

Universidad Distrital Francisco José de Caldas

Facultad de Ingeniería

NetworkX

Estudiante: Melisa Maldonado Melenge

Asignatura: Probabilidad y la estadística

Profesor: Alberto Acosta Lopez

Programa: Ingeniería de sistemas

Ciudad: Bogotá

Año: 2026

Índice

| | |
|---|---|
| 1. ¿Qué es y por qué usarlo? | 2 |
| 2. Tipos de grafos | 2 |
| 2.1. Grafo no dirigido (Graph) | 2 |
| 2.2. Grafo dirigido (DiGraph) | 2 |
| 2.3. Multigrafo (MultiGraph) | 2 |
| 2.4. Multigrafo dirigido (MultiDiGraph) | 2 |
| 3. Nodos | 3 |
| 4. Aristas | 3 |
| 5. Grado de los nodos y vecinos | 4 |
| 6. Caminos y distancias | 4 |
| 7. Centralidad — ¿quién es importante en la red? | 4 |
| 8. Clustering y coeficiente de agrupamiento | 5 |
| 9. Árbol generador mínimo | 5 |
| 10. Grafos que ya vienen armados | 5 |
| 11. Guardar y cargar grafos | 6 |
| 12. Visualización básica | 6 |
| 13. Otras cosas que encontré y me parecieron útiles | 7 |
| 14. Conclusión | 7 |

1. ¿Qué es y por qué usarlo?

NetworkX es una librería de Python para trabajar con grafos. Un grafo es básicamente un conjunto de nodos conectados entre sí mediante aristas. Eso suena abstracto, pero sirve para modelar muchas cosas reales: una red de amigos, las rutas entre ciudades, cómo se enlazan páginas web, dependencias entre tareas, etc.

Lo bueno de `networkx` es que es bastante intuitivo y no necesitas implementar los algoritmos desde cero. Instalarlo es simple:

```
1 pip install networkx
```

Y para importarlo en tu código:

```
1 import networkx as nx
```

2. Tipos de grafos

Hay cuatro tipos de grafo, y depende del problema cuál usar:

2.1. Grafo no dirigido (Graph)

Es el más básico. Las aristas no tienen dirección, o sea que si A está conectado con B, entonces B también está conectado con A. Como cuando dos personas son amigas.

```
1 G = nx.Graph()
```

2.2. Grafo dirigido (DiGraph)

Acá las aristas sí tienen dirección, como las calles de un solo sentido o cuando en una red social alguien sigue a otro pero no al revés.

```
1 G = nx.DiGraph()
```

2.3. Multigrafo (MultiGraph)

Básicamente es un grafo donde dos nodos pueden tener más de una arista entre ellos. Por ejemplo dos ciudades que tienen múltiples rutas distintas entre sí.

```
1 G = nx.MultiGraph()
```

2.4. Multigrafo dirigido (MultiDiGraph)

La combinación de los dos anteriores: múltiples aristas y con dirección.

```
1 G = nx.MultiDiGraph()
```

3. Nodos

Una de las cosas más flexibles de NetworkX es que un nodo puede ser cualquier cosa hashable: un número, un string, una tupla. No tenemos que definir una clase especial ni nada.

```

1 G = nx.Graph()
2
3 G.add_node(1)
4 G.add_node("ciudad_A")
5 G.add_nodes_from([2, 3, 4])
6
7 # Los nodos pueden tener atributos, que es muy util
8 G.add_node("Juan", edad=25, pais="Argentina")

```

Para ver qué hay en el grafo:

```

1 G.nodes()                # lista de nodos
2 G.nodes(data=True)       # nodos con sus atributos
3 len(G)                   # cantidad de nodos
4 G.nodes["Juan"]          # atributos de un nodo especifico

```

Borrar nodos es con `remove_node()` y `remove_nodes_from()`. Ojo: cuando borrás un nodo, también se borran todas sus aristas.

4. Aristas

Las aristas también pueden tener atributos. El más común es el peso, que representa distancia, costo, intensidad de relación, lo que sea.

```

1 G.add_edge(1, 2)
2 G.add_edge(1, 2, peso=4.5)    # con atributo
3 G.add_edge("A", "B", distancia=120, tipo="ruta")
4
5 # Si el nodo no existe todavia, se crea solo
6 G.add_edges_from([(1,2),(2,3),(3,4)])
7
8 # Con pesos en varias a la vez
9 G.add_weighted_edges_from([(1,2,1.0),(2,3,2.5),(3,4,0.8)])

```

Para consultar:

```

1 G.edges()                # todas las aristas
2 G.edges(data=True)       # con atributos
3 G.has_edge(1, 2)         # True o False
4 G[1][2]                  # atributos de esa arista puntual
5 G.number_of_edges()

```

5. Grado de los nodos y vecinos

El grado es cuántas aristas tiene un nodo.

```

1 G.degree(1)          # grado de un nodo
2 list(G.degree())     # lista con (nodo, grado) para todos
3
4 # En DiGraph hay dos grados
5 D.in_degree(1)       # cuantas aristas llegan
6 D.out_degree(1)      # cuantas aristas salen

```

Para ver los vecinos de un nodo (con quiénes está conectado):

```

1 list(G.neighbors(1))
2
3 # En DiGraph:
4 list(D.successors(1)) # a quienes apunta
5 list(D.predecessors(1)) # quienes le apuntan

```

6. Caminos y distancias

Esto es una de las cosas más importantes en la práctica.

```

1 G = nx.Graph()
2 G.add_weighted_edges_from([(1,2,1),(2,3,2),(1,3,6),(3,4,1)])
3
4 # Camino mas corto (en saltos, sin contar peso)
5 nx.shortest_path(G, source=1, target=4)
6
7 # Camino mas corto considerando pesos
8 nx.shortest_path(G, source=1, target=4, weight="weight")
9
10 # Solo la longitud, sin el camino
11 nx.shortest_path_length(G, source=1, target=4, weight="weight")
12
13 # Dijkstra directo (hace las dos cosas a la vez)
14 largo, camino = nx.single_source_dijkstra(G, source=1, target=4)

```

Si el grafo está conectado, también puedes sacar estadísticas globales:

```

1 nx.diameter(G) # la distancia maxima entre cualquier par
2 nx.average_shortest_path_length(G)
3 nx.radius(G)

```

7. Centralidad — ¿quién es importante en la red?

Esta sección me gustó porque conecta con aplicaciones reales. Hay varias formas de medir “importancia” y cada una captura algo diferente.

Centralidad de grado: el nodo con más conexiones.

Betweenness: cuántas veces aparece un nodo en los caminos más cortos entre otros pares. Si lo sacás y la red queda partida, tenía alta betweenness. Sirve para encontrar “puentes”.

Closeness: qué tan cerca está un nodo de todos los demás. Alta closeness significa que podés llegar rápido a cualquier parte.

Eigenvector: no solo importa cuántos vecinos tienes, sino qué tan importantes son esos vecinos. Es la base del PageRank de Google que puntúa la relevancia de una página web basándose en la cantidad y calidad de los enlaces que recibe.

8. Clustering y coeficiente de agrupamiento

El clustering de un nodo mide si sus vecinos también se conocen entre sí. Si todos tus amigos son amigos entre sí, tenés clustering alto.

```
1 nx.clustering(G, 0)           # clustering de un nodo
2 nx.clustering(G)             # de todos los nodos
3 nx.average_clustering(G)     # promedio de la red
4 nx.transitivity(G)           # similar pero calculado distinto
5 nx.triangles(G)              # cuantos triangulos pasan por cada
    nodo
```

9. Árbol generador mínimo

Dado un grafo con pesos, encontrar el subconjunto de aristas que conecta todos los nodos con el menor costo o peso total.

```
1 G = nx.Graph()
2 G.add_weighted_edges_from([
3     (1,2,4), (1,3,2), (2,3,1), (2,4,5), (3,4,8)
4 ])
5
6 mst = nx.minimum_spanning_tree(G) # Kruskal por defecto
7 print(list(mst.edges(data=True)))
8
9 # Tambien hay Prim:
10 mst_prim = nx.minimum_spanning_tree(G, algorithm="prim")
11
12 # Y si quer s el maximo en vez del minimo:
13 mxt = nx.maximum_spanning_tree(G)
```

10. Grafos que ya vienen armados

Una cosa práctica: NetworkX incluye generadores para grafos conocidos, muy útiles para probar cosas sin tener que construir el grafo a mano.

```

1 nx.complete_graph(5)           # todos conectados con todos
2 nx.path_graph(6)               # cadena lineal
3 nx.cycle_graph(5)              # anillo
4 nx.star_graph(4)               # un centro, n hojas
5 nx.grid_2d_graph(3, 3)         # grilla 3x3
6
7 # Modelos aleatorios
8 nx.erdos_renyi_graph(n=20, p=0.3) # cada arista existe con prob
   p

```

11. Guardar y cargar grafos

Cuando el grafo es grande y no lo quieres reconstruir cada vez.

```

1 # Lista de aristas
2 nx.write_edgelist(G, "grafo.txt")
3 G = nx.read_edgelist("grafo.txt")
4
5 # Con pesos
6 nx.write_weighted_edgelist(G, "grafo_peso.txt")
7 G = nx.read_weighted_edgelist("grafo_peso.txt")
8
9 # GraphML: sirve para abrir en Gephi y otros programas
10 nx.write_graphml(G, "grafo.graphml")
11 G = nx.read_graphml("grafo.graphml")

```

12. Visualización básica

Para dibujar grafos, NetworkX usa matplotlib. No es lo más bonito del mundo pero funciona para entender la estructura.

```

1 import matplotlib.pyplot as plt
2
3 G = nx.karate_club_graph()
4 pos = nx.spring_layout(G, seed=42)
5
6 nx.draw_networkx(G, pos,
7     node_color='steelblue',
8     node_size=400,
9     edge_color='gray',
10    font_color='white',
11    with_labels=True
12 )
13 plt.axis('off')
14 plt.tight_layout()
15 plt.show()

```

13. Otras cosas que encontré y me parecieron útiles

Subgrafos: si solo te interesa parte del grafo, podés sacar un subgrafo con ciertos nodos. Ojo que es una vista, no una copia. Si modificás el original, el subgrafo cambia.

```
1 sub = G.subgraph([1, 2, 3])
```

Puentes y puntos de articulación: un puente es una arista que si la quitás desconecta el grafo. Un punto de articulación es lo mismo pero con nodos.

```
1 list(nx.bridges(G))
2 list(nx.articulation_points(G))
```

Isomorfismo: dos grafos son isomorfos si tienen la misma estructura aunque sus nodos tengan nombres distintos.

```
1 nx.is_isomorphic(G1, G2)
```

14. Conclusión

Después de leer bastante sobre NetworkX, la conclusión es que es una librería muy completa para lo que promete. No tienes que implementar casi nada desde cero, y eso ahorra mucho tiempo. Lo que más me costó entender fue la diferencia entre grafos dirigidos y no dirigidos para los algoritmos de conectividad. Lo demás fue bastante intuitivo una vez que entendí la lógica general.
