GIIT_AMA_2223_Practica2_jelanang_alokenveo

Práctica 2. Árboles y Caminos óptimos

Temporalización de la práctica: Lunes 10 de octubre y Lunes 17 de octubre de 2022

Entrega de la práctica: Desde el 17 de octubre hasta el 24 de octubre (ver tarea en el campus virtual o agenda de la asignatura)

Instrucciones:

1. Haz una copia de la hoja pública y renómbrala: si tu correo es **mariomp@alumnos.unex.es** añade al final del título **_mariomp**, por ejemplo GIIT_AMA_2223_Practica2_mariomp

- Para cambiar el nombre pulsa en el título de la hoja (arriba del todo, entre el logo de Sage y el menú "Archivo/File...")

2. Comparte la hoja de trabajo con el usuario **mariomp2223** mediante el botón Compartir/Share de arriba a la derecha. Si lo hacéis en pareja, compartid la hoja con el usuario de tu compañero (ambos usuarios separados por comas), así como añadir también el usuario en el título de la hoja (separados por _).

3. Completa la primera celda y trabaja la práctica.

4. Cuando hayas terminado, haz una copia en un único fichero PDF y ponlo en el campus virtual (Si lo hacéis en pareja, basta que lo suba uno). **Esa será la versión que se evaluará**. La hoja no se considera entregada si no se ha renombrado y compartido (pasos 1 y 2).

- Para generar el PDF lo más sencillo es usar el botón Imprimir/Print de arriba e imprimir la nueva página a fichero.

5. Una vez subido el PDF al campus virtual, no podrá modificarse esta hoja de trabajo. **Hacerlo conllevará la calificación de 0 en esta práctica**.

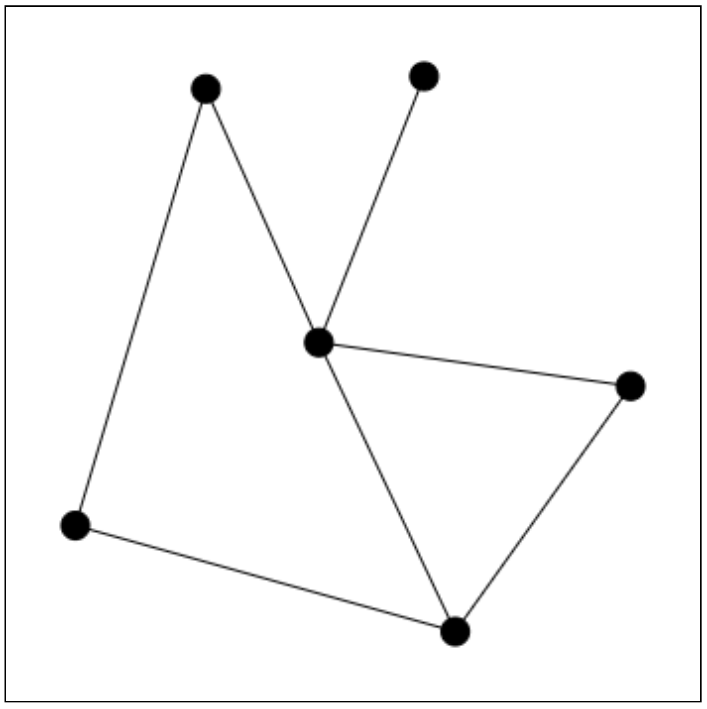
6. Los ejercicios a entregar se representan en Rojo y deben estar correctamente explicados. Los ejercicios indicados con ** **NO** serán obligatorios, pero aquellos que los hagan podrán ir sumando por cada uno de ellos 0.1 puntos adicionales en la calificación de la práctica.

Alumno/s: Jose Luis Obiang Ela Nanguan y Alfredo Mituy Okenve Obiang

Árboles

En primer lugar, recordemos que un árbol es un grafo conexo que no tiene ciclos. Sage permite conocer si un grafo es un árbol o no, con la función ***.is_tree()***, pero también permite generar árboles aleatorios con un número de vértices determinado. Para esto último, se utiliza la función ***graphs.RandomTree(n)***, siendo n el número de vértices.

```
grafo = Graph({0:[3,2,4],1:[2,4],2:[1,3,0,5],3:[2,0],4:[1,0],5:[2]});
grafo.set_pos({0:[217,30],1:[89,291],2:[147,169],3:[307,148],4:[22,81],5:
[201,297]});  graph_editor(grafo);
```



live:

variable name: grafo

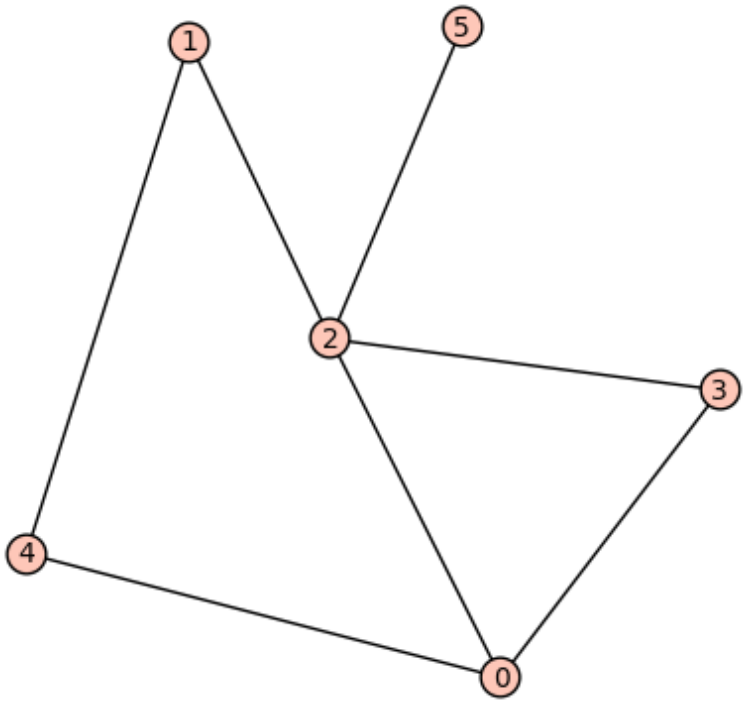
strength:

length:

help

Save Close

```
grafo.show()
```

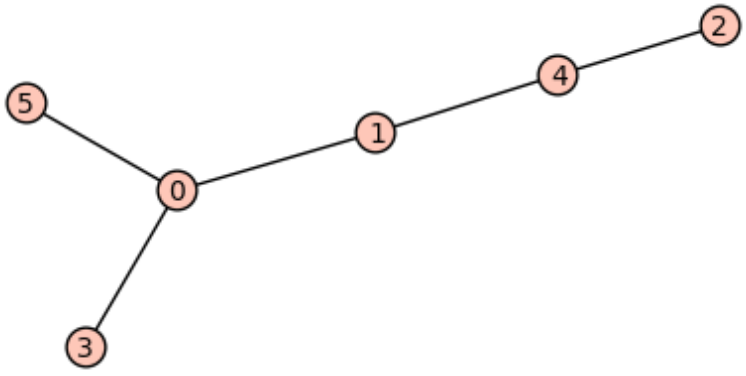



```
grafo.is_tree()
```

```
False
```

Podemos observar que el grafo anterior (que hemos construido no es un árbol), pues a simple vista vemos que tiene 6 vértices y 7 aristas, y para que sea árbol es necesario que tuviese 5 aristas.

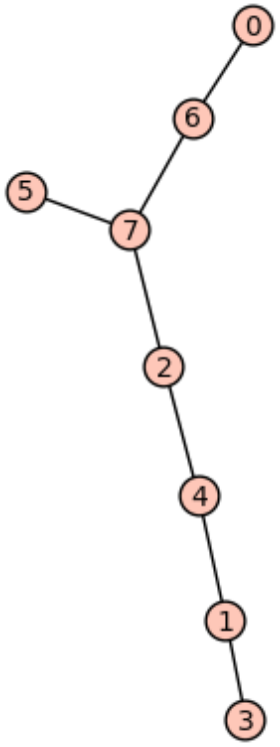
```
grafo = graphs.RandomTree(6); grafo.show()
```



Ejercicio 1. Genera un árbol con 8 vértices, y obtén su lista de adyacencia.

```
#Realiza el ejercicio en esta celda
grafo = graphs.RandomTree(8); grafo.show()
show(grafo.edges())
```

$[(0, 6, \text{None}), (1, 3, \text{None}), (1, 4, \text{None}), (2, 4, \text{None}), (2, 7, \text{None}), (5, 7, \text{None}), (6, 7, \text{None})]$



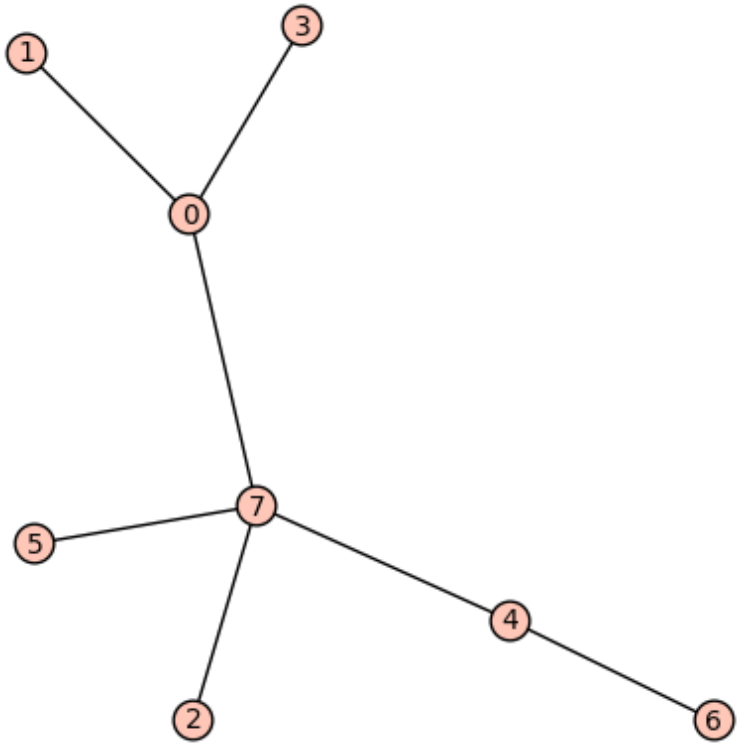
Ejercicio 2. Crea una función que reciba un grafo, y compruebe si es un árbol o no. Y en caso afirmativo, devuelva qué vertices son terminales. En caso negativo, devuelva un aviso (utiliza la función *print*) que indique que el grafo proporcionado no es un árbol.

```
#Realiza el ejercicio en esta celda
def isArbolgetVTerminales(g):
    i = 0
    if g.is_tree():
        print "Es un Arbol"
        for ngrado in g.degree():
            if ngrado == 1:
                print i
            i = i + 1
    else:
        print "El grafo no es un arbol"
```

Ejercicio 3. Genera un grafo y utiliza la función creada en el ejercicio anterior para comprobar que funciona correctamente.

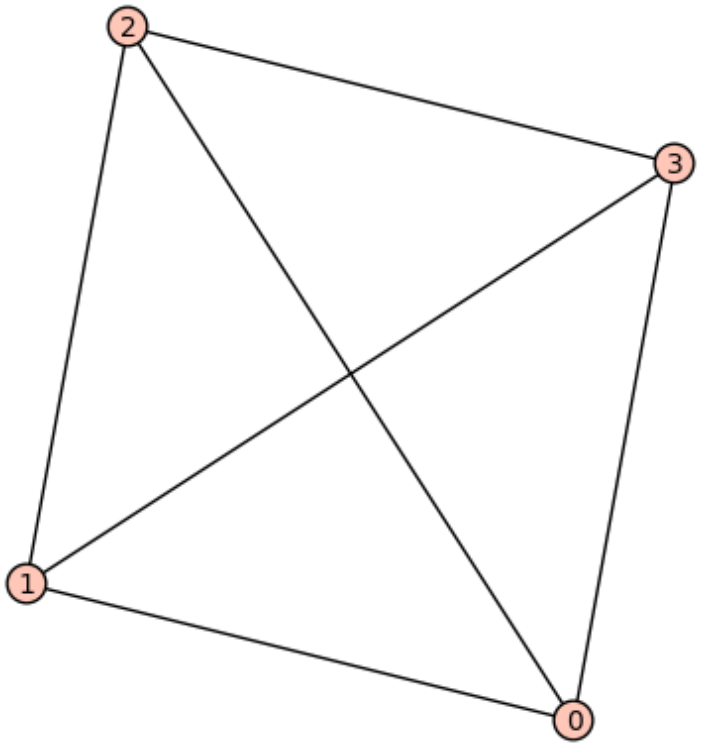
```
#Realiza el ejercicio en esta celda
grafo = graphs.RandomTree(8); grafo.show()
isArbolgetVTerminales(grafo)
```

```
Es un Arbol
1
2
3
5
6
```




```
grafo1 = graphs.RandomGNM(4,6); grafo1.show()
isArbolgetVTerminales(grafo1)
```

```
El grafo no es un arbol
```

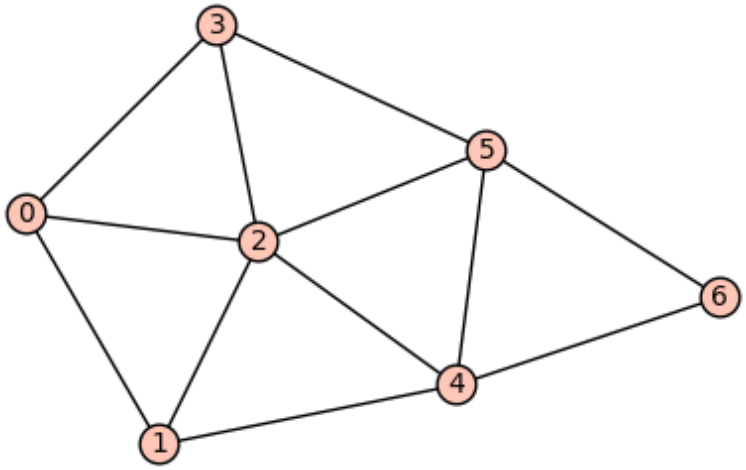


Grafos etiquetados

Un grafo etiquetado es un grafo cuyas aristas tienen etiquetas, que pueden interpretarse como distancias, costes,...

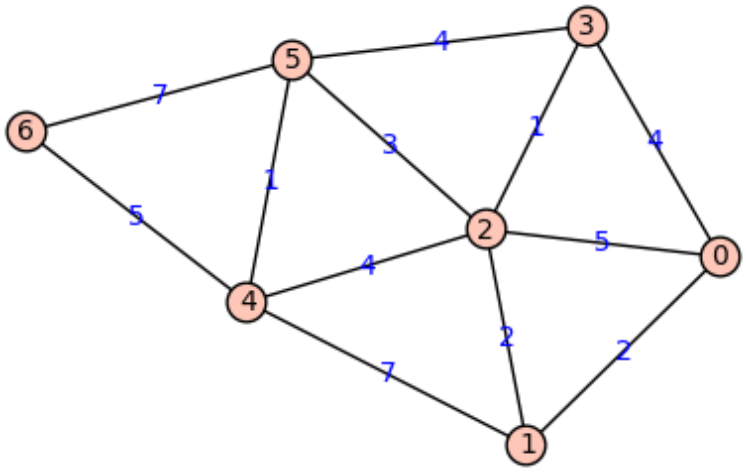
En Sage, se pueden construir grafos etiquetados con la función Graph, pero hay que indicar que las aristas poseen pesos, con el argumento ***weighted = True***. Para ello, generamos el grafo a partir de su lista de adyacencia (entre *{}*), de tal modo que si un vértice *a* es adyacente a otro *b* cuya arista tiene la etiqueta *c*, entonces se escribirá como *a:{b:c}*.

```
lisAdya = {0:{1:2,2:5,3:4},1:{2:2,4:7},2:{3:1,4:4,5:3},3:{5:4},4:{5:1,6:5},5:
{6:7}}
grafoEti = Graph(lisAdya,weighted=True)
grafoEti.show()
```



¿Qué ha ocurrido al representar el grafo? Efectivamente, no se observan los valores de las etiquetas de las aristas, es por ello que para representarlo como un grafo etiquetado, es necesario indicar en la función *show()* el argumento ***edge_labels=True***.

```
grafoEti.show(edge_labels=True)
```



Si pedimos a este grafo etiquetad la lista de vértices y la lista de aristas, ¿qué diferencia encontramos frente a los grafos no etiquetados?

```
print "Lista de vértices:",grafoEti.vertices()
print "Lista de aristas:", grafoEti.edges()
```

```
Lista de vértices: [0, 1, 2, 3, 4, 5, 6]
Lista de aristas: [(0, 1, 2), (0, 2, 5), (0, 3, 4), (1, 2, 2), (1, 4,
7), (2, 3, 1), (2, 4, 4), (2, 5, 3), (3, 5, 4), (4, 5, 1), (4, 6, 5),
(5, 6, 7)]
```

Además, en los grafos etiquetados su matriz de adyacencia se representa a partir de las etiquetas de las aristas, es por ello que Sage proporciona la función ***.weighted_adjacency_matrix()*** para determinar dicha matriz.

```
grafoEti.weighted_adjacency_matrix()
```

```
[0 2 5 4 0 0 0]
[2 0 2 0 7 0 0]
[5 2 0 1 4 3 0]
[4 0 1 0 0 4 0]
[0 7 4 0 0 1 5]
[0 0 3 4 1 0 7]
[0 0 0 0 5 7 0]
```

Sage también presenta una función para saber si un grafo es etiquetado o no, ésta es la función ***.weighted()***.

```
grafoEti.weighted()
```

```
True
```

En esta ocasión, la **longitud** de los caminos es la suma de las etiquetas de las aristas que componen el camino. Además, la **distancia** entre dos vértices es la mínima longitud de todos caminos cuyos extremos son ambos vértices.

Ejercicio 4. Crea una función que reciba un grafo etiquetado (g) y dos vértices (a,b), y devuelva la distancia entre ambos vértices.

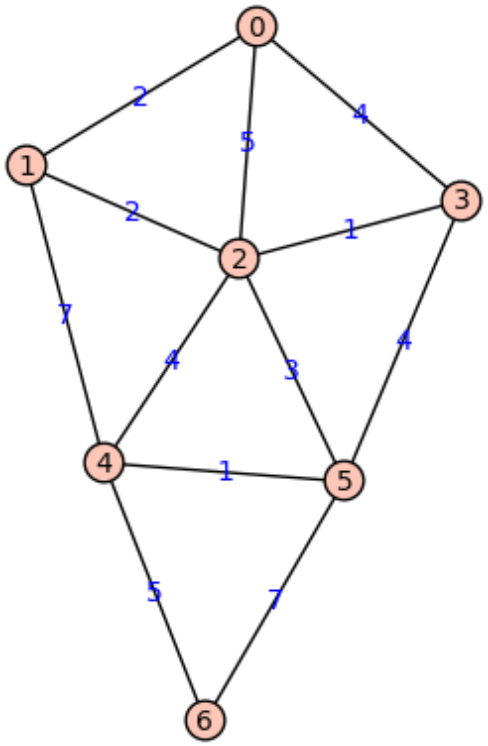
```
#Realiza el ejercicio en esta celda
def getDistancia(g, a, b):
    if g.weighted():
        print "La distancia entre ambos vertices es: ",
g.distance(a,b,by_weight=True)
    else:
        print "El grafo no es etiquetado"
```

```
lisAdya = {0:{1:2,2:5,3:4},1:{2:2,4:7},2:{3:1,4:4,5:3},3:{5:4},4:{5:1,6:5},5:
{6:7}}
grafo=Graph(lisAdya,weighted=True)
getDistancia(grafo,0,5)
```

```
La distancia entre ambos vertices es:  7
```

Por otro lado, el ***Algoritmo de Dijkstra*** calcula la distancia entre dos vértices a partir del camino más corto existente en un grafo etiquetado, Sage tiene incluido dicho algoritmo e implementado en la función ***.shortest_path(a,b,by_weight=True)***, siendo a y b los vértices de los que se quiere calcular la distancia. Sin embargo, esta función solo proporciona el camino más corto, para conocer su distancia se utiliza la función ***.shortest_path_length(a,b,by_weight=True)***.

```
grafoEti.show(edge_labels=True)
```




```
grafoEti.shortest_path(0,6,by_weight=True)
```

```
[0, 1, 2, 5, 4, 6]
```

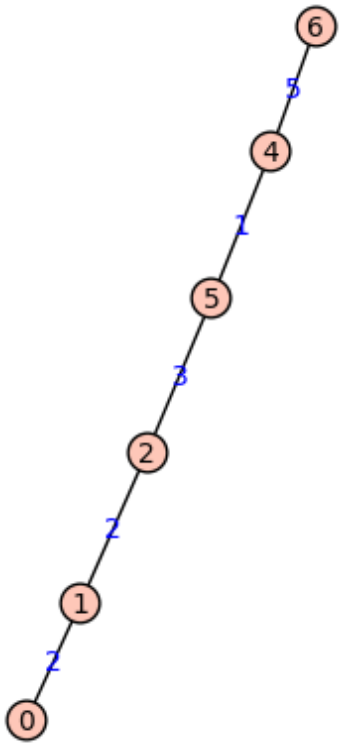
Como observamos el camino más corto que tiene por extremos los vértices 0 y 6, es el 012546. Veamos cuál es la distancia entre ambos vértices.

```
grafoEti.shortest_path_length(0,6,by_weight=True)
```

```
13
```

Vamos a representar dicho camino, que recordemos no deja de ser un subgrafo.

```
lisAdya = {0:{1:2},1:{2:2},2:{5:3},5:{4:1},4:{6:5}}
Graph(lisAdya,weighted=True).show(edge_labels=True)
```



Ejercicio 5. Genera un grafo etiquetado con 9 vértices y 12 aristas, cuyas etiquetas tengan valores entre 5 y 15.

a) Representa el grafo etiquetado.

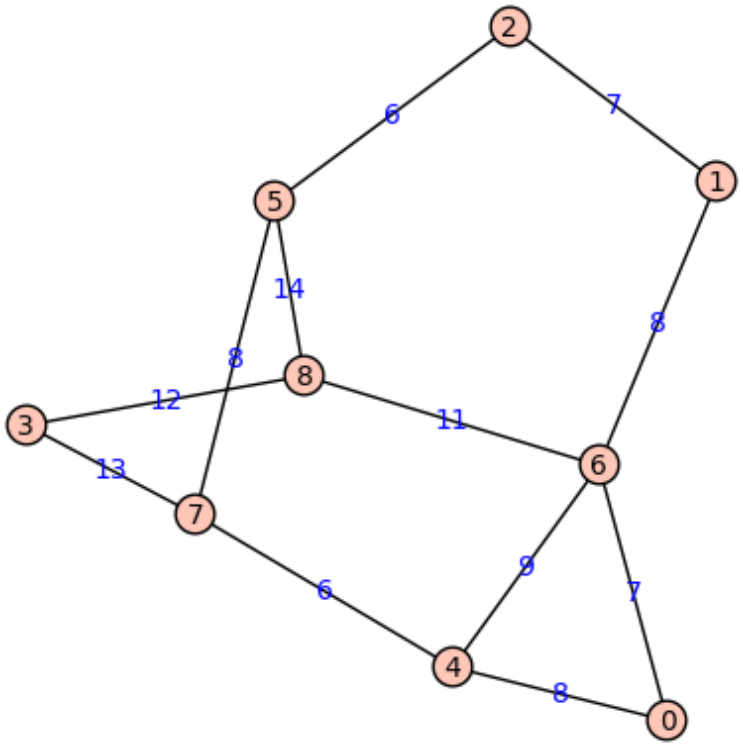
b) Determina la distancia entre los vértices 0 y 8 utilizando la función creada en el ejercicio 4.

c) Determina la distancia entre los vértices 0 y 8 utilizando el Algoritmo de Dijkstra implementado en Sage.

d) ¿Cuál es el camino más corto entre los vértices 0 y 8? Representa dicho camino.

```
#Realiza el ejercicio en esta celda
#a)
lisAdya = {0:{6:7,4:8},1:{6:8},2:{5:6,1:7},3:{8:12,7:13},4:{6:9},5:
{7:8,8:14},6:{8:11},7:{4:6}}
grafoEti=Graph(lisAdya, weighted=True)
grafoEti.show(edge_labels=True)
len(grafoEti.edges())
```

```
12
```



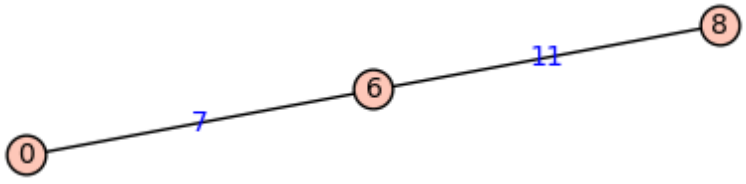

```
#b)
getDistancia(grafoEti,0,8)
```

```
    La distancia entre ambos vertices es:  18
```

```
#c)
grafoEti.shortest_path_length(0,8,by_weight=True)
```

```
    18
```

```
#d)
grafoEti.shortest_path(0,8,by_weight=True)
lisAdya = {0:{6:7},6:{8:11}}
Graph(lisAdya,weighted=True).show(edge_labels=True)
```



Ejercicio 6. Genera un grafo etiquetado con 7 vértices y 10 aristas, cuyas etiquetas tengan valores entre 5 y 15.

a) Representa el grafo etiquetado.

b) Determina la distancia entre los vértices 0 y 4 utilizando la función creada en el ejercicio 4.

c) Determina la distancia entre los vértices 0 y 4 utilizando el Algoritmo de Dijkstra implementado en Sage.

d) ¿Cuál es el camino más corto entre los vértices 0 y 4? Representa dicho camino.

\r\n
\r\n

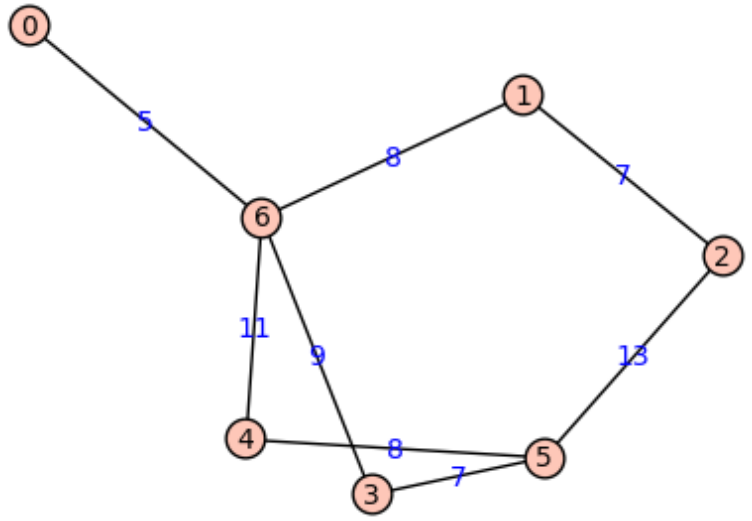
\r\n

\r\n

\r\n\r\n\r\n\r\n\r\n\r\n
' });

```
#Realiza el ejercicio en esta celda
#a)
lisAdya = {0:{6:5},1:{6:8,},2:{5:13,1:7},3:{6:9},4:{6:9},5:{4:8,3:7},6:{4:11}}
grafoEti=Graph(lisAdya, weighted=True)
grafoEti.show(edge_labels=True)
len(grafoEti.edges())
```

```
8
```



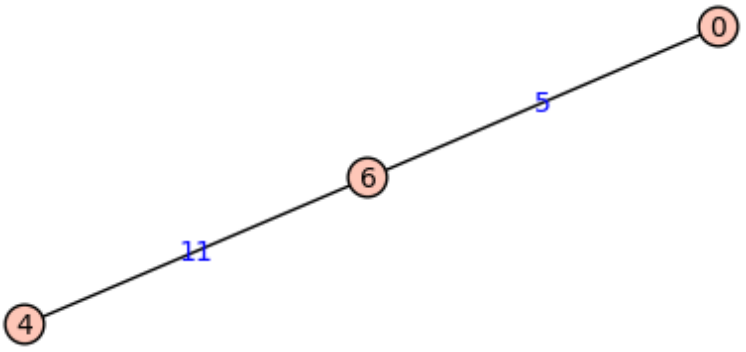

```
#b)
getDistancia(grafoEti,0,4)
```

```
La distancia entre ambos vertices es:  16
```

```
#c)
grafoEti.shortest_path_length(0,4,by_weight=True)
```

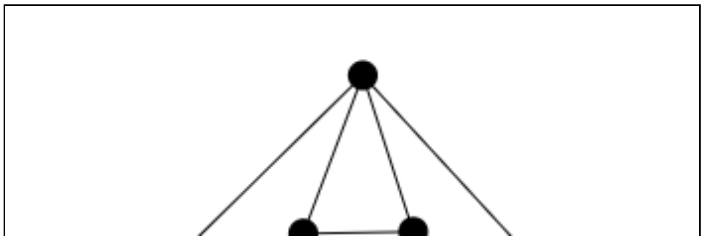
```
16
```

```
#d)
grafoEti.shortest_path(0,4,by_weight=True)
lisAdya = {0:{6:5},6:{4:11}}
Graph(lisAdya,weighted=True).show(edge_labels=True)
```

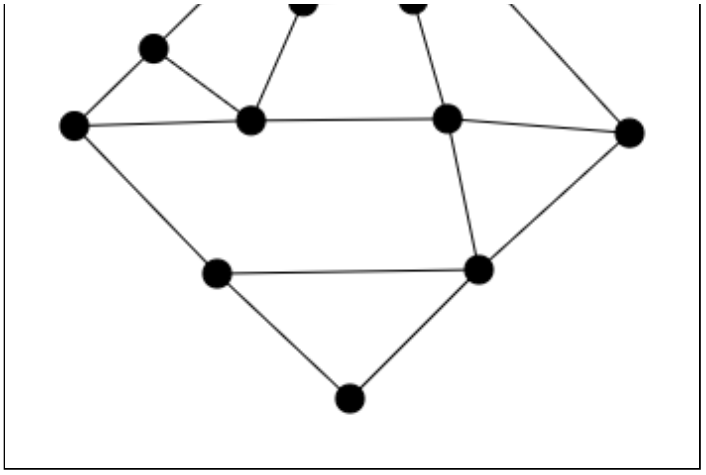


Ejercicio 7. Abre la lista de ejercicios del Tema 1 que aparece en el campus virtual. Considerando el ejercicio 48, representa el grafo etiquetado que se muestra y resuelve dicho problema.

```
grafo = Graph({0:[1,2,4],1:[0,3,4],2:[0,3,5,7],3:[1,2,6,8],4:[0,1,5,8],5:
[2,4,7],6:[3,8,9],7:[2,5,9,10],8:[3,4,6],9:[6,7,10],10:[7,9]});
grafo.set_pos({0:[205.9090909090909,236.1267605633803],1:
[150.45454545454544,235.23047375160053],2:
[223.1818181818182,176.0755441741357],3:
[124.0909090909091,175.17925736235597],4:[180.45454545454544,315],5:
```

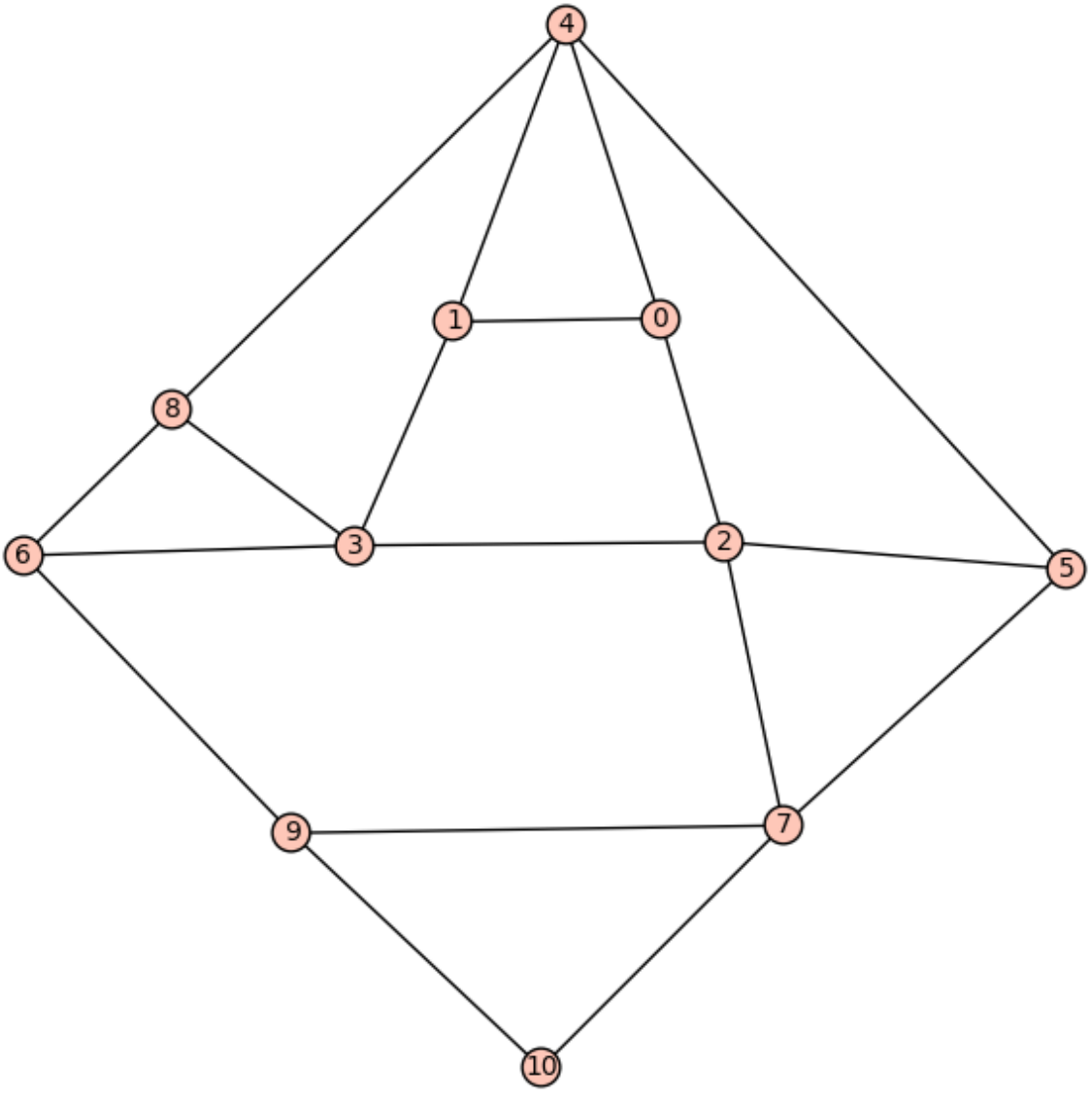


live:
variable name: grafo
strength:
length:
help



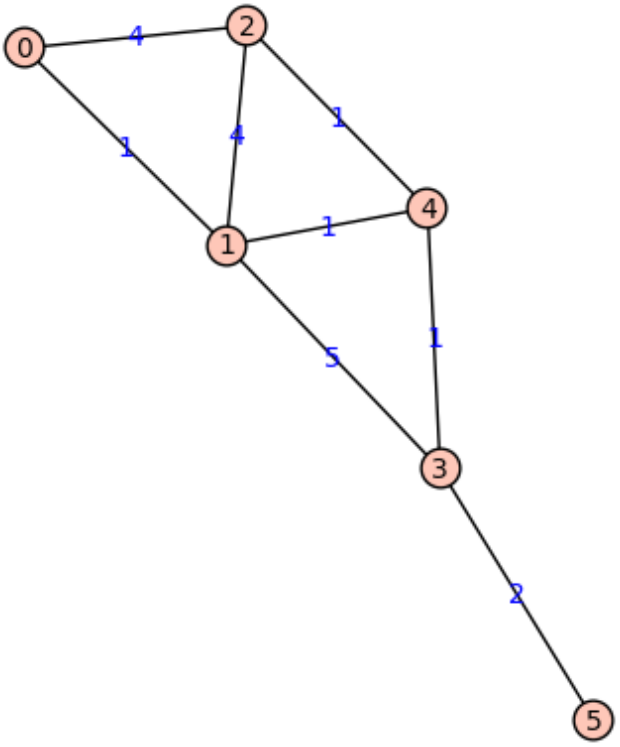
Save Close

```
grafo.plot()
```



Ejercicio 8. Sea la topología de red representada por el siguiente grafo etiquetado. Aplica el algoritmo de Dijkstra para encontrar el camino de coste mínimo entre los puntos 0 y 5.

```
grafoEti = Graph({0:{1:1,2:4},1:{0:1,2:4,3:5,4:1},2:{1:4,0:4,4:1},3:
{1:5,4:1,5:2},4:{1:1,2:1,3:1},5:{3:2}},weighted=True)
grafoEti.show(edge_labels=True)
```

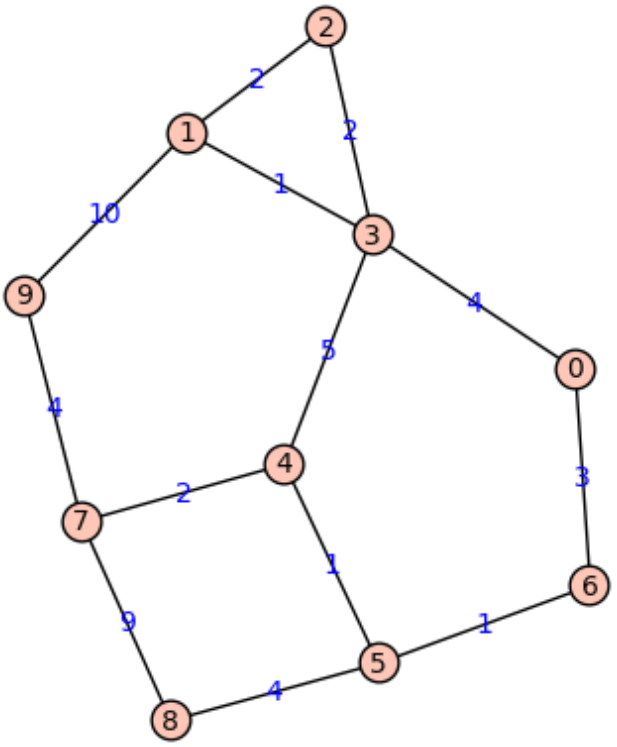



```
#Realiza el ejercicio en esta celda
grafoEti.shortest_path(0,5,by_weight=True)
```

```
[0, 1, 4, 3, 5]
```

Ejercicio 9. Sea la topología de red representada por el siguiente grafo etiquetado. Aplica el algoritmo de Dijkstra para encontrar el camino de coste mínimo entre 0 y 9.

```
grafoEti = Graph({0:{3:4,6:3},1:{2:2,3:1,9:10},2:{1:2,3:2},3:
{0:4,1:1,2:2,4:5},4:{3:5,5:1,7:2},5:{4:1,6:1,8:4},6:{0:3,5:1},7:{4:2,8:9,9:4},8:
{7:9,5:4},9:{1:10,7:4}},weighted=True)
grafoEti.show(edge_labels=True)
```




```
#Realiza el ejercicio en esta celda
grafoEti.shortest_path(0,9,by_weight=True)
```

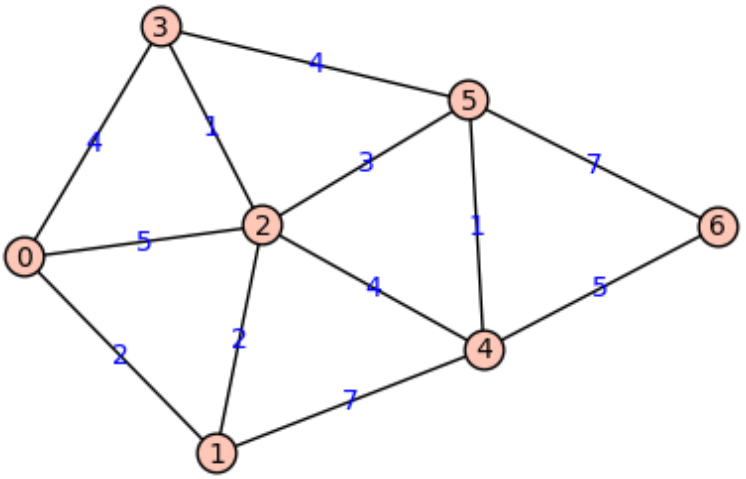
```
[0, 6, 5, 4, 7, 9]
```

Árbol recubridor minimal: Algoritmos de Kruskal y de Prim

En primer lugar, un árbol recubridor minimal es cualquier árbol recubridor tal que la suma de las etiquetas de las aristas sea menor o igual a la de cualquier otro árbol recubridor.

```
lisAdya = {0:{1:2,2:5,3:4},1:{2:2,4:7},2:{3:1,4:4,5:3},3:{5:4},4:{5:1,6:5},5:
{6:7}}
grafoEti = Graph(lisAdya,weighted=True)
grafoEti.show(edge_labels=True),
print list(grafoEti.neighbor_iterator(2))
grafoEti.min_spanning_tree(algorithm='Prim_Boost',starting_vertex=0)
```

```
[0, 1, 3, 4, 5]
[(0, 1, 2), (1, 2, 2), (2, 3, 1), (2, 5, 3), (4, 5, 1), (4, 6, 5)]
```



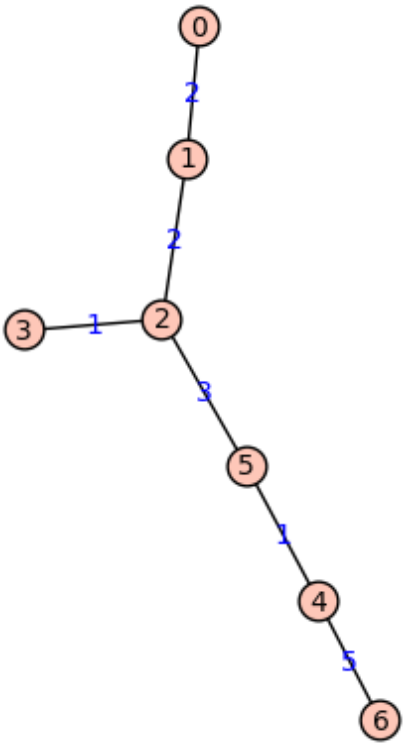
Sabemos que existen dos algoritmos que permiten determinar el árbol recubridor mínimal de un grafo etiqueto. Éstos son el ***Algoritmo de Kruskal*** y el ***Algoritmo de Prim***.

Ejercicio 10. Crea una función que reciba un grafo etiquetado (g) y un vértice de inicio (v), y que devuelva el árbol recubridor minimal que se obtiene cuando se aplica el Algoritmo de Kruskal.

```
#Realiza el ejercicio en esta celda
def getArbolRecubridorMinimalKruskal(g,v):
    if g.weighted():
        exist = false
        if v in g.vertices():
            lisAdya =
grafoEti.min_spanning_tree(algorithm='Kruskal_Boost',starting_vertex=v)
            print lisAdya
            g = Graph(lisAdya,weighted=True)
            g.show(edge_labels=True)
        else:
            print "No existe el vertice ", v
    else:
        print "No es un grafo etiquetado"

lisAdya = {0:{1:2,2:5,3:4},1:{2:2,4:7},2:{3:1,4:4,5:3},3:{5:4},4:{5:1,6:5},5:
{6:7}}
grafoEti = Graph(lisAdya,weighted=True)
getArbolRecubridorMinimalKruskal(grafoEti,0)
```

```
[(0, 1, 2), (1, 2, 2), (2, 3, 1), (2, 5, 3), (4, 5, 1), (4, 6, 5)]
```

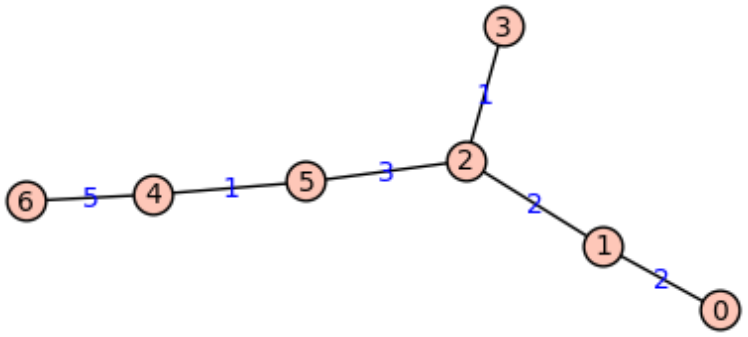


Ejercicio 11. Crea una función que reciba un grafo etiquetado (g) y un vértice de inicio (v), y que devuelva el árbol recubridor minimal que se obtiene cuando se aplica el Algoritmo de Prim.

```
#Realiza el ejercicio en esta celda
#Realiza el ejercicio en esta celda
def getArbolRecubridorMinimalPrim(g,v):
    if g.weighted():
        exist = false
        if v in g.vertices():
            lisAdya =
grafoEti.min_spanning_tree(algorithm='Prim_Boost',starting_vertex=v)
            print lisAdya
            g = Graph(lisAdya,weighted=True)
            g.show(edge_labels=True)
        else:
            print "No existe el vertice ", v
    else:
        print "No es un grafo etiquetado"

lisAdya = {0:{1:2,2:5,3:4},1:{2:2,4:7},2:{3:1,4:4,5:3},3:{5:4},4:{5:1,6:5},5:
{6:7}}
grafoEti = Graph(lisAdya,weighted=True)
getArbolRecubridorMinimalPrim(grafoEti,0)
```

```
[(0, 1, 2), (1, 2, 2), (2, 3, 1), (2, 5, 3), (4, 5, 1), (4, 6, 5)]
```



En Sage, ambos algoritmos están disponibles en la función ***.min_spanning_tree(algorithm,starting_vertex***), donde el argumento *algorithm* indica el algoritmo a utilizar 'Kruskal' o 'Prim_Boost', respectivamente, y el argumento *starting_vertex* (por defecto toma el valor None) toma un valor numérico que indica el vértice por el que se quiera empezar a determinar el árbol recubridor mínimo.

```
grafoEti.min_spanning_tree(algorithm='Prim_Boost',starting_vertex=0)
```

```
[(0, 1, 2), (1, 2, 2), (2, 3, 1), (2, 5, 3), (4, 5, 1), (4, 6, 5)]
```

```
grafoEti.min_spanning_tree(algorithm='Kruskal',starting_vertex=0)
```

```
[(0, 1, 2), (1, 2, 2), (2, 3, 1), (2, 5, 3), (4, 5, 1), (4, 6, 5)]
```

Como se observa ambos algoritmos proporcionan el mismo árbol recubridor mínimal.

Ejercicio 12. Genera un grafo etiquetado con 7 vértices y 10 aristas, cuyas etiquetas tengan valores entre 2 y 14 (todas distintas). Selecciona un vértice del que comenzar y

a) Aplica la función creada en el ejercicio 10 para encontrar el árbol recubridor minimal.

b) Aplica la función creada en el ejercicio 11 para encontrar el árbol recubridor minimal. ¿Coincide con el árbol anterior?

c) Aplica el algoritmo de Kruskal proporcionado por Sage para encontrar el árbol recubridor minimal. ¿Coincide con los árboles anteriores?

d) Aplica el algoritmo de Prim proporcionado por Sage para encontrar el árbol recubridor minimal. ¿Coincide con los árboles anteriores?

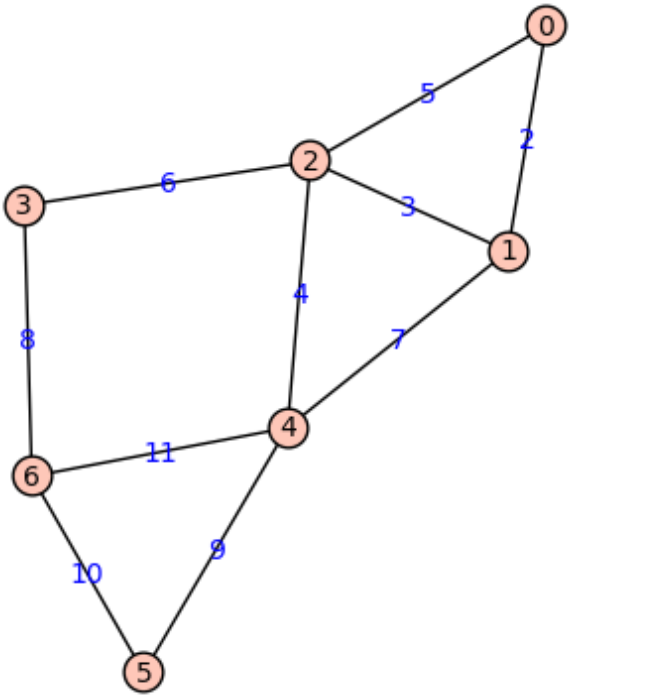
\r\n\r\n

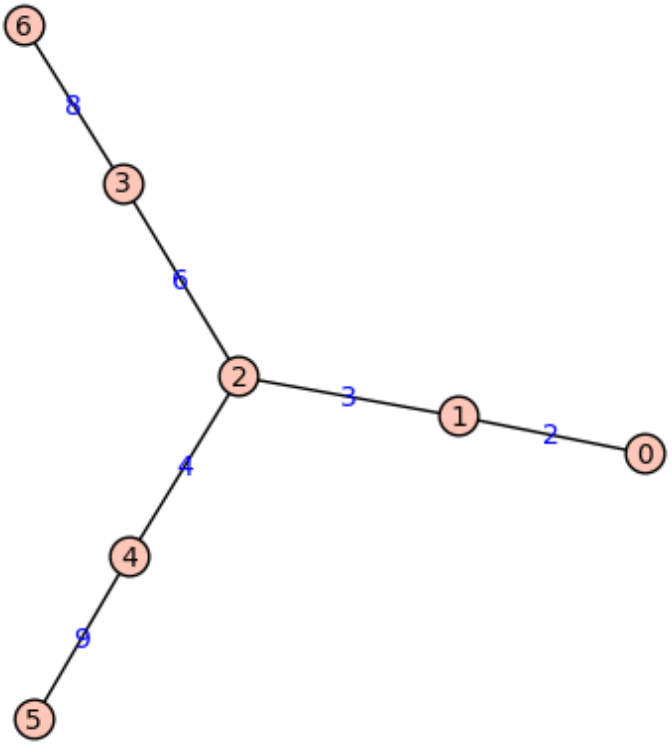
\r\n

\r\n\r\n\r\n\r\n\r\n\r\n\r\n' });

```
#Realiza el ejercicio en esta celda
# a)
lisAdya = {0:{1:2,2:5},1:{2:3,4:7},2:{3:6,4:4},3:{6:8},4:{5:9,6:11},5:{6:10}}
grafoEti = Graph(lisAdya,weighted=True)
grafoEti.plot(edge_labels=True).show(figsize=5)
getArbolRecubridorMinimalKruskal(grafoEti,0)
```

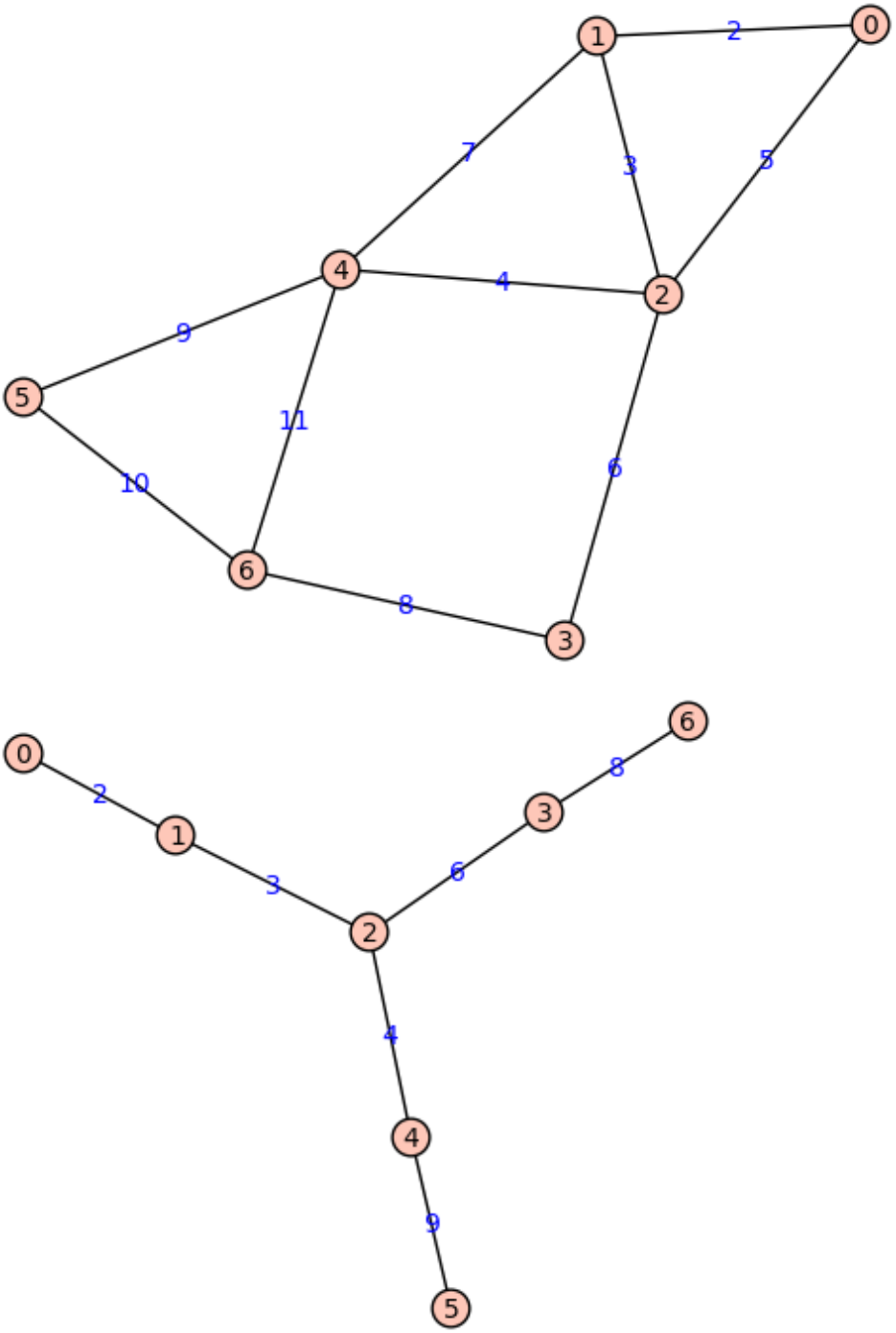
```
[(0, 1, 2), (1, 2, 3), (2, 3, 6), (2, 4, 4), (3, 6, 8), (4, 5, 9)]
```





```
# b)
grafoEti.plot(edge_labels=True).show(figsize=5)
getArbolRecubridorMinimalPrim(grafoEti,0)
```

```
[(0, 1, 2), (1, 2, 3), (2, 3, 6), (2, 4, 4), (3, 6, 8), (4, 5, 9)]
```



Coinciden porque son dos maneras de hallar el arbol recubridor minimal y deben dar el mismo resultado.

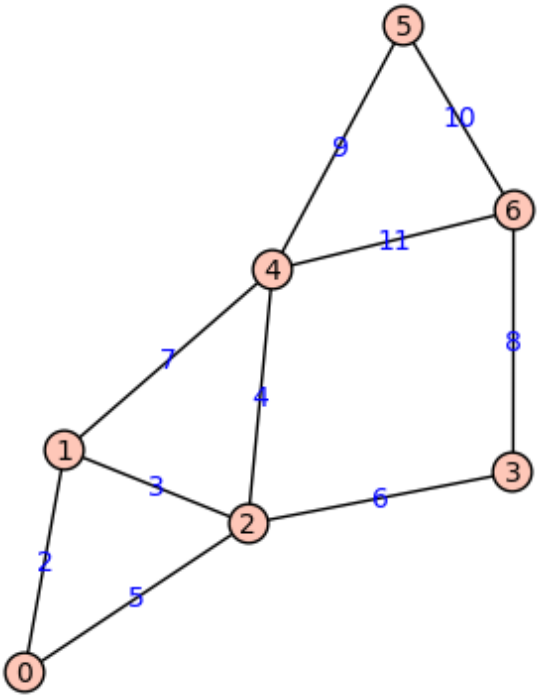
\r\n\r\n

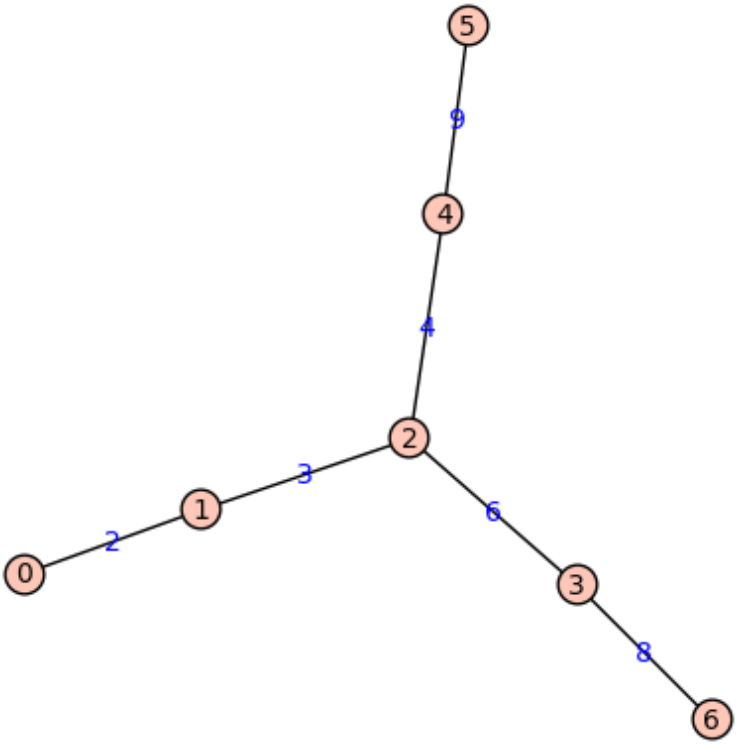
\r\n

\r\n\r\n\r\n\r\n\r\n\r\n\r\n' });

```
# c)
grafoEti.plot(edge_labels=True).show(figsize=5)
lisAdya =
grafoEti.min_spanning_tree(algorithm='Kruskal_Boost',starting_vertex=0)
print lisAdya
g = Graph(lisAdya,weighted=True)
g.show(edge_labels=True)
```

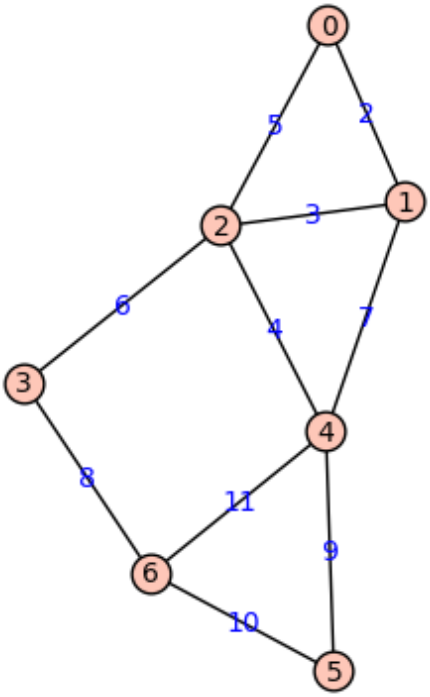
```
[(0, 1, 2), (1, 2, 3), (2, 3, 6), (2, 4, 4), (3, 6, 8), (4, 5, 9)]
```

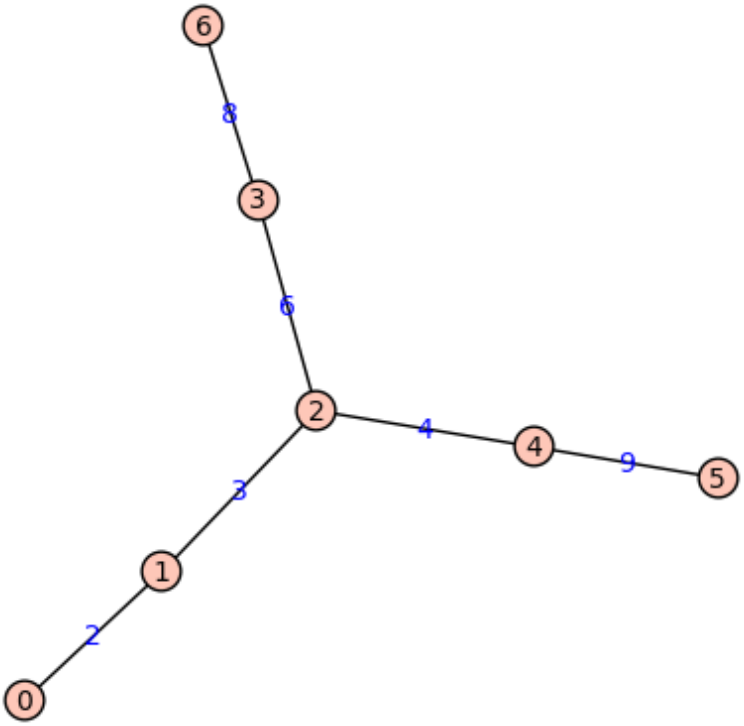




```
# d)
grafoEti.plot(edge_labels=True).show(figsize=5)
lisAdya = grafoEti.min_spanning_tree(algorithm='Prim_Boost',starting_vertex=0)
print lisAdya
g = Graph(lisAdya,weighted=True)
g.show(edge_labels=True)
```

```
[(0, 1, 2), (1, 2, 3), (2, 3, 6), (2, 4, 4), (3, 6, 8), (4, 5, 9)]
```





Coinciden porque son dos maneras de hallar el arbol recubridor minimal y deben dar el mismo resultado.

Ejercicio 13. Considera el grafo (Graph1), utiliza el algoritmo de Kruskal para obtener el árbol recubridor minimal del grafo, y representa dicho árbol.

\r\n\r\n

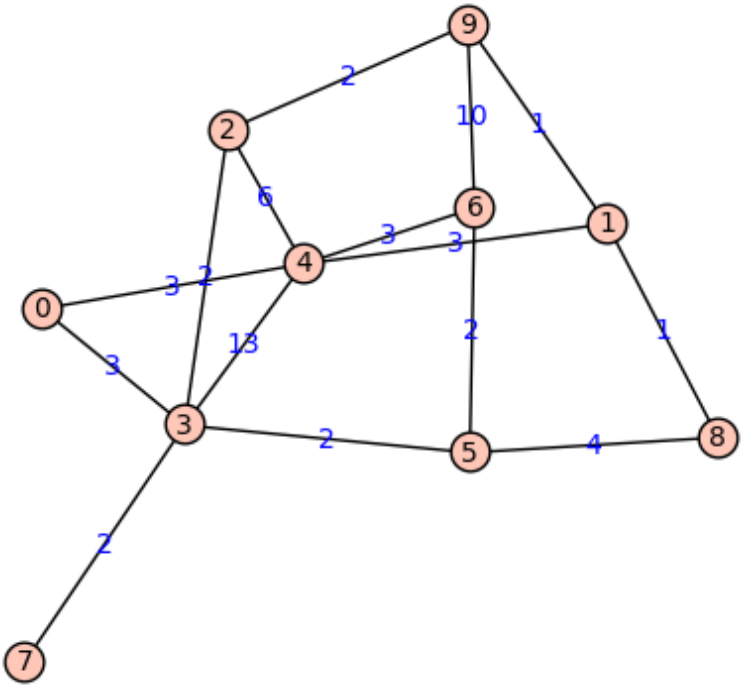
\r\n

\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n

Ejercicio 13. Considera el grafo (Graph1), utiliza el algoritmo de Kruskal para obtener el árbol recubridor minimal del grafo, y representa dicho árbol.

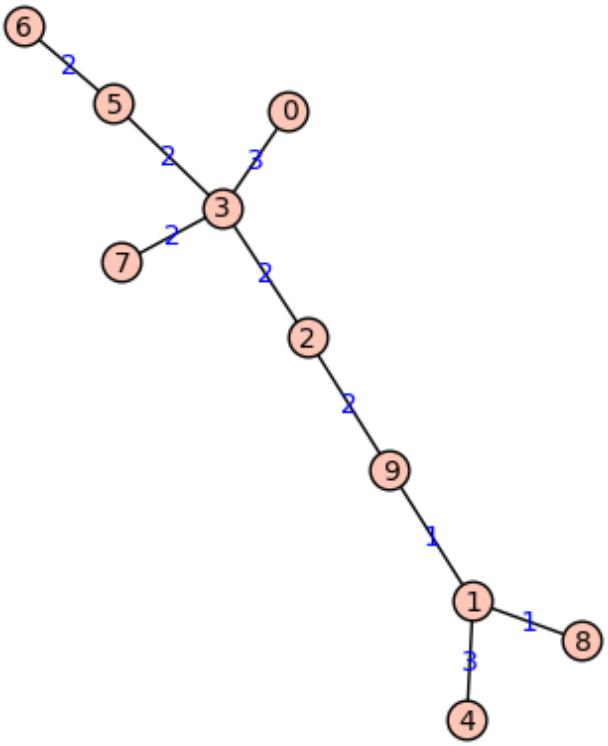
' });

```
Graf1 = Graph({0:{3:1,4:3},1:{4:5,8:4,9:10},2:{3:3,4:2,9:12},3:
{0:3,2:2,4:4,5:2,7:1},4:{0:3,1:3,2:6,3:13,6:11},5:{3:2,6:3,8:4},6:
{4:3,5:2,9:4},7:{3:2},8:{1:1,5:4},9:{6:10,1:1,2:2}},weighted=True)
Graf1.show(edge_labels=True)
```




```
#Realiza el ejercicio en esta celda
lisAdya = Graf1.min_spanning_tree(algorithm='Kruskal_Boost',starting_vertex=0)
print lisAdya
g = Graph(lisAdya,weighted=True)
g.show(edge_labels=True)
```

```
[(0, 3, 3), (1, 4, 3), (1, 8, 1), (1, 9, 1), (2, 3, 2), (2, 9, 2), (3,
5, 2), (3, 7, 2), (5, 6, 2)]
```



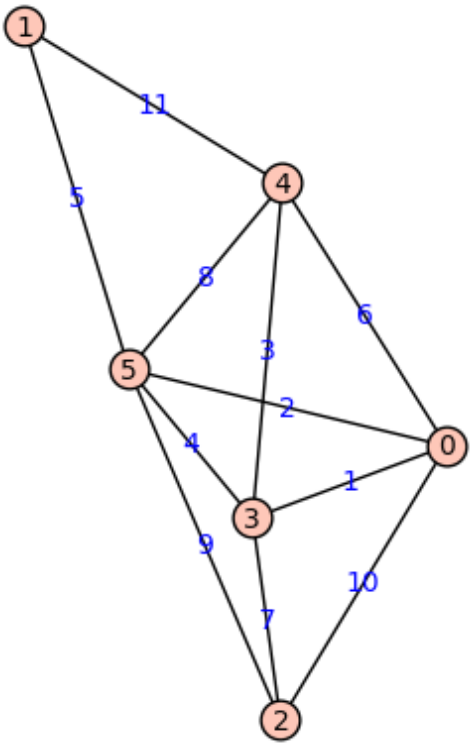
Problema del Flujo Máximo: Algoritmo de Ford-Fulkenson

En los **problemas de flujo**, las aristas representan canales por los que circulan: datos, agua, coches,... entre un origen y un destino. El problema de flujo máximo consiste en encontrar en un grafo etiquetado un subgrafo dirigido con la cantidad máxima de flujo que puede circular entre el origen y el destino.

Para ello es necesario recordar: la *ley de conservación de flujo* (para todo vértice, la suma de las etiquetas de las aristas que salen es igual a la suma de las etiquetas de las aristas que llegan), y además, que las etiquetas de las aristas del subgrafo dirigido no pueden ser mayores a las del grafo inicial.

Para resolver este tipo de problemas, Sage tiene implementeda el ***Algoritmo de Ford-Fulkenson*** en la función ***.flow(a,b,algorithm='FF')***, siendo a y b los vértices origen y destino.

```
lisAdya = {0:{2:10,3:1,4:6, 5:2},1:{4:5,5:11},2:{0:10,3:7,5:9},3:{0:1, 2:7, 4:3,
5:4},4:{0:6,3:3,5:8, 1:11},5:{0:2, 1:5, 2:9, 3:4, 4:8}}
grafoEti = Graph(lisAdya,weighted=True)
grafoEti.show(edge_labels=True)
```




```
grafoEti.flow(2,4,algorithm='FF')
```

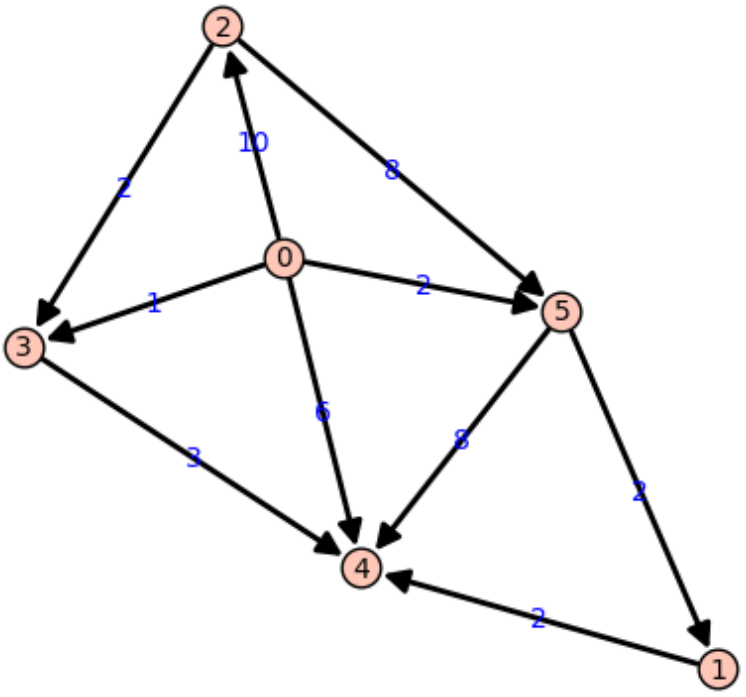
```
22
```

Como podemos observar esta función solo proporciona el valor del flujo máximo que existe entre el vértice origen 0 y el vértice destino 6, pero no nos proporciona el subgrafo dirigido que resuelve el problema. Para ello, podemos añadir el argumento *value_only=False* a la función *.flow()*. Esto permitirá obtener una lista con dos valores, el primero de ellos será el valor del flujo máximo, y el segundo será el subgrafo dirigido que resuelve el problema. Para poder representar dicho subgrafo debemos guardar en una variable esta segunda salida, y a continuación representarla como un grafo etiquetado.

```
grafoEti.flow(0,4,algorithm='FF',value_only=False)
```

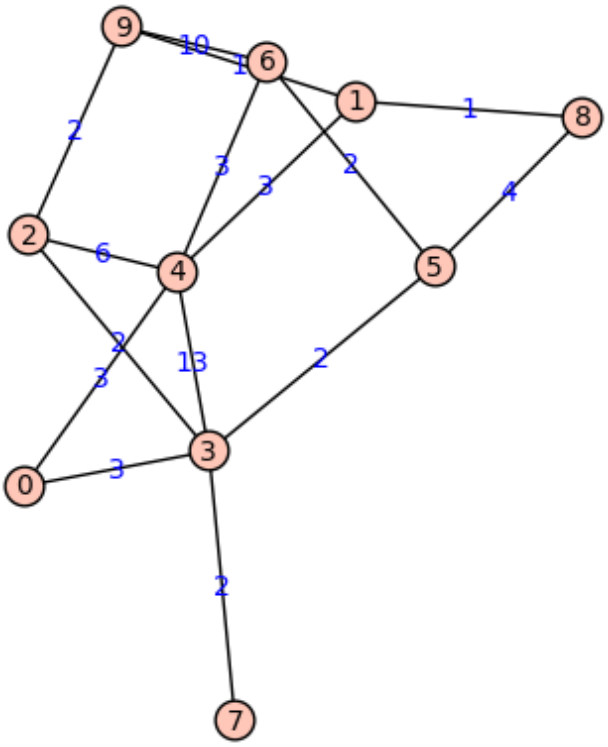
```
(19, Digraph on 6 vertices)
```

```
subgrafo = grafoEti.flow(0,4,algorithm='FF',value_only=False)[1]
subgrafo.show(edge_labels=True)
```



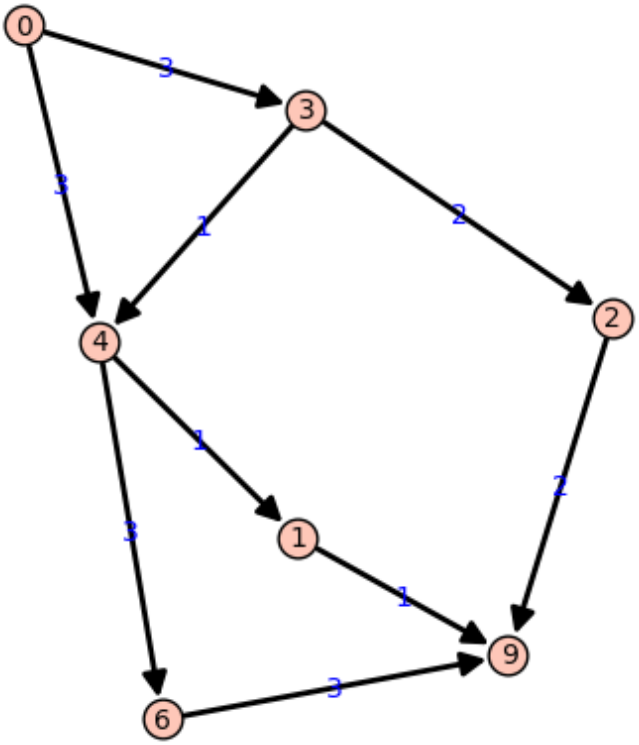
Ejercicio 14. Considerando el grafo (Graph1), aplica el algoritmo de Ford-Fulkenson para resolver el problema de flujo máximo del grafo etiquetado entre el vértice origen 0 y el vértice destino 9. Representa, además, el subgrafo dirigido que resuelve el problema.

```
Graf1 = Graph({0:{3:1,4:3},1:{4:5,8:4,9:10},2:{3:3,4:2,9:12},3:
{0:3,2:2,4:4,5:2,7:1},4:{0:3,1:3,2:6,3:13,6:11},5:{3:2,6:3,8:4},6:
{4:3,5:2,9:4},7:{3:2},8:{1:1,5:4},9:{6:10,1:1,2:2}},weighted=True)
Graf1.show(edge_labels=True)
```




```
#Realiza el ejercicio en esta celda
vInicio = 0
vFin = 9
print "El flujo maximal es: ", Graf1.flow(vInicio,vFin,algorithm='FF')
subgrafo = Graf1.flow(vInicio,vFin,algorithm='FF',value_only=False)[1]
subgrafo.show(edge_labels=True)
```

```
El flujo maximal es:  6
```



Ejercicio 15. Genera un grafo etiquetado, conexo, con 14 vértices y 13 aristas, cuyas etiquetas tomen valores entre 1 y 10.

a) Representa el grafo etiquetado.

b) Encuentra el árbol recubridor minimal del grafo utilizando el algoritmo de Kruskal.

c) Encuentra el árbol recubridor minimal del grafo utilizando el algoritmo de Prim.

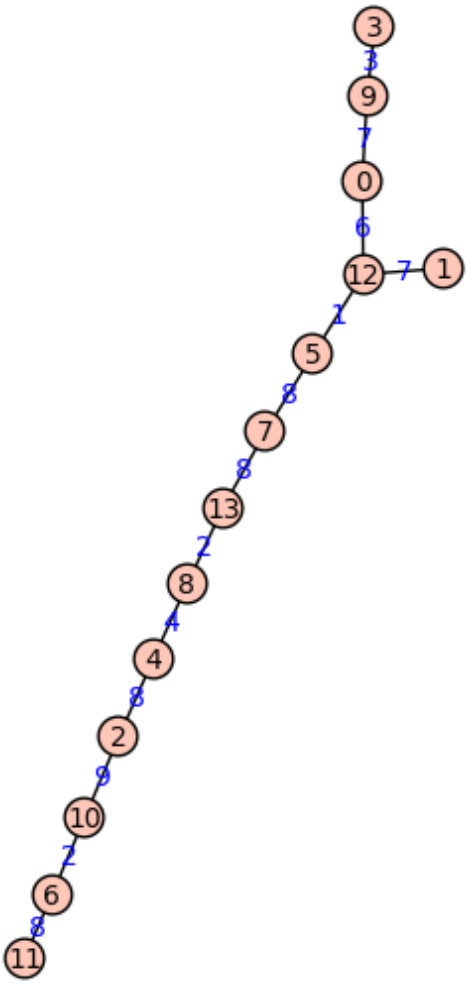
d) Considerando el problema de flujo máximo entre el vértice origen 0 y el vértice destino 12, aplica el algoritmo de Ford-Fulkenson para resolver el problema, indicando tanto el flujo máximo como el subgrafo dirigido que lo resuelve.

\r\n\r\n

\r\n

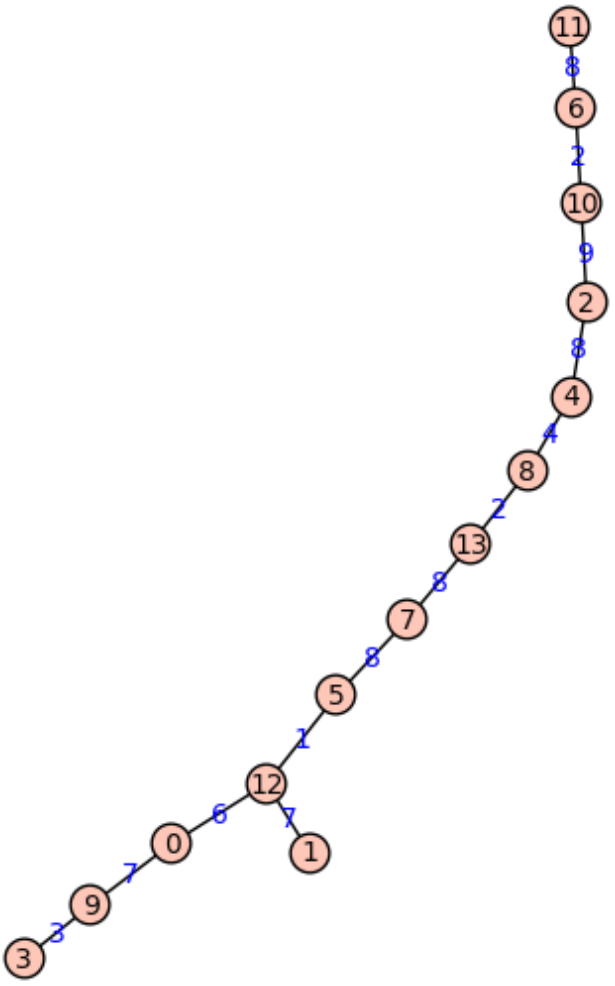
\r\n\r\n\r\n\r\n\r\n\r\n\r\n' });

```
#a)
grafo = Graph({0:{9:1,12:2},1:{12:3},2:{4:4,10:5},3:{9:6},4:{8:7,2:8},5:
{12:9,7:10},6:{10:3,11:5},7:{5:8,13:2},8:{13:10,4:4},9:{0:7,3:3},10:
{2:9,6:2},11:{6:8},12:{0:6,1:7,5:1},13:{7:8,8:2}}); grafo.show(edge_labels=true,
```



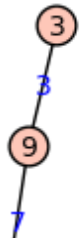

```
#b)
lisAdya = grafo.min_spanning_tree(algorithm='Kruskal_Boost',starting_vertex=2)
print lisAdya
g = Graph(lisAdya,weighted=True)
g.show(edge_labels=True, figsize=7)
```

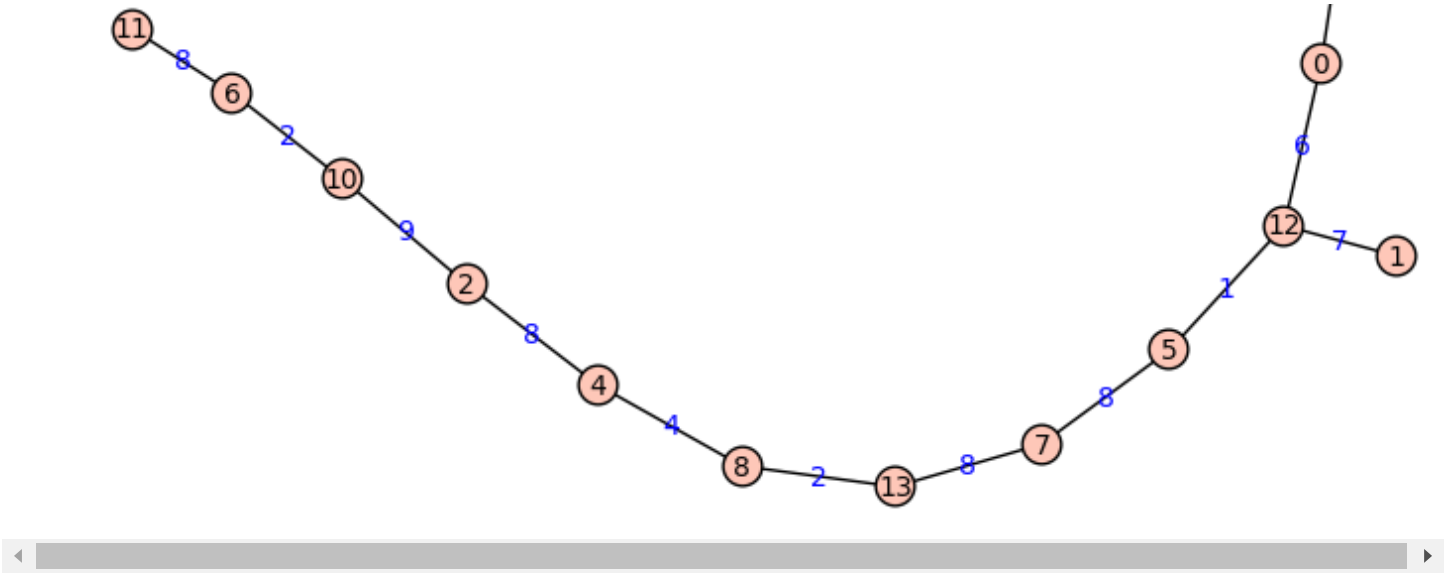
```
[(0, 9, 7), (0, 12, 6), (1, 12, 7), (2, 4, 8), (2, 10, 9), (3, 9, 3),
(4, 8, 4), (5, 7, 8), (5, 12, 1), (6, 10, 2), (6, 11, 8), (7, 13, 8),
(8, 13, 2)]
```




```
#c)
lisAdya = grafo.min_spanning_tree(algorithm='Prim_Boost',starting_vertex=2)
print lisAdya
g = Graph(lisAdya,weighted=True)
g.show(edge_labels=True, figsize=7)
```

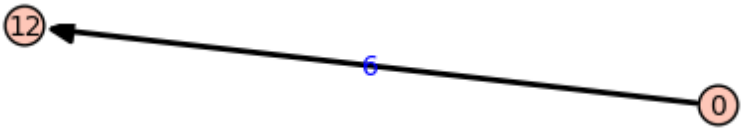
```
[(0, 9, 7), (0, 12, 6), (1, 12, 7), (2, 4, 8), (2, 10, 9), (3, 9, 3),
(4, 8, 4), (5, 7, 8), (5, 12, 1), (6, 10, 2), (6, 11, 8), (7, 13, 8),
(8, 13, 2)]
```






```
#d)
vInicio = 0
vFin = 12
print "El flujo maximal es: ", grafo.flow(vInicio,vFin,algorithm='FF')
subgrafo = grafo.flow(vInicio,vFin,algorithm='FF',value_only=False)[1]
subgrafo.show(edge_labels=True)
```

```
El flujo maximal es:  6
```



**Ejercicio 16. Genera un grafo etiquetado, conexo, con 15 vértices y 14 aristas, cuyas etiquetas tomen valores entre 1 y 10.

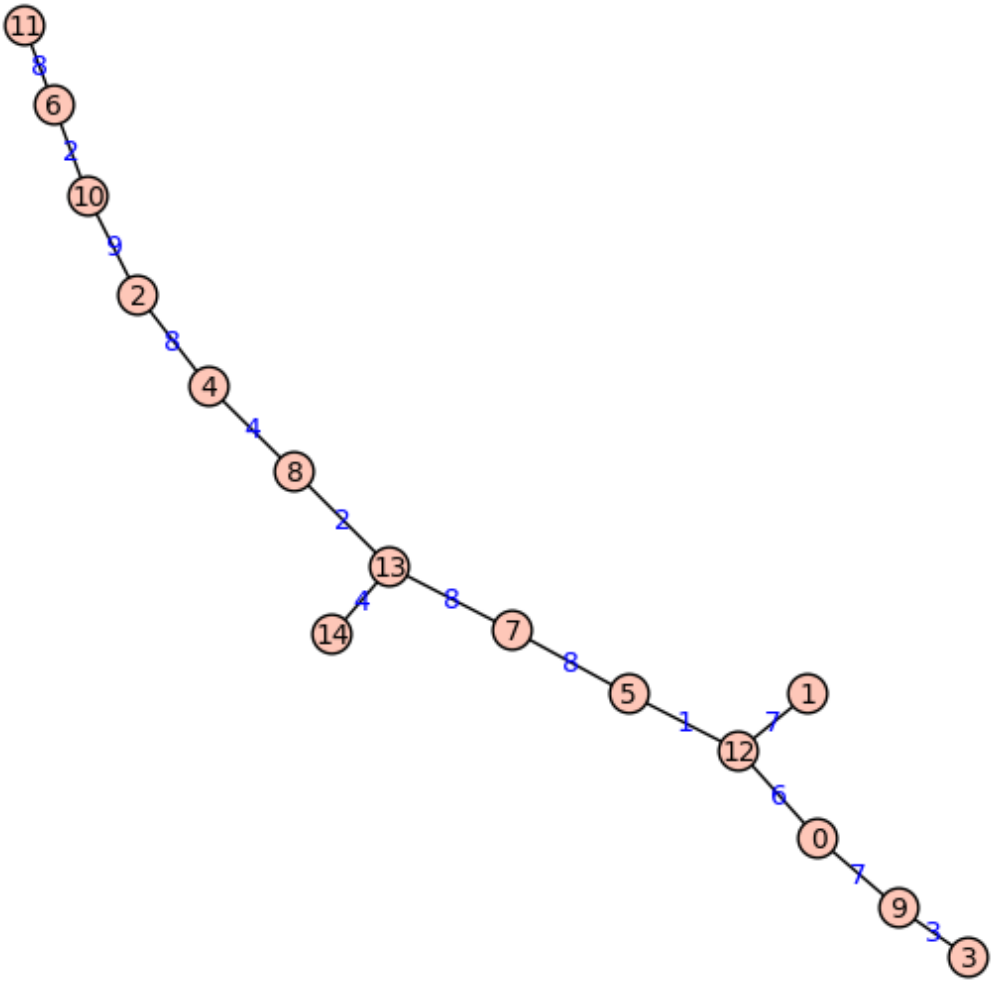
a) Representa el grafo etiquetado.

b) Encuentra el árbol recubridor minimal del grafo utilizando el algoritmo de Kruskal.

c) Encuentra el árbol recubridor minimal del grafo utilizando el algoritmo de Prim.

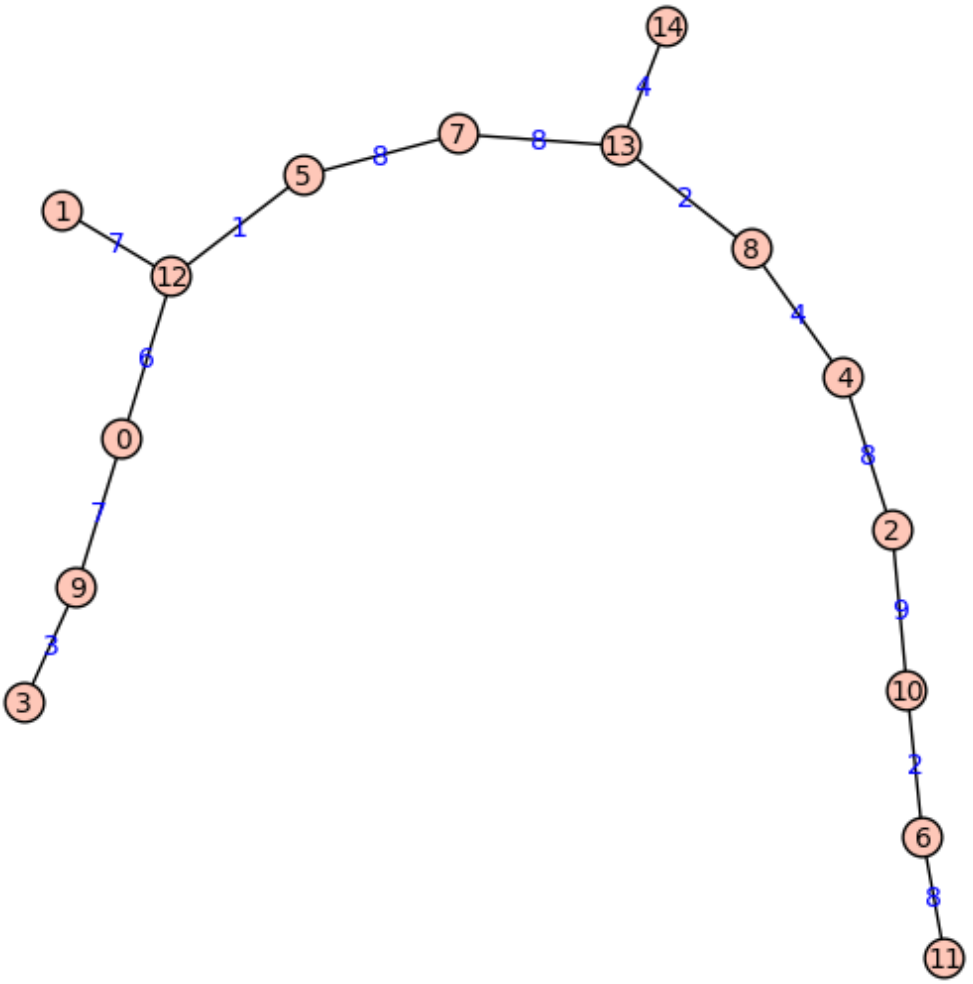
d) Considerando el problema de flujo máximo entre el vértice origen 0 y el vértice destino 12, aplica el algoritmo de Ford-Fulkenson para resolver el problema, indicando tanto el flujo máximo como el subgrafo dirigido que lo resuelve.

```
#Realiza el ejercicio en esta celda
#a)
grafo = Graph({0:{9:1,12:2},1:{12:3},2:{4:4,10:5},3:{9:6},4:{8:7,2:8},5:
{12:9,7:10},6:{10:3,11:5},7:{5:8,13:2},8:{13:10,4:4},9:{0:7,3:3},10:
{2:9,6:2},11:{6:8},12:{0:6,1:7,5:1},13:{7:8,8:2}, 14:{13:4}});
grafo.show(edge_labels=true, figsize=7);
```



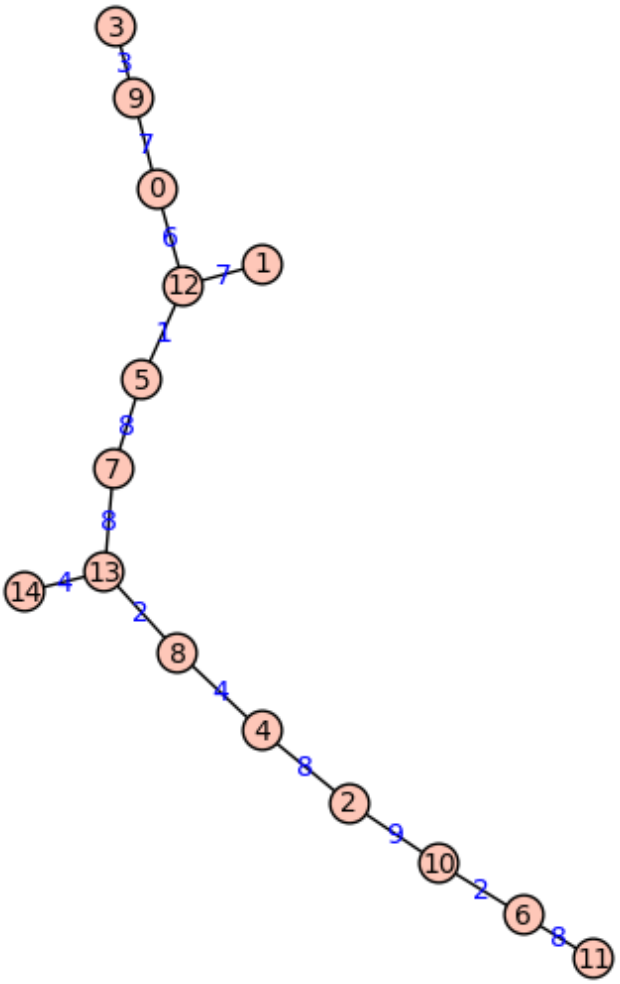

```
#b)
lisAdya = grafo.min_spanning_tree(algorithm='Kruskal_Boost',starting_vertex=2)
print lisAdya
g = Graph(lisAdya,weighted=True)
g.show(edge_labels=True, figsize=7)
```

```
[(0, 9, 7), (0, 12, 6), (1, 12, 7), (2, 4, 8), (2, 10, 9), (3, 9, 3),
(4, 8, 4), (5, 7, 8), (5, 12, 1), (6, 10, 2), (6, 11, 8), (7, 13, 8),
(8, 13, 2), (13, 14, 4)]
```




```
#c)
lisAdya = grafo.min_spanning_tree(algorithm='Prim_Boost',starting_vertex=2)
print lisAdya
g = Graph(lisAdya,weighted=True)
g.show(edge_labels=True, figsize=7)
```

```
[(0, 9, 7), (0, 12, 6), (1, 12, 7), (2, 4, 8), (2, 10, 9), (3, 9, 3),
(4, 8, 4), (5, 7, 8), (5, 12, 1), (6, 10, 2), (6, 11, 8), (7, 13, 8),
(8, 13, 2), (13, 14, 4)]
```




```
#d)
vInicio = 0
vFin = 12
print "El flujo maximal es: ", grafo.flow(vInicio,vFin,algorithm='FF')
subgrafo = grafo.flow(vInicio,vFin,algorithm='FF',value_only=False)[1]
subgrafo.show(edge_labels=True)
```

```
El flujo maximal es:  6
```