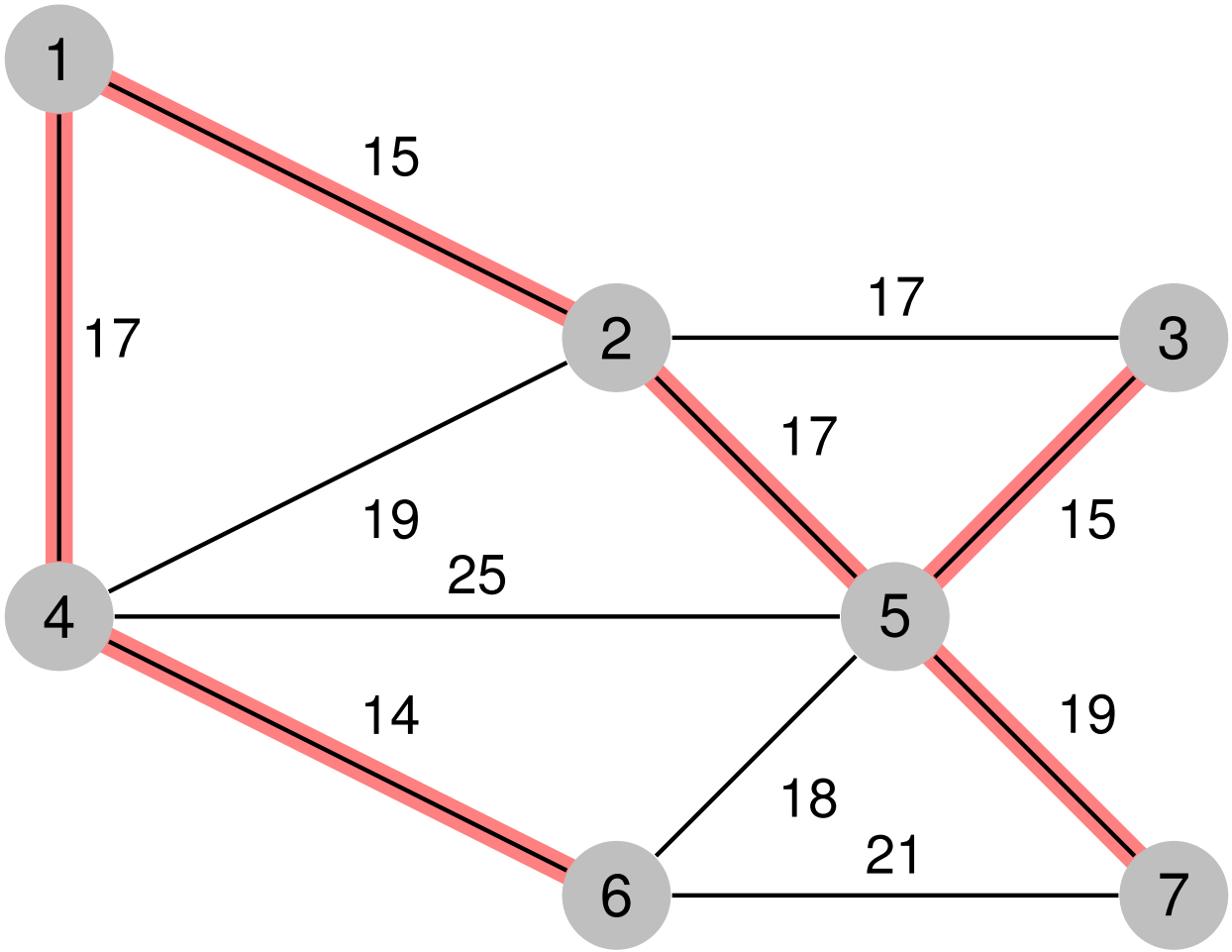
Algoritmo de Prim



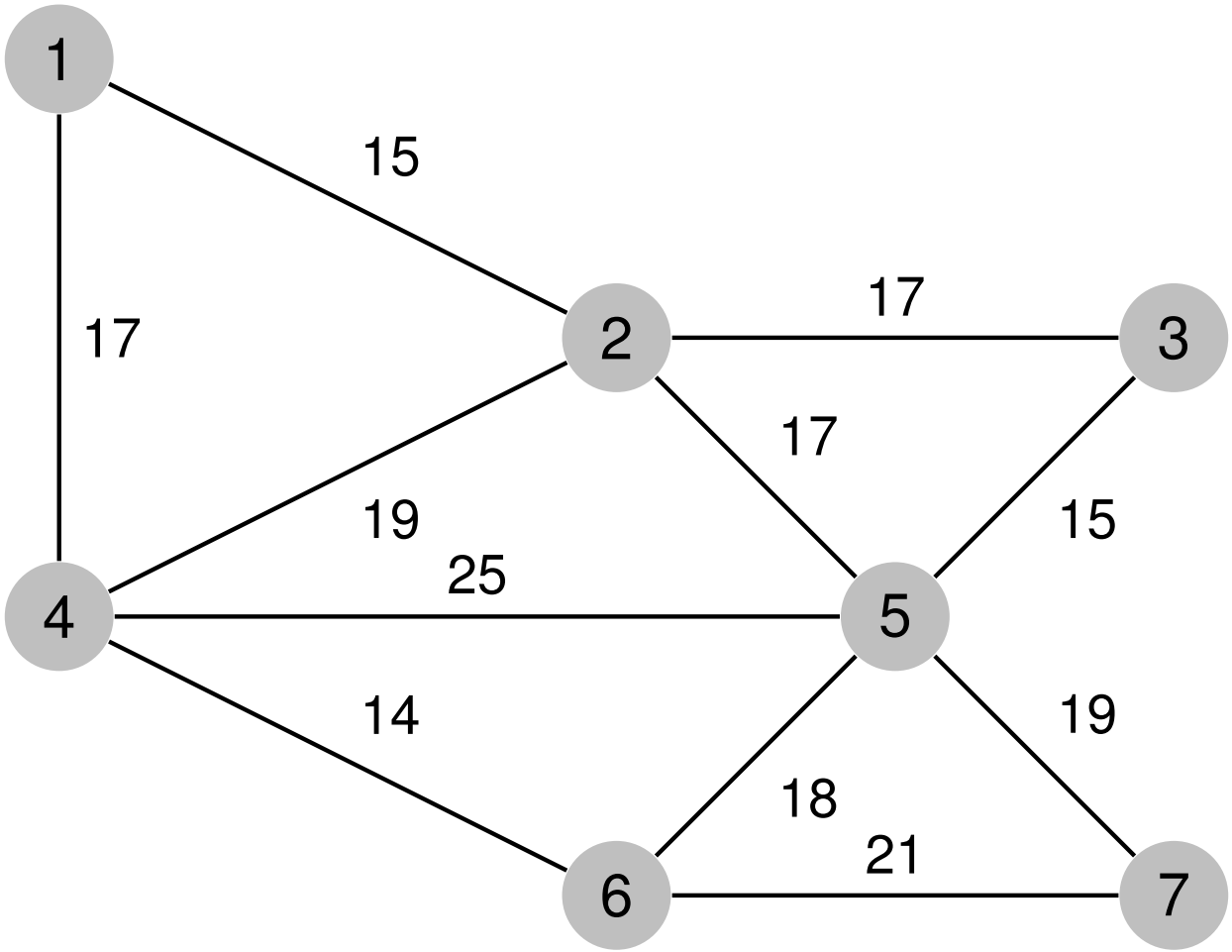
Algoritmo de Prim



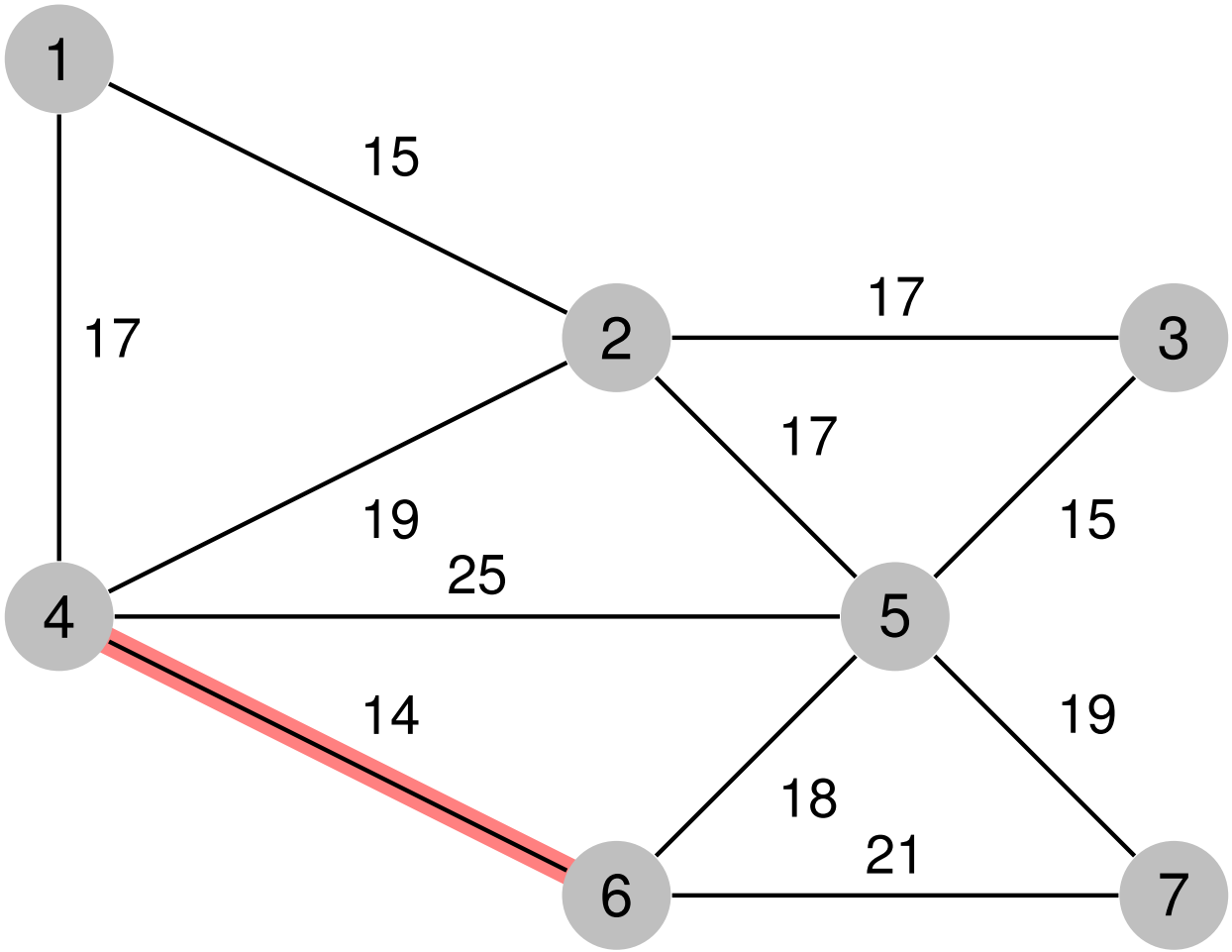
Algoritmo de Prim



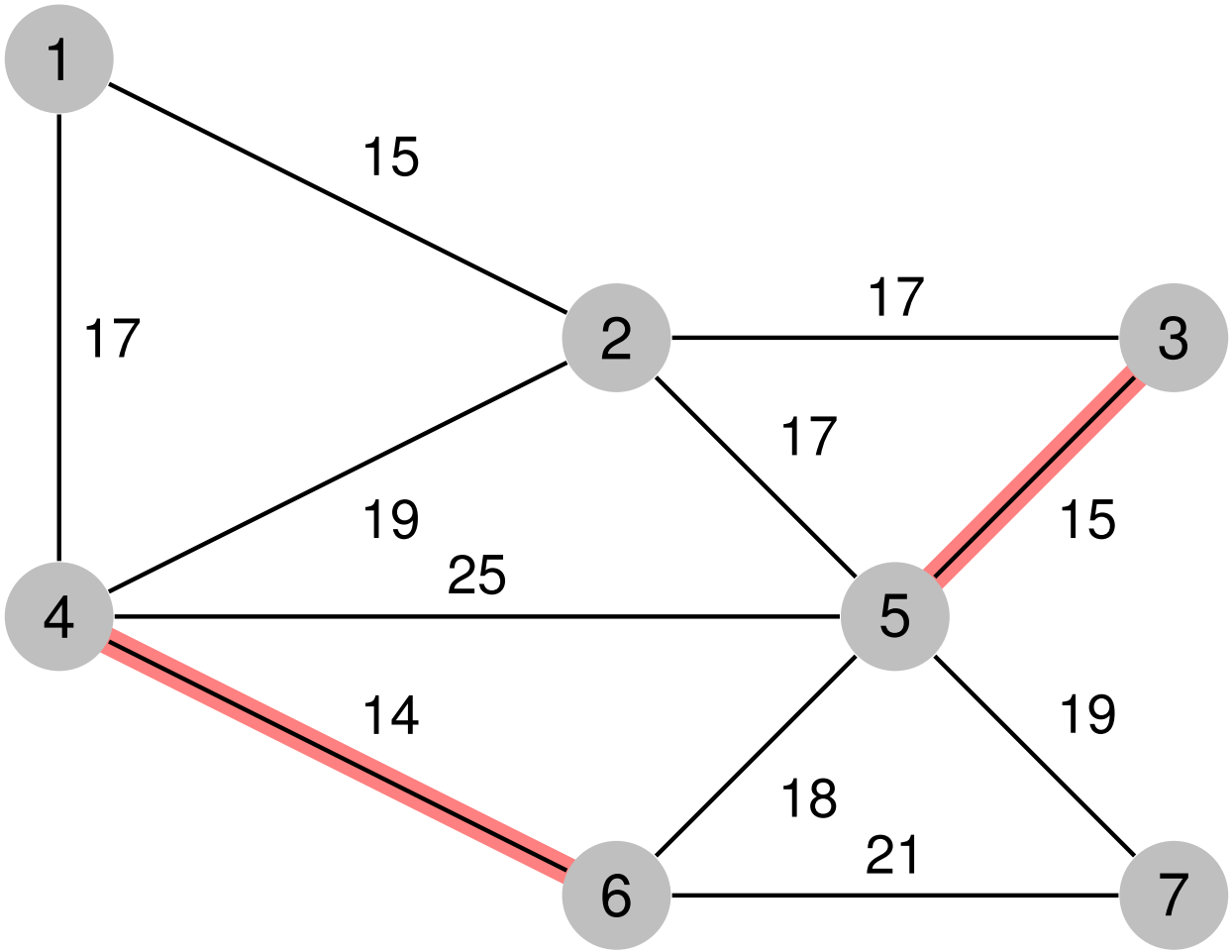
Algoritmo de Kruskal



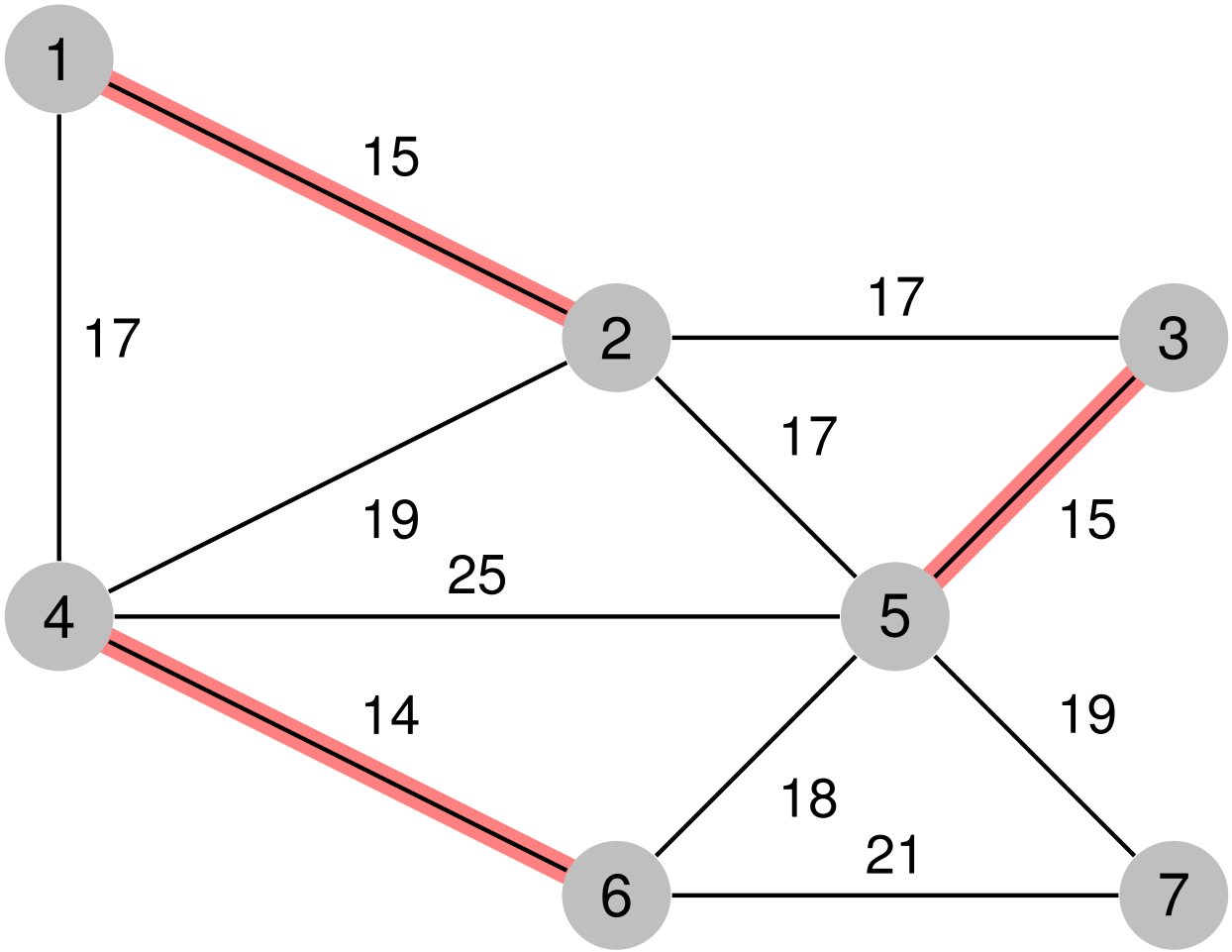
Algoritmo de Kruskal



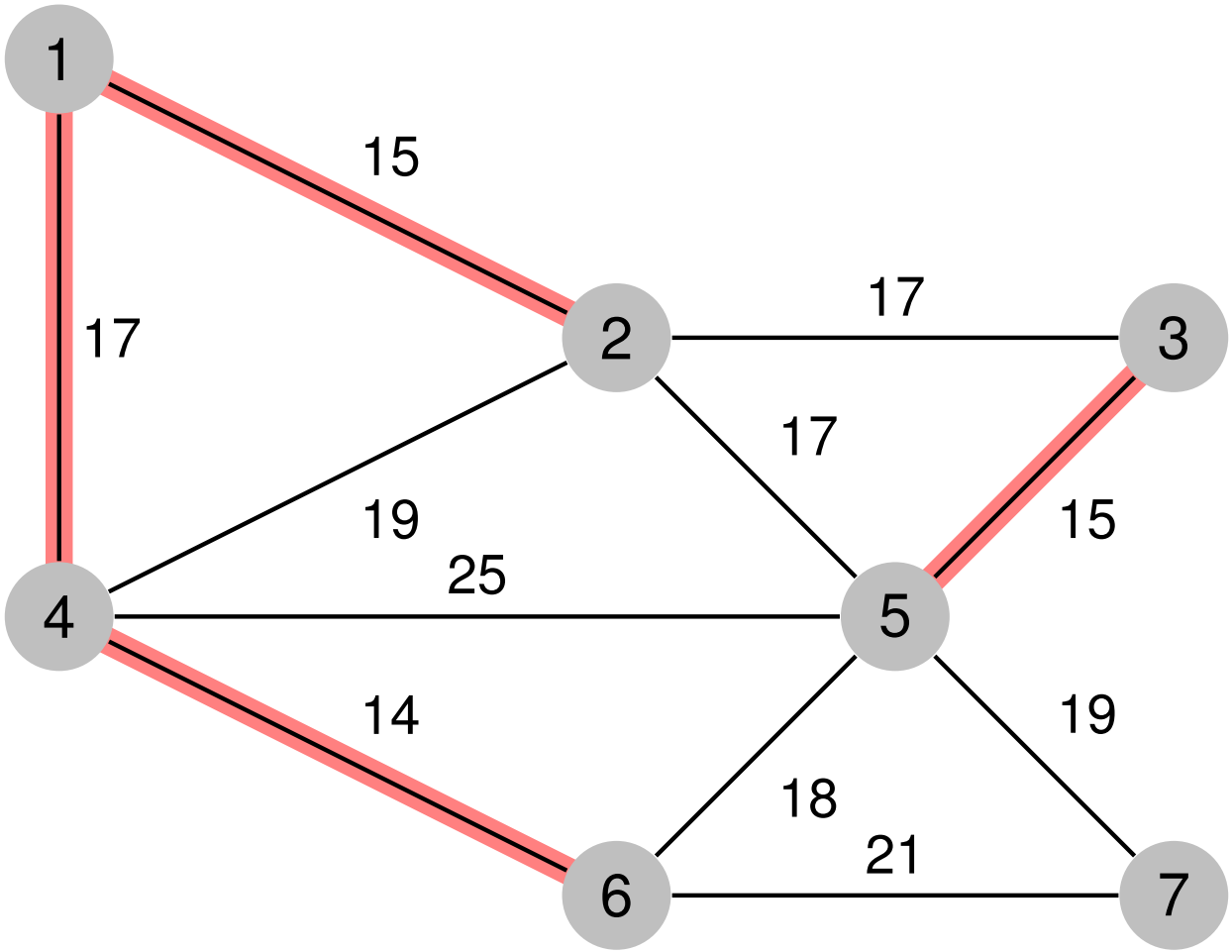
Algoritmo de Kruskal



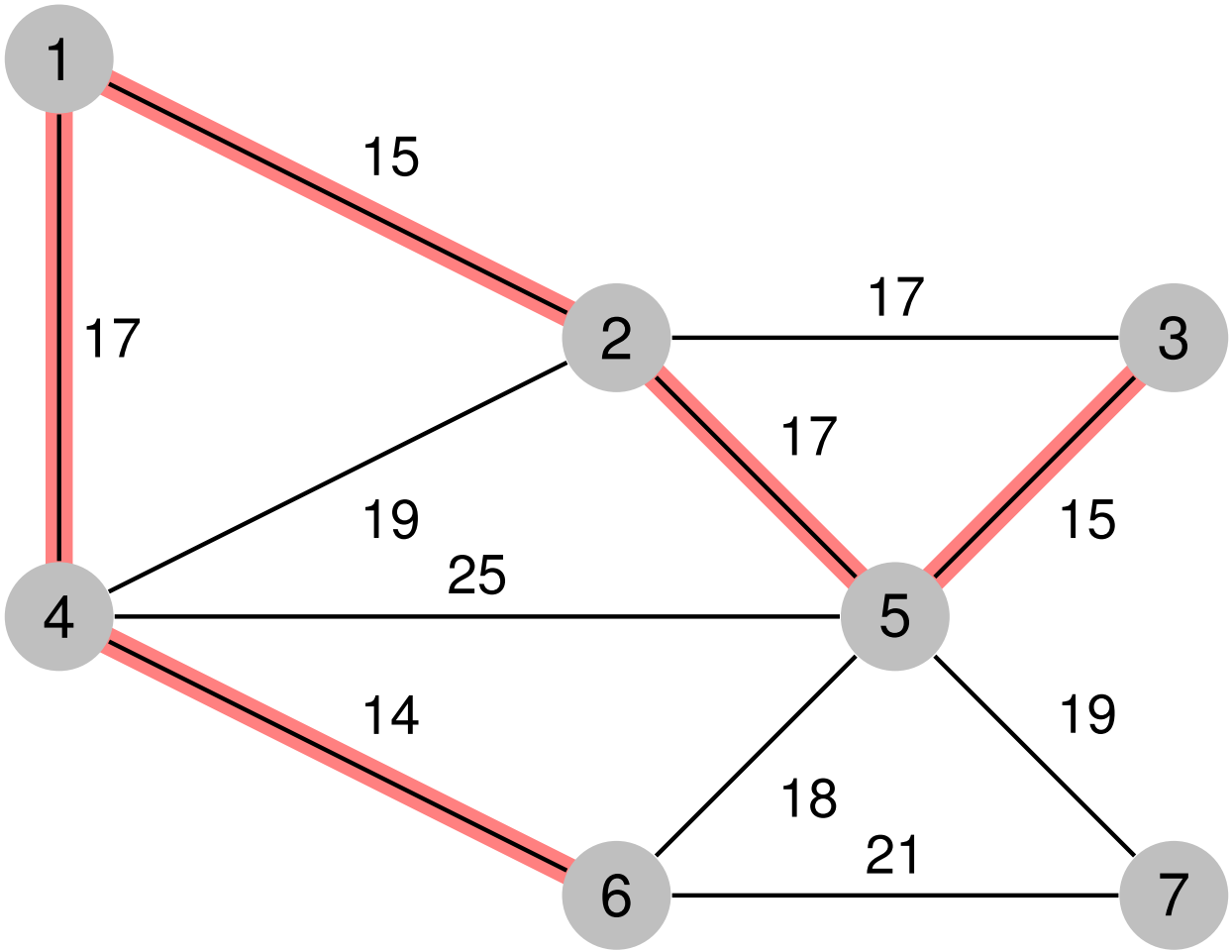
Algoritmo de Kruskal



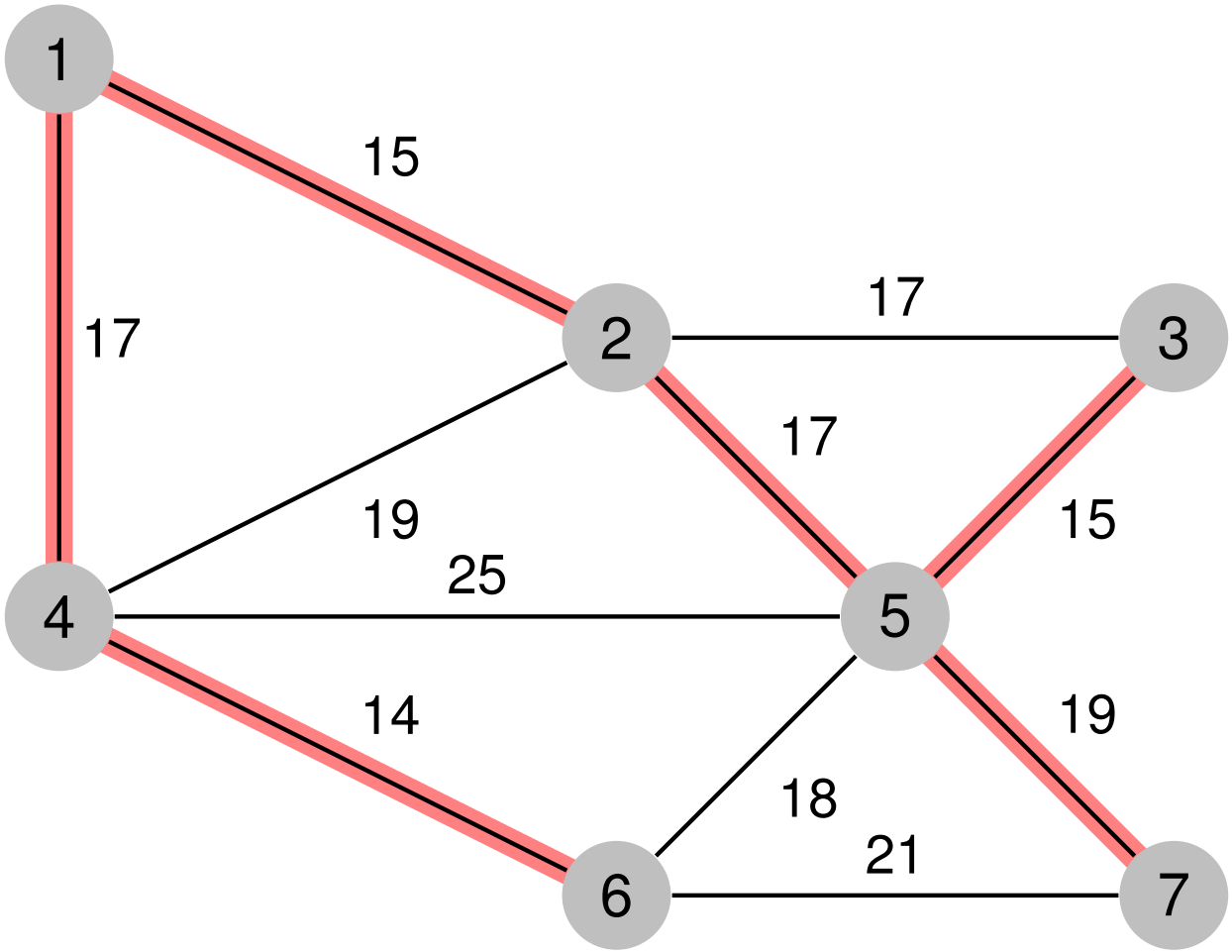
Algoritmo de Kruskal



Algoritmo de Kruskal



Algoritmo de Kruskal



Implementación del Algoritmo de Prim

fun Prim(G=(V,A) con costos en las aristas, k: V)
ret T: conjunto de aristas

var c: arista
C:= V-{k}
T:= {}
do n-1 **times** $\rightarrow$
c:= arista $\{i, j\}$ de costo mínimo tal que $i \in$ C y $j \notin$ C
C:= C-{i}
T:= T $\cup$ {c}
od
end fun

Implementación del Algoritmo de Kruskal

fun Kruskal(G=(V,A) con costos en las aristas)
ret T: conjunto de aristas
var i,j: vértice; u,v: componente conexa; c: arista
C:= A
T:= {}
do |T| < n - 1 $\rightarrow$ c:= arista $\{i, j\}$ de C de costo mínimo
C:= C-{c}
u:= find(i)
v:= find(j)
if u $\neq$ v $\rightarrow$ T:= T $\cup$ {c}
union(u,v)
fi
od
end fun

El problema union-find

Es el problema de cómo mantener un conjunto finito de elementos distribuidos en distintas componentes. Las operaciones que se quieren realizar son tres:

- **init** inicializar diciendo que cada elemento está en una componente integrada exclusivamente por ese elemento,
- **find** encontrar la componente en que se encuentra un elemento determinado,
- **union** unir dos componentes para que pasen a formar una sola que tendrá la unión de los elementos que había en ambas componentes.

El problema union-find

- De sólo manipularse por estas tres operaciones, las componentes serán siempre disjuntas
- y siempre tendremos que la unión de todas ellas dará el conjunto de todos los elementos.
- Una componente corresponde a una clase de equivalencia donde la relación de equivalencia sería “a $\equiv$ b sii a y b pertenecen a la misma componente.”

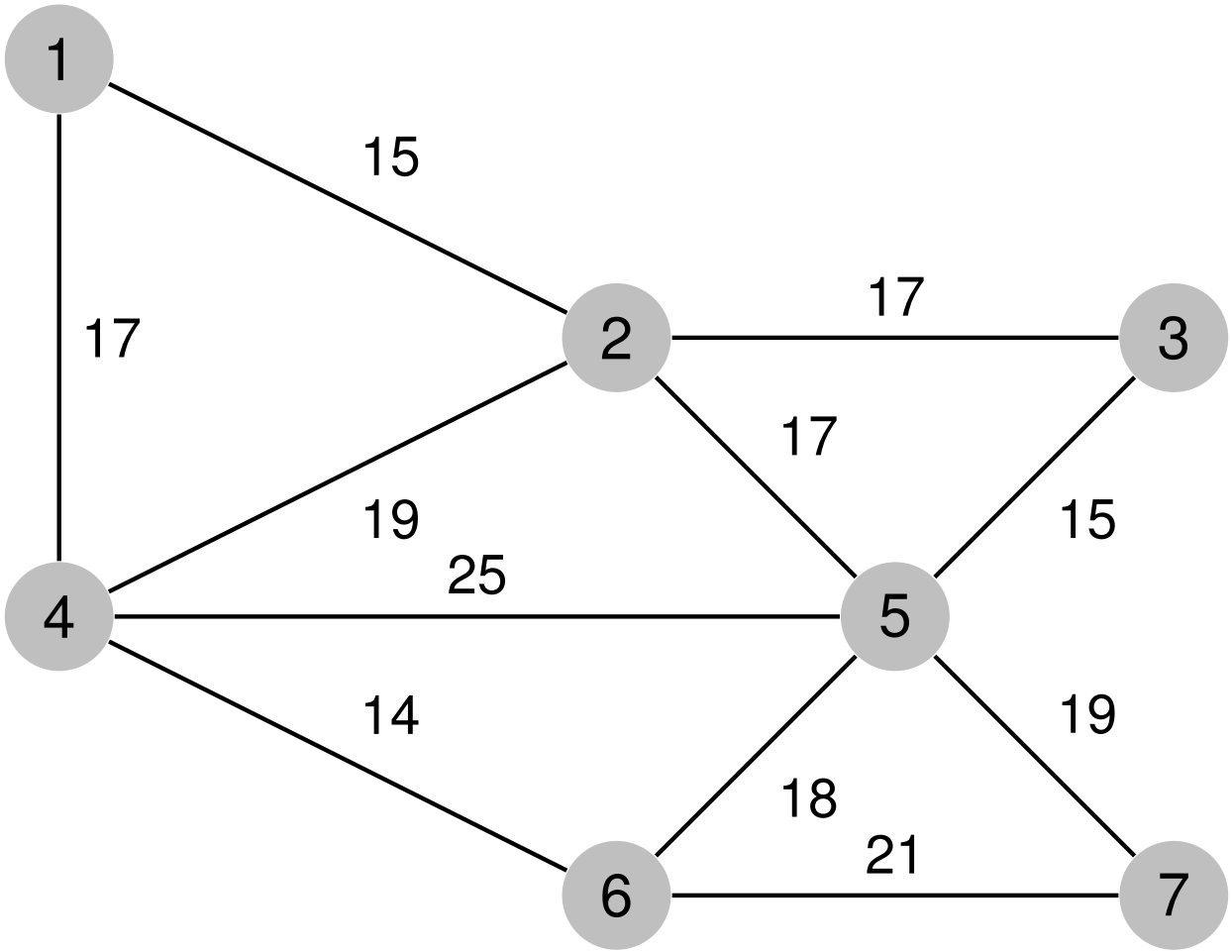
¿cómo implementar una componente?

- Podemos pensar que una componente estará dada por un representante de esa componente.
- Esto permite implementarlas a través de una tabla que indica para cada elemento cuál es el representante de (la componente de) dicho elemento.
- Dado que asumimos una cantidad finita de elementos, los denotamos con números de 1 a n.
- La tabla que indica cuál es el representante de cada elemento será entonces un arreglo indexado por esos números:

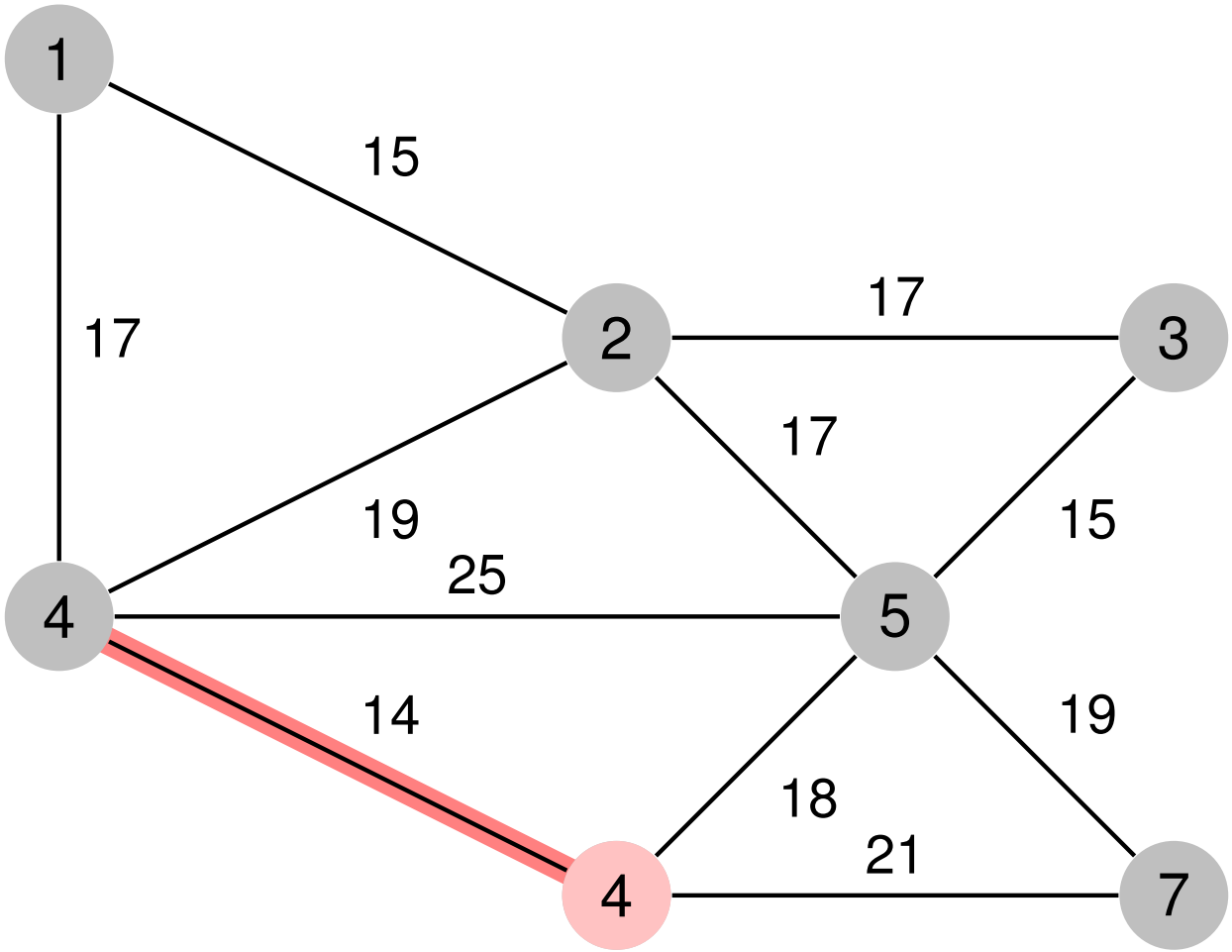
  type trep = **array**[1..n] **of int**

Clase de hoy

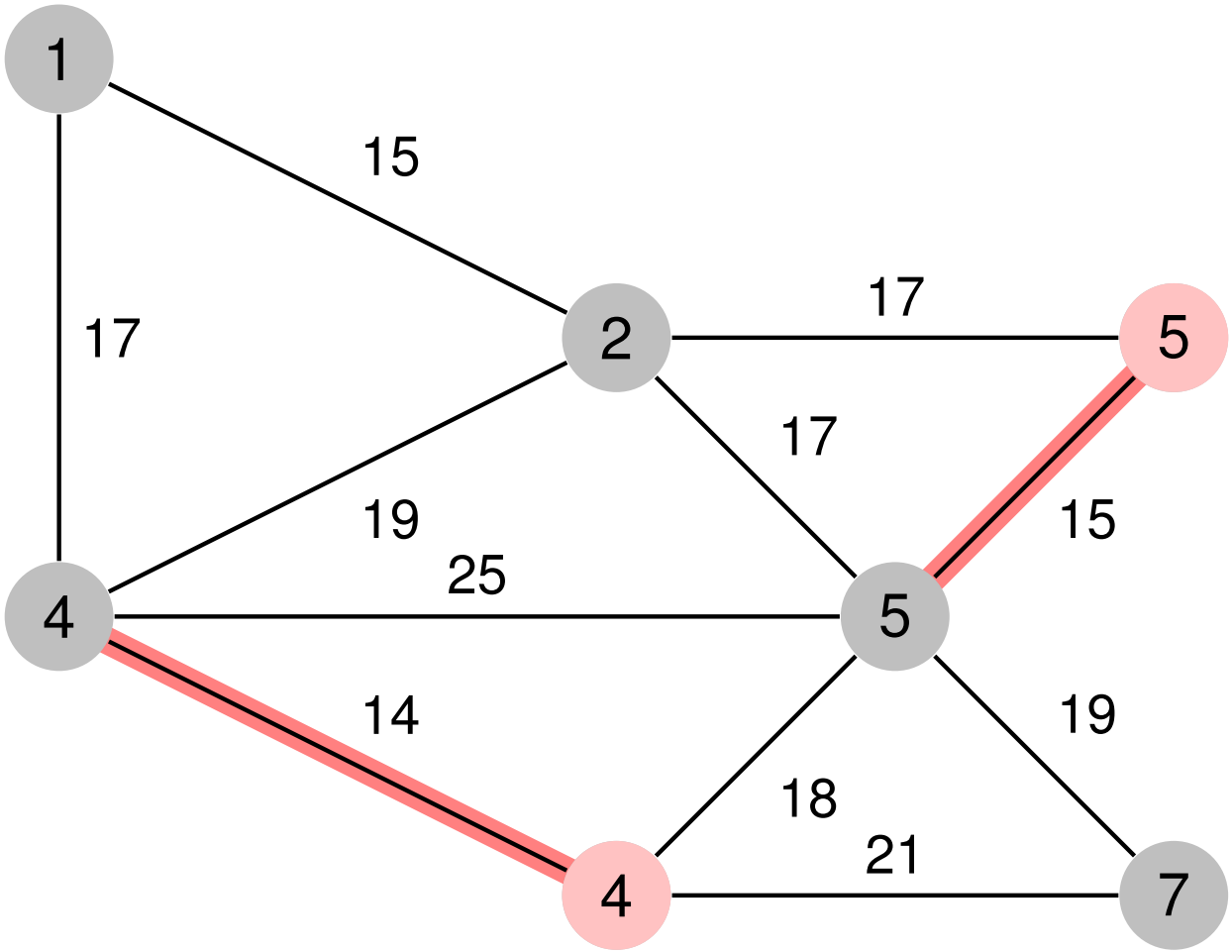
Algoritmo de Kruskal, primer intento



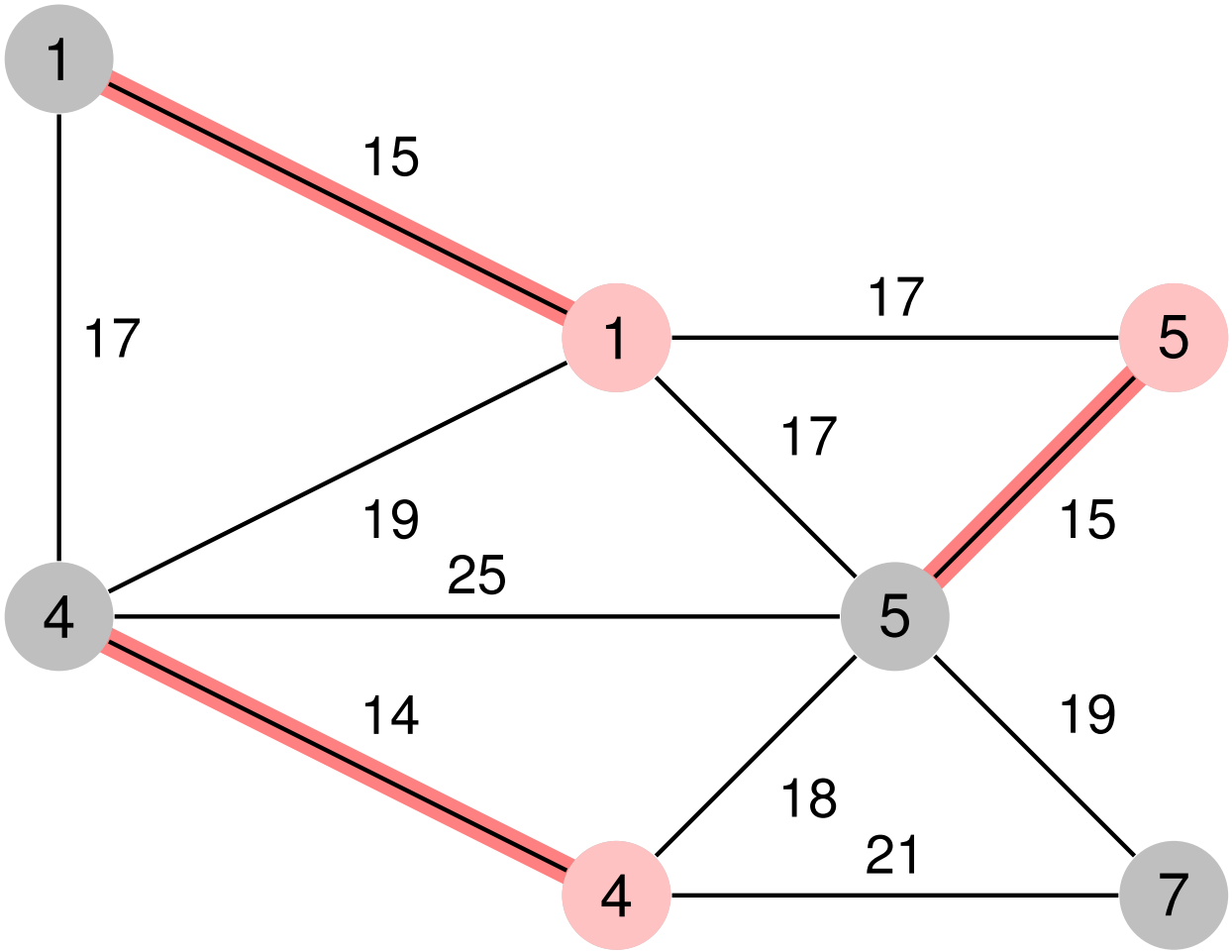
Algoritmo de Kruskal, primer intento



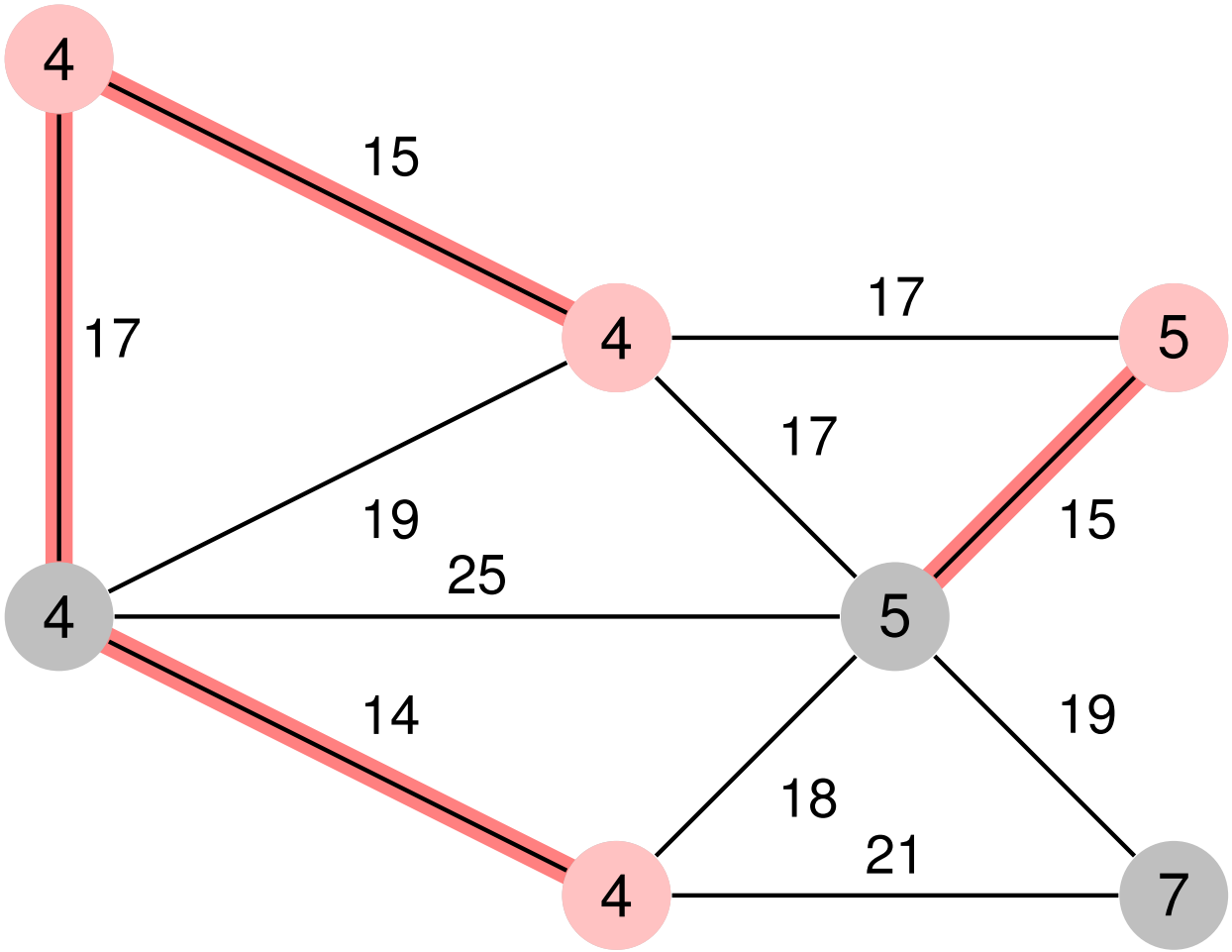
Algoritmo de Kruskal, primer intento



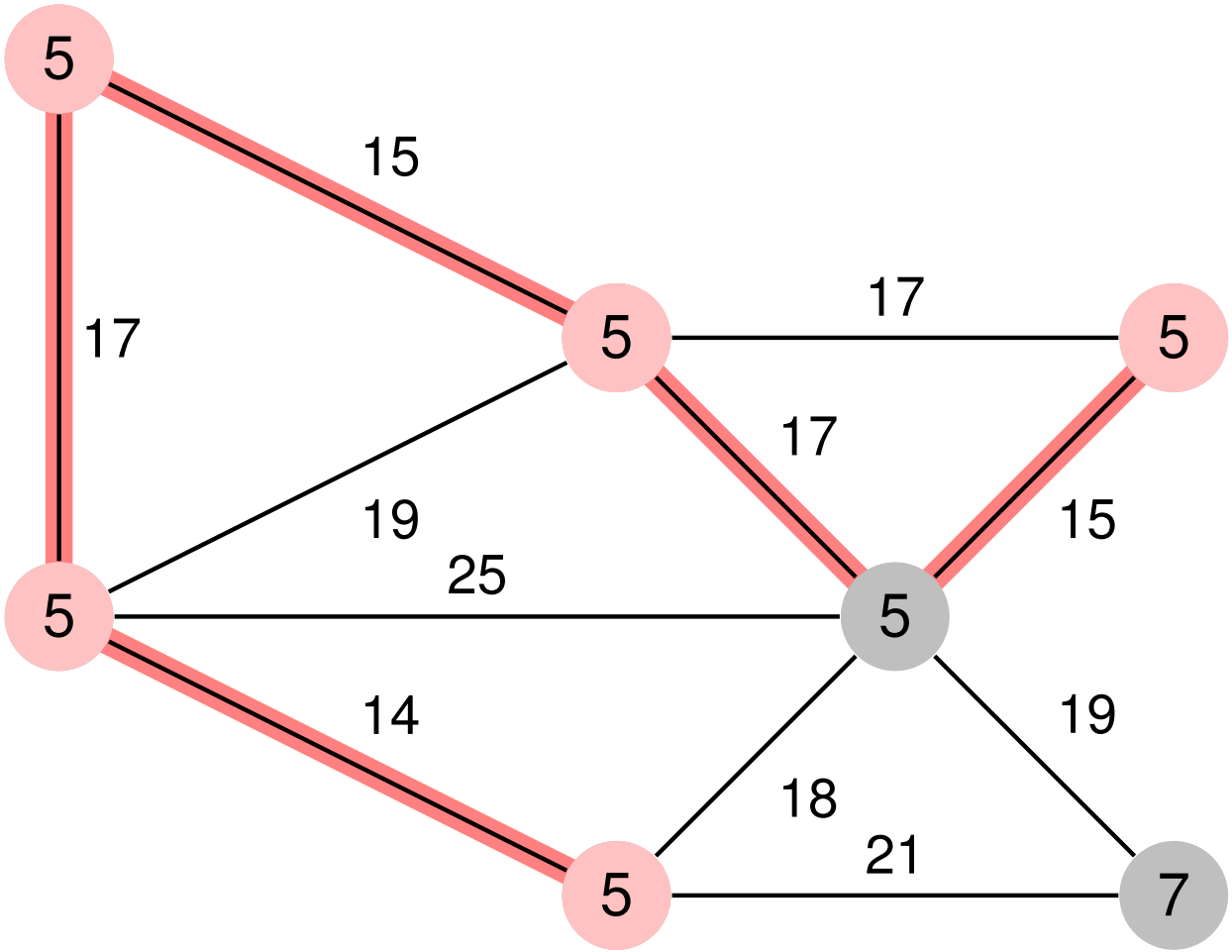
Algoritmo de Kruskal, primer intento



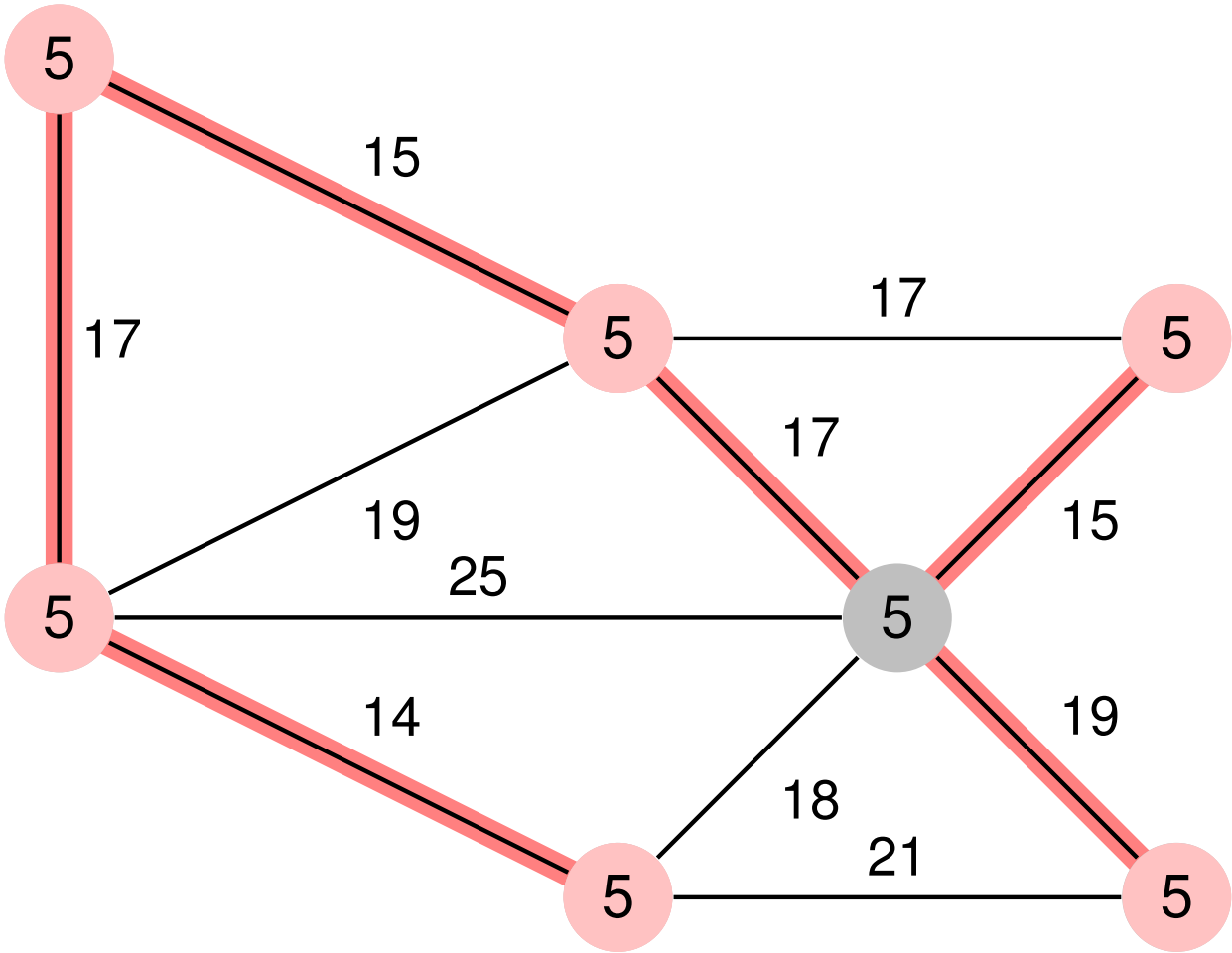
Algoritmo de Kruskal, primer intento



Algoritmo de Kruskal, primer intento



Algoritmo de Kruskal, primer intento



Primer intento

```
proc init(out rep: trep)
        for i:= 1 to n do rep[i]:= i od
end proc

fun find(rep: trep, i: int) ret r: int
     r:= rep[i]
end fun
```

{Pre: $u \neq v \wedge u = rep[u] \wedge v = rep[v]$}

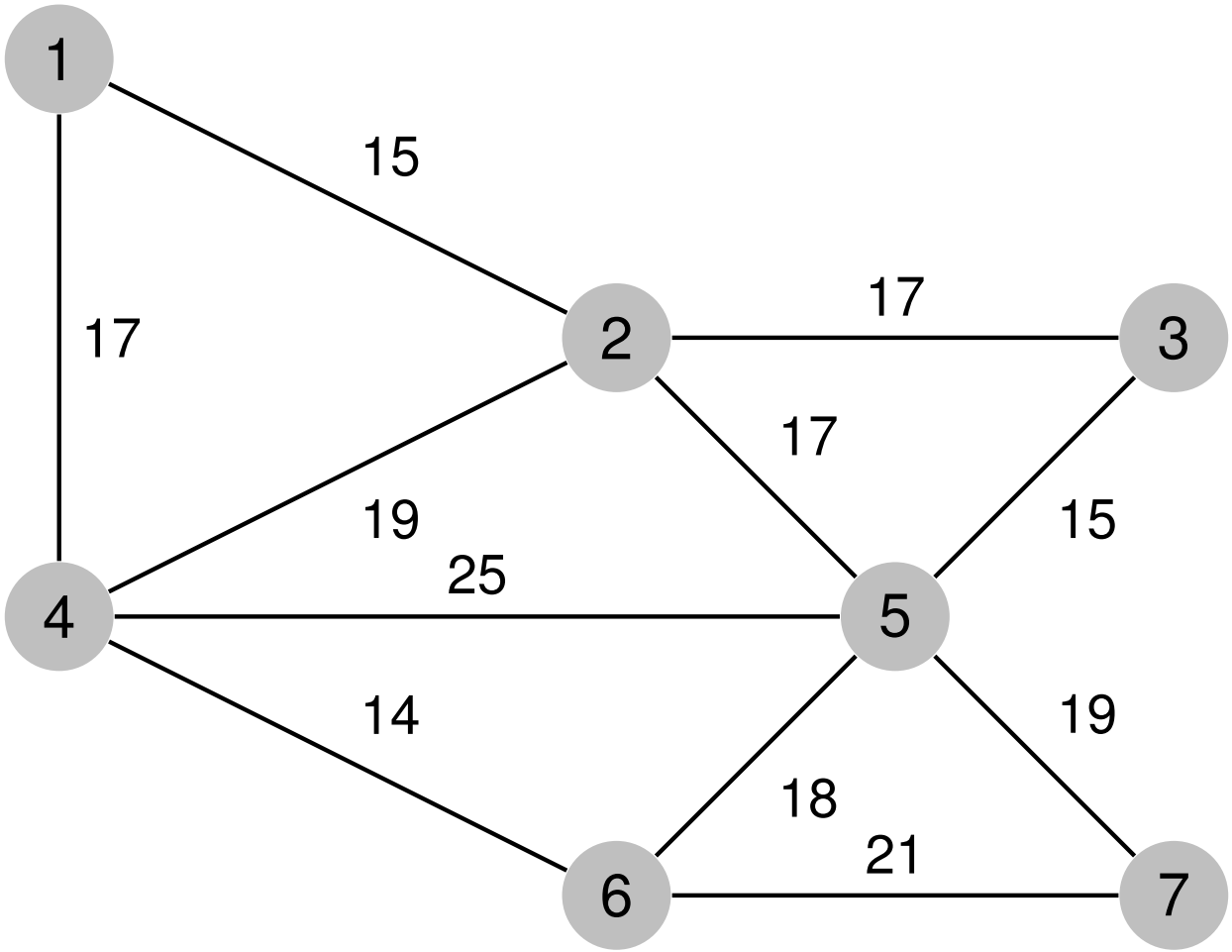
```
proc union(in/out rep: trep, in u,v: int)
        for i:= 1 to n do
             if rep[i]=u → rep[i]:= v fi
        od
end proc
```

Primer intento
Análisis

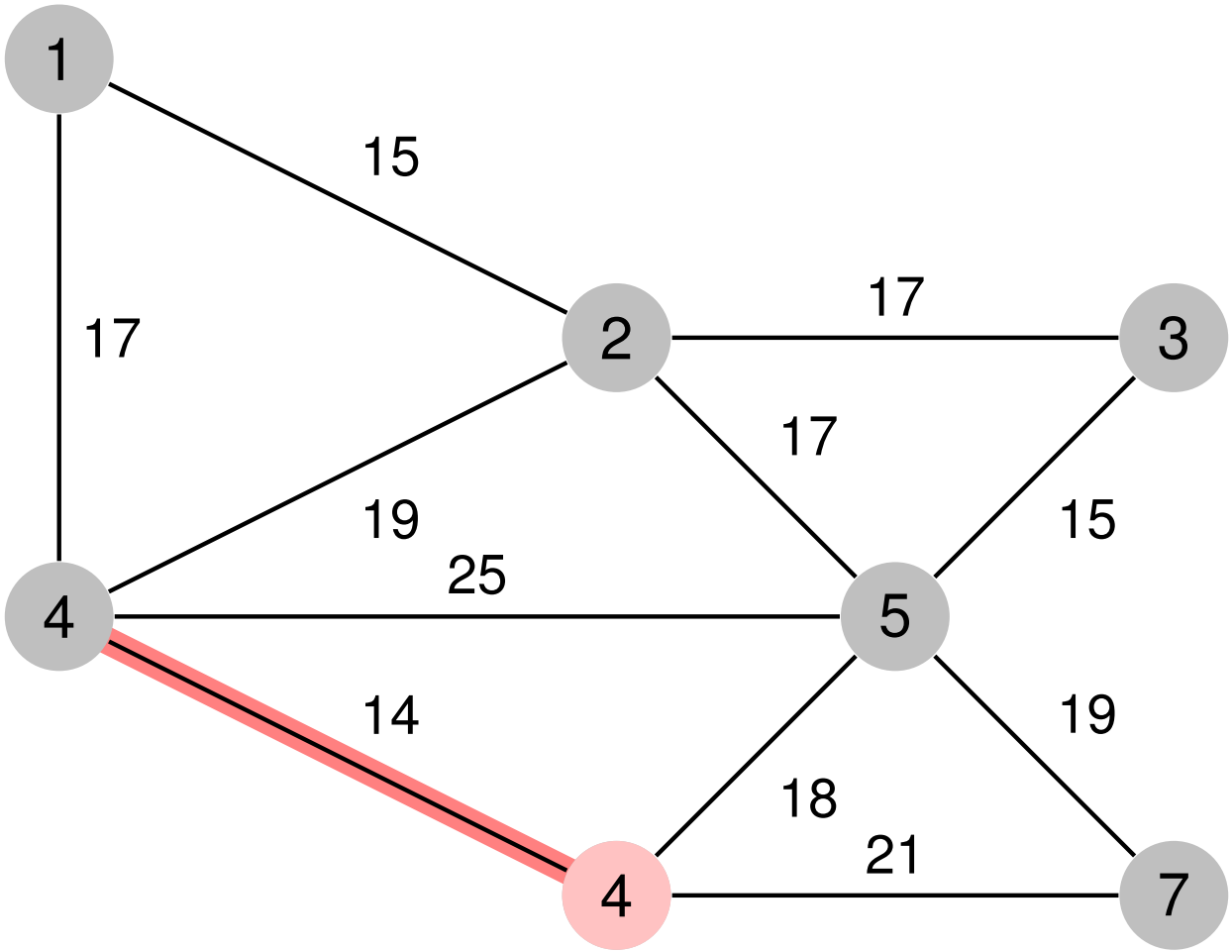
init	es lineal
find	es constante
union	es lineal

Clase de hoy

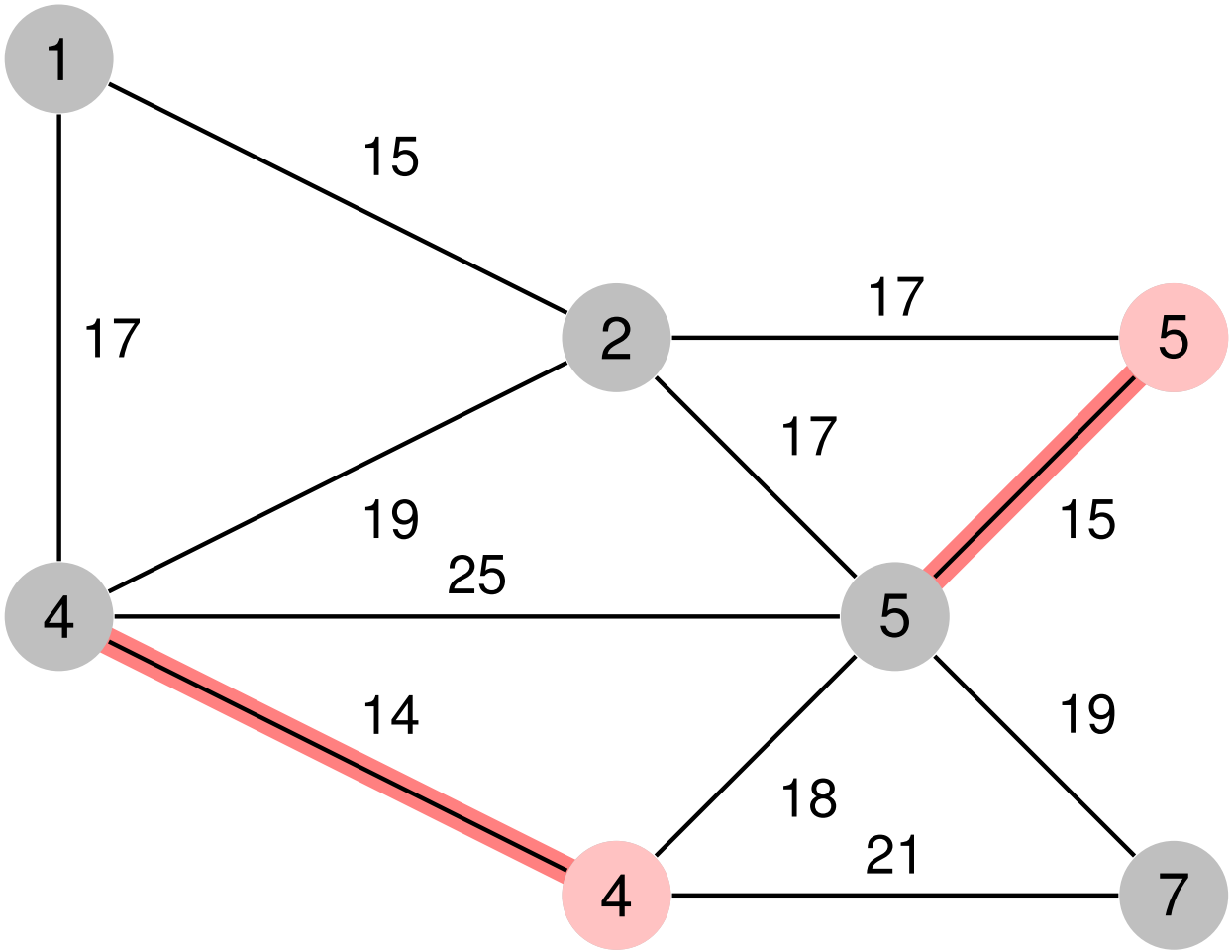
Algoritmo de Kruskal, segundo intento



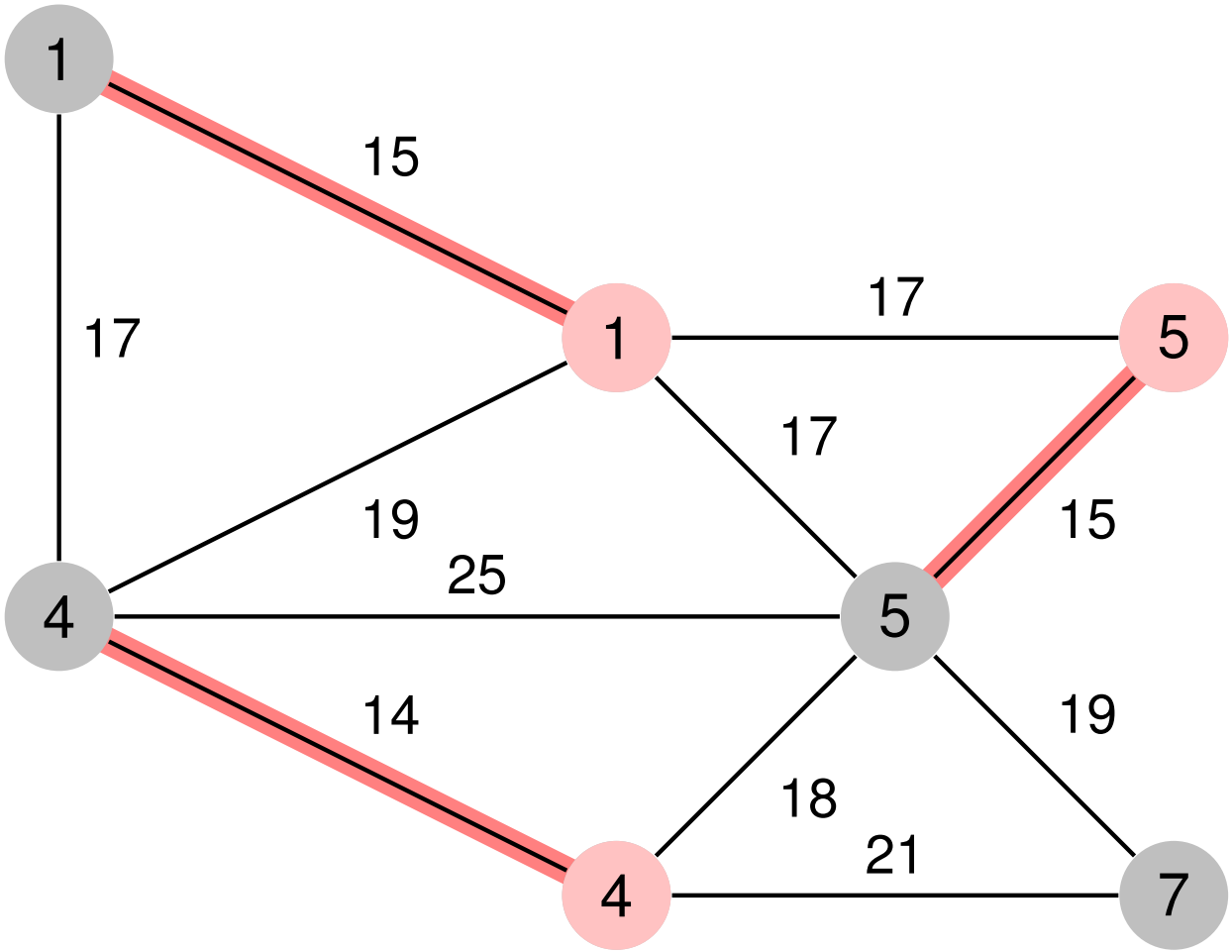
Algoritmo de Kruskal, segundo intento



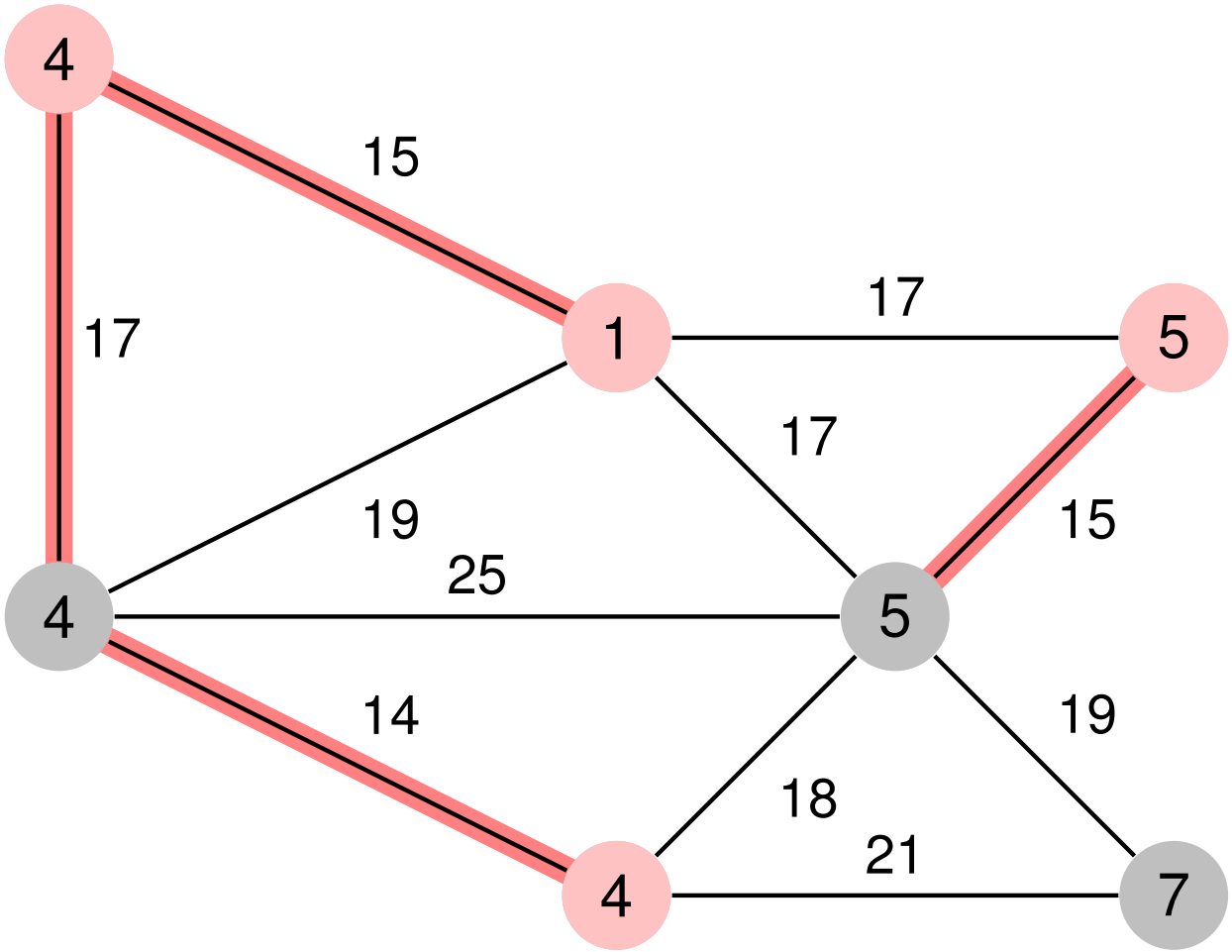
Algoritmo de Kruskal, segundo intento



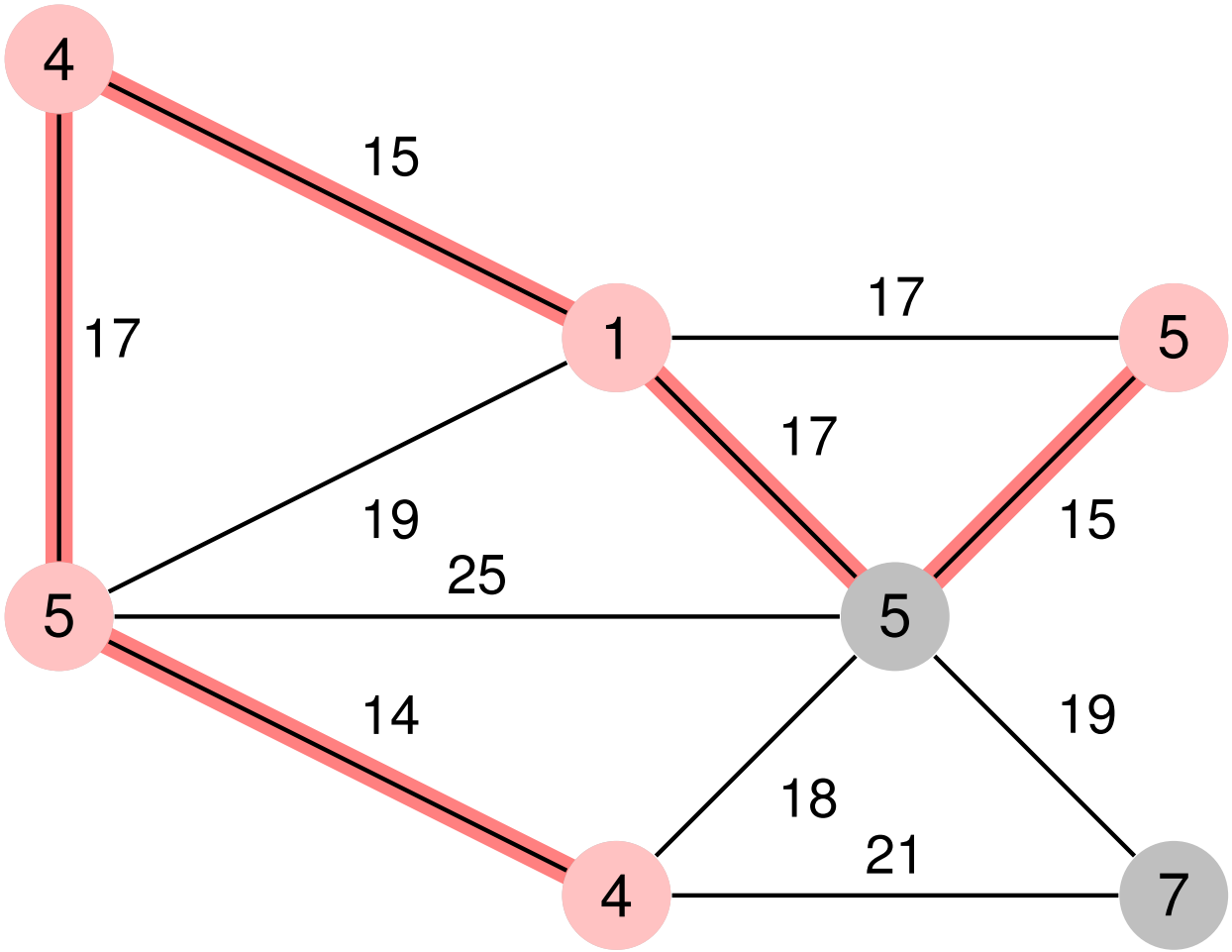
Algoritmo de Kruskal, segundo intento



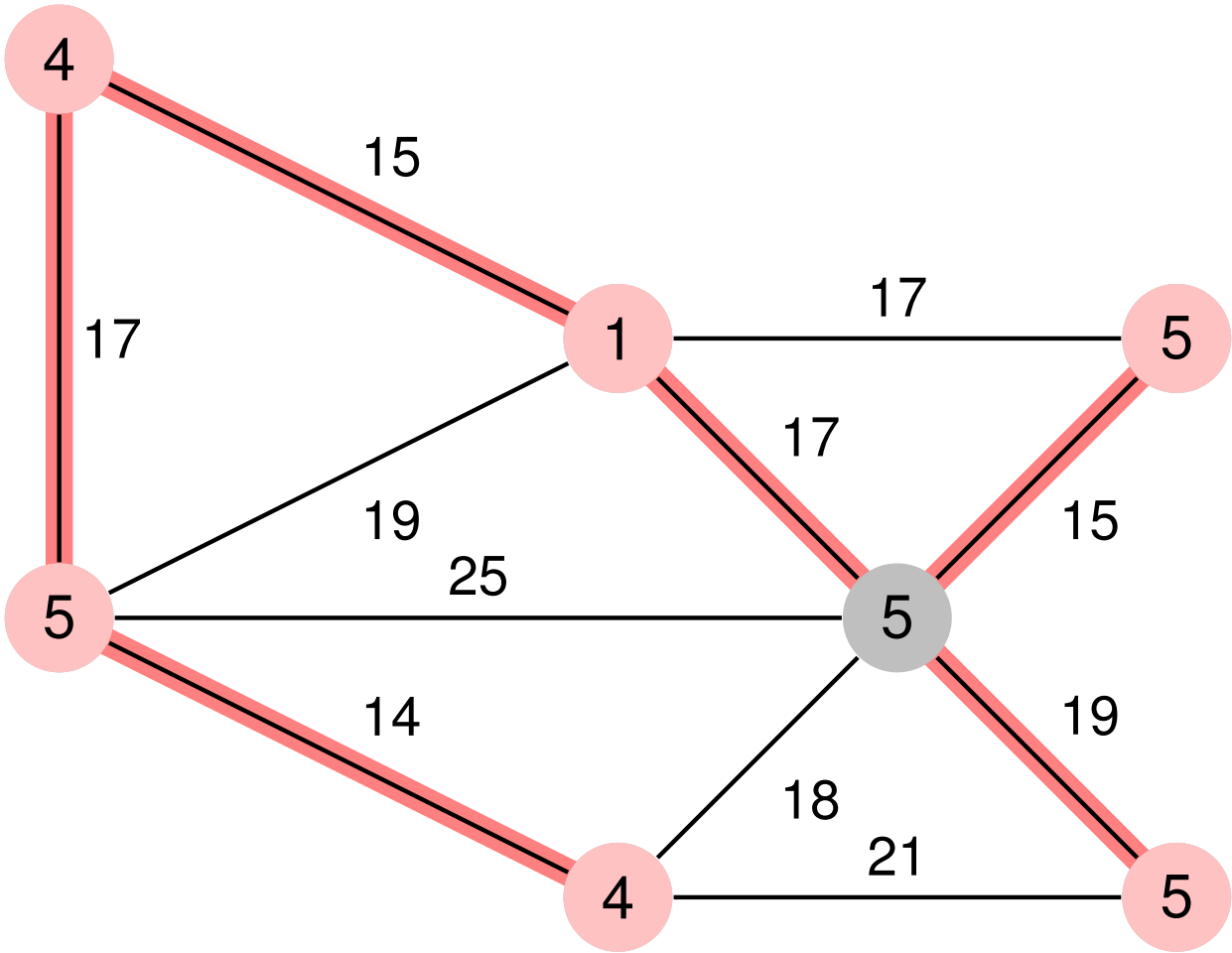
Algoritmo de Kruskal, segundo intento



Algoritmo de Kruskal, segundo intento



Algoritmo de Kruskal, segundo intento



Segundo intento

```
fun is_rep(rep: trep, i: int) ret b: bool
    b:= (rep[i] = i)
end fun
```

{Pre: $u \neq v \wedge$ is_rep(rep,u) $\wedge$ is_rep(rep,v)}

```
proc union(in/out rep: trep, in u,v: int)
      rep[u]:= v
end proc
```

```
fun find(rep: trep, i: int) ret r: int
    var j: int
    j:= i;
    do ¬ is_rep(rep,j) → j:= rep[j] od
    r:= j
end fun
```

Segundo intento
Análisis

init	es lineal
find	es lineal en el peor caso
union	es constante

Clase de hoy

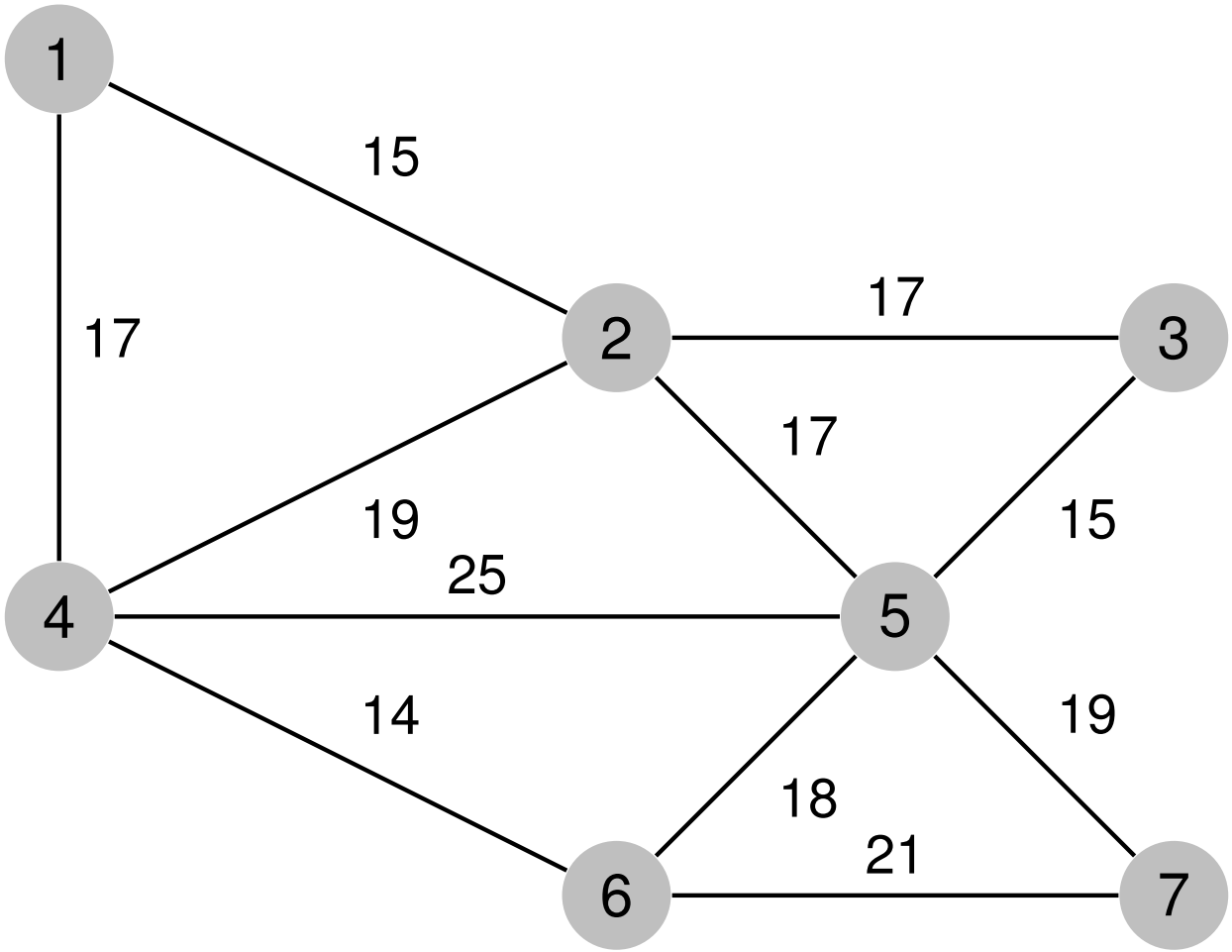
Tercer intento

```
fun find(in/out rep: trep, i: int) ret r: int
    var j,k: int
    j:= i
    while ¬ is_rep(rep,j) do j:= rep[j] od
    r:= j
    j:= i
    while ¬ is_rep(rep,j) do
        k:= rep[j]
        rep[j]:= r
        j:= k
    od
end fun
```

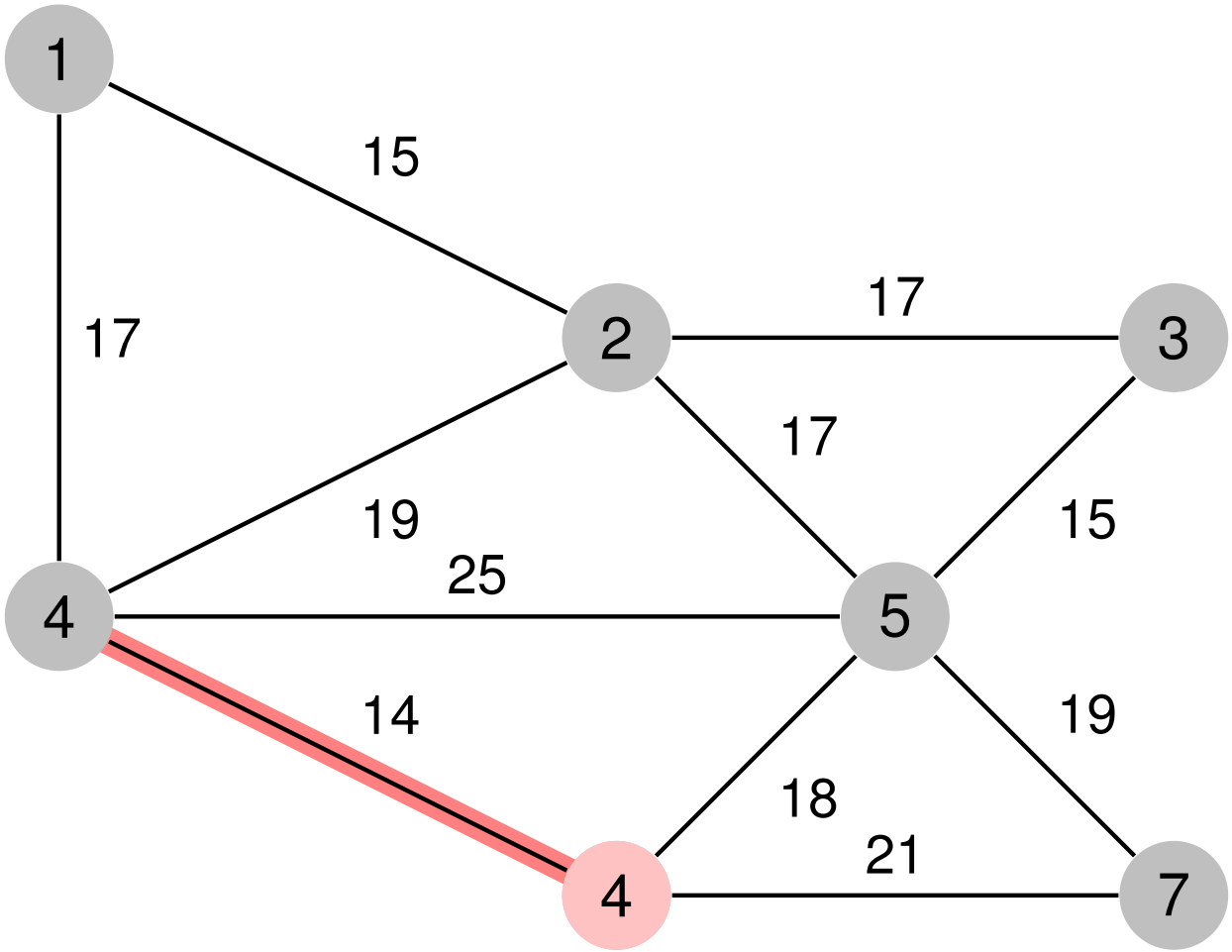
Explicación

- Una vez calculado el representante r, la función find realiza una segunda recorrida desde i actualizando el arreglo rep.
- Tanto en la posición i, como en las posiciones de los j que fueron representantes de i se asigna directamente r.
- Esto vuelve constantes las futuras llamadas a find(rep,i) o find(rep,j).
- Observar el uso **excepcional** de **in**/**out** asociado al parámetro rep.
- Esto se debe a que find no es estrictamente una función ya que modifica dicho parámetro.
- De todas formas se comporta como función ya que find(rep,i) == find(rep,i) siempre da verdadero.

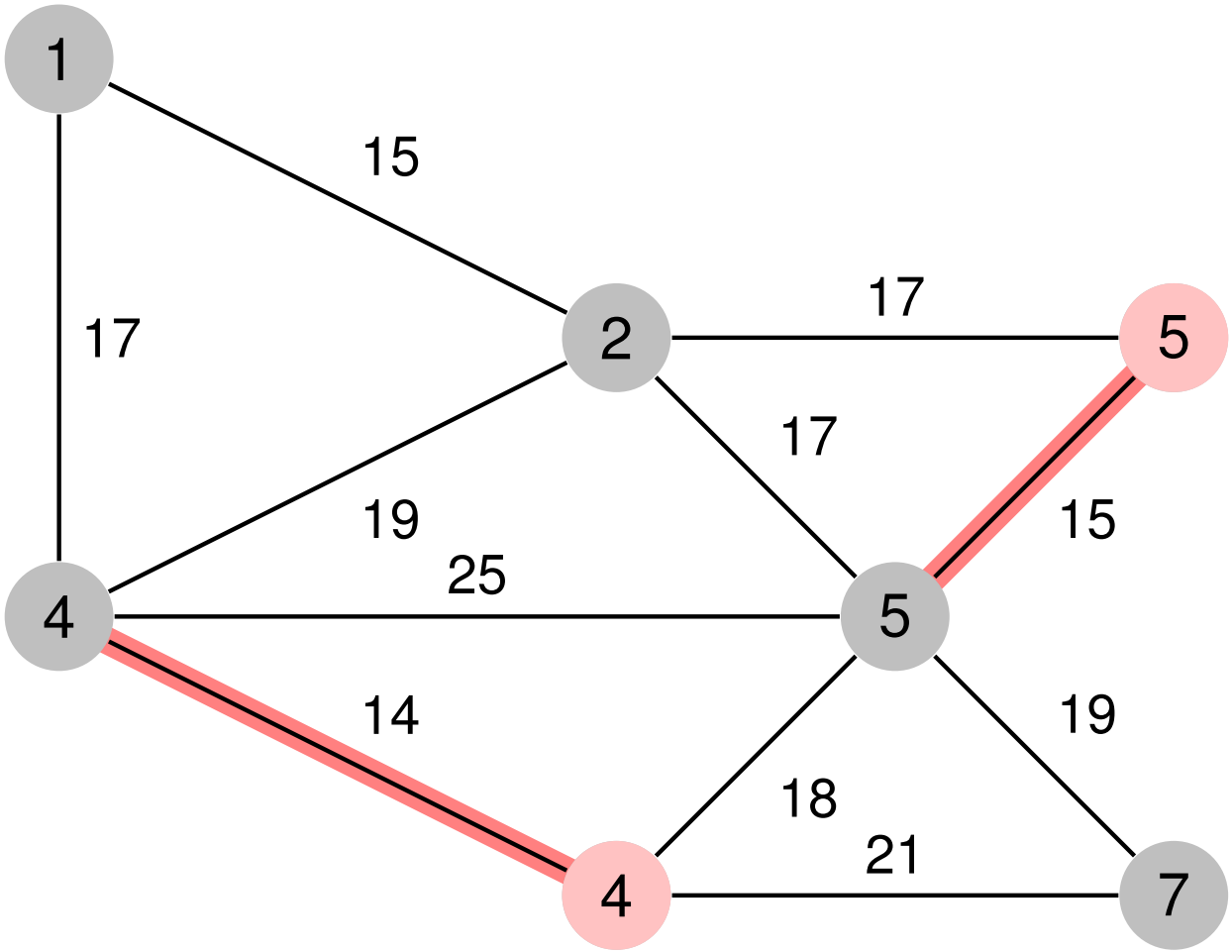
Algoritmo de Kruskal, tercer intento



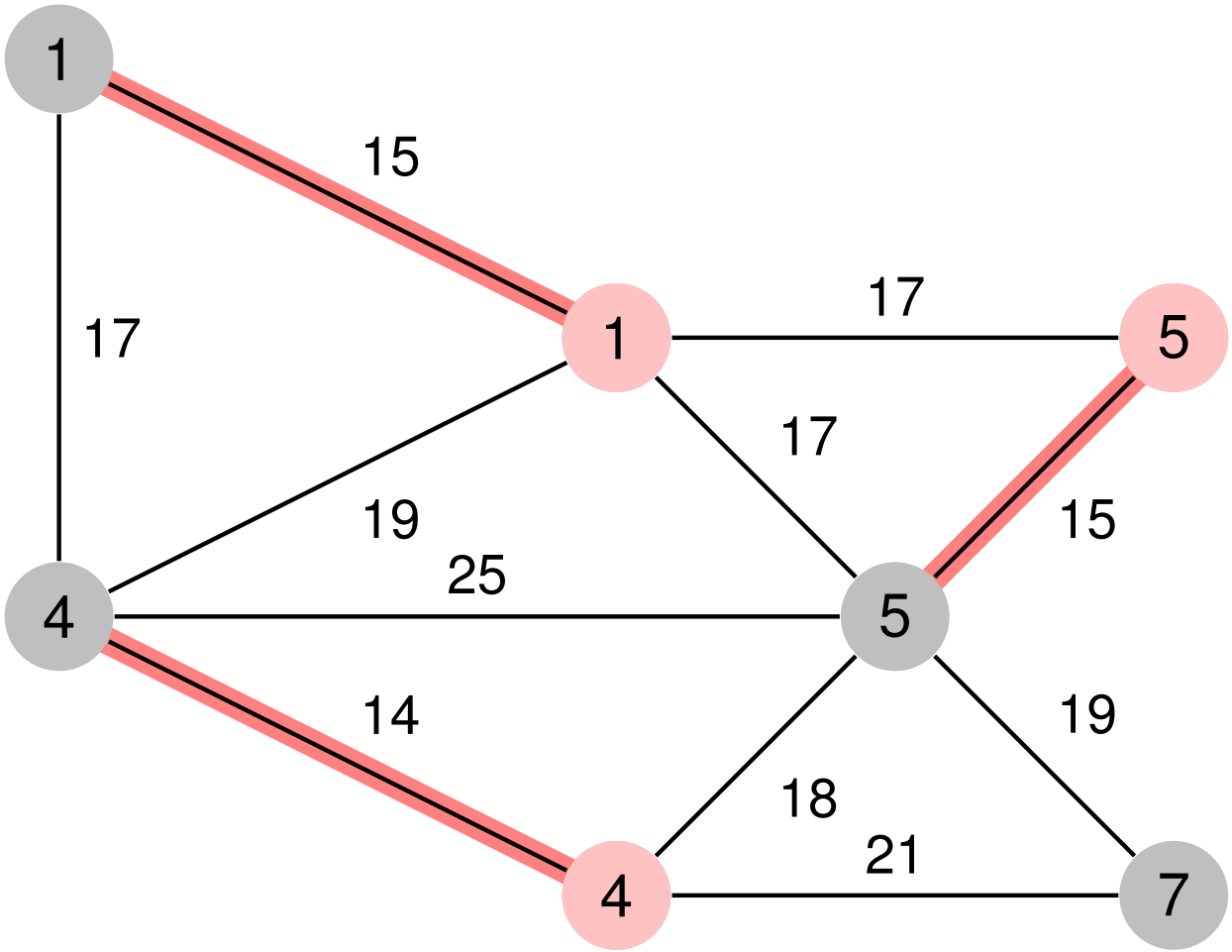
Algoritmo de Kruskal, tercer intento



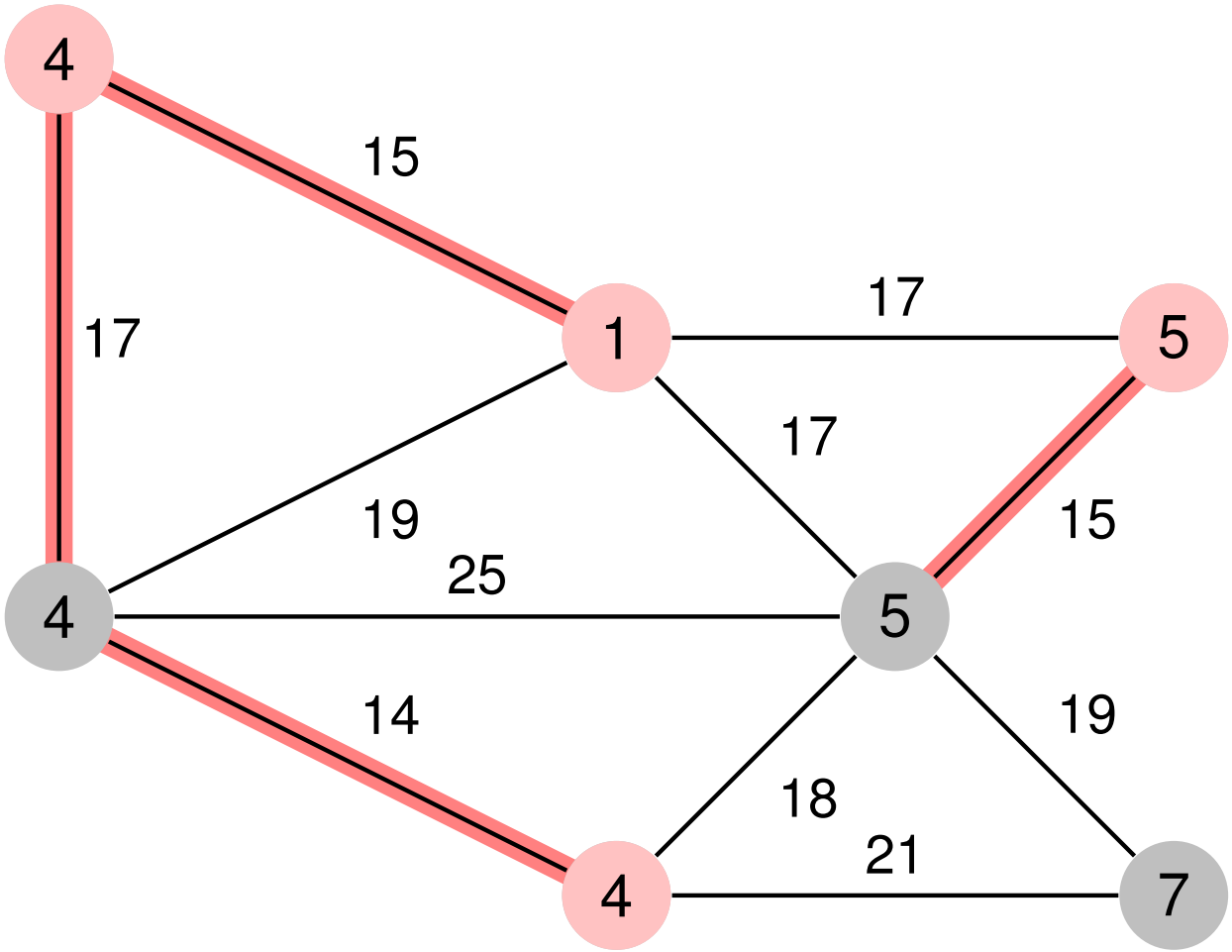
Algoritmo de Kruskal, tercer intento



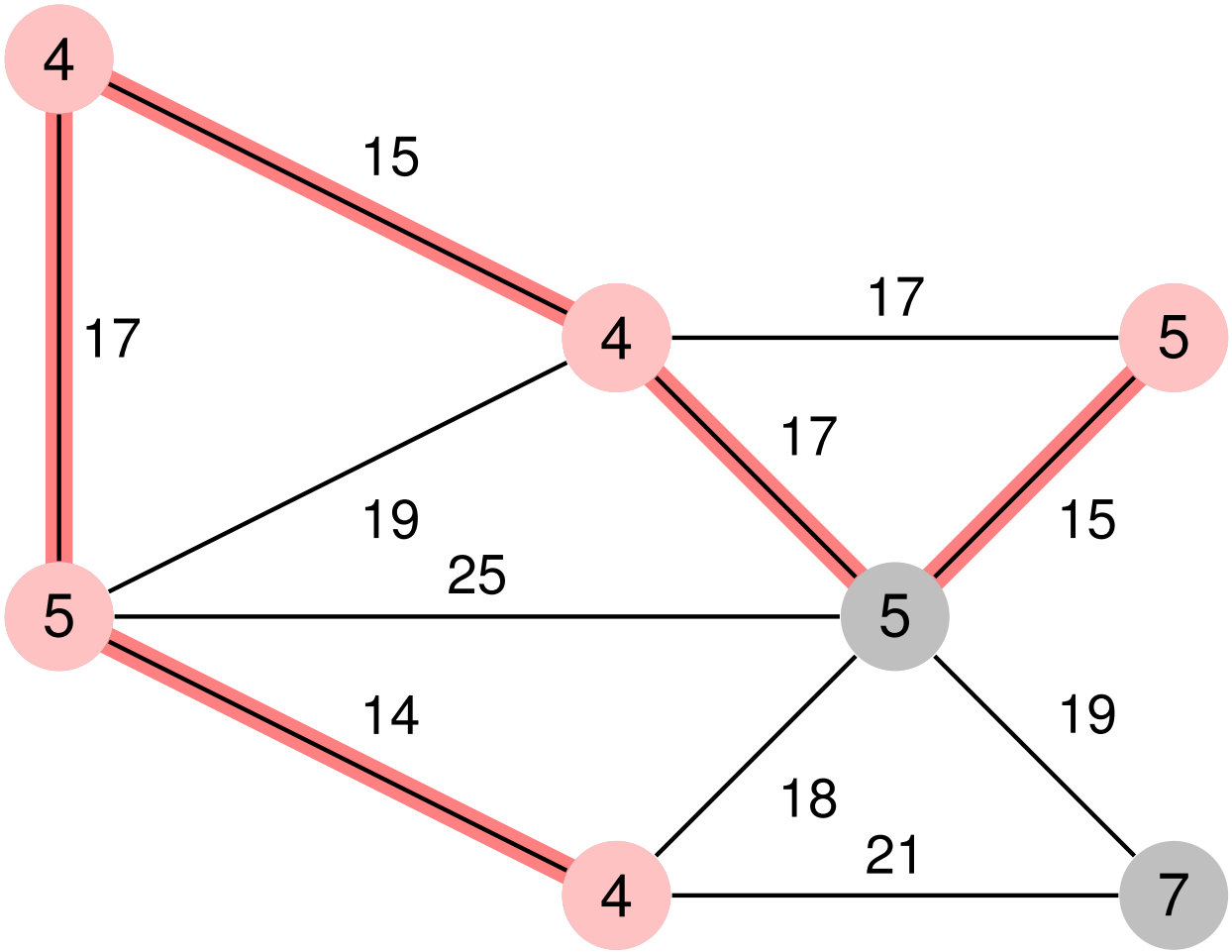
Algoritmo de Kruskal, tercer intento



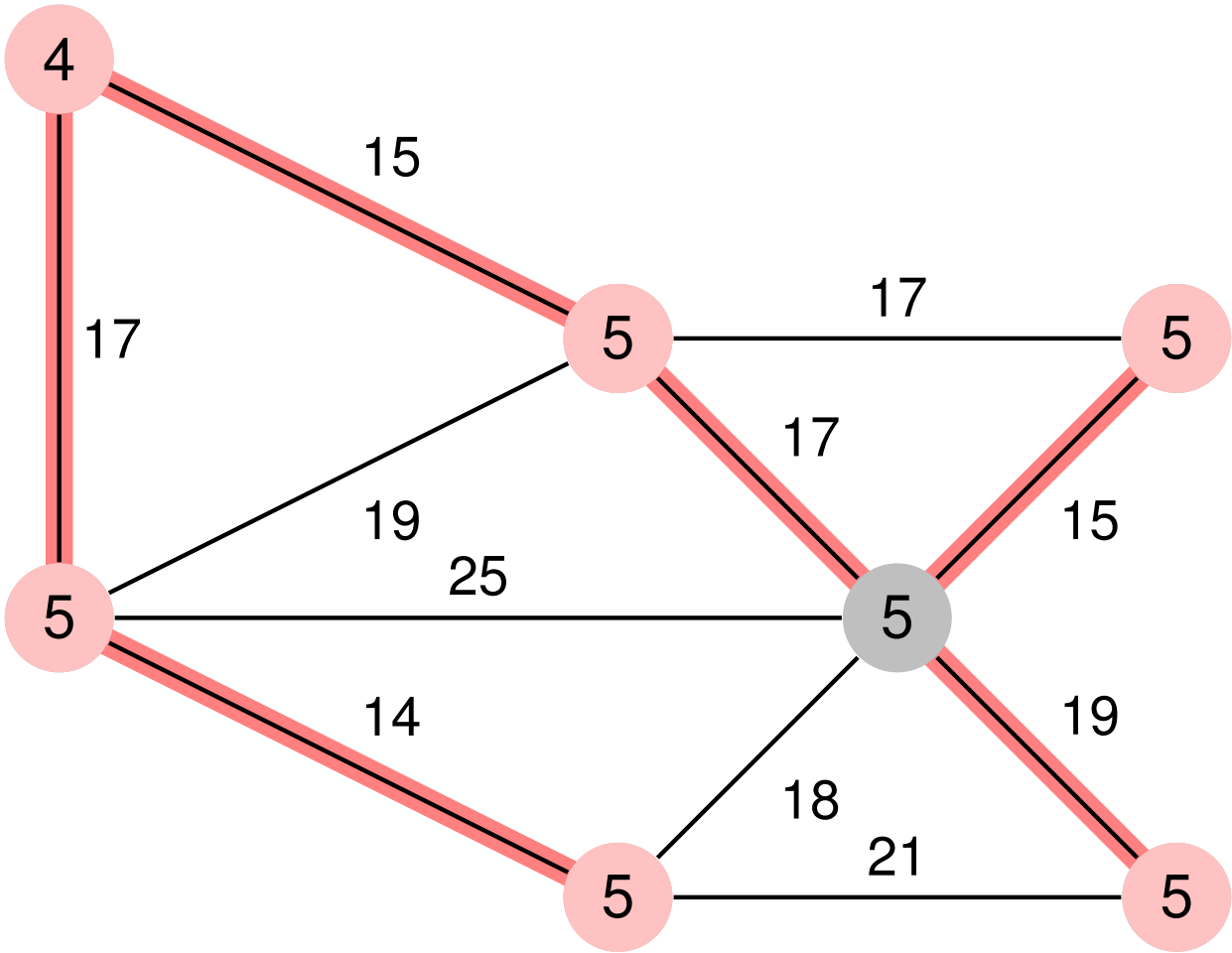
Algoritmo de Kruskal, tercer intento



Algoritmo de Kruskal, tercer intento



Algoritmo de Kruskal, tercer intento



Tercer intento
Análisis

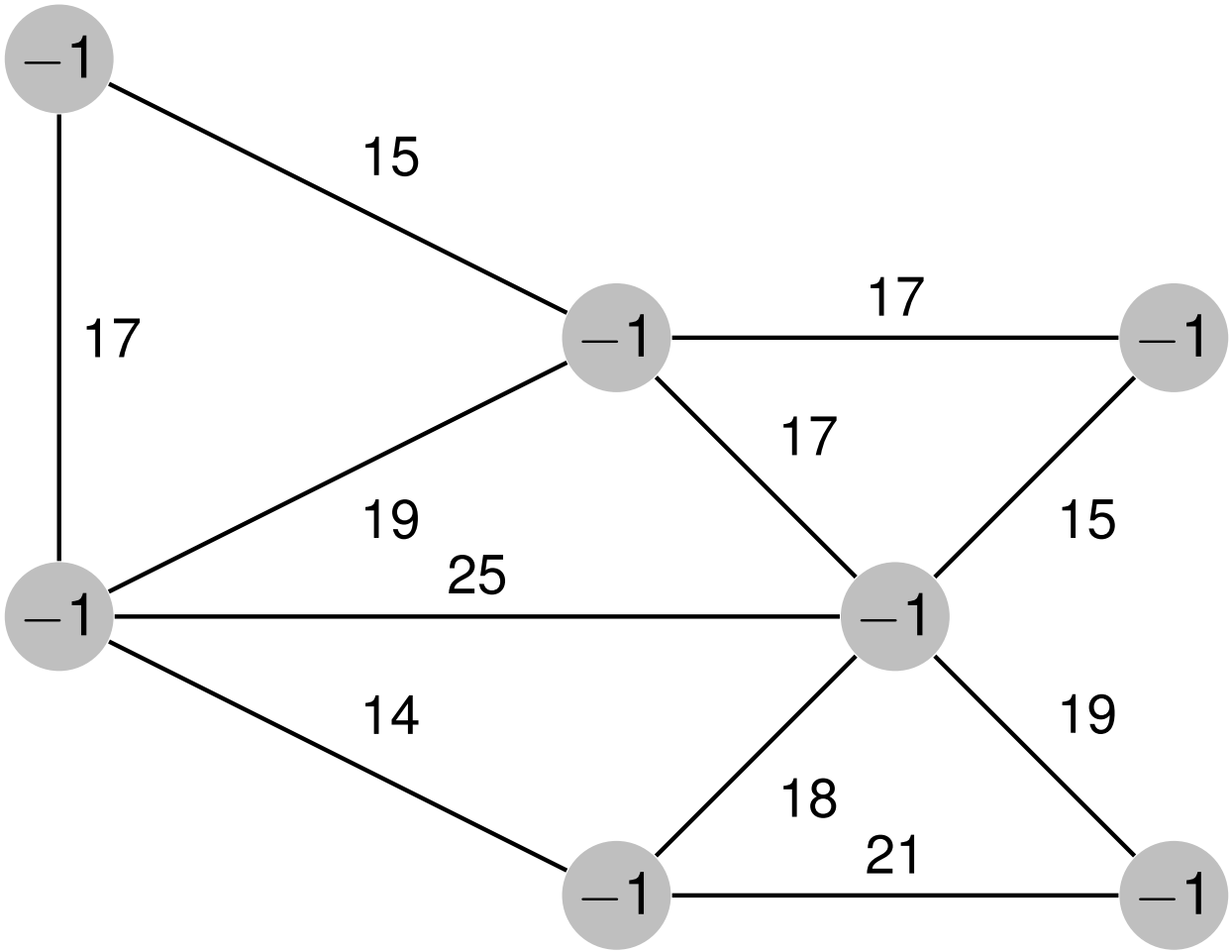
init	es lineal
find	es lineal en el peor caso, pero se torna constante después de ejecutarse
union	es constante

Clase de hoy

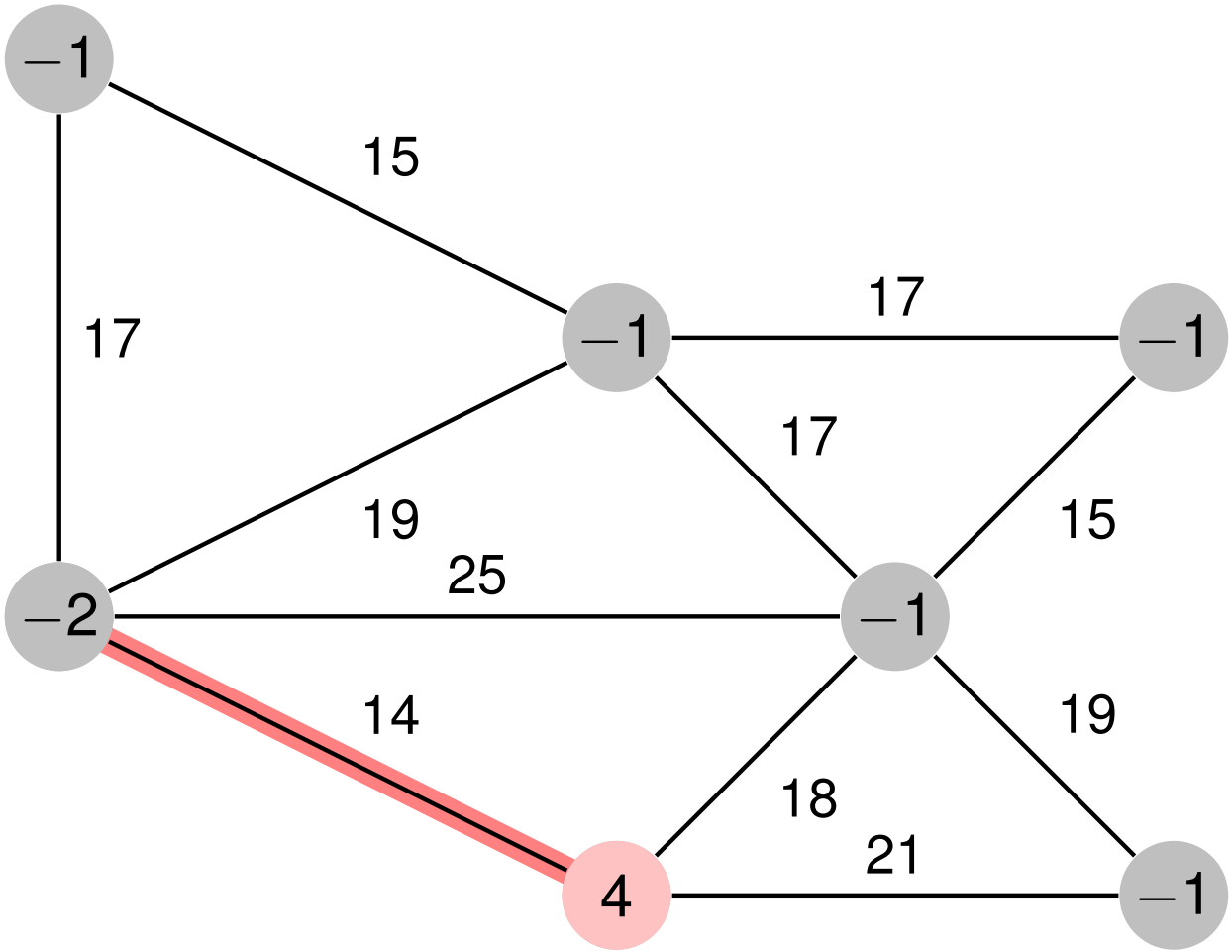
Último intento

```
proc init(out rep: trep)
      for i:= 1 to n do rep[i]:= -1 od
end proc

fun is_rep(rep: trep, i: int) ret b: bool
    b:= (rep[i] < 0)
end fun

proc union(in/out rep: trep, in u,v: int)
      if rep[u] ≥ rep[v] → rep[v]:= rep[u]+rep[v]
                           rep[u]:= v
        rep[u] < rep[v] → rep[u]:= rep[u]+rep[v]
                           rep[v]:= u
      fi
end proc
```

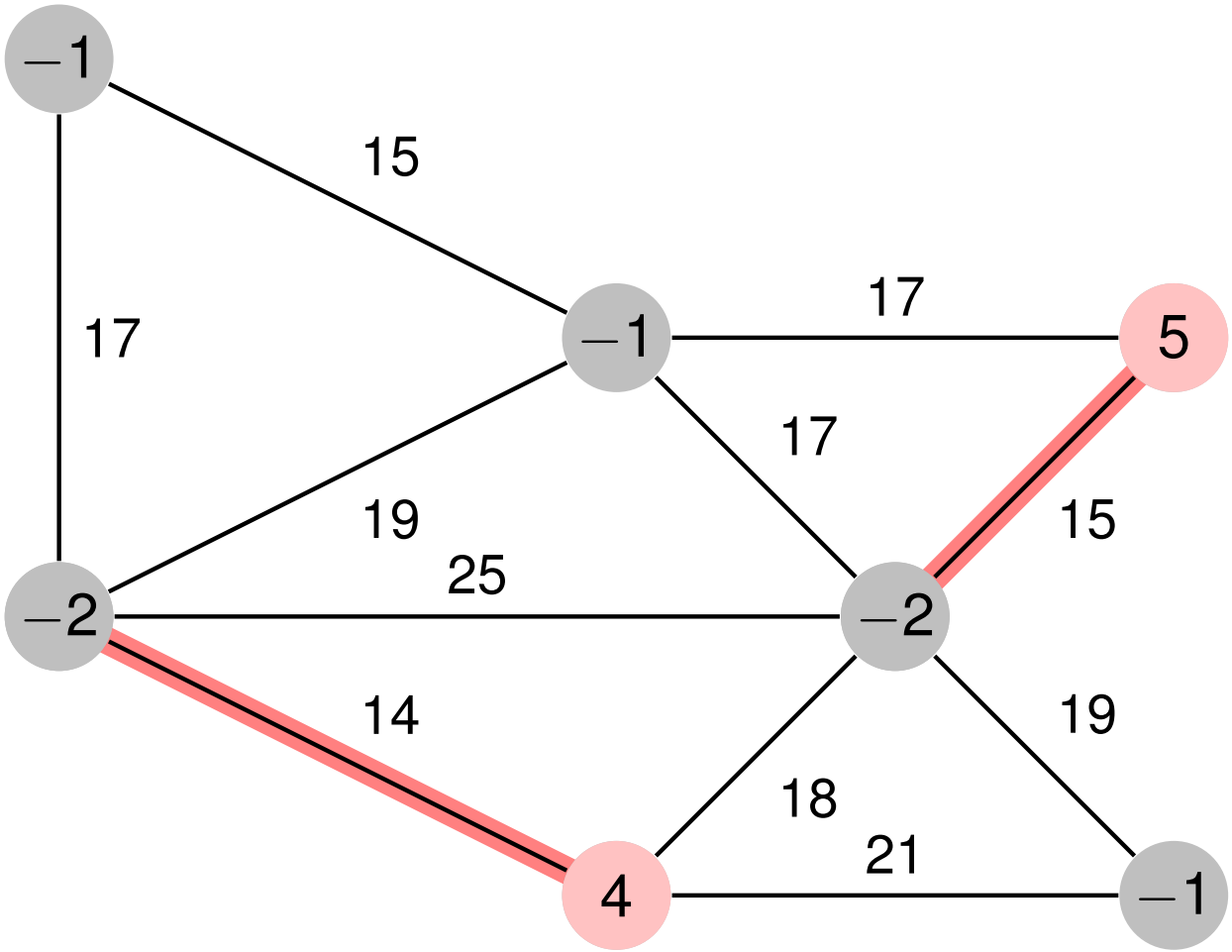
Algoritmo de Kruskal, último intento



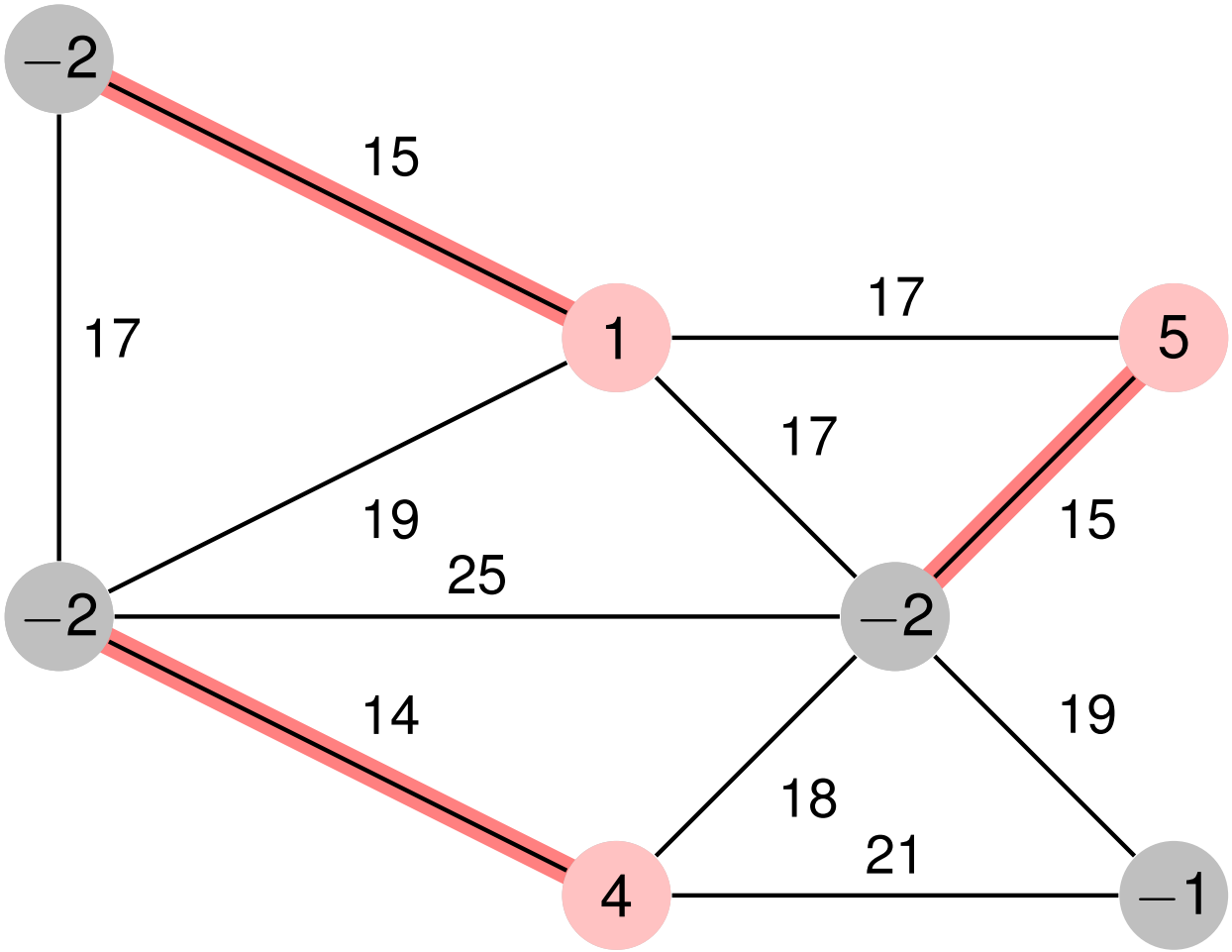
Algoritmo de Kruskal, último intento



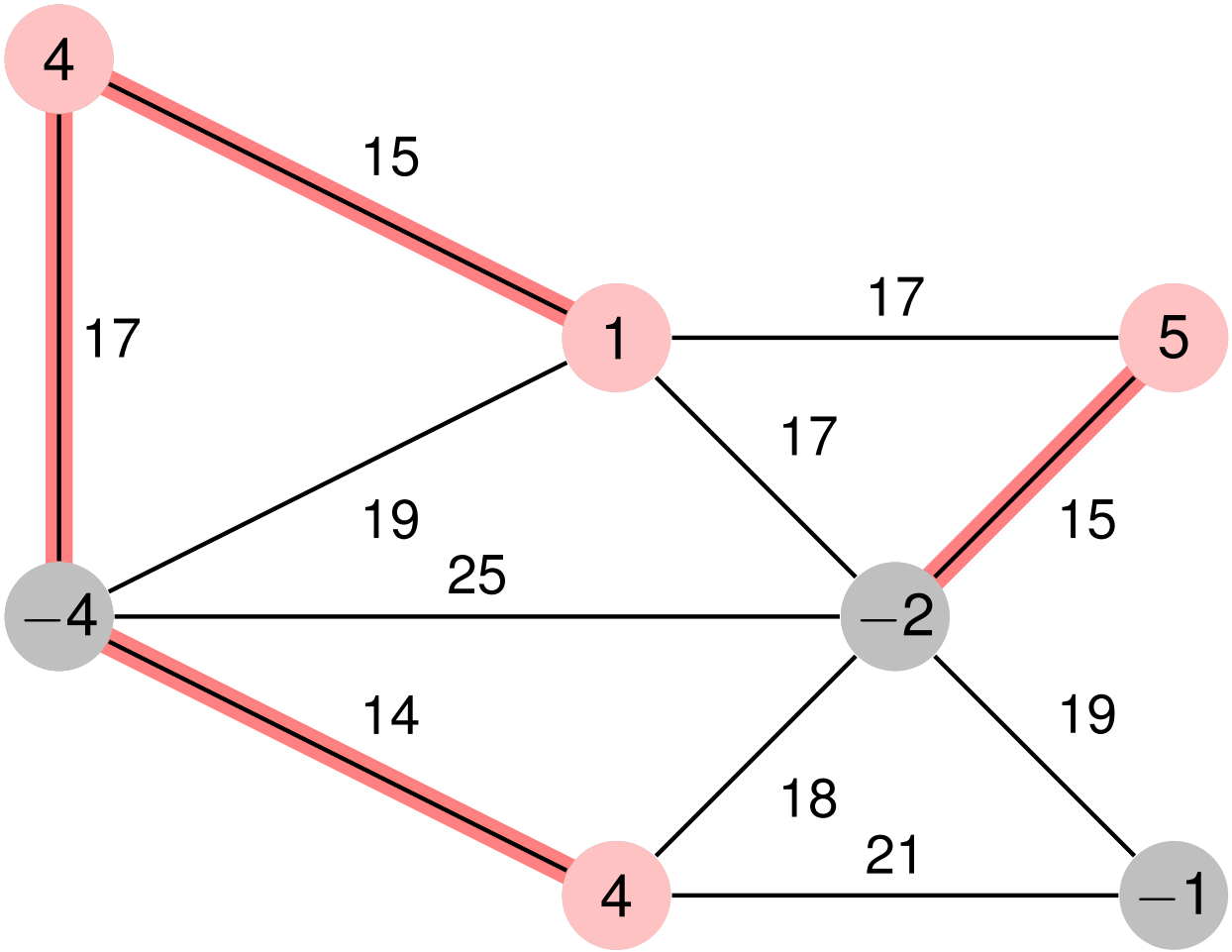
Algoritmo de Kruskal, último intento



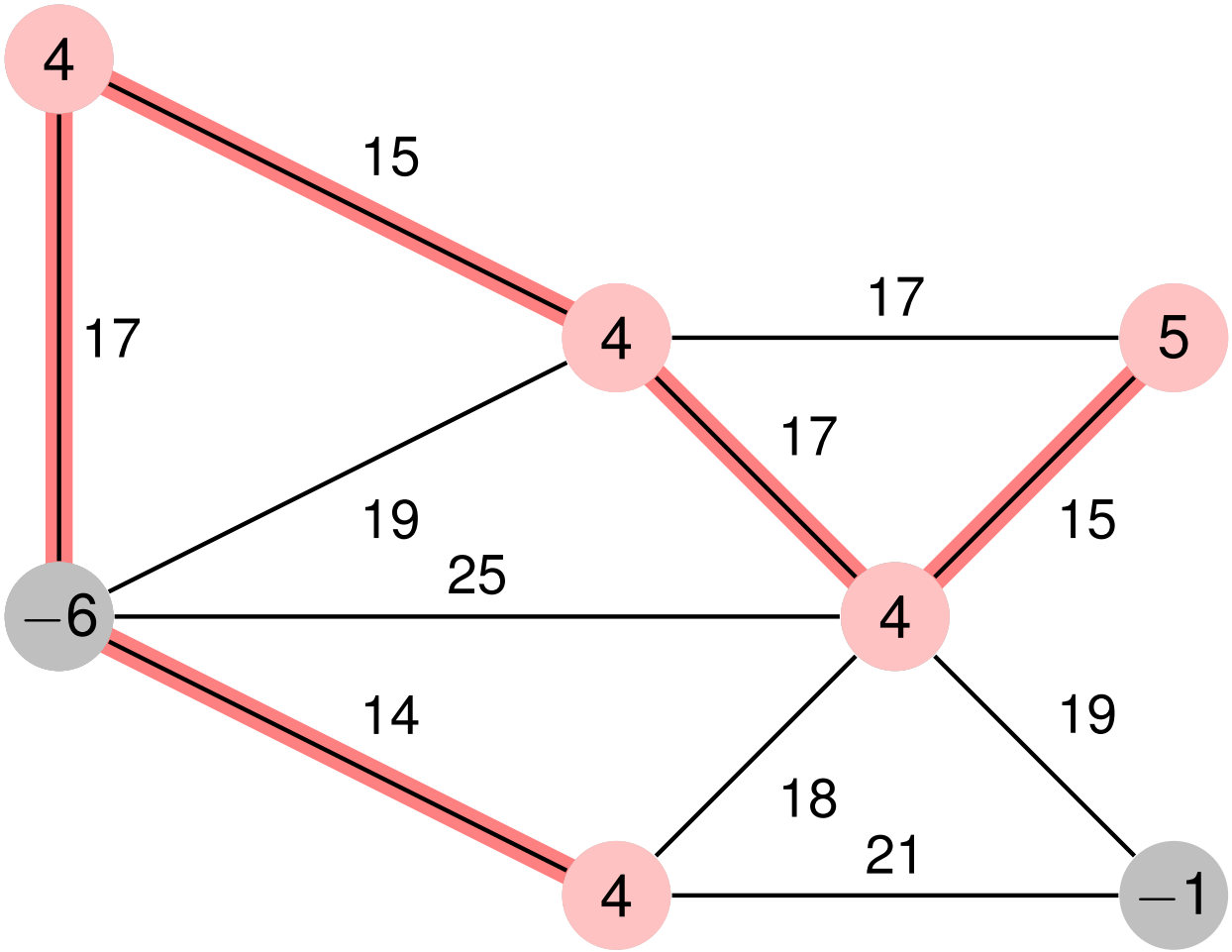
Algoritmo de Kruskal, último intento



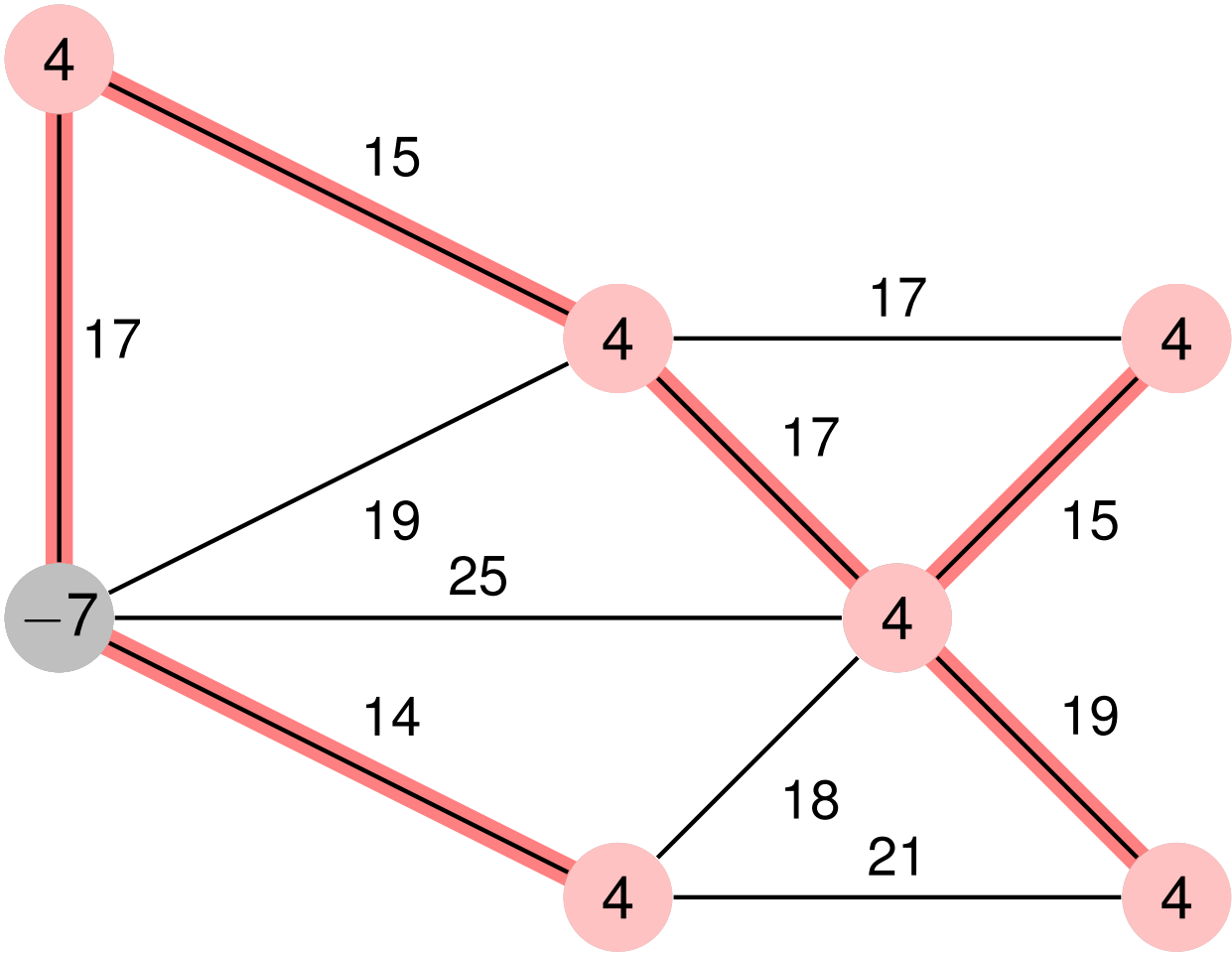
Algoritmo de Kruskal, último intento



Algoritmo de Kruskal, último intento



Algoritmo de Kruskal, último intento



Último intento
Análisis

init	es lineal
find	es en la práctica, es constante
union	es constante