# GIIT_AMA_18_Grafos_Ejercicios

# Práctica de Ampliación de Matemáticas (Tema 2: Teoría de Grafos)

## Alumnos que integran el grupo

1. Alfonso Nguema Ela Nanguan.
2. Carlos Rebolledo Aliseda.
3. David Pozo Nuñez.

## Ejercicios

### PARTE A) (DE LOS TUTORIALES)

TUTORIAL 1

**1. Genera un grafo cuyos vértices sean los números naturales del 1 al 20 y dos vértices sean adyacentes si los números son congruentes módulo 4.**

**2. Crea una función que reciba una lista de números y genere el grafo cuyos vértices son los números de la lista y dos vértices serán adyacentes si la diferencia de los números es 3 o superior.**

**3. Vamos a crear una función que calcule la componente conexa de un vértice. La función recibira un grafo y un vértice v y seguirá el siguiente algoritmo:**

**a) Inicializará una lista L con el vértice v**

**b) Calculará la lista de vértices adyacentes a alguno de L pero que no esté en L**

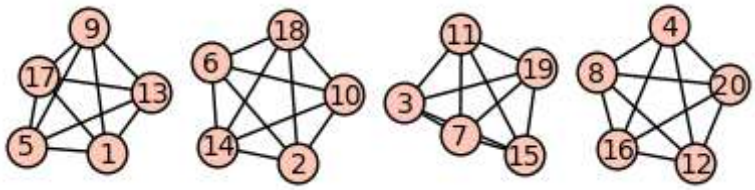
**c) Mientras la lista de vértices adyacentes del apartado anterior no sea vacía, repetirá a) y b)**

### Ejercicio 1:

```
# La funcion que nos genera el grafo
def grafoCondicion(lista,f):
    g=Graph({}) # Generamos un grafo vacío
    for u in lista: # Recorremos la lista
        g.add_vertex(u) # Añadimos sus elementos como vértices
        for v in lista: # Recorremos los pares de elmentos de lista
            if f(u,v)==0: # Si f se anula en un par,
                g.add_edge((u,v)) # añadimos ese par como arista
    return g # Devolvemos el grafo generado
```

```
# La funcion que nos hace la congruencia modulo 4
def f(x,y):
    return (x-y)%4
```

```
# Prueba
g=grafoCondicion([1..20],f)
g.show()
```

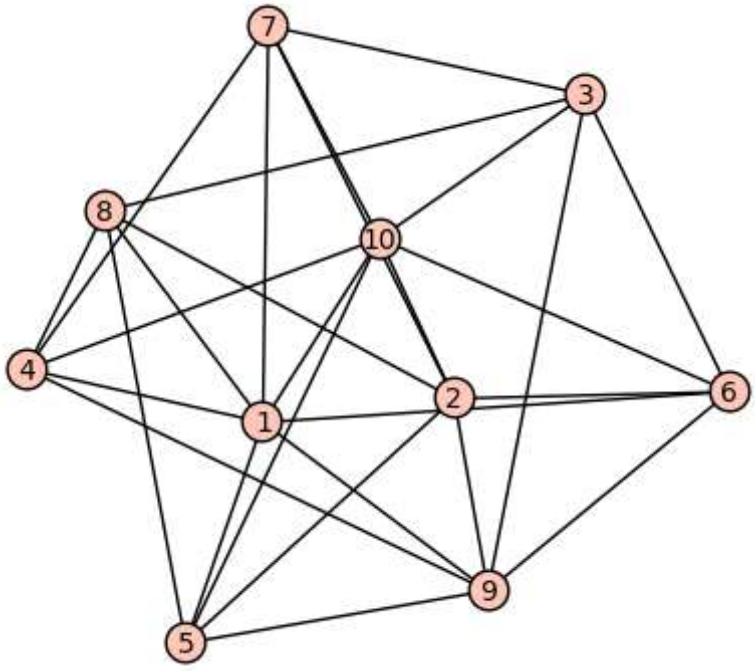



## Ejercicio 2:

```
def grafoCondicion2(lista, f):
    g=Graph({})
    for u in lista:
        g.add_vertex(u)
        for v in lista:
            if f(u,v)>=3: # Hacemos que la resta sea igual o mayor que 3
                g.add_edge((u,v))
    return g
```

```
# La funcion que nos hace la diferencia de los dos numeros
def f(x,y):
    return x-y
```

```
# Prueba
g=grafoCondicion2([1..10],f)
g.show()
```




```

```

## Ejercicio 3:

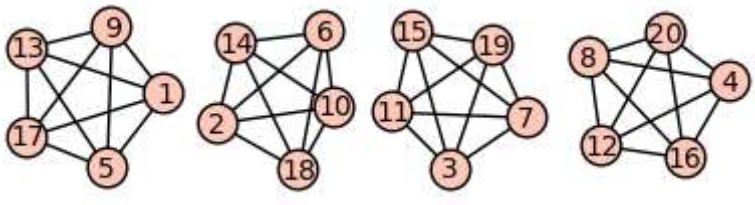
**Para realizar esta tarea, vamos a necesitar varias funciones:**

```
# Es una función que recibe un grafo y una lista de vértices y devuelve la
lista de los
# vértices adyacentes a cualquier vértice de la lista. Los vértices en la
lista devuelta no estén repetidos
def adyacentes(grafo, lista):
    final=[]
    for u in lista:
        for i in g.neighbor_iterator(u):
            final.append(i)
    return list(set(final)) # Hace que no esten repetidos
```

```
# Es una función que recibe dos listas y devuelve la lista obtenida al
borrar en la
# primera lista los elementos de la segunda.
def listaborrar(l1, l2):
    for i in l2:
        if i in l1:
            l1.remove(i)
    return l1
```

```
# Componentes conexas de un vertice
def componenteConexa(grafo, v):
    L = [v]     # Inicializamos la lista L con un vertice dado
    L2 = listaborrar(adyacentes(grafo,L),L)
    while len(L2) > 0:
        for i in L2:
            L.append(i)
        L2 = listaborrar(adyacentes(grafo,L),L)
    return L
```

```
# Visualizamos el grafo que vamos a analizar
show(g)
```




```
# Prueba 1. Sacamos los componentes conexas del vertice 7
componenteConexa(g, 7)
```

```
[7, 3, 11, 19, 15]
```

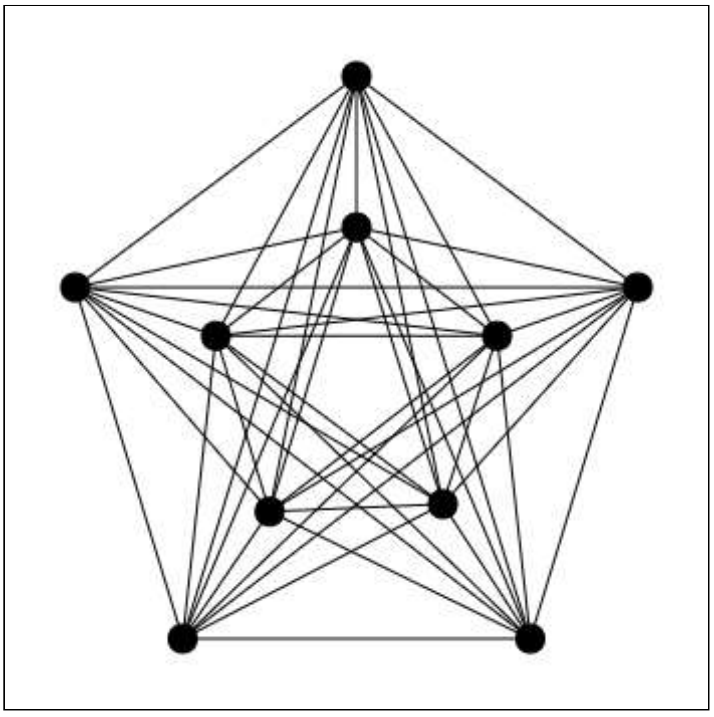
```
# Prueba 2. Sacamos los componentes conexas del vertice 2
componenteConexa(g, 2)
```

```
[2, 10, 18, 14, 6]
```

**TUTORIAL 2**

**1. Representa en una misma gráfica el grafo de Petersen y su complementario (el grafo compuesto por los mismos vértices y las aristas complementarias). Para ello representa el grafo completo, creando una copia del grafo de Petersen (puedes usar *copy* o crear otro nuevo con *graphs*) y añade las aristas que no estén (por ejemplo, recorriendo todas las parejas u,v de vértices del grafo y añadiendo las aristas que no estén). Puedes usar una nueva etiqueta para las aristas que añadas y así simplificas dibujarlas. Dibuja en un color las aristas que no estén en el grafo de Petersen y en otro las que estén.**

```
# Como hay que pintar en el mismo los dos grafos, el complementario y
# el de Petersen, todas las aristas entarán conectadas entre sí.
f = Graph({0:[1,2,3,4,5,6,7,8,9],1:[0,2,3,4,5,6,7,8,9],2:
[0,1,3,4,5,6,7,8,9],3:[0,1,2,4,5,6,7,8,9],4:[0,1,2,3,5,6,7,8,9],5:
[0,1,2,3,4,8,9,6,7],6:[0,1,2,3,4,8,9,7,5],7:[0,1,2,3,4,9,6,8,5],8:
[0,1,2,3,4,5,6,7,9],9:[0,1,2,3,4,6,7,8,5]}); f.set_pos({0:[175,315],1:
[35,209.92369934293058],2:[88.47524157501469,35.00000000000006],3:
[261.5247584249852,35],4:[315,209.92369934293052],5:
[175,239.80423445342365],6:[105,185.7498386375243],7:
[131.7376207875073,98.28798896605909],8:[218,102],9:
[245,185.7498386375243]});
graph_editor(f)
```

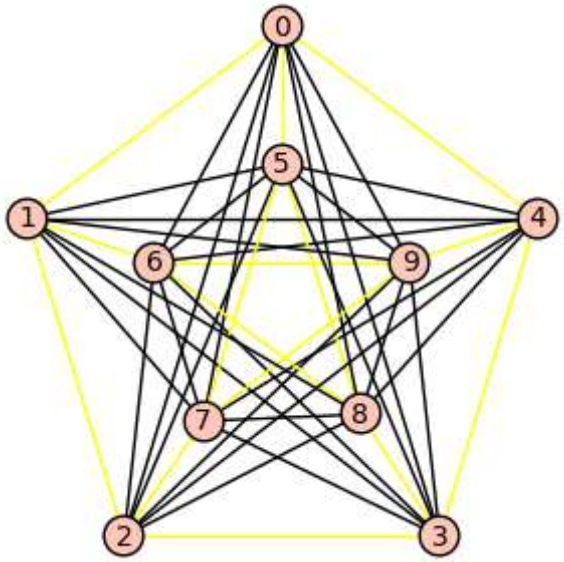



```
# Hemos selecionado a las aristas que corresponden al grafo Pertesen y
# hemos cambiado el color,(las aristas las va cogiendo de una en una,
# es decir, la arista (0,1)=0 (0,2)=1
P = f
E=P.edges();
e={'#FFFF00':[E[k] for k in [0,3,4,9,13,17,21,24,28,34,36,37,40,41,43]]}
e
```

```
{'#FFFF00': [(0, 1, None),
  (0, 4, None),
  (0, 5, None),
  (1, 2, None),
```

```
        (1, 6, None),
        (2, 3, None),
        (2, 7, None),
        (3, 4, None),
        (3, 8, None),
        (4, 9, None),
        (5, 7, None),
        (5, 8, None),
        (6, 8, None),
        (6, 9, None),
        (7, 9, None)]}
```

```
# Mostramos el grafo incluyendo los colores de las aristas
P.plot(edge_colors=e).show(figsize=4)
```




## PARTE B) EJEMPLOS

Elegid dos apartados, el apartado correspondiente a la última cifra de la suma de vuestros DNIs, y el apartado correspondiente a la última cifra de la suma de vuestros DNIs más cinco. Obtener para cada apartado un grafo que cumpla las propiedades pedidas. Debéis incluir el grafo y explicar por qué cumple las propiedades. En caso de que penséis que no existe dicho grafo, justificar porqué.

0. Hamiltoniano con un conjunto independiente formado por 4 vértices.

1. Conexo, 4-regular pero no hamiltoniano.

2. Euleriano con complemento 3-regular.

3. 4 regular, euleriano y no hamiltoniano.

4. No euleriano con secuencia de grados 4,2,2,2,2,2,2,2

5. 4 regular, pero ni plano ni hamiltoniano.

6. 4-regular, no completo y con radio y diámetro igual a 4.

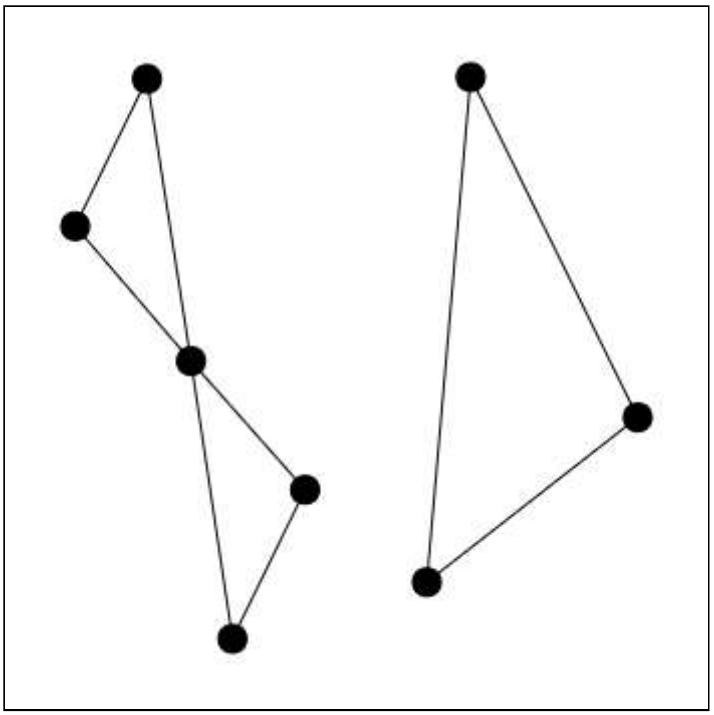
7. Plano, 5-regular.

8. Hamiltoniano, bipartito, 3-regular

9 Plano, bipartito, Euleriano y con al menos un vértice de grado 6.

## 4.

```
# Es un grafo con secuencia de grados [4,2,2,2,2,2,2,2] que no tiene un
# circuito que recorra todas las aristas, ya que no es conexa. Por lo tanto
no es euleriano.
grafo1 = Graph({0:[1,6],1:[0,6],2:[5,7],3:[4,5],4:[3,5],5:[2,3,4,7],6:
[0,1],7:[2,5]});
graph_editor(grafo1)
```

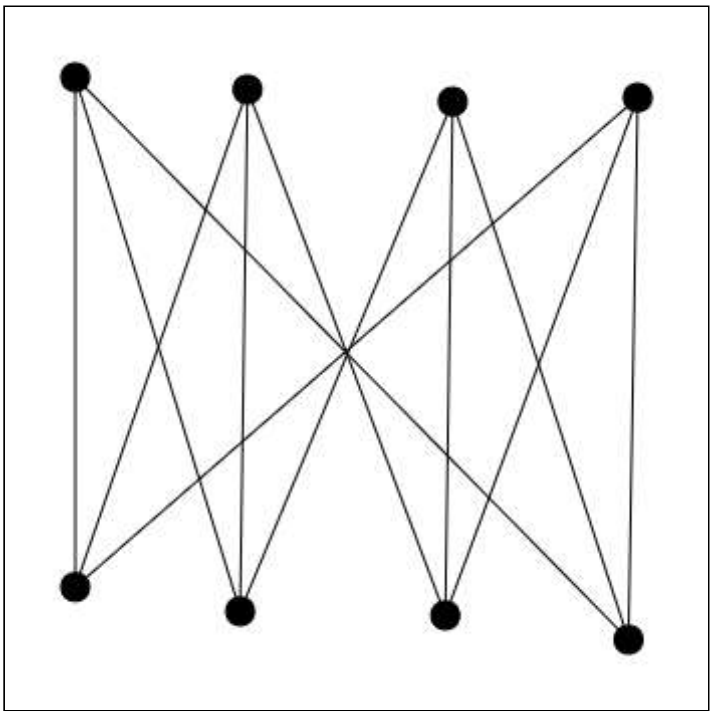



```
# Prueba: Comprobamos que no es euleriano.
grafo1.is_eulerian()
```

```
False
```

## 8.

```
# Hamiltoniano, bipartito, 3-regular
grafo = Graph({0:[1,3,7],1:[0,2,6],2:[1,3,5],3:[0,2,4],4:[3,5,7],5:
[2,4,6],6:[1,5,7],7:[0,4,6]}); grafo.set_pos({0:[35,315],1:
[35,61.1870503597122],2:[120.73289902280132,308.9568345323741],3:
[117.08469055374593,49.10071942446041],4:
[222.88273615635177,302.9136690647482],5:
[219.2345276872964,47.086330935251794],6:[315,304.9280575539568],7:
[310.43973941368074,35]});  graph_editor(grafo);
```




Save Close

```
# Prueba 1. Probamos que es Hamiltoniano
grafo.is_hamiltonian()
```

```
True
```

```
# Prueba 2. Probamos que es bipartito
grafo.is_bipartite()
```

```
True
```

## PARTE C) ALGORITMOS

En los siguientes ejercicios se tratará de codificar funciones sencillas de teoría de grafos, para repasar los conceptos y algoritmos. Sage tiene predefinidas montones de funciones que hacen todas esas operaciones, pero el objetivo es que vosotros aprendáis cómo se trabaja con grafos, por lo que en la resolución deberéis usar únicamente lo que se ha explicado en las prácticas. Las funciones tienen que estar convenientemente explicadas, con un comentario al comienzo de la función como líneas de comentario en cada operación que se considere compleja (en particular en los bucles y llamadas recursivas).

Crea varios grafos de prueba, codifica la función descritas en el apartado correspondiente al resto de dividir entre cinco la suma de los DNI's (hay dos listas según el grado al que pertenezcan los miembros del grupo) y pruébala. Se recomienda crear funciones intermedias para los pasos del algoritmo.

0. Crea una función que reciba un grafo y devuelva un circuito euleriano (en caso de que no exista, devolverá un grafo vacío).

1. Crea una función que reciba un grafo etiquetado y dos vértices y devuelva el flujo máximo entre los dos vértices. (Para calcular un camino entre dos vértices podéis usar algún método de Sage.)

2. Crea una función que reciba un grafo etiquetado (con etiquetas positivas) y dos vértices, a,b, y devuelva

a: la distancia mínima entre a y b.

b: el camino óptimo entre a y b.

3. Crea una función que reciba un grafo y devuelva el árbol recubridor minimal por el algoritmo de Kruskal. Se valorará además que el algoritmo devuelva los pasos intermedios mediante un grafo con etiquetas diferentes para las aristas visitadas y descartadas.

4. Crea una función que reciba un grafo y devuelva el árbol recubridor minimal por el algoritmo de Prim. Se valorará además que el algoritmo devuelva los pasos intermedios mediante un grafo con etiquetas diferentes para las aristas visitadas y descartadas.

# Normas de entrega

1. Rellenar en la parte superior el nombre de los integrantes del grupo.
2. Compartir esta hoja de Sage con el profesor (etlopez18).
3. En el desplegable "File" de la parte superior de la página, elegir "Print"
4. Imprimir la página resultante como pdf.
5. Subirla al campus virtual antes de la fecha límite, junto con la memoria de planificación y trabajo en equipo. Cada día a partir de la fecha límite, la nota sobre la que se puntúa el ejercicio bajará en un punto.