UNIVERSIDAD NACIONAL DE COLOMBIA SEDE BOGOTA
ANALISIS ESTADISTICO DE REDES SOCIALES.
Taller 1: Conceptos y Gestión.

Cesar A Prieto Sarmiento - CC: 1065843742
Alejandro Urrrego Lopez - CC:

25 de febrero de 2024

El siguiente documento contiene la solución propuesta desarrollada en Python a través del paquete igraph, para los ejercicios del taller 1 de la clase de Análisis Estadístico de Redes Sociales, junto con este documento se encuentra el cuaderno de python (archivo .ipynb) que contiene el código con la solución a los ejercicios.

PUNTO 1

Reproducir los ejemplos 3.1, 3.2, 3.3, 4.2 y 4.4 de [este documento](https://rpubs.com/jstats1702/931287) en Python utilizando igraph y/o NetworkX.

Ejemplo 3.1 Este ejemplo contiene la creación y explotacion de una red no dirigida desde 0 y a su vez la creación del gráfico de la red:

```
    # Creamos un grafo no dirigido usando igraph
    g = ig.Graph()
    # Añadimos nodos y aristas
    g.add_vertices(7)
    edges = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 4), (3, 4), (3, 5),
            (3, 6), (4, 5), (5, 6)]
    g.add_edges(edges)

    # Mostramos información sobre el grafo
    print("Clase del grafo:", type(g))
    print("ID del grafo:", id(g))
```

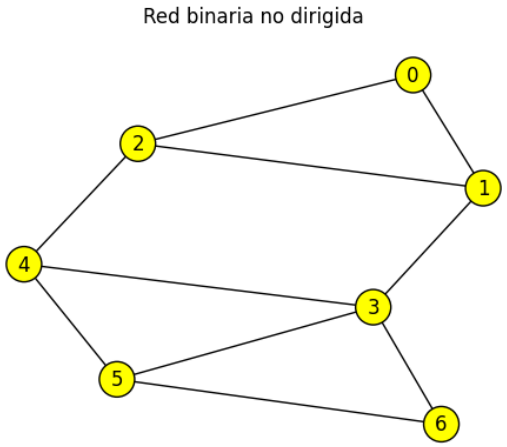
```
Clase del grafo: <class 'igraph.Graph'>
ID del grafo: 2519818251344
```

```
    print(g)
```

```
IGRAPH U--- 7 10 --
+ edges:
0 -- 1 2       2 -- 0 1 4     4 -- 2 3 5     6 -- 3 5
1 -- 0 2 3     3 -- 1 4 5 6   5 -- 3 4 6
```

La exploración del grafo nos arrojó la información conocida de la red, luego de esto procedimos a graficarla:

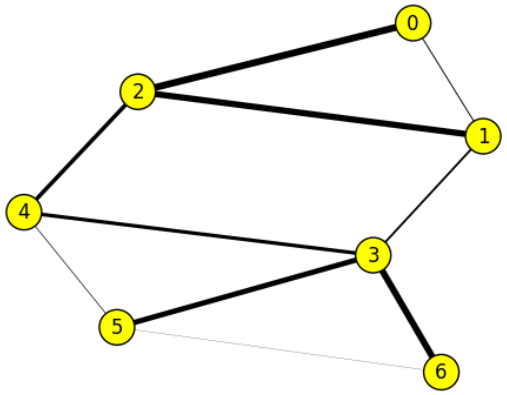
```
    # Gráfica de la red binaria no dirigida
    layout = g.layout("kamada_kawai")
    fig, ax = plt.subplots()

    ig.plot(g, target=ax, layout=layout, vertex_color="yellow", vertex_size=30,
    vertex_label_size=12, vertex_label_dist = 0, edge_color="black", edge_width=1, v
    ertex_label=g.vs.indices)
    plt.title("Red binaria no dirigida")
    plt.show()
```



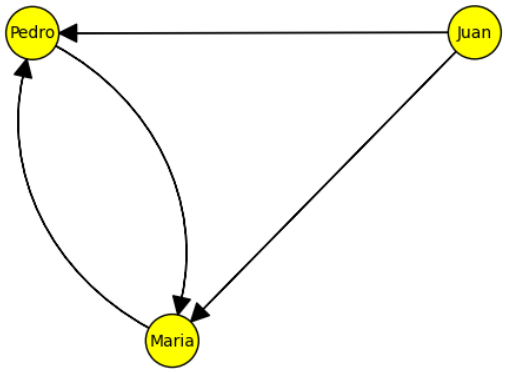
Ejemplo 3.2 Este ejemplo al igual que el anterior contiene la creacion y exploracion de una red ponderada y no dirigida desde 0 y a su vez la creación del gráfico de la red:

```
    wg.es["weight"] = [round(random.uniform(0, 1), 3) for _ in range(wg.ecount())]
    # pesos
    print(wg.es["weight"])
[0.134, 0.847, 0.764, 0.255, 0.495, 0.449, 0.652, 0.789, 0.094, 0.028]

    # ponderada?
    print("¿El grafo ponderado?", wg.is_weighted())
¿El grafo ponderado? True
```

Ahora el grafico de la red ponderado se veria así:



Ejemplo 3.3 Este ejemplo plantea la creación de una red binaria a la cual se le agregan atributos a los vértices:



Ejemplo 4.2 Este ejemplo utiliza la red del ejemplo 3.1 para explorar la matrix de adyacencia y verificar sus propiedades

```
    # matriz de adyacencia
    A = g.get_adjacency()
    # clase de objeto
    print("Clase de objeto:", type(A))
    # imprime la matriz de adyacencia
    print("Matriz de adyacencia:")
    print(A)
```

```
Clase de objeto: <class 'igraph.datatypes.Matrix'>
Matriz de adyacencia:
[[0, 1, 1, 0, 0, 0, 0]
 [1, 0, 1, 1, 0, 0, 0]
 [1, 1, 0, 0, 1, 0, 0]
 [0, 1, 0, 0, 1, 1, 1]
 [0, 0, 1, 1, 0, 1, 0]
 [0, 0, 0, 1, 1, 0, 1]
 [0, 0, 0, 1, 0, 1, 0]]
```

Por otra parte, tendríamos que convertir esta matriz en formato numpy.ndarray para poder tratarla y/o transformarla y hacer operaciones matriciales con ella.

```
    # formato 'matrix array'
    Y = np.array(g.get_adjacency().data)
    print("¿Es simétrica?", np.array_equal(Y, Y.T))


    # imprime la matriz de adyacencia en formato 'matrix array'
    print("Matriz de adyacencia:")
    print(Y)
```

```
Clase de objeto: <class 'numpy.ndarray'>
¿Es simétrica? True
Matriz de adyacencia:
```

```
[[0 1 1 0 0 0 0]
 [1 0 1 1 0 0 0]
 [1 1 0 0 1 0 0]
 [0 1 0 0 1 1 1]
 [0 0 1 1 0 1 0]
 [0 0 0 1 1 0 1]
 [0 0 0 1 0 1 0]]
```

Así mismo podemos obtener los arreglos donde se encuentra la relación entre los 2 nodos de modo que obtendríamos lo siguiente:

```
    # versión vectorizada exhaustiva
    yvec1 = Y[np.tril_indices(Y.shape[0], k=-1)]
    print("Versión vectorizada exhaustiva:", yvec1)
```

```
Versión vectorizada exhaustiva: [1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 0 1]
```

```
     #versión vectorizada indexada
    yvec2 = np.where(yvec1 == 1)
    print("Versión vectorizada indexada:", yvec2)
```

```
Versión vectorizada indexada: (array([ 0,  1,  2,  4,  8,  9, 13, 14, 18, 20],
    dtype=int64),)
```

Ejemplo 4.4 Este ejemplo utiliza la red del ejemplo 3.1 para seguir explorando las diferentes opciones que tenemos para visualizar arreglos de las relaciones de la red

```
    n = Y.shape[0]
    A = []

    for i in range(n - 1):
        for j in range(i + 1, n):
            if Y[i, j] == 1:
                A.append([i, j])
    A = np.array(A)

    print("Clase de objeto:", type(A))
    print(A)
```

```
Clase de objeto: <class 'numpy.ndarray'>
[[0 1]
 [0 2]
 [1 2]
 [1 3]
 [2 4]
 [3 4]
 [3 5]
 [3 6]
 [4 5]
 [5 6]]
```

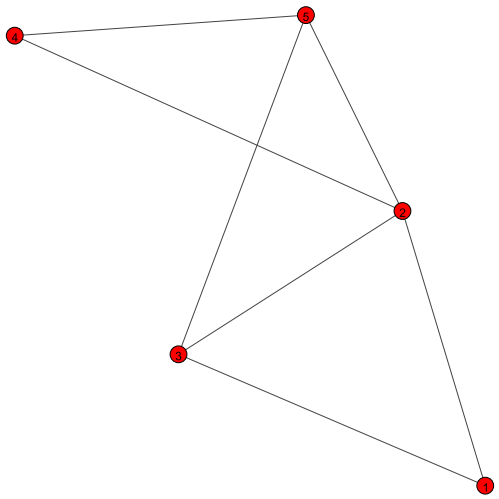
PUNTO 2

Grafo $G = (V, E)$ Considere el grafo $G = (V, E)$, con:

$$V = \{1, 2, 3, 4, 5\} \qquad E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{4, 5\}\}$$

- Visualizar G

```
g = ig.Graph()
g.add_vertices(5) # Indicamos cuántos vértices tiene
g.add_edges([(0,1),(0,2),(1,2),(1,3),(1,4),(2,4), (3,4)]) #
g.vs["label"] = [1,2,3,4,5]
ig.plot(g)
```



- Calcular orden, tamaño y diámetro del grafo.

  El orden de la red se representa por $|V|$, el diámetro de la red se representa por diam(G) y el tamaño de la red se representa por $|E|$.

```
print('El orden del grafo es: ' + str(g.vcount()) +
        ' El Diametro del grafo es: ' + str(g.diameter()) +
        '\nEl tamaño del grafo es: ' + str(g.ecount()))
```

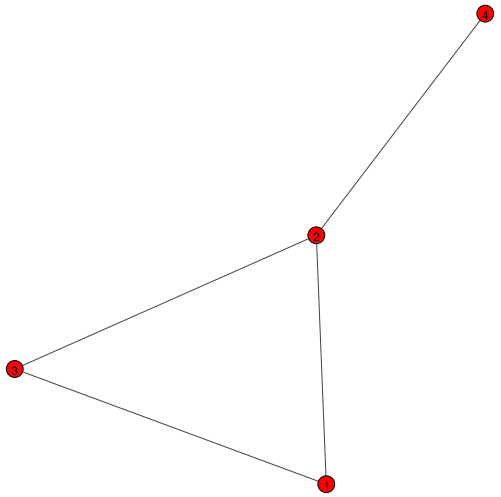
```
El orden del grafo es: 5
El Diámetro del grafo es: 2
El tamaño del grafo es: 7
```

- Calcular el grado de cada vértice

```
print('El grado de cada vertice es :' + str( g.degree()) +
    ' El indegree es :' +  str( g.indegree()) +
    '\nEl out degree es :' + str(g.outdegree()))
```

```
El grado de cada vertice es :[2, 4, 3, 2, 3]
El indegree es :[2, 4, 3, 2, 3]
El out degree es :[2, 4, 3, 2, 3]
```

- Graficar el subgrafo generado por los nodos 1, 2, 3 y 4



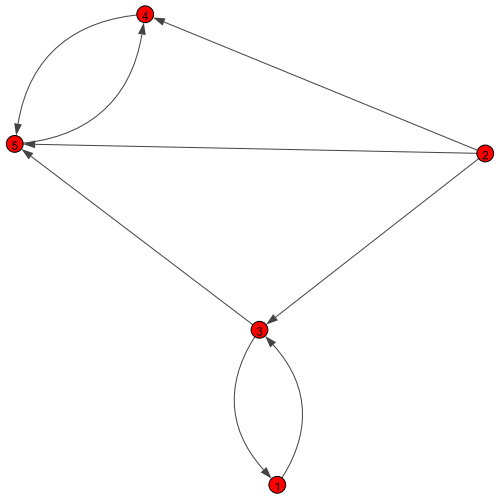
PUNTO 3

Digrafo $G = (V, E)$ Considere el digrafo $G = (V, E)$, con:

$$V = \{1, 2, 3, 4, 5\} \qquad E = \{(1, 3), (2, 3), (2, 4), (2, 5), (3, 1), (3, 5), (4, 5), (5, 4)\}$$

- Visualizar G

```
g = ig.Graph(directed=True)
g.add_vertices(5) # Indicamos cuantos vertices tiene
g.add_edges([(0,2),(1,2),(1,3),(1,4),(2,0),(2,4),(3,4),(4,3)])
g.vs["label"] = [1,2,3,4,5]
ig.plot(g)
```



- Calcular orden, tamaño y diámetro del grafo

```
print('El orden del grafo es: ' + str(g.vcount()) +
    ' El Diametro del grafo es: ' + str(g.diameter()) +
    '\nEl tamaño del grafo es: ' + str(g.ecount()))
```

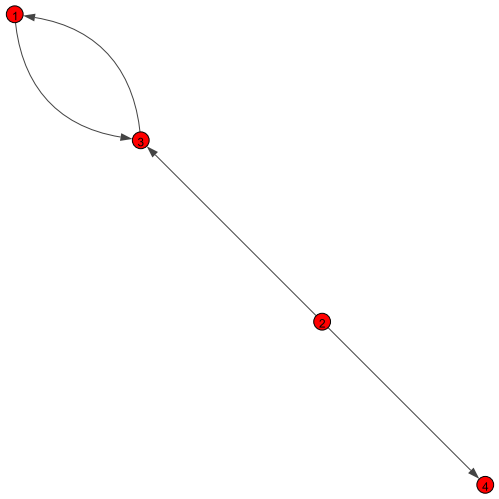
```
El orden del grafo es: 5
El Diametro del grafo es: 3
El tamaño del grafo es: 8
```

- Calcular el grado de cada vértice del grafo

```
print('El grado de cada vertice es :' + str( g.degree()) +
    ' El indegree es :' +  str( g.indegree()) +
    '\nEl out degree es :' + str(g.outdegree()))
```

```
aEl grado de cada vertice es :[2, 3, 4, 3, 4]
El indegree es :[1, 0, 2, 2, 3]
El out degree es :[1, 3, 2, 1, 1]
```

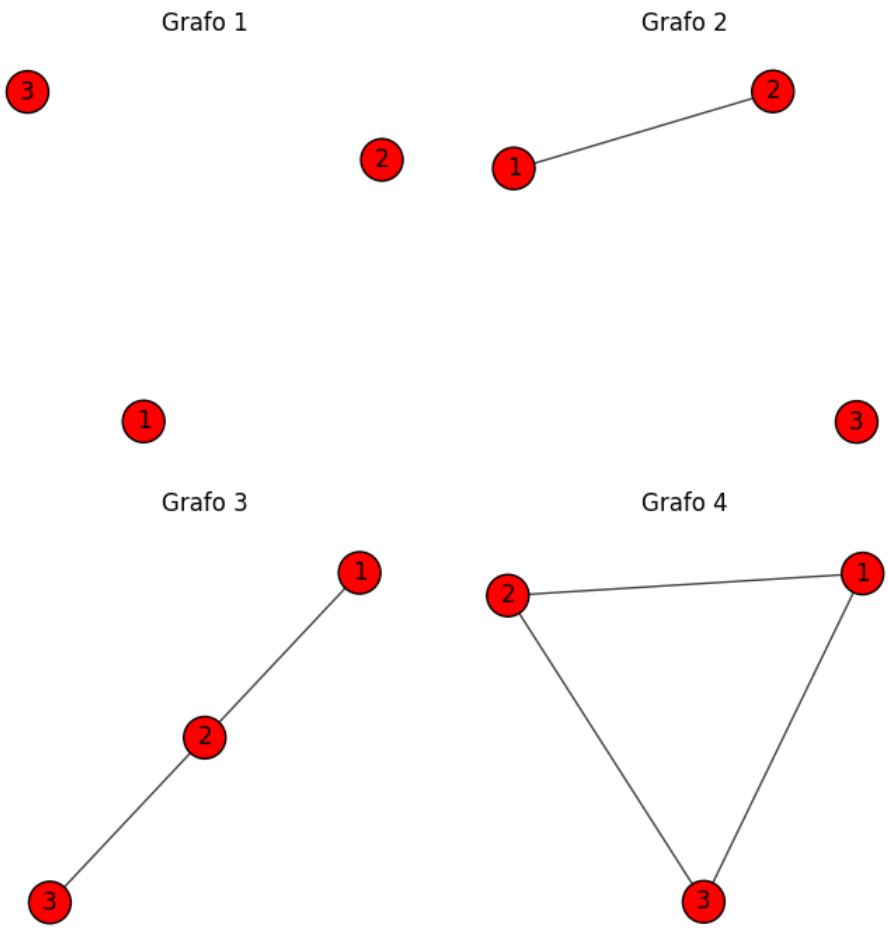
- Graficar el subgrafo generado por los nodos 1, 2, 3 y 4



PUNTO 4

Estados triádicos no dirigidos

- Graficar todos los posibles estados triádicos no dirigidos.



- Identificar los estados isomorfos.

  En términos matemáticos, para verificar si dos estados A y B son isomorfos, necesitaríamos encontrar una biyección f entre los vértices de los grafos asociados a los estados A y B respectivamente, tal que para cada par de vértices u, v en A, existe una arista entre u y v si y solo si hay una arista entre $f(u)$ y $f(v)$ en B.

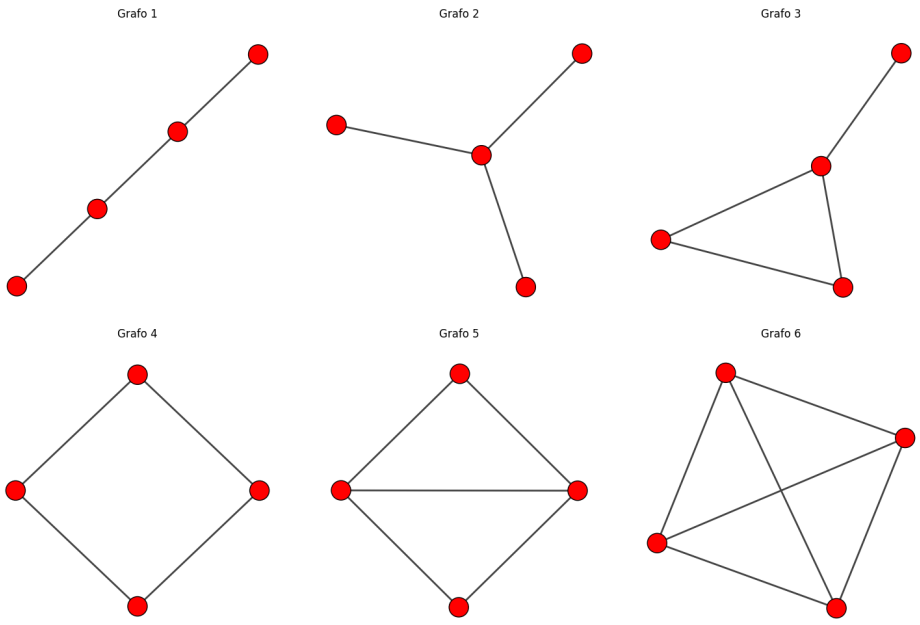
  Para este punto creamos la siguiente función en python que nos ayudaba a identificar los estados isomorfos en los grafos anteriores de modo que la salida nos indicará cuáles estados son isomorfos

```
# Lista para almacenar los estados isomorfos
isomorphic_states = []

# Verificar isomorfismo entre todos los pares de grafos
for i in range(len([g0, g, g1, g2])):
    for j in range(i+1, len([g0, g, g1, g2])):
        if [g0, g, g1, g2][i].isomorphic([g0, g, g1, g2][j]):
            isomorphic_states.append((i+1, j+1))

if isomorphic_states:
    print("Los estados isomorfos son:", isomorphic_states)
else:
    print("No hay estados isomorfos.")

plt.show()
```

```
No hay estados isomorfos.
```

PUNTO 5

Visualizar todos los grafos (no dirigidos) conectados con 4 vértices.



Como se puede visualizar en la gráfica anterior, solo encontramos 6 grafos no dirigidos y conectados con 4 vértices

PUNTO 6

Reconstrucción de la matriz de adyacencia

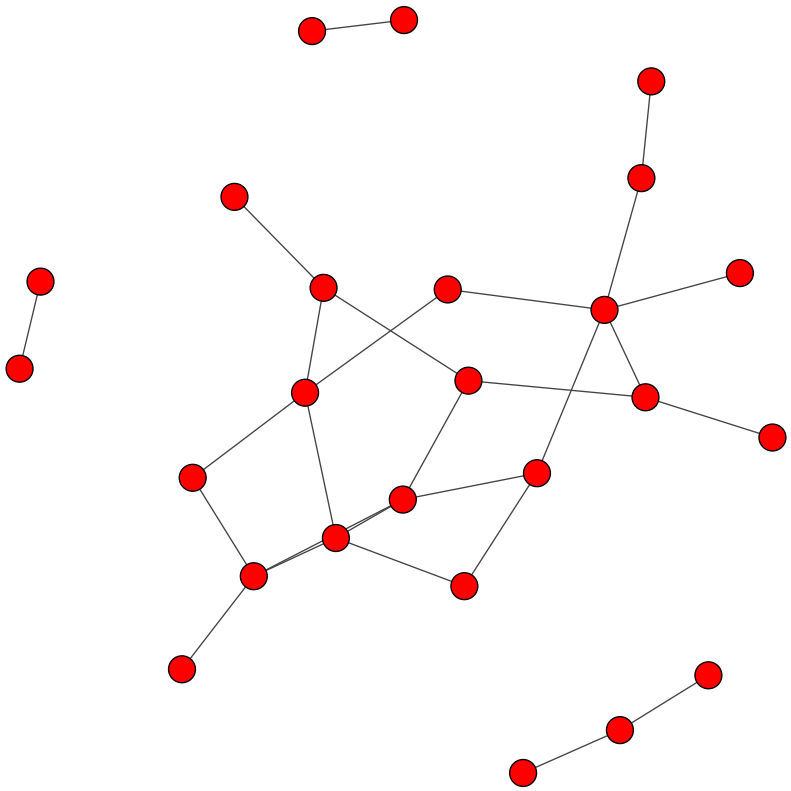
- Escribir una rutina que reconstruya la matriz de adyacencia a partir de la matriz de aristas y una lista de vértices.

  Ver codigo punto 6A

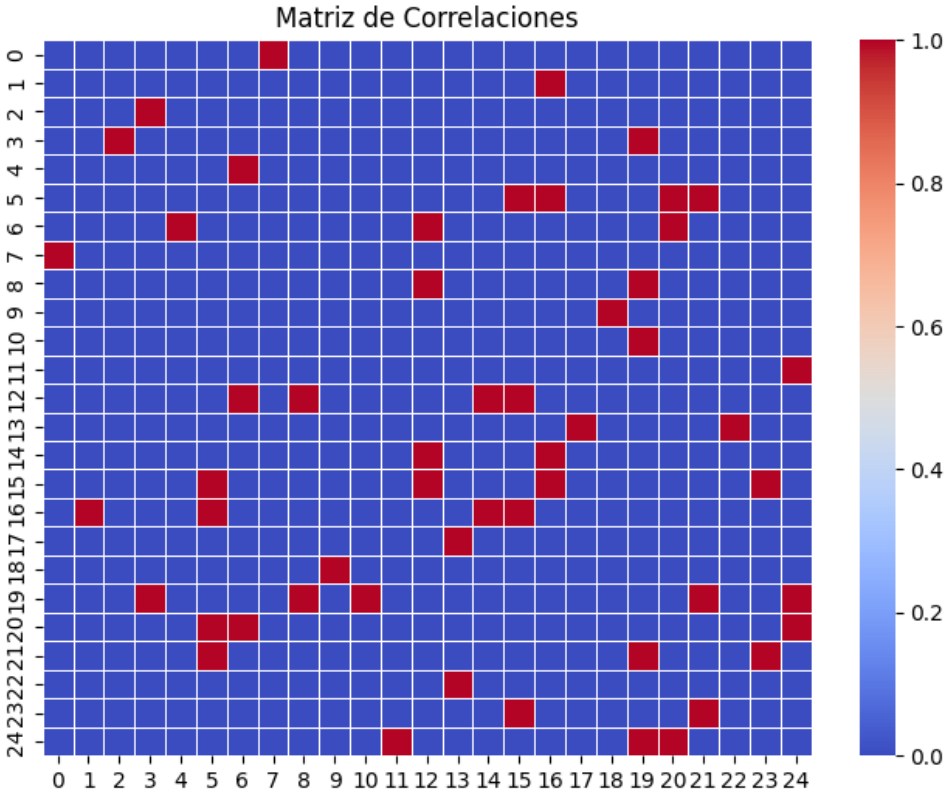
- Simular una red no dirigida de 25 nodos generada a partir de enlaces aleatorios con probabilidad de éxito 0.1 y probar la rutina con la red simulada.

  Ver código punto 6B

- Visualizar la red simulada mediante un grafo y una socio-matriz.



El anterior es el gráfico de la red generada con la función creada, luego el siguiente sería la socio matriz en forma de mapa de calor.



Como podemos ver el gráfico anterior, es un mapa de calor de la socio matriz de la red generada, donde los cuadros rojos representan los vértices que presentan una relación y los azules la ausencia de relación.

PUNTO 7

Escribir una rutina que reconstruya la matriz de aristas y la lista de vértices a partir de la matriz de adyacencia.

Para la rutina creada ver codigo 7A

- Simular una red no dirigida de 25 nodos generada a partir de enlaces aleatorios independientes e idénticamente distribuidos con probabilidad de éxito 0.1

```
#Simular una red no dirigida de 25 nodos (...)

Matriz=redA(n=25,Aristas=False,p=0.1)
```

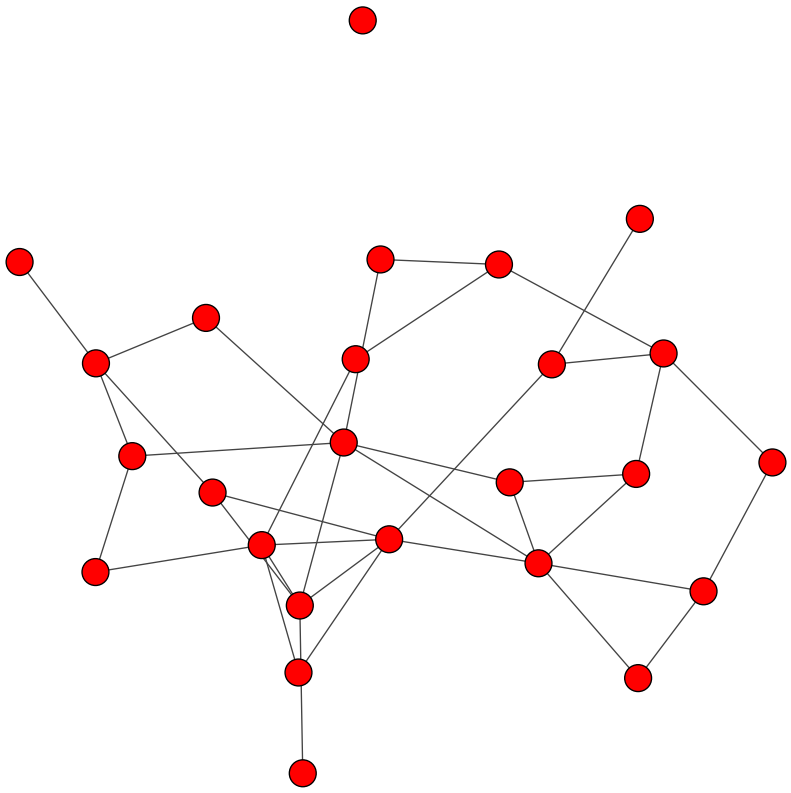
- Probar la rutina con la red simulada.

  Lo siguiente es solo una parte de la salida de la función original, para ver la salida completa **ver Codigo 7B**

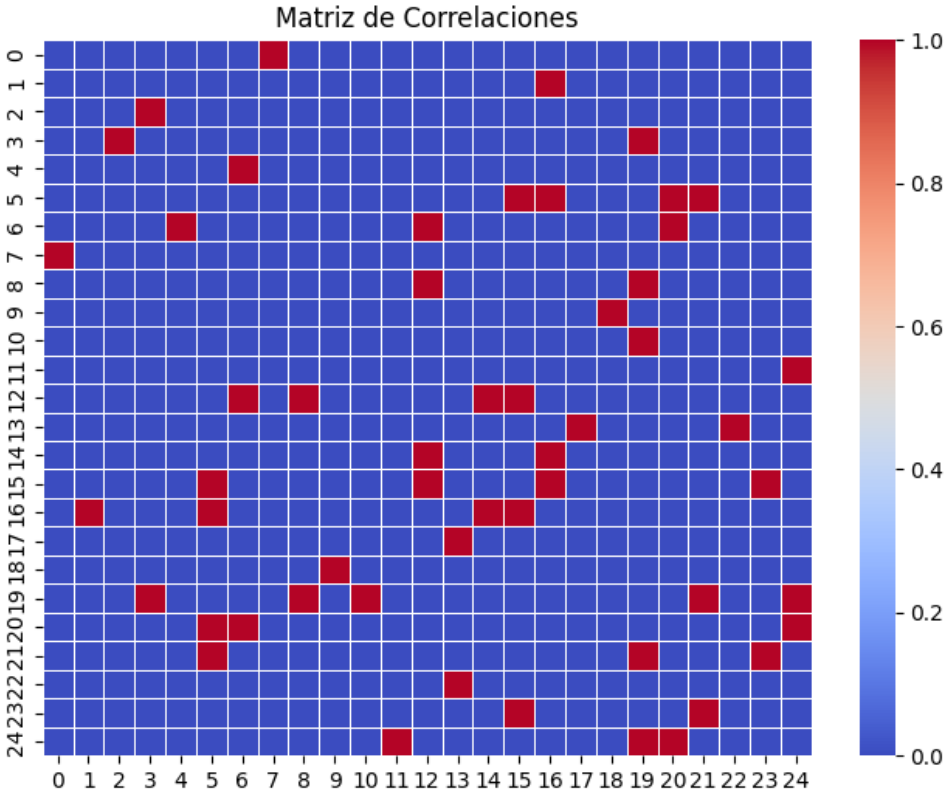
```
print(ArVe(Matriz)[1])
print(ArVe(Matriz)[0])
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ... ,24]
[[ 0  6]
 [ 1 13]
 [ 1 22]
 [ 3  9]
  (...)
 [22 23]]
```

- Visualizar la red simulada por medio de un grafo y una socio-matriz.



El anterior es el gráfico de la red generada con la función creada, luego el siguiente sería la socio matriz en forma de mapa de calor.



Como podemos ver el gráfico anterior, es un mapa de calor de la socio matriz de la red generada, donde los cuadros rojos representan los vértices que presentan una relación y los azules la ausencia de relación.

PUNTO 8

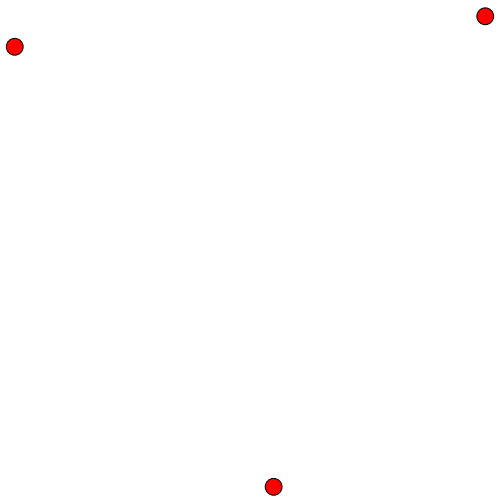
Escribir una rutina que simule redes tanto dirigidas como no dirigidas a partir de enlaces aleatorios independientes e idénticamente distribuidos con una probabilidad de éxito dada. Esta rutina debe tener como argumentos el orden de la red, la probabilidad de interacción (por defecto 0.5), el tipo de red (por defecto como no dirigida) y la semilla (por defecto 123), y además, tener como retorno una versión vectorizada de la matriz de adyacencia y una visualización. Probar esta rutina generando cuatro casos diferentes.

Para ver la rutina creada ver el codigo 8A

```
    funcion(n=3)[0]
```

```
array([0, 0, 0])
```

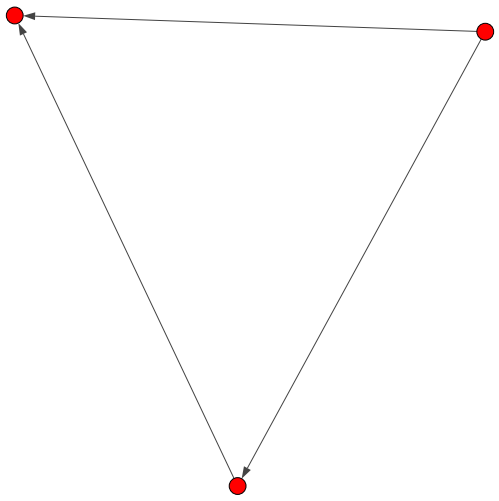
```
    funcion(n=3)[1]
```



```
    funcion(n=3,dirigida=True)[0]
```

```
array([0, 0, 0, 1, 0, 0, 1, 1, 0])
```

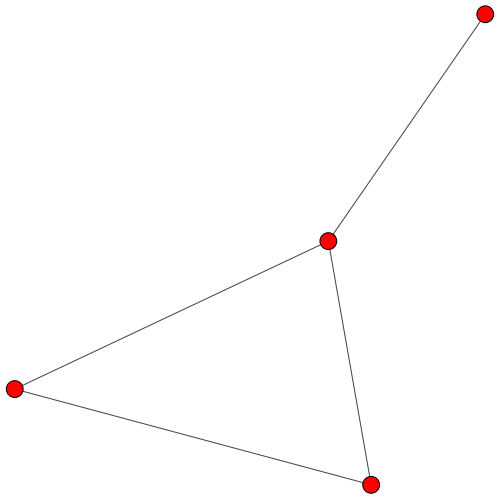
```
    funcion(n=3,dirigida=True)[1]
```



```
    funcion(n=4)[0]
```

```
array([0, 0, 1, 1, 1, 1])
```

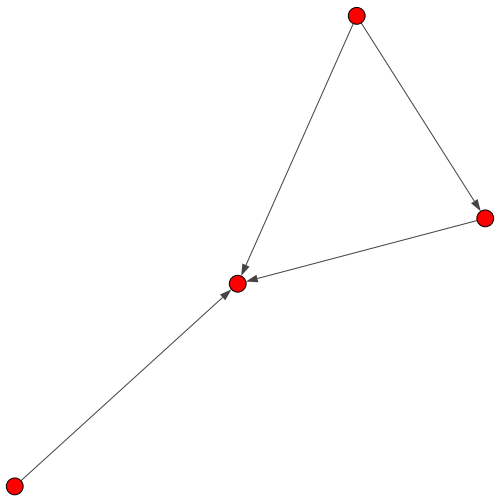
```
    funcion(n=4)[1]
```



```
    funcion(n=4,dirigida=True,semilla=1)[0]
```

```
array([0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0])
```

```
    funcion(n=4,dirigida=True,semilla=1)[1]
```



PUNTO 9

Considere el conjunto de datos dado en el archivo *addhealth.RData* disponible en la página web del curso. Estos datos fueron recopilados por *The National Longitudinal Study of Adolescent Health* y están asociados con un estudio escolar sobre salud y comportamientos sociales de adolescentes de varias escuelas en los Estados Unidos. Los participantes nominaron hasta 5 niños y 5 niñas como amigos y reportaron el número de actividades extracurriculares en las que participaron juntos.

El archivo *addhealth.RData* contiene una lista con dos arreglos, X y E. X tiene tres campos: *female* (0 = No, 1 = Sí), *race* (1 = Blanco, 2 = Negro, 3 = Hispano, 4 = Otro), y *grade* (grado del estudiante). E también tiene tres campos: *V1* (vértice de salida), *V2* (vértice de llegada), y *activities* (número de actividades extracurriculares).

- Identificar y clasificar las variables nodales

```
# Variables nodales
variables_nodales = X.columns.tolist()
print("Las variables nodales son: \n", variables_nodales)
```

```
Las variables nodales son:
['Unnamed: 0', 'female', 'race', 'grade']
```

- Identificar y clasificar las variables relacionales

```
# Variables relacionales
variables_relacionales = E.columns.tolist()
print("Las variables relacionales son: \n", variables_relacionales)
```

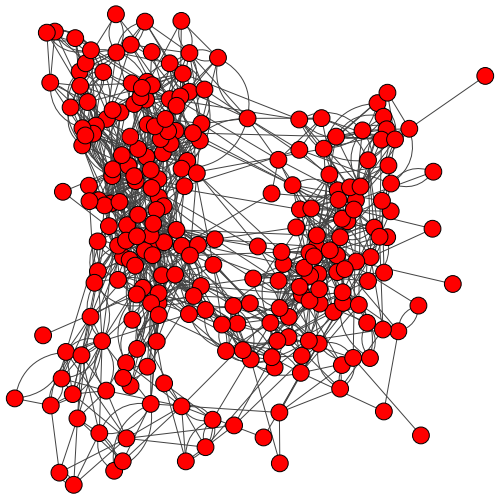
```
Las variables relacionales son:
['Unnamed: 0', 'V1', 'V2', 'activities']
```

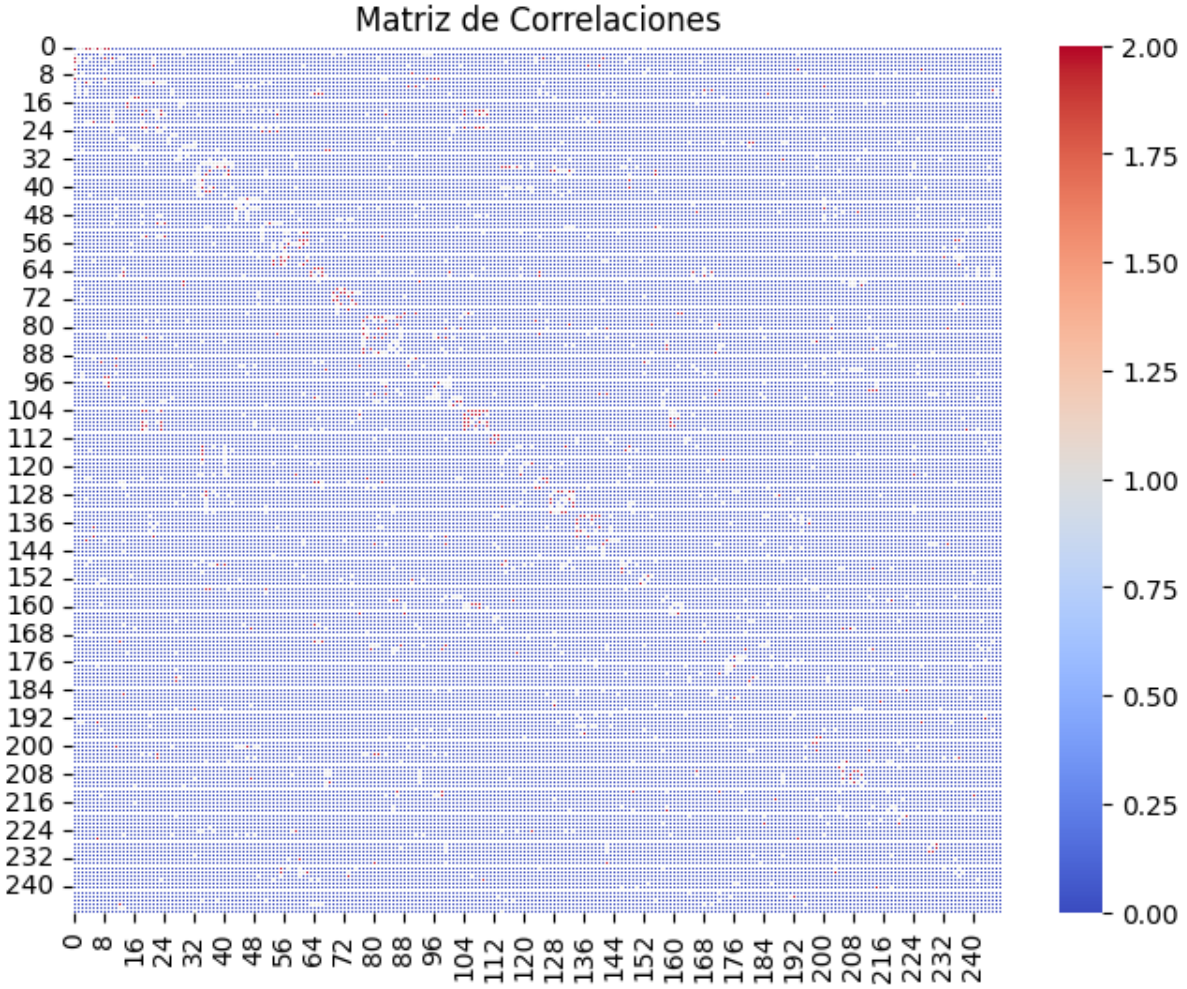
- Calcular el orden, el tamaño y el diámetro de la red

```
print('El orden del grafo es: ' + str(g.vcount()) +
' El Diametro del grafo es: ' + str(g.diameter()) + '\n'+
' El tamaño del grafo es: ' + str(g.ecount()))
```

```
El orden del grafo es: 248
El Diámetro del grafo es: 7
El tamaño del grafo es: 1264
```

- Visualizar la red sin tener en cuenta las variables nodales por medio de un grafo y una socio-matriz





- Identificar el top 5 de los nodos más propensos a emitir/recibir relaciones

```
# Calcular los nodos
nodos = np.unique(A.flatten())
```

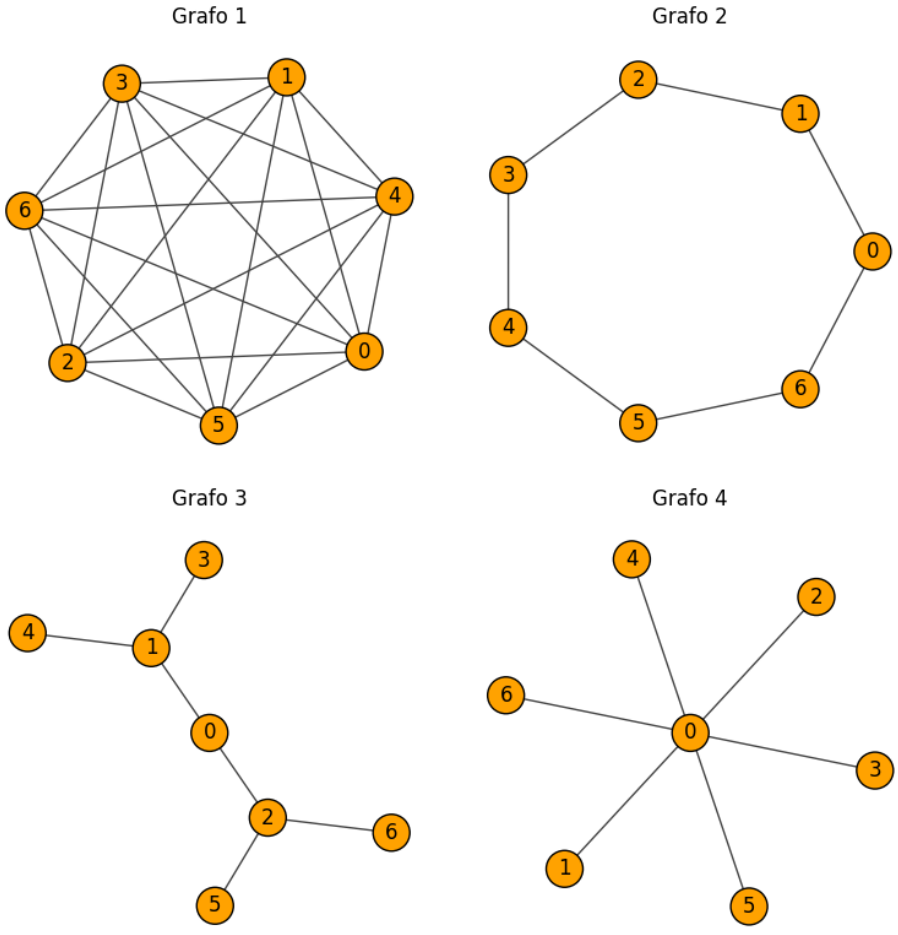
```
# Calcular la centralidad de grado
centralidad_grado = g.degree()

# Obtener los índices de los nodos con mayor centralidad de grado
top_indices = np.argsort(centralidad_grado)[-5:][::-1]

# Obtener los nombres de los nodos correspondientes a los índices
top_nodos = [nodos[idx] for idx in top_indices]

print("Top 5 nodos más propensos a emitir/recibir relaciones:", top_nodos)
```

```
Top 5 nodos más propensos a emitir/recibir relaciones: [38, 21, 26, 86, 89]
```

PUNTO 10

Sintetizar y replicar la Sección 2.4.2 (*Special Types of Graphs*, p. 24) de *Kolaczyk and Csárdi* (2020).

Special Types of Graphs
Graphs come in all shapes and sizes, as it were, but there are a number of families of graphs that are commonly encountered in practice. We illustrate this notion wit the examples of four such families shown in Fig. 2.2.



A complete graph is a graph where every vertex is joined to every other vertex by an edge. This

concept is perhaps most useful i practice through its role in defining clique , which is a complete subgraph. Shown in Fig. 2.2 is a complete graph order Nv N = 7, meaning that each vertex is connected to all of he other six vertices.

A regular graph is a graph in which every vertex has the same degree. A regular graph with common degree d is called d-regular. An example of a 2-regular graph is the ring shown in Fig. 2.2. The standard lattice, such as is associated visually with a checker board, is an example of a 4-regular graph.

A connected graph with no cycles is called a tree. The disjoint union of such graphs is called a forest. Trees are of fundamental importance in the analysis of networks. They serve, for example, as a key data structure in the efficient design of many computational algorithms. A digraph whose underlying graph is a tree is called a directed tree. Often such trees have associated with them a special vertex called a root, which is distinguished by being the only vertex from which there is a directed path to every other vertex in the graph. Such a graph is called a rooted tree. A vertex preceding another vertex on a path from the root is called an ancestor, while a vertex following another vertex is called a descendant. Immediate ancestors are called parents, and immediate descendants, children. A vertex without any children is called a leaf. The distance from the root to the farthest leaf is called the depth of the tree.

Given a rooted tree of this sort, it is not uncommon to represent it diagrammatically without any indication of its directedness, as this is to be understood from the definition of the root. Such a representation of a tree is shown in Fig. 2.2. Treating vertex 1 as the root, this is a tree of depth 2, wherein each vertex (excluding the leafs) is the ancestor of two descendants.

A k-star is a special case of a tree, consisting only of one root and k leaves. Such graphs are useful for conceptualizing a vertex and its immediate neighbors (ignoring any connectivity among the neighbors). A representation of a 7-star is given in Fig. 2.2.

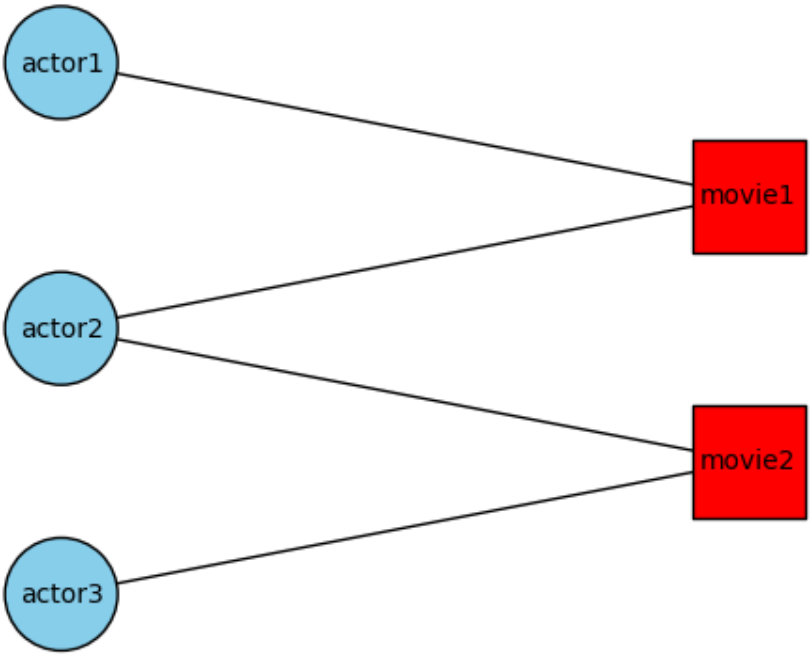
An important generalization of the concept of a tree is that of a directed acyclic graph (i.e., the DAG). A DAG, as its name implies, is a graph that is directed and that has no directed cycles. However, unlike a directed tree, its underlying graph is not a tree, in that replacing the arcs with undirected edges leaves a (simple) graph that contains cycles. Our toy graph dg, for example, is directed but not a DAG.

```
g.is_dag()
```

```
False
```

since the underlying graph is a triangle and hence a 3-cycle. Nevertheless, it is often possible to still design efficient computational algorithms on DAGs that take advantage of this near-tree-like structure.

Lastly, a bipartite graph is a graph G = (V, E) such that the vertex set V may be partitioned into two disjoint sets, say V1 and V2, and each edge in E has on endpoint in V1 and the other in V2. Such graphs typically are used to represet ‘membership’ networks, for example, with ‘members’ denoted by vertices in V1, nd the corresponding ‘organizations’, by vertices in V2. For example, they are pop larin studying the relationship between actors and movies, where actors and m vies play the roles of membes and organizations, respectively.



La grafica anterior es el final del capitulo 2.4.2 del libro (*Special Types of Graphs*, p. 24) de *Kolaczyk and Csárdi* (2020). Para ver el codigo con el cual se crearon los graficos **Ver codigo punto 10**