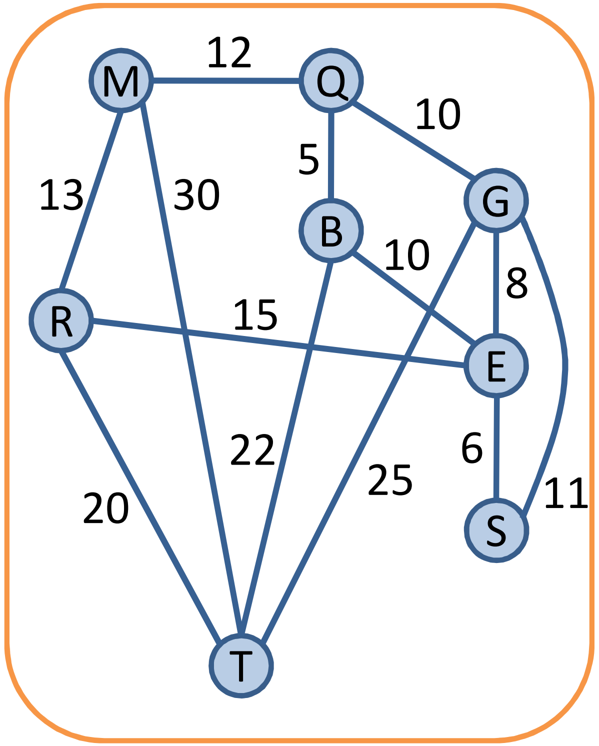
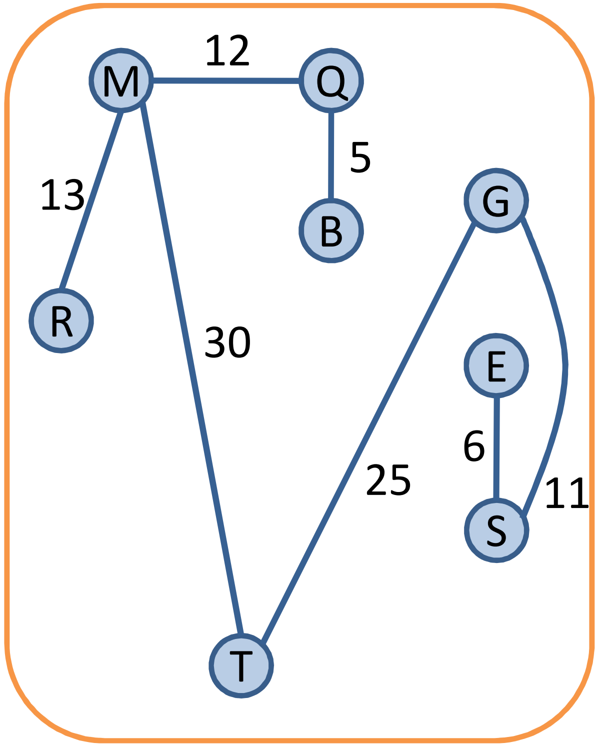
Árboles generadores de coste mínimo

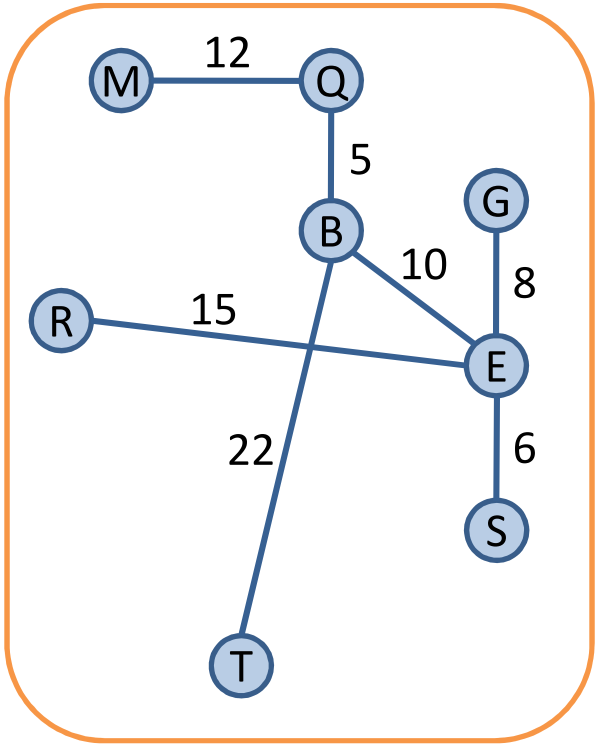
Dado un grafo no dirigido y conexo G = (V, A), se define un **árbol generador** (o de expansión) **de G** como un árbol que conecta todos los vértices de V; su coste es la suma de los costes de las aristas del árbol. Un árbol es un grafo conexo acíclico.



Grafo G



Árbol generador de G
Coste = 102



Árbol generador de G
Coste = 78

TAD Partición

<u>Numeración de elementos</u>:

Definimos una aplicación biyectiva de un conjunto *C* en el rango de enteros [0, *n*-1], tal que *n* es el cardinal de *C*, mediante dos funciones:

int IndiceElto (tElemento x)

<u>Pre</u>: $x \in C$.

<u>Post</u>: Devuelve el índice del elemento x en el rango $[0, n-1]$.

tElemento NombreElto (int i)

<u>Pre</u>: $0 \leq i \leq n-1$

<u>Post</u>: Devuelve el elemento de C cuyo índice es i.

<u>Especificación del TAD Partición</u>

Definición:

Una partición del conjunto de enteros $C = \{0, 1, \ldots, n-1\}$ es un conjunto de subconjuntos disjuntos cuya unión es el conjunto total C.

Operaciones:

Particion(int n);

<u>Post</u>: Construye una partición de subconjuntos unitarios del intervalo de enteros $[0, n-1]$.

void unir (int a, int b);

<u>Pre</u>: $0 \le a, b \le n-1$, a y b son los representantes de sus clases y $a \neq b$.
<u>Post</u>: Une el subconjunto del elemento *a* y el del elemento *b* en uno de los dos subconjuntos arbitrariamente. La partición queda con un miembro menos.

int encontrar(int x) const;

<u>Pre</u>: $0 \le x \le n-1$.
<u>Post</u>: Devuelve el representante del subconjunto al que pertenece el elemento *x*.

Implementación del TAD Partición

1. Vector de pertenencia

P = { {2,3,**7**} {0,**4**} {**5**} {**1**,6,8,9} }

	0	1	2	3	4	5	6	7	8	9
P	4	1	7	7	4	5	1	7	1	1

P.unir(4,1)

P = { {2,3,**7**} {0,1,**4**,6,8,9} {**5**} }

	0	1	2	3	4	5	6	7	8	9
P	4	4	7	7	4	5	4	7	4	4

Particion() $\in$ *O(n)*
encontrar() $\in$ *O(1)*
unir() $\in$ *O(n)*

Implementación del TAD Partición

2.1. Listas de elementos

P = {{2,3,**7**} {0,**4**} {**5**} {**1**,6,8,9}}

	0	1	2	3	4	5	6	7	8	9
P	4	1	7	7	4	5	1	7	1	1
	-1	6	3	-1	0	-1	8	2	9	-1

P.unir(4,1)

P = {{2,3,**7**} {0,1,**4**,6,8,9} {**5**}}

	0	1	2	3	4	5	6	7	8	9
P	4	4	7	7	4	5	4	7	4	4
	-1	6	3	-1	1	-1	8	2	9	0

encontrar() $\in$ *O(1)*

unir() $\in$ *O(n), pero el tiempo de ejecución es menor.*

Implementación del TAD Partición

2.2. Listas de elementos (con longitud)

$P = \Big\{\{2,3,\mathbf{7}\}\ \{0,\mathbf{4}\}\ \{\mathbf{5}\}\ \{\mathbf{1},6,8,9\}\Big\}$

	0	1	2	3	4	5	6	7	8	9
P	4	1	7	7	4	5	1	7	1	1
	-1	6	3	-1	0	-1	8	2	9	-1
	■	4	■	■	2	1	■	3	■	■

P.unir(4,1)

$P = \Big\{\{2,3,\mathbf{7}\}\ \{0,\mathbf{1},4,6,8,9\}\ \{\mathbf{5}\}\Big\}$

	0	1	2	3	4	5	6	7	8	9
P	1	1	7	7	1	5	1	7	1	1
	6	4	3	-1	0	-1	8	2	9	-1
	■	6	■	■	■	1	■	3	■	■

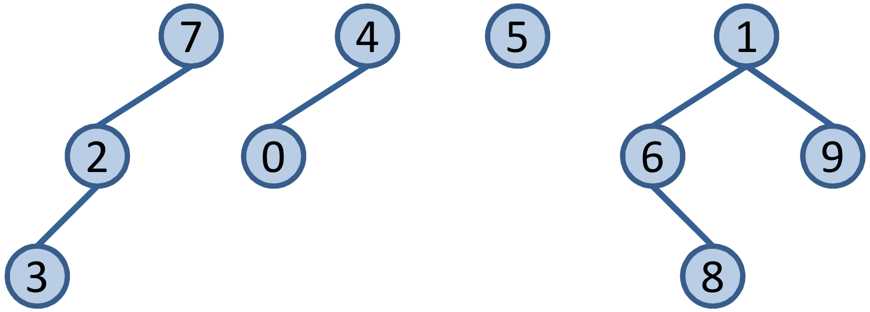
encontrar() $\in$ *O(1)*
unir() $\in$ *O(n), pero el tiempo de ejecución se reduce por lo menos a la mitad.*

Implementación del TAD Partición

3.1. Bosque de árboles

P = {{2,3,**7**} {0,**4**} {**5**} {**1**,6,8,9}}

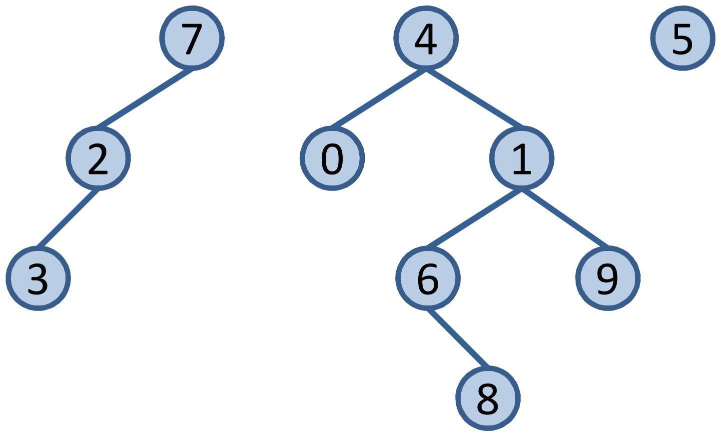
	0	1	2	3	4	5	6	7	8	9
P	4	-1	7	2	-1	-1	1	-1	6	1



P.unir(4,1)

P = {{2,3,**7**} {0,1,**4**,6,8,9} {**5**}}

	0	1	2	3	4	5	6	7	8	9
P	4	4	7	2	-1	-1	1	-1	6	1



encontrar() $\in$ *O(n)*
unir() $\in$ *O(1)*

Implementación del TAD Partición mediante bosque de árboles

```
/*---------------------------------------*/
/* particion.h                           */
/*---------------------------------------*/
#ifndef PARTICION_H
#define PARTICION_H

#include <vector>

class Particion {
public:
   Particion(int n): padre(n, -1) {}
   void unir(int a, int b) { padre[b] = a; }
   int encontrar(int x) const;
private:
   std::vector<int> padre;
};

#endif   // PARTICION_H
```

```
/*------------------------------------*/
/* particion.cpp                      */
/*------------------------------------*/
#include "particion.h“

int Particion::encontrar(int x) const
{
   while (padre[x] != -1)
      x = padre[x];
   return x;
}
```

Implementación del TAD Partición

3.2. Bosque de árboles (con control de altura)

a) **Unión por tamaño**
   El árbol con menos nodos se convierte en subárbol del que tiene mayor número de nodos.

b) **Unión por altura**
   El árbol menos alto se convierte en subárbol del otro.

unir() $\in$ ***O(1)***

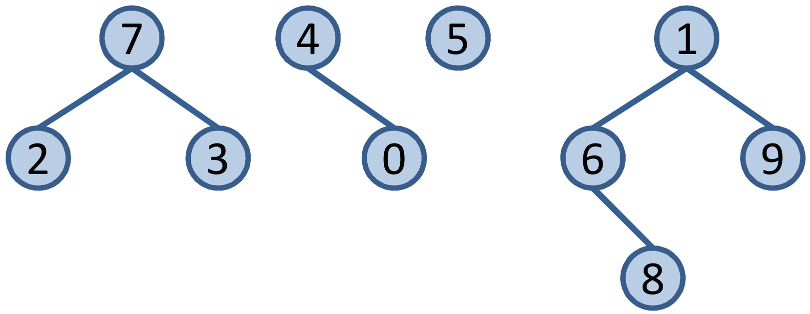
encontrar() $\in$ ***O(log n)***

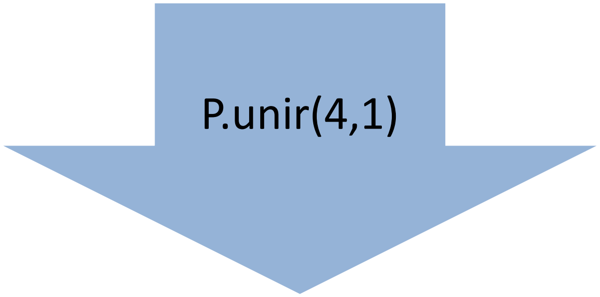
Implementación del TAD Partición

3.2. Bosque de árboles (con control de altura). Unión por altura

P = {{2,3,**7**} {0,**4**} {**5**} {**1**,6,8,9}}

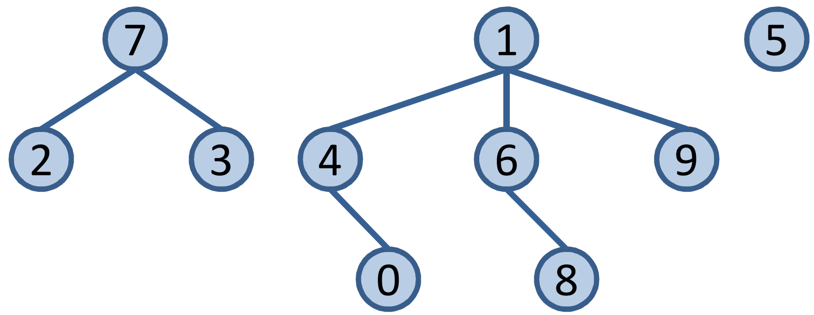
	0	1	2	3	4	5	6	7	8	9
P	4	-3	7	7	-2	-1	1	-2	6	1





P = {{2,3,**7**} {0,**1**,4,6,8,9} {**5**}}

	0	1	2	3	4	5	6	7	8	9
P	4	-3	7	7	1	-1	1	-2	6	1



encontrar() $\in$ *O(log n)*
unir() $\in$ *O(1)*

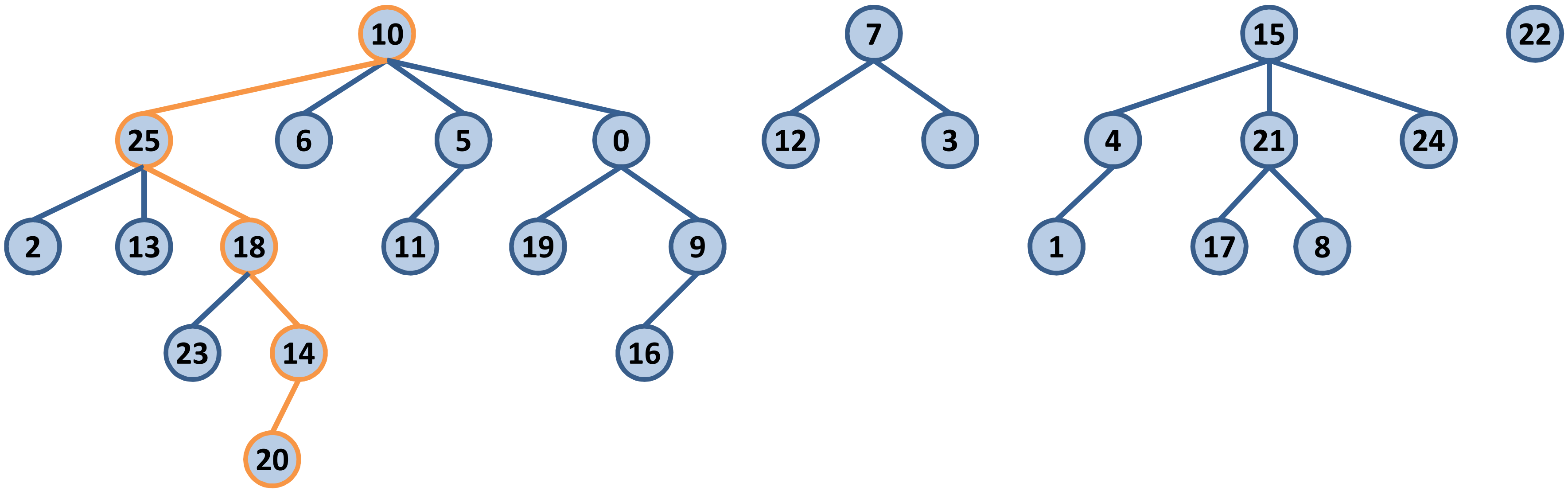
Implementación del TAD Partición

3.2. Bosque de árboles (con control de altura). Compresión de caminos

encontrar(20)

P = {{0,2,5,6,9,**10**,11,13,14,16,18,19,20,23,25} {3,**7**,12} {1,4,8,**15**,17,21,24} {**22**}}

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
P	10	4	25	7	15	10	10	-2	21	0	-5	5	7	25	18	-3	9	21	25	0	14	15	-1	18	15	10



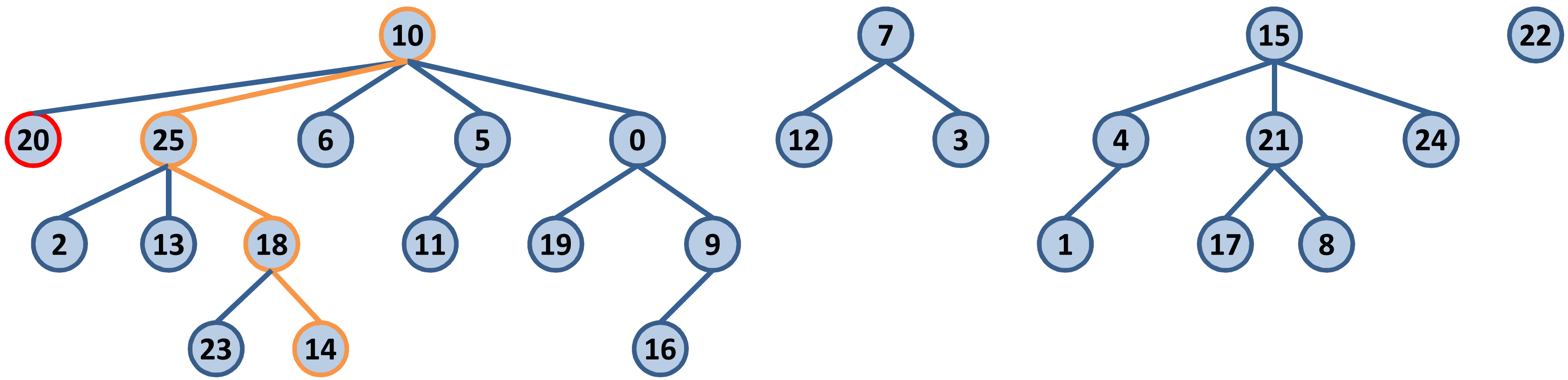
Implementación del TAD Partición

3.2. Bosque de árboles (con control de altura). Compresión de caminos

encontrar(20)

P = {{0,2,5,6,9,**10**,11,13,14,16,18,19,20,23,25} {3,**7**,12} {1,4,8,**15**,17,21,24} {**22**}}

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
P	10	4	25	7	15	10	10	-2	21	0	-5	5	7	25	18	-3	9	21	25	0	10	15	-1	18	15	10



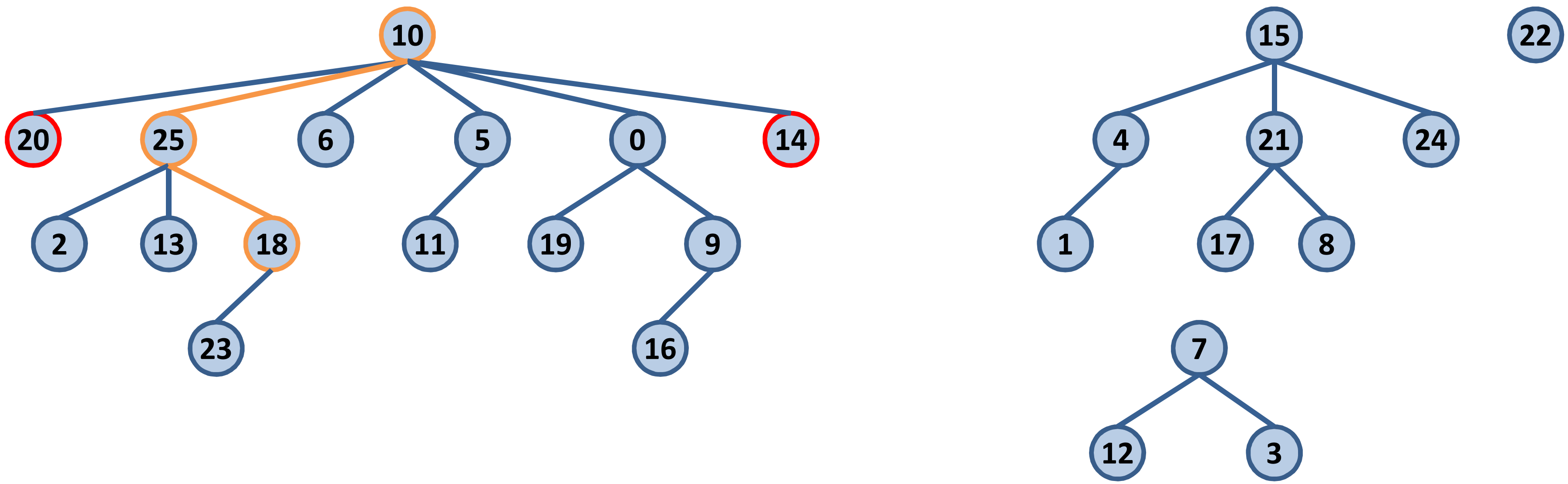
Implementación del TAD Partición

3.2. Bosque de árboles (con control de altura). Compresión de caminos

encontrar(20)

P = {{0,2,5,6,9,**10**,11,13,14,16,18,19,20,23,25} {3,**7**,12} {1,4,8,**15**,17,21,24} {**22**}}

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
P	10	4	25	7	15	10	10	-2	21	0	-5	5	7	25	10	-3	9	21	25	0	10	15	-1	18	15	10



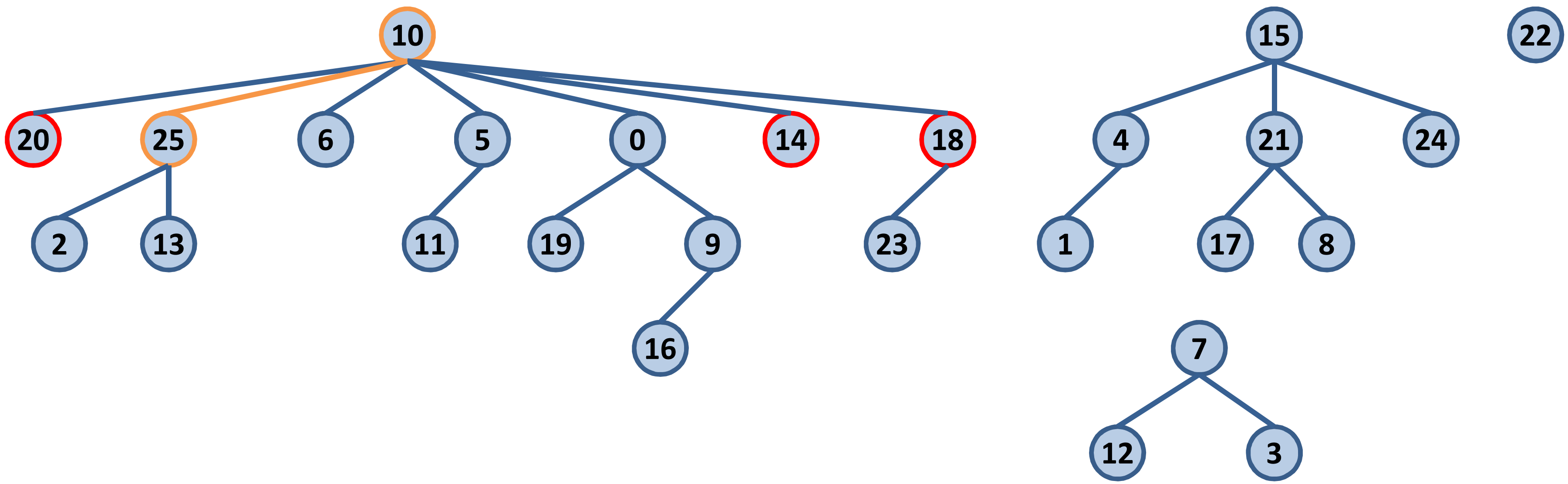
Implementación del TAD Partición

3.2. Bosque de árboles (con control de altura). Compresión de caminos

Encontrar(20)

P = {{0,2,5,6,9,**10**,11,13,14,16,18,19,20,23,25} {3,**7**,12} {1,4,8,**15**,17,21,24} {**22**}}

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
P	10	4	25	7	15	10	10	-2	21	0	-5	5	7	25	10	-3	9	21	10	0	10	15	-1	18	15	10



Implementación del TAD Partición mediante bosque de árboles

```
/*------------------------------------------------*/
/* particion.h                                     */
/*------------------------------------------------*/
#ifndef PARTICION_H
#define PARTICION_H

#include <vector>

class Particion {
public:
   Particion(int n): padre(n, -1) {}
   void unir(int a, int b);
   int encontrar(int x) const;
private:
   mutable std::vector<int> padre;
};

#endif   // PARTICION_H
```

```
/*-----------------------------------------------------------*/
/* particion.cpp                                             */
/*                                                           */
/* Implementación de la clase Particion:                     */
/* Bosque de árboles con unión por altura y búsqueda con    */
/* compresión de caminos.                                    */
/*-----------------------------------------------------------*/
#include "particion.h"
// El árbol con mayor altura se convierte en subárbol del otro.
void Particion::unir(int a, int b)
{
   if (padre[b] < padre[a])
      padre[a] = b;
   else {
      if (padre[a] == padre[b])
         padre[a]--; // El árbol resultante tiene un nivel más.
      padre[b] = a;
   }
}
```

```
int Particion::encontrar(int x) const
{
   int y, raiz = x;

   while (padre[raiz] > -1)
      raiz = padre[raiz];
   // Compresión del camino de x a raíz: Los nodos
   // del camino se hacen hijos de la raíz.
   while (padre[x] > -1) {
      y = padre[x];
      padre[x] = raiz;
      x = y;
   }
   return raiz;
}
```

Árboles generadores de coste mínimo
Algoritmo de Kruskal

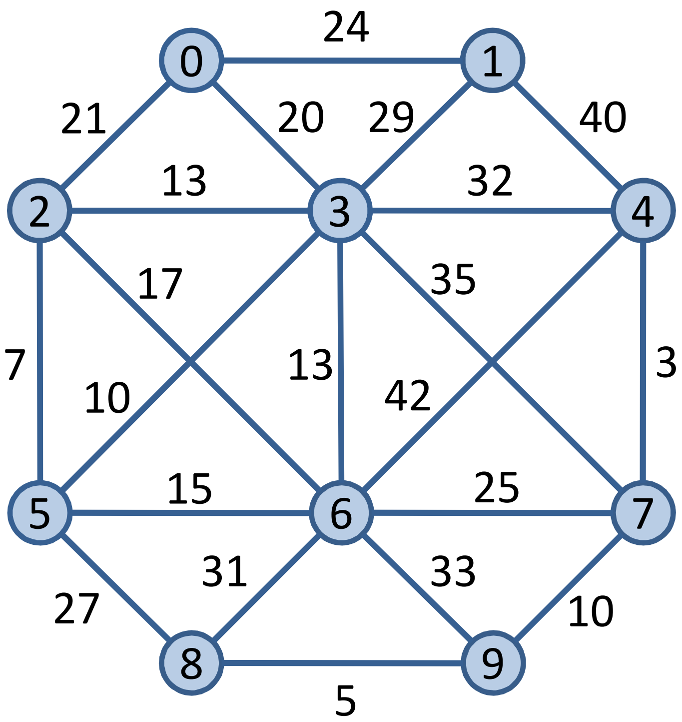
```
template <typename T> class GrafoP {
public:
   typedef T tCoste;
   typedef size_t vertice;
   struct arista {
      vertice orig, dest;
      tCoste coste;
      explicit arista(vertice v=vertice(), vertice w=vertice(),
             tCoste c=tCoste()): orig(v), dest(w), coste(c) {}
      // Orden de aristas para Prim y Kruskall
      bool operator <(const arista& a) const
      { return coste < a.coste; }
   };
   // resto de miembros de la clase GrafoP<T> ...
};
```

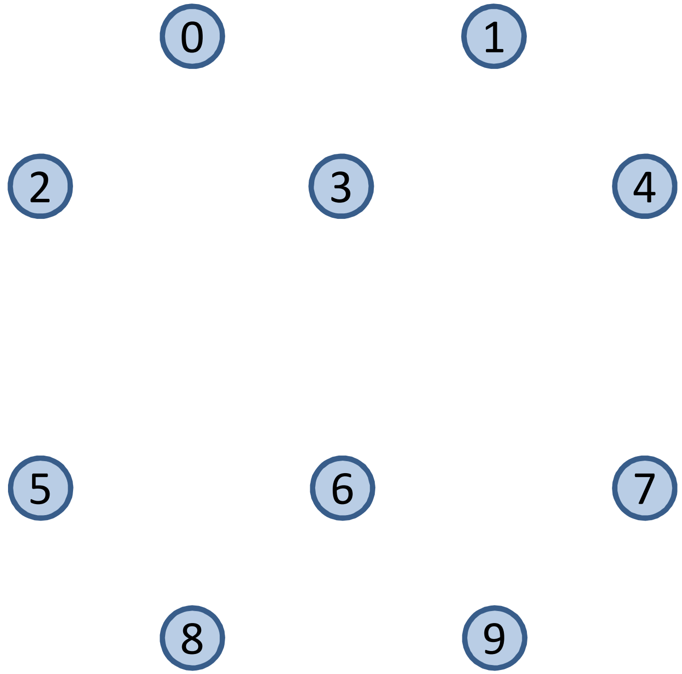
```
#include "particion.h"
#include "apo.h"

template <typename tCoste>
GrafoP<tCoste> Kruskall(const GrafoP<tCoste>& G)
// Devuelve un árbol generador de coste mínimo
// de un grafo no dirigido ponderado y conexo G.
{
   typedef typename GrafoP<tCoste>::vertice vertice;
   typedef typename GrafoP<tCoste>::arista arista;
   const tCoste INFINITO = GrafoP<tCoste>::INFINITO;

   const size_t n = G.numVert();
   GrafoP<tCoste> g(n);     // Árbol generador de coste mínimo.
   Particion P(n);   // Partición inicial de los vértices de G.
   Apo<arista> A(n*n); // Aristas de G ordenadas por costes.
```

```
   // Copiar aristas del grafo G en el APO A.
   for (vertice u = 0; u < n; u++)
      for (vertice v = u+1; v < n; v++)
         if (G[u][v] != INFINITO)
            A.insertar(arista(u, v, G[u][v]));

   size_t i = 1;
   while (i <= n-1) {                  // Seleccionar n-1 aristas.
      arista a = A.cima();             // arista de menor coste
      A.suprimir();
      vertice u = P.encontrar(a.orig);
      vertice v = P.encontrar(a.dest);
      if (u != v) {// extremos de a pertenecen a distintos componentes
         P.unir(u, v);
         // Incluir la arista a en el árbol g.
         g[a.orig][a.dest] = g[a.dest][a.orig] = a.coste;
         i++;
      }
   }
   return g;
}
```

Algoritmo de Kruskal
Ejemplo

Inicialización

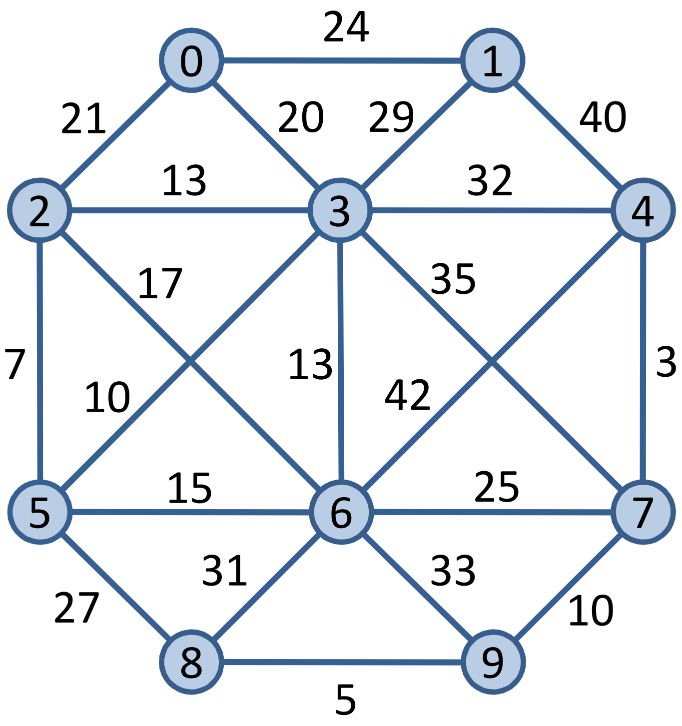


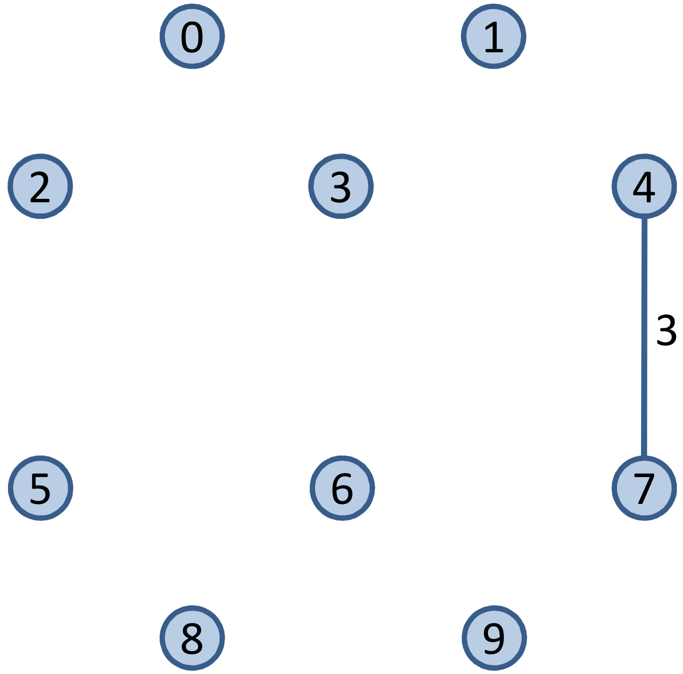


P = {{0} {1} {2} {3} {4} {5} {6} {7} {8} {9}}

Algoritmo de Kruskal
Ejemplo

i = 1 a = (4, 7, 3)

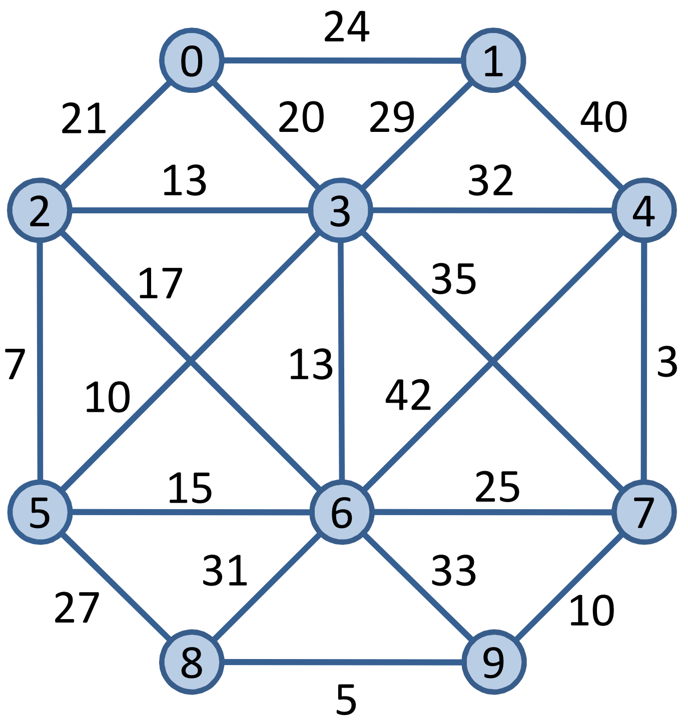


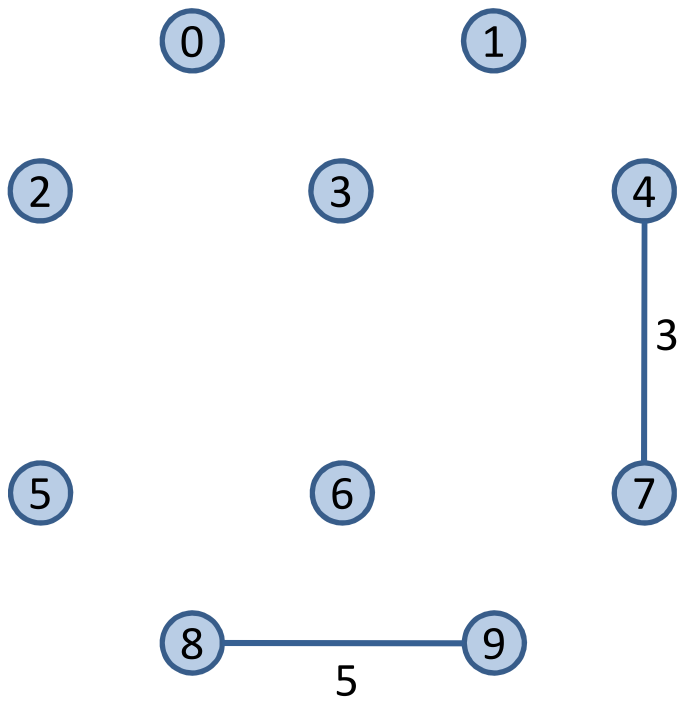


P = {{0} {1} {2} {3} {4,7} {5} {6} {8} {9}}

Algoritmo de Kruskal
Ejemplo

i = 2 a = (8, 9, 5)

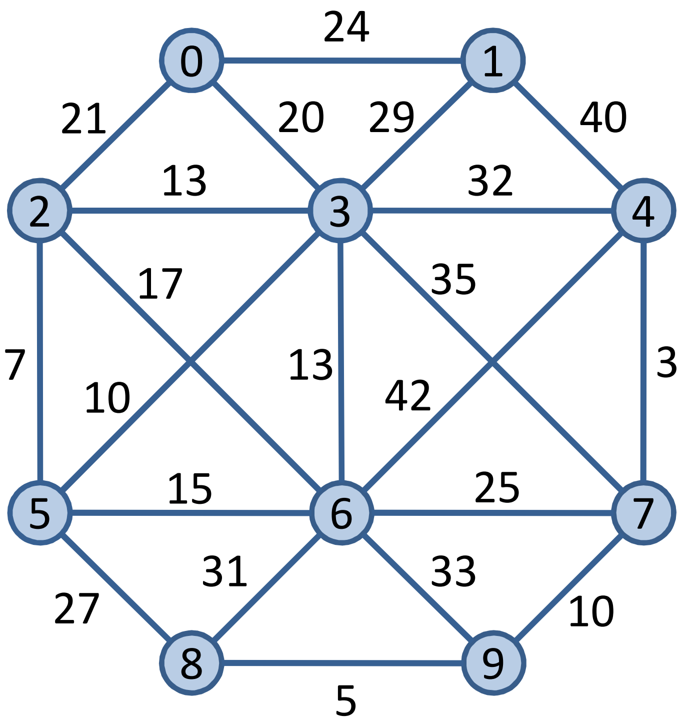


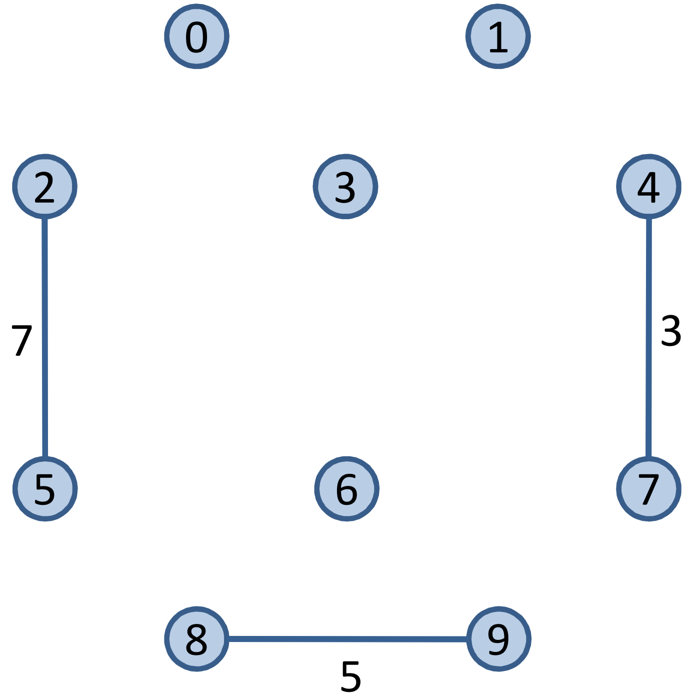


P = {{0} {1} {2} {3} {4,7} {5} {6} {8,9}}

Algoritmo de Kruskal
Ejemplo

i = 3 a = (2, 5, 7)

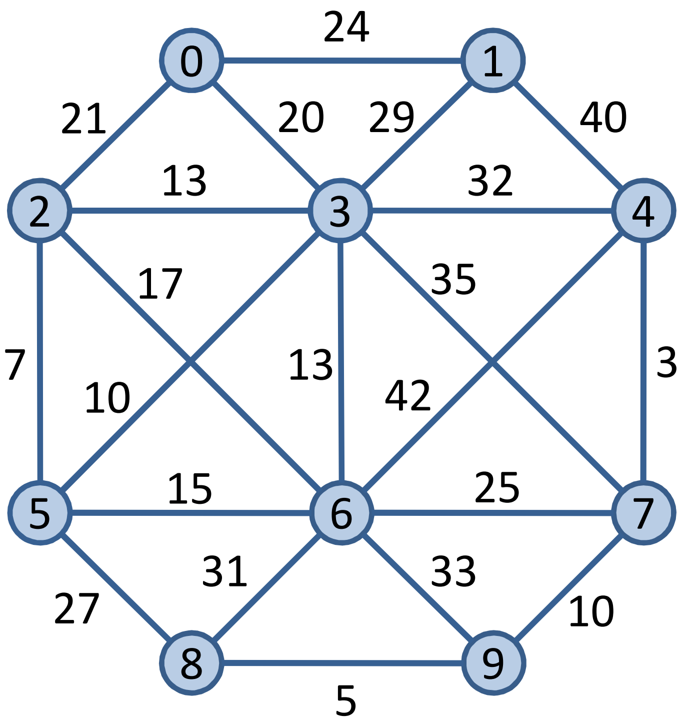


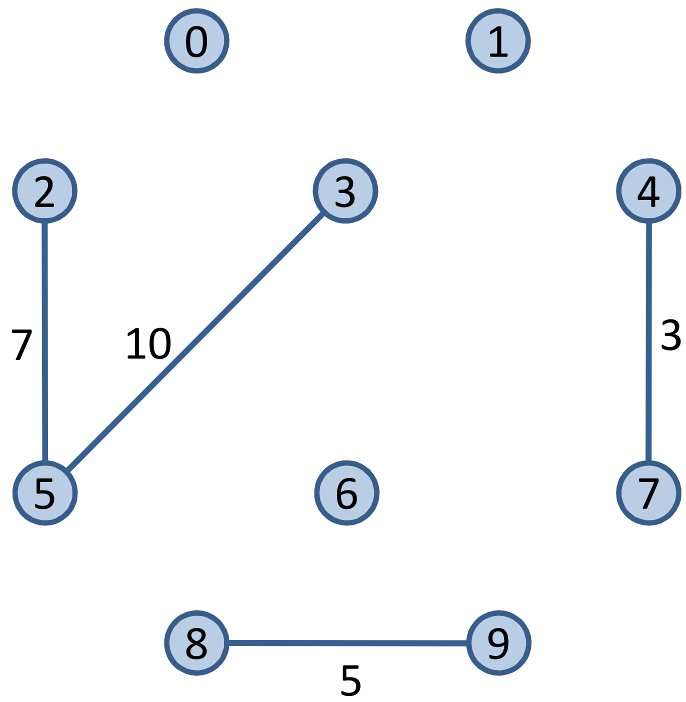


P = {{0} {1} {2,5} {3} {4,7} {6} {8,9}}

Algoritmo de Kruskal
Ejemplo

i = 4 a = (3, 5, 10)

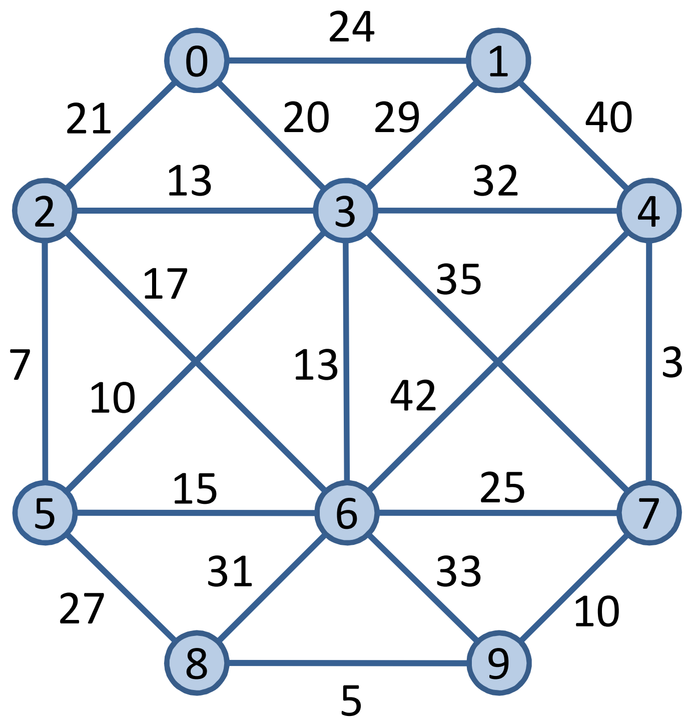


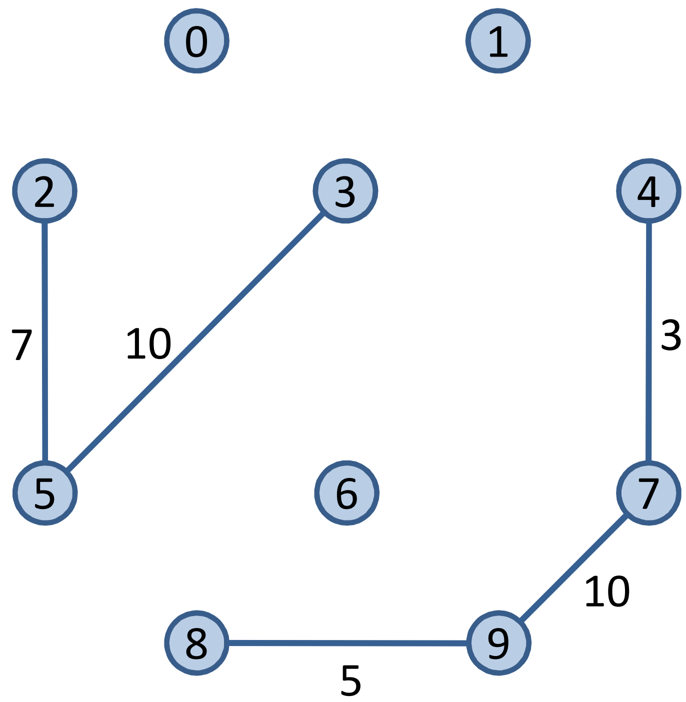


P = {{0} {1} {2,3,5} {4,7} {6} {8,9}}

Algoritmo de Kruskal
Ejemplo

i = 5 a = (7, 9, 10)

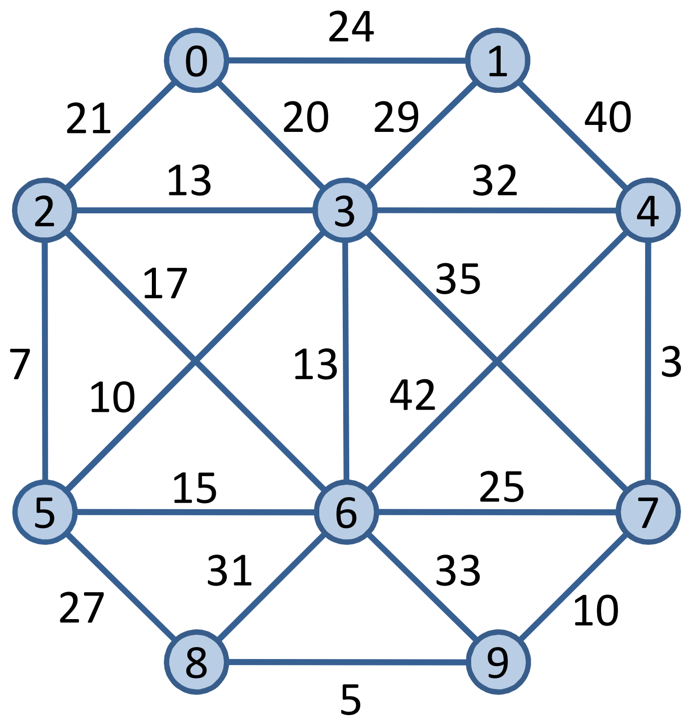


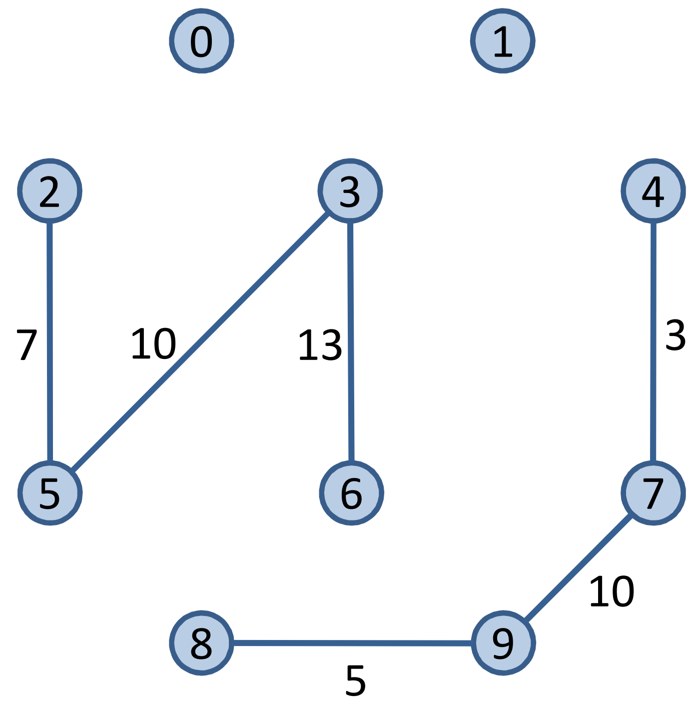


P = {{0} {1} {2,3,5} {4,7,8,9} {6}}

Algoritmo de Kruskal
Ejemplo

i = 6 a = (3, 6, 10)

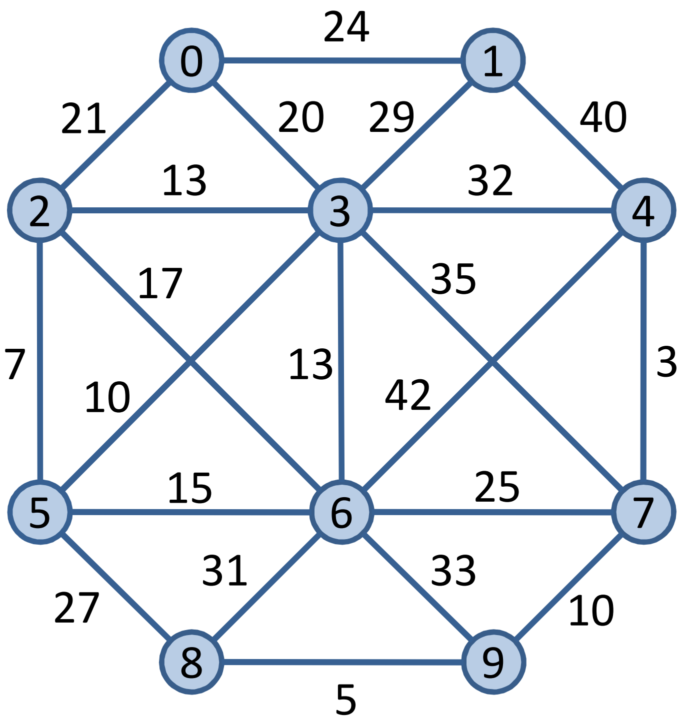


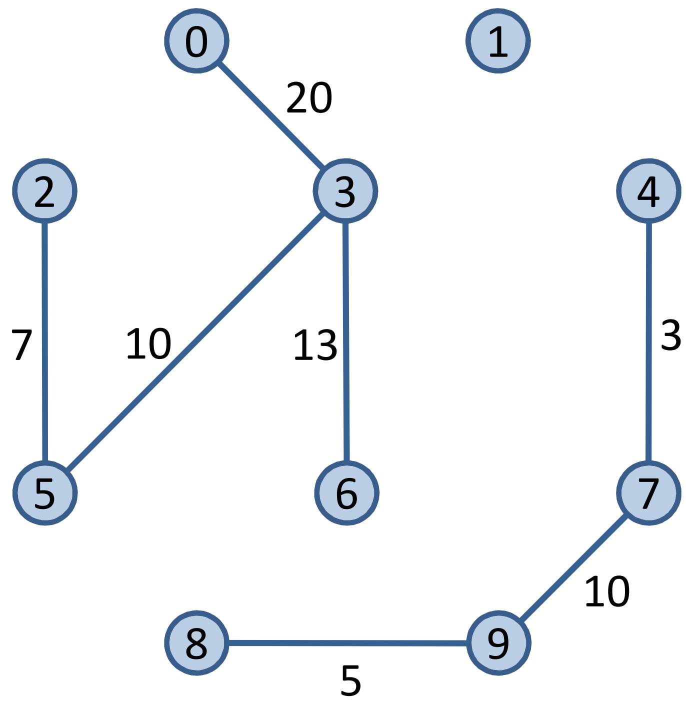


P = {{0} {1} {2,3,5,6} {4,7,8,9}}

Algoritmo de Kruskal
Ejemplo

i = 7 a = (0, 3, 20)

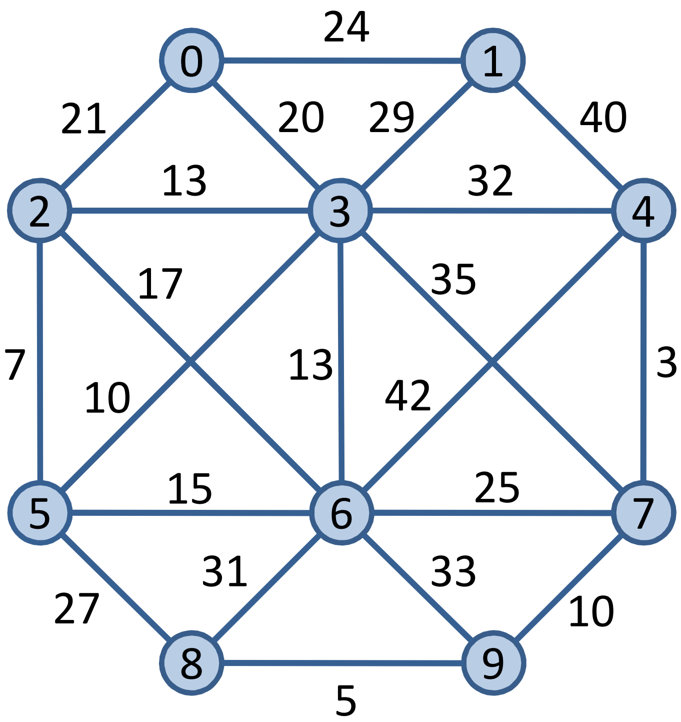


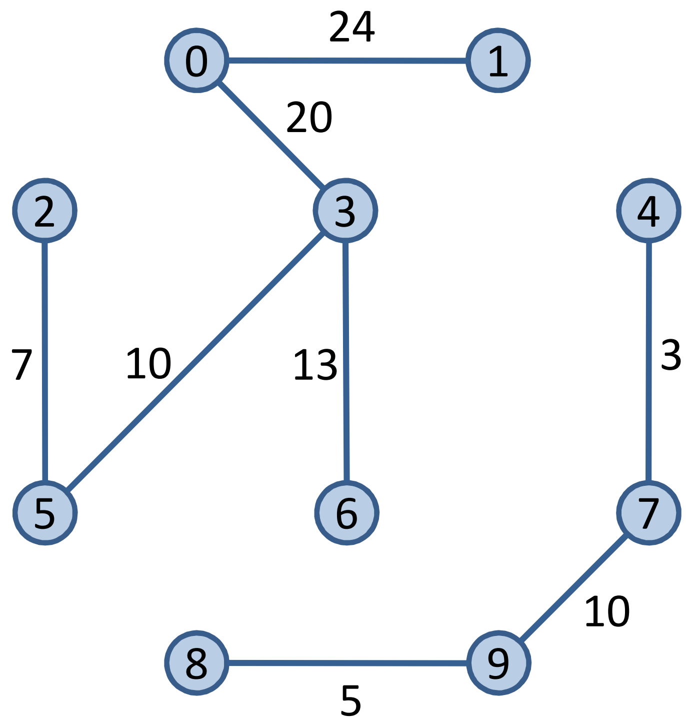


P = {{1} {0,2,3,5,6} {4,7,8,9}}

Algoritmo de Kruskal
Ejemplo

i = 8 a = (0, 1, 24)

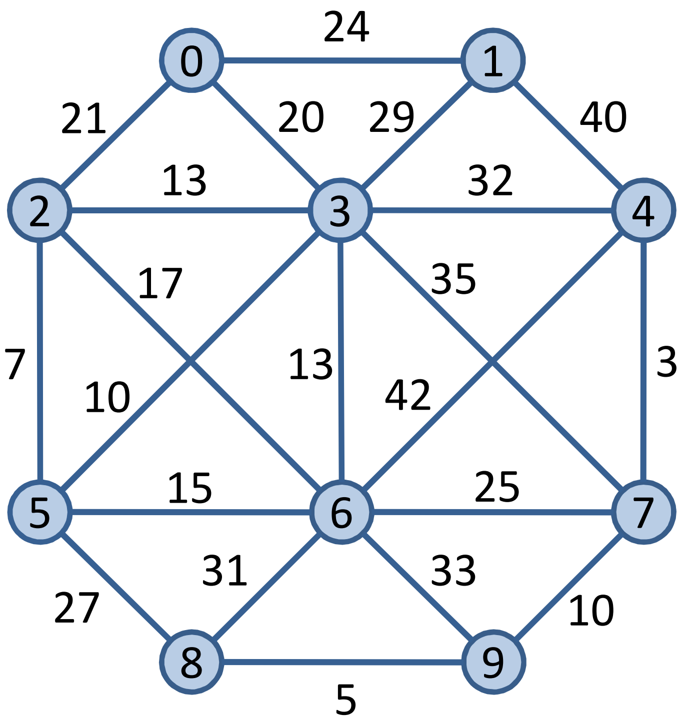


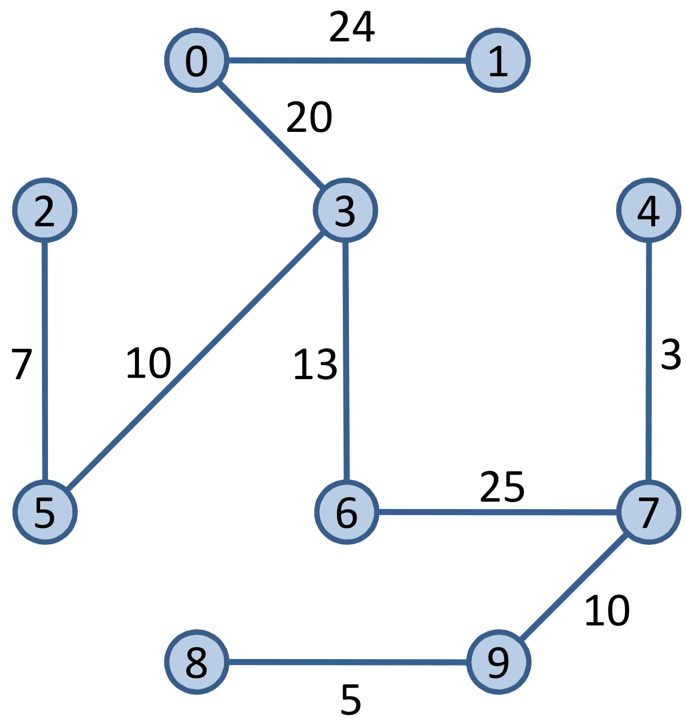


P = {{0,1,2,3,5,6} {4,7,8,9}}

Algoritmo de Kruskal
Ejemplo

i = 9 a = (6, 7, 25)





P = {{0,1,2,3,4,5,6,7,8,9}}