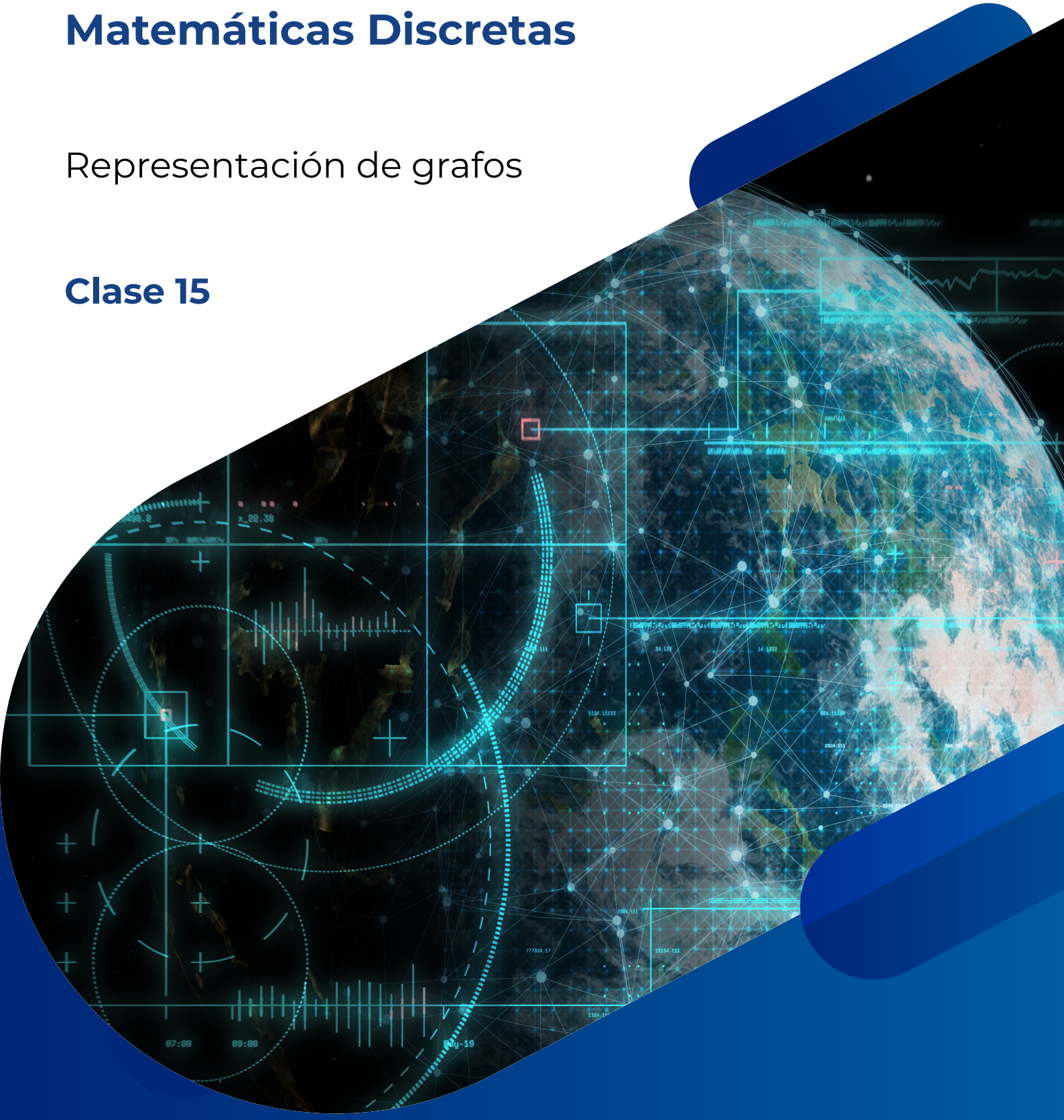
# Matemáticas Discretas

## Representación de grafos

## Clase 15



La excelencia no se improvisa

## Introducción de la clase

La teoría de grafos no se limita únicamente a la definición de vértices y aristas; para poder analizarlos y utilizarlos en aplicaciones prácticas, necesitamos una forma adecuada de representarlos. En esta clase exploraremos cómo expresar un grafo en estructuras que puedan ser fácilmente comprendidas, manipuladas y procesadas por un computador. Esto nos permitirá pasar de la idea abstracta de un grafo a una herramienta concreta para resolver problemas en distintas áreas de la ciencia y la tecnología.

Asimismo, estudiaremos dos de las formas más comunes de representar grafos: la *matriz de adyacencia*, que utiliza una estructura tabular, y la *lista de adyacencia*, que organiza la información de manera más eficiente en muchos casos prácticos. Finalmente, veremos cómo distintas herramientas y software permiten visualizar y analizar grafos de manera intuitiva, lo que facilita la comprensión de su estructura y aplicaciones en situaciones reales como el análisis de redes de comunicación, seguridad informática o interacciones sociales.

## Clase 15: Representación de grafos

RDA 3: Aplicar los principios y métodos de la Matemática Discreta en una variedad de situaciones, en contextos tanto teóricos como prácticos.

## 15. Representación de grafos

Cuando trabajamos con relaciones, aprendimos que un mismo conjunto de pares ordenados podía representarse de diferentes formas: como un diagrama de puntos y flechas, o como una matriz que registraba las conexiones. Algo similar ocurre con los grafos. Aunque la manera más intuitiva de verlos es a través de un dibujo con vértices y aristas, también podemos traducir su estructura a formatos algebraicos o tabulares que permiten analizarlos con mayor precisión.

Esta posibilidad de cambiar de representación es valiosa, porque un mismo grafo puede dibujarse de distintas maneras en el plano y, a simple vista, parecer diferente. Sin embargo, al representarlo como matriz o lista de adyacencia, descubrimos que la estructura de conexiones es la misma. Esto nos ayuda a reconocer grafos

equivalentes aunque sus dibujos sean distintos, centrándonos en lo que realmente importa: las relaciones entre los vértices.

Cada representación tiene ventajas y limitaciones. La matriz de adyacencia es clara y facilita ciertos cálculos, como contar aristas o verificar caminos, pero puede resultar costosa en términos de memoria cuando el grafo es muy grande y tiene pocas conexiones. En cambio, la lista de adyacencia aprovecha mejor el espacio y se adapta mejor a grafos dispersos, lo que la convierte en una herramienta eficiente para aplicaciones en la informática y la ciberseguridad.

Finalmente, veremos cómo estas representaciones se utilizan en software especializado para la visualización y el análisis de grafos. Herramientas como *NetworkX*, permiten transformar las estructuras matemáticas en diagramas interactivos, facilitando la comprensión de redes complejas como las de comunicación, transporte o interacciones sociales. De esta manera, un concepto matemático abstracto se convierte en una herramienta para resolver problemas reales.

## 15.1. Matriz de adyacencia

Dado un grafo $G = (V, A)$, podemos representarlo mediante una matriz cuadrada que describa qué vértices están conectados entre sí. Tenemos que en un grafo no dirigido, la adyacencia, define una relación simétrica sobre el conjunto de vértices $V$, donde dos vértices $v_i$ y $v_j$ son adyacentes si $\{v_i, v_j\} \in A$. La matriz que representa esta relación recibe el nombre de **matriz de adyacencia** (Epp 2020).

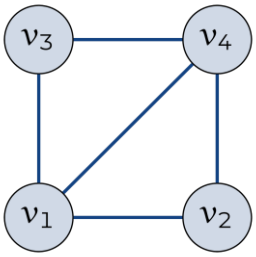
Así, si el conjunto de vértices es $V = \{v_1, v_2, \dots, v_n\}$, la matriz asociada al grafo es una matriz cuadrada $M_G = (m_{ij})$ de orden $n$, definida como:

$$m_{ij} = \begin{cases} 1 & \text{si existe una arista que conecta } v_i \text{ con } v_j, \\ 0 & \text{en caso contrario.} \end{cases}$$

En el caso de los grafos no dirigidos, la relación de adyacencia es simétrica, es decir, si $v_i$ es adyacente a $v_j$, entonces $v_j$ también es adyacente a $v_i$. Como consecuencia, la matriz de adyacencia es simétrica respecto a la diagonal principal. Esta propiedad permite identificar con facilidad si un grafo es dirigido o no, simplemente observando la forma de su matriz.

**Ejemplo 1**. *Consideremos el grafo $G$ con vértices $V = \{v_1, v_2, v_3, v_4\}$ y aristas $A = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}, \{v_1, v_4\}\}$. Su representación gráfica se muestra en la figura 1.*

**Figura 1.** *Grafo $G$ con cuatro vértices*




*Nota: creación propia (Merino, A., 2025).*

La matriz de adyacencia correspondiente es:

$$M_G = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

**Ejemplo 2**. *Podemos usar la matriz de adyacencia para calcular el **grado** de cada vértice. Si $M_G = (m_{ij})$, entonces:*

$$\text{grad}(v_j) = \sum_{i=1}^{n} m_{ij} = \sum_{i=1}^{n} m_{ji}.$$

*En nuestro grafo, el vértice $v_1$ tiene grado:*

$$\text{grad}(v_1) = m_{11} + m_{21} + m_{31} + m_{41} = 0 + 1 + 1 + 1 = 3.$$

*De manera similar,* $\text{grad}(v_2) = 2$, $\text{grad}(v_3) = 2$ *y* $\text{grad}(v_4) = 3$.

Otra aplicación que podemos realizar con las matrices de adyacencia de un grafo es el estudio de los caminos dentro del grafo (Johnsonbaugh 2018).

***TEOREMA 1.***

Sea $G$ un grafo con $V = \{v_1, \dots, v_n\}$ y matriz de adyacencia $M_G = (m_{ij})$. El número de caminos distintos de longitud $k$ que van de $v_i$ a $v_j$ es igual al elemento en la posición $(i, j)$ de la matriz $M_G^k$.

**Ejemplo 3**. *En el grafo de la figura anterior, al calcular la potencia $M_G^2$ obtenemos:*

$$M_G^2 = \begin{bmatrix} 3 & 1 & 1 & 2 \\ 1 & 2 & 2 & 1 \\ 1 & 2 & 2 & 1 \\ 2 & 1 & 1 & 3 \end{bmatrix}.$$

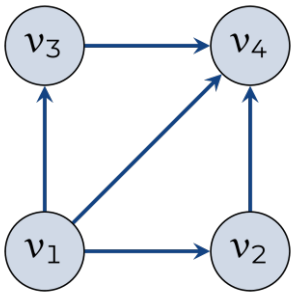
*El valor en la posición* $(1,4)$ *es* $2$*, lo cual significa que existen exactamente dos caminos distintos de longitud* $2$ *desde* $v_1$ *hasta* $v_4$ *(el que pasa por* $v_3$ *y el que pasa por* $(v_2)$*).*

Cuando el grafo es dirigido, la relación de adyacencia ya no es simétrica: puede ocurrir que $(v_i, v_j) \in A$ pero $(v_j, v_i) \notin A$. En este caso, la matriz de adyacencia no será necesariamente simétrica.

**Ejemplo 4**. *Consideremos el grafo* $G$ *con vértices* $V = \{v_1, v_2, v_3, v_4\}$ *y aristas* $A = \{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_3, v_4), (v_1, v_4)\}$*. La representación gráfica se muestra en la siguiente figura.*

**Figura 2**

*Grafo dirigido D*




*Nota: creación propia (Merino, A., 2025).*

*La matriz de adyacencia en este caso es:*

$$M_G = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

*la cual no es simétrica, reflejando la direccionalidad de las aristas.*

En los grafos dirigidos, el cálculo de los grados se realiza a partir de la matriz de adyacencia de la siguiente manera:

$$grad_s(v_i) = \sum_{j=1}^{n} m_{ij},$$

que es la suma de los elementos de la fila $i$,

$$grad_s(v_j) = \sum_{i=1}^{n} m_{ij},$$

que es la suma de los elementos de la columna $j$.

Aplicando estas fórmulas al grafo $G$, tenemos:

$$(v_1) = 3, \quad (v_1) = 0, \qquad (v_2) = 1, \quad (v_2) = 1,$$

y

$$(v_3) = 1, \quad (v_3) = 1, \qquad (v_4) = 0, \quad (v_4) = 3.$$

Con esta información podemos identificar dos casos especiales:

- $v_1$ es una **fuente**, ya que su grado de entrada es 0. En la matriz se refleja porque la columna correspondiente a $v_1$ está formada únicamente por ceros.
- $v_4$ es un **sumidero**, ya que su grado de salida es 0. En la matriz se observa porque la fila correspondiente a $v_4$ está compuesta solo por ceros.

Una de las aplicaciones que podemos obtener de la matriz de adyacencia es el cálculo de la **distancia entre dos vértices** en un grafo. Recordemos que, si $M_G$ es la matriz de adyacencia de un grafo $G$, entonces la entrada $(i, j)$ de la matriz $M_G^k$ indica cuántos *caminos distintos* de longitud $k$ existen de $v_i$ a $v_j$. Con esta idea, podemos construir una **matriz de distancias** $D_G = (d_{ij})$, donde:

$$d_{ij} = \min\{\, k \in \mathbb{N} : \left(M_G^k\right)_{ij} > 0 \,\}.$$

Es decir, $d_{ij}$ es el menor número $k$ tal que existe al menos un camino de longitud $k$ desde $v_i$ hasta $v_j$ (White and Ray 2021).

**Ejemplo 5**. *Consideremos nuevamente el grafo G del primer ejemplo. Su matriz de adyacencia y su cuadrado son*

$$M_G = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}, \qquad M_G^2 = \begin{bmatrix} 3 & 1 & 1 & 2 \\ 1 & 2 & 2 & 1 \\ 1 & 2 & 2 & 1 \\ 2 & 1 & 1 & 3 \end{bmatrix}.$$

*Consideremos $v_2$ y $v_3$, como $(M_G)_{23} = 0$, es decir, no hay un camino de longitud 1, tenemos que la distancia entre estos vértices no es 1; como $(M_G^2)_{23} = 2$, es decir, hay dos caminos de longitud 2, tenemos que la distancia entre estos vértices es menor o igual a 2. Por lo tanto, $d(v_1, v_2) = 2$.*

*Con el mismo razonamiento, tenemos que la matriz de distancias del grafo es:*

$$D_G = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 2 & 1 \\ 1 & 2 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}.$$

Una noción global relacionada con la distancia es la del diámetro del grafo.

***DEFINICIÓN 1.***

El **diámetro** de un grafo $G$ se define como la mayor de las distancias entre dos vértices distintos del grafo:

$$\operatorname{diam}(G) = \max_{v_i, v_j \in V} d(v_i, v_j).$$

El diámetro mide qué tan «lejanos» pueden estar dos vértices dentro del grafo y, por lo tanto, indica la máxima distancia que se necesita recorrer para comunicar cualquier par de vértices.

**Ejemplo 6**. *En el grafo G del ejemplo anterior, las mayores distancias entre vértices son de 2 (el cual es el mayor valor de $D_G$). Por lo tanto, el diámetro del grafo es:*

$$\operatorname{diam}(G) = 2.$$

## 15.2. Lista de adyacencia

Otra forma muy común de representar un grafo es mediante la **lista de adyacencia** (Erciyes 2021). Mientras que la matriz de adyacencia utiliza un arreglo bidimensional donde se registra la existencia de una arista entre cada par de vértices, la lista de adyacencia se centra en cada vértice por separado y guarda únicamente la información de sus vecinos inmediatos.

Esta representación resulta especialmente útil cuando se trabaja con grafos *dispersos* (es decir, con pocas aristas en relación al número total de vértices), ya que ocupa menos memoria y permite acceder de manera más directa a los vértices conectados a un nodo dado. Por esta razón, es ampliamente utilizada en la implementación de algoritmos de recorrido y búsqueda como BFS y DFS.

***DEFINICIÓN 2.***

Sea $G = (V, A)$ un grafo. Para cada vértice $v_i \in V$, llamamos **lista de adyacencia** de $v_i$ al conjunto de vértices que están conectados con $v_i$ por una arista.

$$L(v_i) = \{ v_j \in V : \{v_i, v_j\} \in A \}.$$

A los elementos de $L(v_i)$ se los denomina **vecinos** de $v_i$. En el caso de grafos dirigidos, la lista de adyacencia de $v_i$ está conformada por lo sucesores de $v_i$:

$$L(v_i) = \{ v_j \in V : (v_i, v_j) \in A \}.$$

**Ejemplo 7**. *Consideremos el mismo grafo dirigido G que vimos anteriormente, con vértices* $V = \{v_1, v_2, v_3, v_4\}$ *y aristas*

$$A = \{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_3, v_4), (v_1, v_4)\}.$$

*Su lista de adyacencia se describe de la siguiente manera:*

$$\begin{aligned} L(v_1) &= \{v_2, v_3, v_4\}, \\ L(v_2) &= \{v_4\}, \\ L(v_3) &= \{v_4\}, \\ L(v_4) &= \varnothing. \end{aligned}$$

Podemos observar que la lista de adyacencia se obtiene directamente de las filas de la matriz de adyacencia. Por ejemplo, en la primera fila de $M_G$, los valores distintos de cero aparecen en las posiciones 2,3,4, lo que indica que $v_1$ tiene como vecinos a $v_2, v_3$ y $v_4$.

**Ejemplo 8**. *Si consideramos ahora el mismo grafo, pero como* ***no dirigido****, es decir:*

$$A = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}, \{v_1, v_4\}\},$$

*su lista de adyacencia sería*

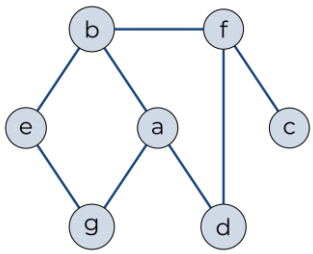
$$
\begin{aligned}
L(v_1) &= \{v_2, v_3, v_4\},\\
L(v_2) &= \{v_1, v_4\},\\
L(v_3) &= \{v_1, v_4\},\\
L(v_4) &= \{v_1, v_2, v_3\}.
\end{aligned}
$$

*En este caso, cada arista aparece reflejada en dos listas, una para cada extremo, lo cual corresponde a la simetría de la relación de adyacencia en grafos no dirigidos.*

**Ejemplo 9**. *Consideremos el grafo de la siguiente figura:*

**Figura 3**

*Grafo G con siete vértices*




*Nota: creación propia (Merino, A., 2025).*

*El código en Python para guardar un grafo anterior como lista de adyacencias es:*

**Código 1**

*Lista de adyacencias en Python*

```
G = { "a" : ["b","d","g"],
    "b" : ["a","e","f"],
    "c" : ["f"],
    "d" : ["a","f"],
    "e" : ["b","g"],
    "f" : ["b","c","d"],
    "g" : ["a","e"]
  }
```

Al concluir esta parte, hemos aprendido que un mismo grafo puede representarse de distintas maneras, ya sea mediante una matriz de adyacencia o mediante una lista de adyacencia. Ambas formas contienen la misma información sobre las conexiones entre vértices, aunque cada una presenta ventajas según el tipo de grafo y el análisis que queramos realizar. La matriz facilita operaciones algebraicas como el cálculo de caminos o el grado de los vértices, mientras que la lista de adyacencia es más eficiente para grafos dispersos y recorridos.

Para profundizar en estos conceptos, puedes revisar el siguiente enlace:

- **Título del enlace:** Grafos básicos, lista y matriz de adyacencia, definiciones y propiedades
- **Descripción del enlace:** Video explicativo sobre las representaciones de grafos, incluyendo matrices y listas de adyacencia.

**Enlace:** https://youtu.be/vnNFiNVy9KM?si=bFBC-1YMi2DsP1y0

### 15.3. Herramientas y visualización en software

En las secciones anteriores vimos cómo un mismo grafo puede representarse de manera formal mediante una matriz o una lista de adyacencia. Esto nos permitió comprender qué operaciones son más eficientes en términos de tiempo y espacio según la representación elegida. Sin embargo, en la práctica, muchas de estas estructuras ya están implementadas en librerías de uso general, lo que nos permite concentrarnos en el análisis y el diseño de algoritmos sin tener que programar desde cero la gestión de grafos.

En esta sección presentaremos **NetworkX**, una librería de Python especializada en el manejo de grafos. Con ella podremos crear grafos, consultar sus vértices y aristas, explorar vecinos y también visualizar las conexiones entre nodos.

#### 15.3.1. Comandos básicos

Podemos importar NetworkX y crear una gráfica no dirigida con un conjunto de aristas de manera muy sencilla:

**Código 2**

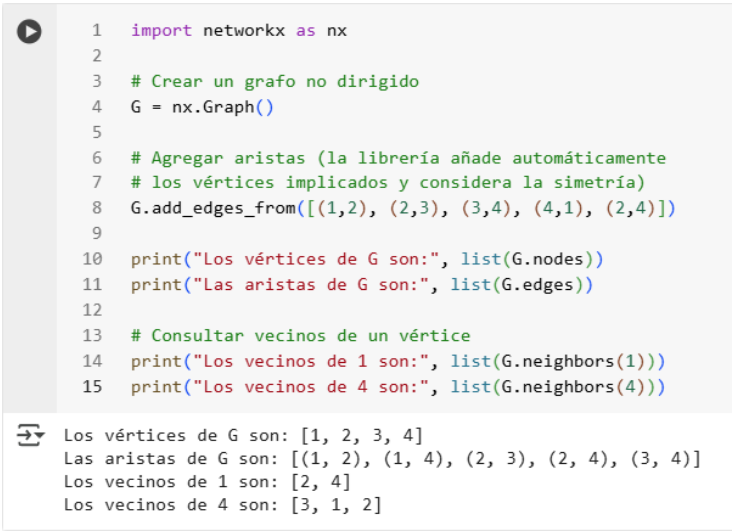
*Creación de una gráfica no dirigida en NetworX*

```
import networkx as nx

# Crear un grafo no dirigido
G = nx.Graph()

# Agregar aristas
G.add_edges_from([(1,2), (2,3), (3,4), (4,1), (2,4)])

print("Los vértices de G son:", list(G.nodes))
print("Las aristas de G son:", list(G.edges))

# Consultar vecinos de un vértice
print("Los vecinos de 1 son:", list(G.neighbors(1)))
print("Los vecinos de 4 son:", list(G.neighbors(4)))
```

La ejecución de este código produce lo que podemos observar en la siguiente figura:

**Figura 4.**

Salida de la ejecución en Python para la gráfica no dirigida




*Nota: creación propia (Merino, A., 2025).*

Como se observa, NetworkX construye automáticamente la lista de vértices a partir de las aristas añadidas, y permite consultar fácilmente los vecinos de cada vértice.

### 15.3.2. Grafos ponderados

Otra característica de NetworkX es la posibilidad de asignar **pesos** a las aristas, lo cual resulta útil cuando trabajamos con grafos ponderados (por ejemplo, en problemas de rutas más cortas o costos en redes). Para esto se usa el método add_weighted_edges_from, donde cada arista se describe mediante una terna: (vértice_origen, vértice_destino, peso).

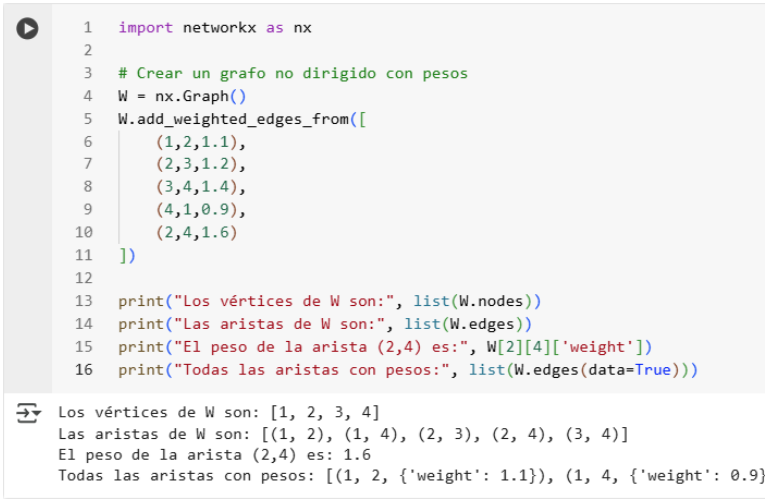
**Código 3**

*Creación de un grafo ponderado en NetworkX*

```
import networkx as nx

# Crear un grafo no dirigido con pesos
W = nx.Graph()
W.add_weighted_edges_from([
    (1,2,1.1),
    (2,3,1.2),
    (3,4,1.4),
    (4,1,0.9),
    (2,4,1.6)
])

print("Los vértices de W son:", list(W.nodes))
print("Las aristas de W son:", list(W.edges))
print("El peso de la arista (2,4) es:", W[2][4]['weight'])
print("Todas las aristas con pesos:", list(W.edges(data=True)))
```

**Figura 5.**

*Visualización de un grafo ponderado en NetworkX*



```python
import networkx as nx

# Crear un grafo no dirigido con pesos
W = nx.Graph()
W.add_weighted_edges_from([
    (1,2,1.1),
    (2,3,1.2),
    (3,4,1.4),
    (4,1,0.9),
    (2,4,1.6)
])

print("Los vértices de W son:", list(W.nodes))
print("Las aristas de W son:", list(W.edges))
print("El peso de la arista (2,4) es:", W[2][4]['weight'])
print("Todas las aristas con pesos:", list(W.edges(data=True)))
```

```
Los vértices de W son: [1, 2, 3, 4]
Las aristas de W son: [(1, 2), (1, 4), (2, 3), (2, 4), (3, 4)]
El peso de la arista (2,4) es: 1.6
Todas las aristas con pesos: [(1, 2, {'weight': 1.1}), (1, 4, {'weight': 0.9}
```

*Nota: creación propia (Merino, A., 2025).*

En la última línea del código se observa cómo la librería almacena los pesos en un diccionario asociado a cada arista. Esto permite no solo trabajar con costos, sino también añadir información adicional, como capacidades, colores o etiquetas relevantes en contextos como el análisis de redes de comunicación o el estudio de flujos de datos en ciberseguridad.

### 15.3.3. Gráfos dirigidos

Para trabajar con grafos dirigidos se utiliza el objeto DiGraph. En este caso, el orden de los vértices en cada arista importa, por lo que NetworkX interpreta correctamente la dirección de las flechas. El método neighbors devuelve únicamente los vértices hacia los cuales sale una arista desde $v_i$, mientras que el método predecessors permite identificar los vértices desde los que llegan aristas a $v_i$.

**Código 4**

*Creación de un grafo dirigido en NetworkX*

```
import networkx as nx

# Crear un grafo dirigido
D = nx.DiGraph()
D.add_edges_from([(1,2),(2,3),(3,4),(4,1),(2,4)])

print("En D hay flecha de 1 a:", list(D.neighbors(1)))
print("En D hay flecha de 4 a:", list(D.neighbors(4)))
```

```
print("En D hay flecha hacia 1 desde:", list(D.predecessors(1)))
print("En D hay flecha hacia 4 desde:", list(D.predecessors(4)))
```

La salida muestra la diferencia entre *vecinos sucesores* y *vecinos predecesores*, lo que refleja la asimetría de la matriz de adyacencia en los grafos dirigidos.

### 15.3.4. Visualización de gráficas

Una ventaja adicional de NetworkX es la posibilidad de generar representaciones gráficas de los grafos de forma sencilla. Con la función nx.draw, podemos obtener un dibujo automático de la gráfica, y con draw_networkx_edge_labels es posible mostrar etiquetas o pesos en las aristas. Además, la librería permite usar diferentes *layouts* para distribuir los vértices: en círculo, en cuadrícula o mediante algoritmos de fuerza.

**Código 5**

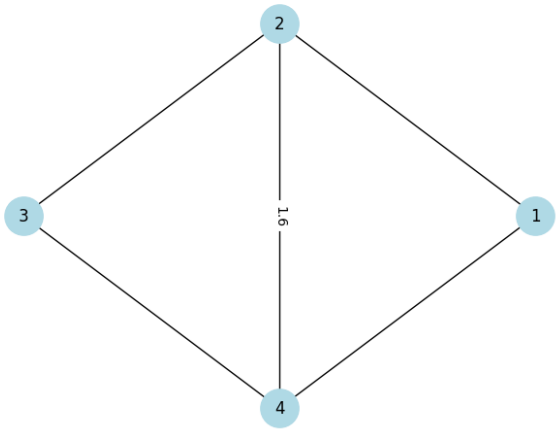
*Dibujo de una gráfica con NetworkX*

```
import matplotlib.pyplot as plt
import networkx as nx

G = nx.Graph()
G.add_edges_from([(1,2),(2,3),(3,4),(4,1),(2,4)])

# Dibujar la gráfica
pos = nx.circular_layout(G)  # disposición circular
nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=800, font_size=
12)
nx.draw_networkx_edge_labels(G, pos, edge_labels={(2,4):"1.6"})
plt.show()
```

**Figura 6**

*Visualización de una gráfica no dirigida con disposición circular en NetworkX*

```python
import matplotlib.pyplot as plt
import networkx as nx

G = nx.Graph()
G.add_edges_from([(1,2),(2,3),(3,4),(4,1),(2,4)])

# Dibujar la gráfica
pos = nx.circular_layout(G)  # disposición circular
nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=800, font_size=12)
nx.draw_networkx_edge_labels(G, pos, edge_labels={(2,4):"1.6"})
plt.show()
```




*Nota:* creación propia (Merino, A., 2025).

Para profundizar en el uso de la librería NetworkX, puedes revisar el siguiente enlace:

- **Título del enlace:** Uso básico de NetworkX
- **Descripción del enlace:** Tutoría introductorio al uso de la librería NetworkX, con ejemplos prácticos.

**Enlace:** https://madi.nekomath.com/P5/NetworkX.html

## Referencias

Epp, S. S. (2020). Discrete Mathematics with Applications (5.a ed.). Cengage Learning.

Erciyes, K. (2021). Discrete Mathematics and Graph Theory. Springer.

Johnsonbaugh, R. (2018). Discrete Mathematics (8.a ed.). Pearson Educación.

White, R. y Ray, A. (2021). Practical Discrete Mathematics. Packt Publishing.

## Términos

- **Matriz de adyacencia:** Es una matriz cuadrada que indica si dos vértices de un grafo están conectados o no. Se coloca un 1 cuando existe una arista entre dos vértices y un 0 cuando no existe.

- **Lista de adyacencia:** Es una forma de representar un grafo en la que a cada vértice se le asocia una lista con los vértices que están conectados directamente a él.

## Profundización

### Recurso 1

- **Título del recurso:** Práctica de grafos en Python
- **Descripción del enlace:** Este recurso presenta un ejemplo práctico de uso de grafos en Python.
- **Enlace:** https://colab.research.google.com/github/andres-merino/MatematicasDiscretas-Virtual-05-N0279/blob/main/pages/clases/clase15/AplicacionGrafos.ipynb

PUCE Virtual

**La excelencia no se improvisa**

síguenos