eccentricity(G, v=None, sp=None)

Essa função calcula a excentricidade de um nó v ou de todos os nós no grafo G, que é a maior distância entre v e todos os outros nós no grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

# Calcula a excentricidade de todos os nós no grafo G

eccentricities = nx.eccentricity(G)

print(eccentricities)

# Calcula a excentricidade do nó 1 no grafo G

eccentricity\_node\_1 = nx.eccentricity(G, v=1)

print(eccentricity\_node\_1)

edge\_betweenness\_centrality(G, normalized=True, weight=None, seed=None)

Essa função calcula a centralidade de intermediação de arestas para todas as arestas do grafo G. A centralidade de intermediação de arestas é uma medida que quantifica a importância de uma aresta na comunicação entre outros pares de nós no grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

# Calcula a centralidade de intermediação de arestas para todas as arestas no grafo G

edge\_betweenness = nx.edge\_betweenness\_centrality(G)

print(edge\_betweenness)

Espero que essas explicações e s tenham sido úteis! Se você tiver mais dúvidas ou precisar de mais esclarecimentos, sinta-se à vontade para perguntar!

User

O mesmo para: 'edge\_betweenness\_centrality\_subset',

'edge\_bfs',

'edge\_boundary',

'edge\_connectivity',

'edge\_current\_flow\_betweenness\_centrality',

'edge\_current\_flow\_betweenness\_centrality\_subset',

'edge\_dfs',

'edge\_disjoint\_paths',

'edge\_expansion',

'edge\_load\_centrality',

ChatGPT

Claro! Vamos continuar com as explicações e s para as funções restantes:

edge\_betweenness\_centrality\_subset(G, sources, targets, normalized=True, weight=None, seed=None)

Essa função calcula a centralidade de intermediação de arestas apenas para as arestas no caminho mais curto entre os pares de nós fornecidos em sources e targets.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4), (4, 5)])

# Calcula a centralidade de intermediação de arestas apenas para as arestas no caminho mais curto entre os pares de nós (1, 5) e (2, 4)

edge\_betweenness\_subset = nx.edge\_betweenness\_centrality\_subset(G, sources=[1, 2], targets=[5, 4])

print(edge\_betweenness\_subset)

edge\_bfs(G, source, orientation='original', reverse\_order=False)

Essa função realiza uma busca em largura (BFS) a partir do nó de origem source no grafo direcionado G, retornando as arestas alcançáveis.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

# Realiza uma busca em largura a partir do nó 1 no grafo direcionado G

edges\_reachable\_from\_1 = nx.edge\_bfs(G, source=1)

print(list(edges\_reachable\_from\_1))

edge\_boundary(G, ebunch, \*args)

Essa função retorna as arestas na fronteira do conjunto ebunch de arestas em um grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

# Retorna as arestas na fronteira do conjunto [(1, 2), (3, 4)] de arestas em G

edge\_boundary\_set = nx.edge\_boundary(G, [(1, 2), (3, 4)])

print(list(edge\_boundary\_set))

edge\_connectivity(G, s=None, t=None, flow\_func=None)

Essa função retorna a conectividade de arestas entre os nós s e t em um grafo G. A conectividade de arestas é o número mínimo de arestas cuja remoção desconecta s e t no grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1), (3, 5), (5, 6)])

# Retorna a conectividade de arestas entre os nós 1 e 6 em G

edge\_connectivity\_value = nx.edge\_connectivity(G, s=1, t=6)

print(edge\_connectivity\_value)

edge\_current\_flow\_betweenness\_centrality(G, normalized=True, weight=None, dtype=None, solver=None, cutoff=None)

Essa função calcula a centralidade de intermediação de corrente de arestas para todas as arestas do grafo G. A centralidade de intermediação de corrente de arestas é uma medida que quantifica a importância de uma aresta na transferência de corrente entre pares de nós no grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

# Calcula a centralidade de intermediação de corrente de arestas para todas as arestas no grafo G

edge\_current\_flow\_betweenness = nx.edge\_current\_flow\_betweenness\_centrality(G)

print(edge\_current\_flow\_betweenness)

edge\_current\_flow\_betweenness\_centrality\_subset(G, sources, targets, normalized=True, weight=None, dtype=None, solver=None, cutoff=None)

Essa função calcula a centralidade de intermediação de corrente de arestas apenas para as arestas no caminho mais curto entre os pares de nós fornecidos em sources e targets.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4), (4, 5)])

# Calcula a centralidade de intermediação de corrente de arestas apenas para as arestas no caminho mais curto entre os pares de nós (1, 5) e (2, 4)

edge\_current\_flow\_betweenness\_subset = nx.edge\_current\_flow\_betweenness\_centrality\_subset(G, sources=[1, 2], targets=[5, 4])

print(edge\_current\_flow\_betweenness\_subset)

edge\_dfs(G, source=None, orientation='original', reverse\_order=False)

Essa função realiza uma busca em profundidade (DFS) a partir da aresta de origem source no grafo direcionado G, retornando as arestas alcançáveis.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

# Realiza uma busca em profundidade a partir da aresta (1, 2) no grafo direcionado G

edges\_reachable\_from\_12 = nx.edge\_dfs(G, source=(1, 2))

print(list(edges\_reachable\_from\_12))

edge\_disjoint\_paths(G, source, target, k=None)

Essa função retorna um gerador de k caminhos de comprimento mínimo disjuntos entre os nós source e target em um grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

# Retorna um gerador de 2 caminhos de comprimento mínimo disjuntos entre os nós 1 e 3 em G

disjoint\_paths = nx.edge\_disjoint\_paths(G, source=1, target=3, k=2)

print(list(disjoint\_paths))

edge\_expansion(G, s, t)

Essa função calcula a expansão de arestas entre os conjuntos de nós s e t em um grafo G. A expansão de arestas é uma medida que quantifica a qualidade dos conjuntos s e t em termos de seu vínculo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1), (4, 5)])

# Calcula a expansão de arestas entre os conjuntos de nós {1, 2, 3} e {4, 5} em G

edge\_expansion\_value = nx.edge\_expansion(G, s={1, 2, 3}, t={4, 5})

print(edge\_expansion\_value)

Espero que isso seja útil! Se você tiver mais alguma dúvida, por favor, não hesite em perguntar.

User

O mesmo para: 'edge\_subgraph',

'edgebfs',

'edgedfs',

'edgelist',

'edges',

'effective\_size',

'efficiency',

'efficiency\_measures',

'ego',

'ego\_graph',

ChatGPT

Claro! Vamos continuar com as explicações e s para as funções restantes:

edge\_subgraph(G, edges)

Essa função retorna uma subgráfico de G composto apenas pelas arestas especificadas em edges.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 5)])

# Retorna um subgráfico de G composto apenas pelas arestas [(1, 2), (2, 3)]

subgraph = nx.edge\_subgraph(G, [(1, 2), (2, 3)])

print(subgraph.edges())

edgebfs(G, source, orientation='original', reverse\_order=False)

Essa função realiza uma busca em largura (BFS) a partir da aresta de origem source no grafo direcionado G, retornando as arestas alcançáveis.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

# Realiza uma busca em largura a partir da aresta (1, 2) no grafo direcionado G

edges\_reachable\_from\_12 = nx.edgebfs(G, source=(1, 2))

print(list(edges\_reachable\_from\_12))

edgedfs(G, source=None, orientation='original', reverse\_order=False)

Essa função realiza uma busca em profundidade (DFS) a partir da aresta de origem source no grafo direcionado G, retornando as arestas alcançáveis.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

# Realiza uma busca em profundidade a partir da aresta (1, 2) no grafo direcionado G

edges\_reachable\_from\_12 = nx.edgedfs(G, source=(1, 2))

print(list(edges\_reachable\_from\_12))

edgelist(G, data=False)

Essa função retorna uma lista das arestas do grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

# Retorna uma lista das arestas em G

edge\_list = nx.edgelist(G)

print(edge\_list)

edges(G, nbunch=None, data=False, default=None)

Essa função retorna uma visualização de todas as arestas de G ou as arestas associadas aos nós em nbunch.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

# Retorna uma visualização de todas as arestas em G

all\_edges = nx.edges(G)

print(all\_edges)

# Retorna uma visualização das arestas associadas ao nó 1 em G

edges\_of\_node\_1 = nx.edges(G, nbunch=1)

print(edges\_of\_node\_1)

effective\_size(G, weight='weight')

Essa função calcula o tamanho efetivo de cada nó em um grafo ponderado G. O tamanho efetivo de um nó é uma medida da importância do nó em relação a todas as trajetórias no grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=2)

G.add\_edge(2, 3, weight=3)

G.add\_edge(3, 4, weight=1)

G.add\_edge(4, 1, weight=4)

# Calcula o tamanho efetivo de cada nó em G

effective\_size\_values = nx.effective\_size(G)

print(effective\_size\_values)

efficiency(G, u, v, weight='weight')

Essa função calcula a eficiência do par de nós (u, v) em um grafo G. A eficiência é uma medida de quão eficiente é a comunicação entre o nó u e o nó v no grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=2)

G.add\_edge(2, 3, weight=3)

G.add\_edge(3, 4, weight=1)

G.add\_edge(4, 1, weight=4)

# Calcula a eficiência do par de nós (1, 3) em G

efficiency\_value = nx.efficiency(G, u=1, v=3)

print(efficiency\_value)

efficiency\_measures(G, weight='weight')

Essa função calcula várias medidas de eficiência para cada par de nós em um grafo ponderado G. Retorna um dicionário que mapeia pares de nós para seus respectivos valores de eficiência.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=2)

G.add\_edge(2, 3, weight=3)

G.add\_edge(3, 4, weight=1)

G.add\_edge(4, 1, weight=4)

# Calcula várias medidas de eficiência para cada par de nós em G

efficiency\_measures\_values = nx.efficiency\_measures(G)

print(efficiency\_measures\_values)

ego(G, n, radius=1, center=True, undirected=False, distance=None)

Essa função retorna o subgráfico do ego de um nó n em relação a um raio especificado radius. O subgráfico do ego de um nó inclui o nó n e todos os seus vizinhos dentro do raio.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (1, 4), (2, 3), (3, 4)])

# Retorna o subgráfico do ego do nó 1 com raio 1

ego\_subgraph = nx.ego(G, n=1, radius=1)

print(ego\_subgraph.edges())

ego\_graph(G, n, radius=1, center=True, undirected=False, distance=None)

Essa função retorna um subgráfico do ego de um nó n em relação a um raio especificado radius como um novo grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (1, 4), (2, 3), (3, 4)])

# Retorna o subgráfico do ego do nó 1 com raio 1 como um novo grafo

ego\_subgraph = nx.ego\_graph(G, n=1, radius=1)

print(ego\_subgraph.edges())

Espero que estas explicações e s sejam úteis para você! Se precisar de mais ajuda ou tiver mais dúvidas, sinta-se à vontade para perguntar.

User

O mesmo para: 'eigenvector',

'eigenvector\_centrality',

'eigenvector\_centrality\_numpy',

'empty\_graph',

'enumerate\_all\_cliques',

'equitable\_color',

'equivalence\_classes',

'erdos\_renyi\_graph',

'estrada\_index',

'euler',

ChatGPT

Claro! Continuaremos com as explicações e s para as funções restantes:

eigenvector

Essa função retorna o vetor de autovalores da matriz de adjacência de um grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

# Retorna o vetor de autovalores da matriz de adjacência de G

eigenvalues = nx.eigenvector(G)

print(eigenvalues)

eigenvector\_centrality

Essa função calcula a centralidade de vetor próprio para cada nó em um grafo ponderado. A centralidade de vetor próprio mede a importância relativa de um nó em relação a seus vizinhos.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

# Calcula a centralidade de vetor próprio para cada nó em G

eigenvector\_centrality\_values = nx.eigenvector\_centrality(G)

print(eigenvector\_centrality\_values)

eigenvector\_centrality\_numpy

Essa função calcula a centralidade de vetor próprio para cada nó em um grafo ponderado, usando o módulo NumPy.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

# Calcula a centralidade de vetor próprio para cada nó em G usando NumPy

eigenvector\_centrality\_values = nx.eigenvector\_centrality\_numpy(G)

print(eigenvector\_centrality\_values)

empty\_graph(n=0)

Essa função retorna um grafo vazio (sem nós e sem arestas) com n nós.

:

import networkx as nx

# Retorna um grafo vazio com 5 nós

empty\_graph = nx.empty\_graph(n=5)

print(empty\_graph.nodes())

enumerate\_all\_cliques(G)

Essa função retorna um gerador que gera todas as cliques do grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

# Gera todas as cliques em G

cliques\_generator = nx.enumerate\_all\_cliques(G)

for clique in cliques\_generator:

print(clique)

equitable\_color(G, num\_colors=None, value\_only=False)

Essa função encontra uma coloração equitativa mínima para o grafo G. Uma coloração equitativa é uma atribuição de cores aos nós de forma que o número de nós com cores diferentes tenha um valor mínimo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4), (4, 5)])

# Encontra uma coloração equitativa mínima para G

coloring = nx.equitable\_color(G)

print(coloring)

equivalence\_classes(iterable, relation)

Essa função agrupa os elementos do iterable em classes de equivalência usando a relação fornecida.

:

from networkx.utils import equivalence\_classes

# Agrupa os elementos em classes de equivalência usando a relação "x % 2"

iterable = [1, 2, 3, 4, 5, 6]

classes = equivalence\_classes(iterable, relation=lambda x: x % 2)

print(list(classes))

erdos\_renyi\_graph(n, p, seed=None, directed=False)

Essa função retorna um grafo aleatório com n nós, onde cada par de nós é conectado com probabilidade p.

:

import networkx as nx

# Retorna um grafo aleatório com 10 nós e probabilidade de aresta 0.3

random\_graph = nx.erdos\_renyi\_graph(n=10, p=0.3)

print(random\_graph.edges())

estrada\_index(G)

Essa função calcula o índice de estrada para um grafo G. O índice de estrada é uma medida que combina informações de centralidade de grau e centralidade de intermediação dos nós.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4), (4, 5)])

# Calcula o índice de estrada para G

estrada\_index\_value = nx.estrada\_index(G)

print(estrada\_index\_value)

euler(G)

Esta função retorna um ciclo Euleriano no grafo G, se existir. Um ciclo Euleriano é um ciclo que passa por cada aresta exatamente uma vez.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

# Retorna um ciclo Euleriano em G

eulerian\_cycle = nx.euler(G)

print(eulerian\_cycle)

Espero que estas explicações e s sejam úteis para você! Se precisar de mais ajuda ou tiver mais dúvidas, sinta-se à vontade para perguntar.

User

O mesmo para: 'eulerian\_circuit',

'eulerian\_path',

'eulerize',

'exception',

'expanders',

'expected\_degree\_graph',

'extended\_barabasi\_albert\_graph',

'fast\_could\_be\_isomorphic',

'fast\_gnp\_random\_graph',

'faster\_could\_be\_isomorphic',

ChatGPT

Claro! Continuaremos com as explicações e s para as funções restantes:

eulerian\_circuit(G, source=None, keys=False)

Essa função retorna um circuito Euleriano fechado em G, se existir. Um circuito Euleriano é um ciclo que passa por cada aresta exatamente uma vez. O parâmetro source é usado como o nó inicial para começar a busca do circuito.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

# Retorna um circuito Euleriano em G

eulerian\_circuit = nx.eulerian\_circuit(G)

print(list(eulerian\_circuit))

eulerian\_path(G, source=None, keys=False)

Essa função retorna um caminho Euleriano em G, se existir. Um caminho Euleriano é um caminho que passa por cada aresta exatamente uma vez. O parâmetro source é usado como o nó inicial para começar a busca do caminho.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

# Retorna um caminho Euleriano em G

eulerian\_path = nx.eulerian\_path(G)

print(list(eulerian\_path))

eulerize(G)

Essa função modifica o grafo G para torná-lo Euleriano adicionando arestas ausentes. Isso é feito garantindo que todos os nós tenham grau par.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

# Torna G Euleriano adicionando arestas ausentes

eulerized\_graph = nx.eulerize(G)

print(eulerized\_graph.edges())

exception(\*args, \*\*kwargs)

Essa função é usada para levantar uma exceção no contexto das funções do NetworkX.

:

import networkx as nx

try:

# Chama a função que levanta uma exceção

nx.exception()

except Exception as e:

print("Exceção capturada:", e)

expanders.ramanujan\_graph(d, k, seed=None)

Essa função retorna um grafo Ramanujan com grau d e parâmetro k.

:

import networkx as nx

# Retorna um grafo Ramanujan com grau 3 e parâmetro 7

ramanujan\_graph = nx.expanders.ramanujan\_graph(d=3, k=7)

print(ramanujan\_graph.edges())

expected\_degree\_graph(w, seed=None)

Essa função retorna um grafo cujo grau médio é w.

:

import networkx as nx

# Retorna um grafo com grau médio de 4

expected\_degree\_graph = nx.expected\_degree\_graph(w=4)

print(expected\_degree\_graph.edges())

extended\_barabasi\_albert\_graph(n, m1, m2, p, seed=None)

Essa função retorna um grafo estendido de Barabási-Albert. O grafo cresce inicialmente por anexação de um grau m1 do nó novo. Em seguida, para cada novo nó adicionado, dois links são adicionados ao grafo: um link é anexado a um nó existente com probabilidade p, enquanto o outro link é anexado a um nó existente aleatoriamente.

:

import networkx as nx

# Retorna um grafo estendido de Barabási-Albert com 10 nós, grau inicial 2 e 2 links adicionais com probabilidade 0.5

extended\_ba\_graph = nx.extended\_barabasi\_albert\_graph(n=10, m1=2, m2=2, p=0.5)

print(extended\_ba\_graph.edges())

fast\_could\_be\_isomorphic(G1, G2)

Essa função verifica se os grafos G1 e G2 podem ser isomórficos sem calcular explicitamente o isomorfismo. Ela retorna True se os grafos puderem ser isomórficos e False caso contrário.

:

import networkx as nx

G1 = nx.Graph()

G1.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

G2 = nx.Graph()

G2.add\_edges\_from([(4, 5), (5, 6), (6, 4)])

# Verifica se G1 e G2 podem ser isomórficos

isomorphic = nx.fast\_could\_be\_isomorphic(G1, G2)

print(isomorphic)

fast\_gnp\_random\_graph(n, p, seed=None, directed=False)

Essa função retorna um grafo aleatório com n nós, onde cada par de nós é conectado com probabilidade p, de forma mais rápida que gnp\_random\_graph. O parâmetro directed determina se o grafo é direcionado ou não.

:

import networkx as nx

# Retorna um grafo aleatório com 10 nós e probabilidade de aresta 0.3

random\_graph = nx.fast\_gnp\_random\_graph(n=10, p=0.3)

print(random\_graph.edges())

faster\_could\_be\_isomorphic(G1, G2)

Essa função verifica se os grafos G1 e G2 podem ser isomórficos de forma mais rápida que could\_be\_isomorphic. Ela retorna True se os grafos puderem ser isomórficos e False caso contrário.

:

import networkx as nx

G1 = nx.Graph()

G1.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

G2 = nx.Graph()

G2.add\_edges\_from([(4, 5), (5, 6), (6, 4)])

# Verifica se G1 e G2 podem ser isomórficos de forma mais rápida

isomorphic = nx.faster\_could\_be\_isomorphic(G1, G2)

print(isomorphic)

Espero que estas explicações e s adicionais tenham sido úteis! Se precisar de mais ajuda ou tiver mais dúvidas, sinta-se à vontade para perguntar.

User

O mesmo para: 'fiedler\_vector',

'filters',

'find\_asteroidal\_triple',

'find\_cliques',

'find\_cliques\_recursive',

'find\_cycle',

'find\_induced\_nodes',

'find\_negative\_cycle',

'florentine\_families\_graph',

'flow',

ChatGPT

Claro! Continuaremos com as explicações e s para as funções restantes:

fiedler\_vector(G, weight='weight', normalized=True, tol=1e-08, method='tracemin\_lu')

Essa função retorna o vetor de Fiedler de G. O vetor de Fiedler é o segundo menor autovalor não nulo da matriz Laplaciana de G e é usado em várias aplicações de álgebra linear e teoria dos grafos.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 3}), (2, 3, {'weight': 2}), (3, 1, {'weight': 1})])

# Retorna o vetor de Fiedler do grafo G

fiedler\_vector = nx.fiedler\_vector(G)

print(fiedler\_vector)

filters

Esse é um módulo no NetworkX que fornece funções para criar filtros de grafos.

find\_asteroidal\_triple(G)

Essa função retorna uma tripla asteroidal em G, se existir. Uma tripla asteroidal é um conjunto de três nós que são mutuamente adjacentes no grafo complementar.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (1, 3), (3, 4)])

# Retorna uma tripla asteroidal em G

asteroidal\_triple = nx.find\_asteroidal\_triple(G)

print(asteroidal\_triple)

find\_cliques(G)

Essa função retorna todas as cliques (subgrafos completos) de G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (1, 3), (3, 4)])

# Retorna todas as cliques em G

cliques = list(nx.find\_cliques(G))

print(cliques)

find\_cliques\_recursive(G)

Essa função também retorna todas as cliques de G, mas usando uma abordagem recursiva.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (1, 3), (3, 4)])

# Retorna todas as cliques em G usando uma abordagem recursiva

cliques = list(nx.find\_cliques\_recursive(G))

print(cliques)

find\_cycle(G, source=None, orientation='original')

Essa função retorna um ciclo em G, se existir. O parâmetro source é usado como o nó inicial para começar a busca do ciclo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (3, 4), (4, 5), (5, 3)])

# Retorna um ciclo em G

cycle = nx.find\_cycle(G)

print(cycle)

find\_induced\_nodes(G, nbunch)

Essa função retorna os nós de G que são induzidos por nbunch, ou seja, os vizinhos de nbunch que estão todos conectados entre si.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4), (4, 5)])

# Retorna os nós induzidos pelo nó 1

induced\_nodes = nx.find\_induced\_nodes(G, nbunch=1)

print(induced\_nodes)

find\_negative\_cycle(G, weight='weight')

Essa função retorna um ciclo de peso negativo em G, se existir. O grafo deve ser direcionado.

:

import networkx as nx

G = nx.DiGraph()

G.add\_weighted\_edges\_from([(1, 2, -1), (2, 3, -2), (3, 1, -3)])

# Retorna um ciclo de peso negativo em G

negative\_cycle = nx.find\_negative\_cycle(G)

print(negative\_cycle)

florentine\_families\_graph()

Essa função retorna o grafo das famílias florentinas do Renascimento da Itália.

:

import networkx as nx

# Retorna o grafo das famílias florentinas

florentine\_families\_graph = nx.florentine\_families\_graph()

print(florentine\_families\_graph.edges())

flow

Esse é um módulo no NetworkX que contém funções relacionadas a fluxo em grafos.

Espero que estas explicações e s adicionais tenham sido úteis! Se precisar de mais ajuda ou tiver mais dúvidas, sinta-se à vontade para perguntar.

User

'flow\_hierarchy',

'flow\_matrix',

'floyd\_warshall',

'floyd\_warshall\_numpy',

'floyd\_warshall\_predecessor\_and\_distance',

'forest\_str',

'freeze',

'from\_dict\_of\_dicts',

'from\_dict\_of\_lists',

'from\_edgelist',

ChatGPT

Claro! Continuaremos com as explicações e s para as funções restantes:

flow\_hierarchy(G, root=None, capacity='capacity', weight='weight')

Essa função retorna a hierarquia de fluxo do grafo G, a partir do nó raiz especificado.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edge(1, 2, capacity=10, weight=5)

G.add\_edge(1, 3, capacity=5, weight=3)

G.add\_edge(2, 4, capacity=4, weight=2)

G.add\_edge(3, 4, capacity=8, weight=6)

# Retorna a hierarquia de fluxo do grafo G, com nó raiz 1

flow\_hierarchy = nx.flow\_hierarchy(G, root=1)

print(flow\_hierarchy)

flow\_matrix(G, nodelist=None, capacity='capacity', weight='weight', dtype=None, format='full')

Essa função retorna uma matriz de fluxo para o grafo G, usando o algoritmo de Ford-Fulkerson.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edge(1, 2, capacity=10, weight=5)

G.add\_edge(1, 3, capacity=5, weight=3)

G.add\_edge(2, 4, capacity=4, weight=2)

G.add\_edge(3, 4, capacity=8, weight=6)

# Retorna a matriz de fluxo para o grafo G

flow\_matrix = nx.flow\_matrix(G)

print(flow\_matrix)

floyd\_warshall(G, weight='weight', nodes=None)

Essa função retorna as menores distâncias entre todos os pares de nós em G, usando o algoritmo de Floyd-Warshall.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=5)

G.add\_edge(1, 3, weight=3)

G.add\_edge(2, 4, weight=2)

G.add\_edge(3, 4, weight=6)

# Retorna as menores distâncias entre todos os pares de nós em G

shortest\_distances = nx.floyd\_warshall(G)

print(shortest\_distances)

floyd\_warshall\_numpy(G, nodelist=None, weight='weight')

Essa função retorna as menores distâncias entre todos os pares de nós em G, usando o algoritmo de Floyd-Warshall, representando a matriz de distâncias como um array NumPy.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=5)

G.add\_edge(1, 3, weight=3)

G.add\_edge(2, 4, weight=2)

G.add\_edge(3, 4, weight=6)

# Retorna as menores distâncias entre todos os pares de nós em G como um array NumPy

shortest\_distances = nx.floyd\_warshall\_numpy(G)

print(shortest\_distances)

floyd\_warshall\_predecessor\_and\_distance(G, weight='weight')

Essa função retorna um dicionário contendo todos os pares de nós e suas menores distâncias e predecessores no caminho mais curto entre eles, usando o algoritmo de Floyd-Warshall.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=5)

G.add\_edge(1, 3, weight=3)

G.add\_edge(2, 4, weight=2)

G.add\_edge(3, 4, weight=6)

# Retorna os pares de nós e suas menores distâncias e predecessores no caminho mais curto

predecessor\_and\_distance = nx.floyd\_warshall\_predecessor\_and\_distance(G)

print(predecessor\_and\_distance)

forest\_str(G, nodelist=None, root=None, args=None)

Essa função retorna uma string que representa a árvore de caminhos mais curtos do grafo G a partir do nó raiz especificado.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edge(1, 2, weight=5)

G.add\_edge(1, 3, weight=3)

G.add\_edge(2, 4, weight=2)

G.add\_edge(3, 4, weight=6)

# Retorna a representação em string da árvore de caminhos mais curtos de G a partir do nó raiz 1

forest\_string = nx.forest\_str(G, root=1)

print(forest\_string)

freeze(G)

Essa função retorna uma cópia congelada do grafo G, o que significa que a cópia não pode ser modificada.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

# Retorna uma cópia congelada do grafo G

frozen\_graph = nx.freeze(G)

print(frozen\_graph.edges())

from\_dict\_of\_dicts(d, create\_using=None)

Essa função retorna um grafo a partir de um dicionário de dicionários, onde as chaves exteriores representam os nós de origem, as chaves interiores representam os nós de destino e os valores representam os atributos das arestas.

:

import networkx as nx

d = {

1: {2: {'weight': 5}, 3: {'weight': 3}},

2: {4: {'weight': 2}},

3: {4: {'weight': 6}}

}

# Retorna um grafo a partir do dicionário de dicionários

G = nx.from\_dict\_of\_dicts(d)

print(G.edges(data=True))

from\_dict\_of\_lists(d, create\_using=None)

Essa função retorna um grafo a partir de um dicionário de listas, onde as chaves representam os nós de origem e os valores representam os nós de destino.

:

import networkx as nx

d = {

1: [2, 3],

2: [4],

3: [4]

}

# Retorna um grafo a partir do dicionário de listas

G = nx.from\_dict\_of\_lists(d)

print(G.edges())

from\_edgelist(edgelist, create\_using=None, edge\_key='weight')

Essa função retorna um grafo a partir de uma lista de arestas representadas como tuplas (u, v, d), onde u e v são os nós e d é um dicionário contendo os atributos da aresta.

:

import networkx as nx

edgelist = [

(1, 2, {'weight': 5}),

(1, 3, {'weight': 3}),

(2, 4, {'weight': 2}),

(3, 4, {'weight': 6})

]

# Retorna um grafo a partir da lista de arestas

G = nx.from\_edgelist(edgelist)

print(G.edges(data=True))

Espero que estas explicações e s tenham sido úteis! Se precisar de mais ajuda ou tiver mais dúvidas, sinta-se à vontade para perguntar.

User

O mesmo para: 'from\_graph6\_bytes',

'from\_nested\_tuple',

'from\_numpy\_array',

'from\_pandas\_adjacency',

'from\_pandas\_edgelist',

'from\_prufer\_sequence',

'from\_scipy\_sparse\_array',

'from\_sparse6\_bytes',

'frucht\_graph',

'fruchterman\_reingold\_layout',

ChatGPT

Claro! Continuaremos com as explicações e s para as funções restantes:

from\_graph6\_bytes(graph6\_bytes)

Essa função retorna um grafo a partir de sua representação em formato graph6 (formato de codificação compacta para grafos não direcionados).

:

import networkx as nx

graph6\_bytes = b'IyF~'

G = nx.from\_graph6\_bytes(graph6\_bytes)

print(G.edges())

from\_nested\_tuple(X, create\_using=None)

Essa função retorna um grafo a partir de uma tupla aninhada que representa as arestas do grafo.

:

import networkx as nx

X = (('A', 'B'), ('B', 'C'), ('C', 'D'))

G = nx.from\_nested\_tuple(X)

print(G.edges())

from\_numpy\_array(A, create\_using=None, edge\_attribute='weight')

Essa função retorna um grafo a partir de uma matriz NumPy que representa a matriz de adjacência do grafo.

:

import networkx as nx

import numpy as np

A = np.array([[0, 1, 0, 0],

[1, 0, 1, 1],

[0, 1, 0, 0],

[0, 1, 0, 0]])

G = nx.from\_numpy\_array(A)

print(G.edges())

from\_pandas\_adjacency(df, create\_using=None)

Essa função retorna um grafo a partir de um DataFrame do pandas que representa a matriz de adjacência do grafo.

:

import networkx as nx

import pandas as pd

data = {

'A': [0, 1, 0, 0],

'B': [1, 0, 1, 1],

'C': [0, 1, 0, 0],

'D': [0, 1, 0, 0]

}

df = pd.DataFrame(data, index=['A', 'B', 'C', 'D'])

G = nx.from\_pandas\_adjacency(df)

print(G.edges())

from\_pandas\_edgelist(df, source, target, edge\_attr=None, create\_using=None)

Essa função retorna um grafo a partir de um DataFrame do pandas que representa a lista de arestas do grafo.

:

import networkx as nx

import pandas as pd

data = {

'source': ['A', 'B', 'B', 'C'],

'target': ['B', 'C', 'D', 'D'],

'weight': [5, 2, 6, 4]

}

df = pd.DataFrame(data)

G = nx.from\_pandas\_edgelist(df, 'source', 'target', edge\_attr='weight')

print(G.edges(data=True))

from\_prufer\_sequence(sequence)

Essa função retorna um grafo a partir de uma sequência de Prufer.

:

import networkx as nx

sequence = [2, 2, 3]

G = nx.from\_prufer\_sequence(sequence)

print(G.edges())

from\_scipy\_sparse\_array(A, parallel\_edges=False, create\_using=None, edge\_attribute='weight')

Essa função retorna um grafo a partir de uma matriz esparsa do SciPy que representa a matriz de adjacência do grafo.

:

import networkx as nx

import scipy.sparse as sp

A = sp.csr\_matrix([[0, 1, 0, 0],

[1, 0, 1, 1],

[0, 1, 0, 0],

[0, 1, 0, 0]])

G = nx.from\_scipy\_sparse\_array(A)

print(G.edges())

from\_sparse6\_bytes(sparse6\_bytes)

Essa função retorna um grafo a partir de sua representação em formato sparse6 (formato de codificação compacta para grafos não direcionados esparsos).

:

import networkx as nx

sparse6\_bytes = b'>>sparse6<<:I~'

G = nx.from\_sparse6\_bytes(sparse6\_bytes)

print(G.edges())

frucht\_graph()

Essa função retorna o grafo Frucht, um grafo cúbico não direcionado de 12 nós e 18 arestas, que é o menor grafo cúbico sem vértices de corte.

:

import networkx as nx

G = nx.frucht\_graph()

print(G.edges())

fruchterman\_reingold\_layout(G, k=None, pos=None, fixed=None, iterations=50, threshold=0.0001, weight='weight', scale=1.0)

Essa função retorna um layout de posicionamento dos nós para o grafo G usando o algoritmo de Fruchterman-Reingold.

:

import networkx as nx

import matplotlib.pyplot as plt

G = nx.cycle\_graph(5)

# Calcula o layout dos nós usando o algoritmo de Fruchterman-Reingold

pos = nx.fruchterman\_reingold\_layout(G)

# Desenha o grafo usando o layout calculado

nx.draw(G, pos, with\_labels=True, node\_color='skyblue', node\_size=1000, font\_size=10)

plt.show()

Espero que estas explicações e s tenham sido úteis! Se precisar de mais ajuda ou tiver mais dúvidas, sinta-se à vontade para perguntar.

User

'full\_join',

'full\_rary\_tree',

'function',

'gaussian\_random\_partition\_graph',

'general\_random\_intersection\_graph',

'generalized\_degree',

'generate\_adjlist',

'generate\_edgelist',

'generate\_gexf',

'generate\_gml',

ChatGPT

Continuando a lista de funções do NetworkX:

full\_join(m, n)

Essa função retorna o grafo do produto cartesiano de dois grafos completos: K\_m e K\_n.

:

import networkx as nx

G = nx.full\_join(3, 4)

print(G.edges())

full\_rary\_tree(r, n)

Essa função retorna uma árvore r-ária completa com n nós.

:

import networkx as nx

G = nx.full\_rary\_tree(3, 7)

print(G.edges())

function(func)

Essa função é usada para converter uma função em uma função NetworkX que pode ser usada para atribuir pesos às arestas.

:

import networkx as nx

def my\_weight\_func(u, v, data):

return u + v

G = nx.Graph()

# Use a função function() para converter a função em uma função NetworkX

G.add\_edge(1, 2, weight=nx.function(my\_weight\_func))

print(G.edges(data=True))

gaussian\_random\_partition\_graph(n, s, v, p\_in, p\_out, directed=False, seed=None)

Essa função retorna um grafo particionado aleatório onde os nós são distribuídos em grupos de acordo com uma distribuição normal.

:

import networkx as nx

import matplotlib.pyplot as plt

G = nx.gaussian\_random\_partition\_graph(100, 5, 5, 0.5, 0.1)

pos = nx.spring\_layout(G)

nx.draw(G, pos, with\_labels=False, node\_size=30)

plt.show()

general\_random\_intersection\_graph(degseq, k, p, seed=None)

Essa função retorna um grafo de interseção aleatório com a sequência de graus dada.

:

import networkx as nx

import matplotlib.pyplot as plt

degseq = [4, 3, 2, 1, 3, 2]

G = nx.general\_random\_intersection\_graph(degseq, 3, 0.5)

pos = nx.spring\_layout(G)

nx.draw(G, pos, with\_labels=False, node\_size=30)

plt.show()

generalized\_degree(G, nodes=None, weight=None)

Essa função retorna o grau generalizado de um ou vários nós em um grafo.

:

import networkx as nx

G = nx.cycle\_graph(5)

degree = nx.generalized\_degree(G, nodes=[0, 1, 2], weight='weight')

print(degree)

generate\_adjlist(G, delimiter=' ', comments='#', encoding='utf-8')

Essa função retorna uma representação em formato de lista de adjacência do grafo.

:

import networkx as nx

G = nx.cycle\_graph(4)

adjlist\_str = nx.generate\_adjlist(G, delimiter=',')

print(adjlist\_str)

generate\_edgelist(G, delimiter=' ', comments='#', data=True, encoding='utf-8')

Essa função retorna uma representação em formato de lista de arestas do grafo.

:

import networkx as nx

G = nx.cycle\_graph(4)

edgelist\_str = nx.generate\_edgelist(G, delimiter='|', data=False)

print(edgelist\_str)

generate\_gexf(G)

Essa função retorna uma representação em formato GEXF (Graph Exchange XML Format) do grafo.

:

import networkx as nx

G = nx.cycle\_graph(4)

gexf\_str = nx.generate\_gexf(G)

print(gexf\_str)

generate\_gml(G)

Essa função retorna uma representação em formato GML (Graph Modeling Language) do grafo.

:

import networkx as nx

G = nx.cycle\_graph(4)

gml\_str = nx.generate\_gml(G)

print(gml\_str)

Espero que esta continuação seja útil! Se você tiver mais perguntas ou precisar de mais s, por favor, fique à vontade para perguntar.

User

O mesmo para: 'generate\_graphml',

'generate\_multiline\_adjlist',

'generate\_network\_text',

'generate\_pajek',

'generate\_random\_paths',

'generators',

'generic',

'geographical\_threshold\_graph',

'geometric',

'geometric\_edges',

ChatGPT

Continuando a lista de funções do NetworkX:

generate\_graphml(G, encoding='utf-8')

Essa função retorna uma representação em formato GraphML (Graph Markup Language) do grafo.

:

import networkx as nx

G = nx.cycle\_graph(4)

graphml\_str = nx.generate\_graphml(G)

print(graphml\_str)

generate\_multiline\_adjlist(G, delimiter=' ', comments='#', encoding='utf-8')

Essa função retorna uma representação em formato de lista de adjacência multilinhas do grafo.

:

import networkx as nx

G = nx.cycle\_graph(4)

multiline\_adjlist\_str = nx.generate\_multiline\_adjlist(G, delimiter=',')

print(multiline\_adjlist\_str)

generate\_network\_text(G, delimiter=' ', data=True, encoding='utf-8')

Essa função retorna uma representação de texto do grafo.

:

import networkx as nx

G = nx.cycle\_graph(4)

network\_text\_str = nx.generate\_network\_text(G, delimiter='|', data=False)

print(network\_text\_str)

generate\_pajek(G)

Essa função retorna uma representação em formato Pajek do grafo.

:

import networkx as nx

G = nx.cycle\_graph(4)

pajek\_str = nx.generate\_pajek(G)

print(pajek\_str)

generate\_random\_paths(G, source=None, targets=None, length=10, seed=None)

Essa função gera caminhos aleatórios em um grafo.

:

import networkx as nx

G = nx.cycle\_graph(4)

paths = nx.generate\_random\_paths(G, source=0, targets=[2, 3], length=5)

print(list(paths))

generators

Essa é uma submódulo do NetworkX que contém várias funções para criar diferentes tipos de grafos, como grafos aleatórios, grafos clássicos e muito mais.

:

import networkx as nx

# de criação de um grafo completo com 5 nós

G = nx.generators.complete\_graph(5)

print(G.edges())

generic

Essa é uma submódulo do NetworkX que fornece funções para criar um grafo vazio, adicionar e remover nós e arestas, verificar a existência de nós e arestas, entre outras operações básicas.

:

import networkx as nx

G = nx.generic.Graph()

G.add\_node(1)

G.add\_edge(1, 2)

print(G.edges())

geographical\_threshold\_graph(n, theta, dim=2, pos=None, weight=None, metric=None, p\_dist=None, seed=None)

Essa função retorna um grafo baseado em um modelo de limiar geográfico, onde os nós são posicionados em um espaço euclidiano e as arestas são adicionadas entre nós que estão a uma distância menor ou igual a um limite (theta).

:

import networkx as nx

import matplotlib.pyplot as plt

G = nx.geographical\_threshold\_graph(50, 0.2)

pos = nx.get\_node\_attributes(G, 'pos')

nx.draw(G, pos, node\_size=50, node\_color='blue', with\_labels=False)

plt.show()

geometric

Essa é uma submódulo do NetworkX que contém funções relacionadas à geometria dos grafos, como cálculo de distâncias, posições dos nós, entre outros.

:

import networkx as nx

G = nx.cycle\_graph(4)

distance = nx.geometric.euclidean\_distance((0, 0), (3, 4))

print(distance)

Espero que estas informações sejam úteis! Se você tiver mais perguntas ou precisar de mais s, não hesite em perguntar.

User

O mesmo para: 'geometric\_edges',

'get\_edge\_attributes',

'get\_node\_attributes',

'gexf',

'global\_efficiency',

'global\_parameters',

'global\_reaching\_centrality',

'gml',

'gn\_graph',

'gnc\_graph',

ChatGPT

Continuando a lista de funções do NetworkX:

geometric\_edges(G, pos, radius=1.0)

Essa função retorna uma lista de tuplas representando as arestas em um grafo cujas distâncias euclidianas são menores ou iguais ao raio fornecido.

:

import networkx as nx

G = nx.cycle\_graph(4)

pos = nx.circular\_layout(G)

edges\_within\_radius = nx.geometric\_edges(G, pos, radius=0.5)

print(edges\_within\_radius)

get\_edge\_attributes(G, name)

Essa função retorna um dicionário de atributos das arestas do grafo G com o nome fornecido.

:

import networkx as nx

G = nx.cycle\_graph(4)

G[0][1]['weight'] = 5

G[1][2]['weight'] = 10

edge\_attributes = nx.get\_edge\_attributes(G, 'weight')

print(edge\_attributes)

get\_node\_attributes(G, name)

Essa função retorna um dicionário de atributos dos nós do grafo G com o nome fornecido.

:

import networkx as nx

G = nx.cycle\_graph(4)

G.nodes[0]['color'] = 'red'

G.nodes[1]['color'] = 'blue'

node\_attributes = nx.get\_node\_attributes(G, 'color')

print(node\_attributes)

gexf

Essa é uma submódulo do NetworkX que fornece funções para ler e gravar grafos em formato GEXF (Graph Exchange XML Format), que é um formato de arquivo usado para representar grafos.

:

import networkx as nx

G = nx.cycle\_graph(4)

nx.write\_gexf(G, "graph.gexf")

global\_efficiency(G, weight=None)

Essa função retorna a eficiência global de um grafo, que é uma medida da capacidade de comunicação do grafo como um todo.

:

import networkx as nx

G = nx.cycle\_graph(4)

efficiency = nx.global\_efficiency(G)

print(efficiency)

global\_parameters(G, weight=None)

Essa função retorna um dicionário contendo várias medidas de centralidade global para o grafo.

:

import networkx as nx

G = nx.cycle\_graph(4)

parameters = nx.global\_parameters(G)

print(parameters)

global\_reaching\_centrality(G, cutoff=None)

Essa função retorna a centralidade de alcance global dos nós no grafo. A centralidade de alcance global de um nó é o número de outros nós que podem ser alcançados a partir desse nó em até um determinado número de etapas (cutoff).

:

import networkx as nx

G = nx.cycle\_graph(4)

reaching\_centrality = nx.global\_reaching\_centrality(G)

print(reaching\_centrality)

gml

Essa é uma submódulo do NetworkX que fornece funções para ler e gravar grafos em formato GML (Graph Modeling Language), que é um formato de arquivo para representação de grafos.

:

import networkx as nx

G = nx.cycle\_graph(4)

nx.write\_gml(G, "graph.gml")

gn\_graph(n, kernel=lambda x: (x + 1)\*\*-1, create\_using=None, seed=None)

Essa função retorna um grafo da hierarquia de Girvan-Newman.

:

import networkx as nx

G = nx.gn\_graph(5)

print(G.edges())

gnc\_graph(n, create\_using=None, seed=None)

Essa função retorna um grafo de Gilbert randomizado por cluster.

:

import networkx as nx

G = nx.gnc\_graph(10)

print(G.edges())

Espero que estas informações sejam úteis! Se você tiver mais perguntas ou precisar de mais s, não hesite em perguntar.

User

O mesmo para: 'gnm\_random\_graph',

'gnp\_random\_graph',

'gnr\_graph',

'goldberg\_radzik',

'gomory\_hu\_tree',

'google\_matrix',

'graph',

'graph6',

'graph\_atlas',

'graph\_atlas\_g',

ChatGPT

Continuando a lista de funções do NetworkX:

gnm\_random\_graph(n, m, seed=None, directed=False)

Essa função retorna um grafo aleatório com n nós e m arestas, usando o modelo G(n, m) (Erdős-Rényi) para grafos não direcionados e o modelo D(n, m) para grafos direcionados.

:

import networkx as nx

G = nx.gnm\_random\_graph(6, 10)

print(G.edges())

gnp\_random\_graph(n, p, seed=None, directed=False)

Essa função retorna um grafo aleatório com n nós, onde cada par de nós é conectado por uma aresta com probabilidade p. Para grafos não direcionados, cada possível aresta é adicionada independentemente com probabilidade p. Para grafos direcionados, cada possível arco é adicionado independentemente com probabilidade p.

:

import networkx as nx

G = nx.gnp\_random\_graph(6, 0.3)

print(G.edges())

gnr\_graph(n, radius, seed=None)

Essa função retorna um grafo aleatório com n nós gerado pelo modelo G(n, r), onde os nós são dispostos aleatoriamente dentro de um círculo de raio r e cada par de nós é conectado por uma aresta se a distância entre eles for menor ou igual a r.

:

import networkx as nx

G = nx.gnr\_graph(10, 0.3)

print(G.edges())

goldberg\_radzik(G, demand, capacity='capacity', weight='weight')

Essa função encontra o fluxo máximo em uma rede usando o algoritmo de Goldberg-Radzik.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edge('A', 'B', capacity=5)

G.add\_edge('A', 'C', capacity=3)

G.add\_edge('B', 'D', capacity=4)

G.add\_edge('C', 'D', capacity=2)

flow\_value, flow\_dict = nx.goldberg\_radzik(G, demand='A', capacity='capacity', weight=None)

print(flow\_value)

print(flow\_dict)

gomory\_hu\_tree(G, flow\_func=None, capacity='capacity')

Essa função retorna a árvore de Gomory-Hu de um grafo usando um algoritmo de fluxo máximo.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edge('A', 'B', capacity=5)

G.add\_edge('A', 'C', capacity=3)

G.add\_edge('B', 'D', capacity=4)

G.add\_edge('C', 'D', capacity=2)

tree = nx.gomory\_hu\_tree(G, capacity='capacity')

print(tree.edges())

google\_matrix(G, alpha=0.85, personalization=None, nodelist=None, weight='weight', dangling=None)

Essa função retorna a matriz do Google (matriz de PageRank) para um grafo. É uma matriz estocástica que modela a probabilidade de um random walker passar de um nó para outro.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edge('A', 'B', weight=0.5)

G.add\_edge('A', 'C', weight=0.5)

G.add\_edge('B', 'D', weight=0.5)

G.add\_edge('C', 'D', weight=0.5)

matrix = nx.google\_matrix(G, alpha=0.85, weight='weight')

print(matrix)

graph

Esse é um submódulo do NetworkX que contém classes e funções para trabalhar com grafos em geral.

graph6

Esse é um submódulo do NetworkX que fornece funções para ler e gravar grafos em formato graph6, que é um formato compacto para representação de grafos.

graph\_atlas

Esse é um submódulo do NetworkX que fornece uma lista de todos os grafos rotulados e não rotulados com até 11 nós.

graph\_atlas\_g

Esse é um submódulo do NetworkX que fornece uma lista de todos os grafos não rotulados com até 7 nós.

Espero que estas informações sejam úteis! Se você tiver mais perguntas ou precisar de mais s, não hesite em perguntar.

User

O mesmo para: 'graph\_clique\_number',

'graph\_edit\_distance',

'graph\_hashing',

'graph\_number\_of\_cliques',

'graphical',

'graphmatrix',

'graphml',

'graphviews',

'greedy\_color',

'grid\_2d\_graph',

ChatGPT

Continuando a lista de funções do NetworkX:

graph\_clique\_number(G, cliques=None)

Essa função retorna o número de cliques de um tamanho máximo em um grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (2, 4), (3, 4)])

clique\_number = nx.graph\_clique\_number(G)

print(clique\_number)

graph\_edit\_distance(G1, G2, node\_match=None, edge\_match=None)

Essa função calcula a distância de edição (edit distance) entre dois grafos. A distância de edição é o número mínimo de operações necessárias para transformar um grafo no outro. As operações permitidas são: inserção, remoção e substituição de nós ou arestas.

:

import networkx as nx

G1 = nx.Graph()

G1.add\_edges\_from([(1, 2), (1, 3), (2, 3)])

G2 = nx.Graph()

G2.add\_edges\_from([(1, 2), (2, 3)])

distance = nx.graph\_edit\_distance(G1, G2)

print(distance)

graph\_hashing(G, method='tried-and-true')

Essa função calcula uma assinatura hash de um grafo usando um dos métodos disponíveis.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3)])

hash\_value = nx.graph\_hashing(G)

print(hash\_value)

graph\_number\_of\_cliques(G, cliques=None)

Essa função retorna o número de cliques em um grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (2, 4), (3, 4)])

num\_cliques = nx.graph\_number\_of\_cliques(G)

print(num\_cliques)

graphical(sequence)

Essa função verifica se uma sequência de graus é gráfica, ou seja, se ela pode ser o grau de um grafo válido.

:

import networkx as nx

sequence = [3, 3, 2, 2]

is\_graphical = nx.graphical(sequence)

print(is\_graphical) # True

graphmatrix

Esse é um submódulo do NetworkX que contém funções para calcular matrizes de grafos.

graphml

Esse é um submódulo do NetworkX que fornece funções para ler e gravar grafos em formato GraphML.

graphviews

Esse é um submódulo do NetworkX que contém classes para criar visualizações (views) de grafos com restrições, como subconjuntos de nós ou arestas.

greedy\_color(G, strategy='largest\_first', interchange=False)

Essa função colore os nós de um grafo usando o algoritmo de coloração guloso. O parâmetro "strategy" define a estratégia de seleção do próximo nó a ser colorido, e "interchange" indica se a coloração pode ser trocada por cores melhores.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (2, 4), (3, 4)])

coloring = nx.greedy\_color(G)

print(coloring)

grid\_2d\_graph(m, n, periodic=False, create\_using=None)

Essa função retorna um grafo de grade 2D com m linhas e n colunas. O grafo é criado conectando cada nó a seus vizinhos imediatos na horizontal e vertical.

:

import networkx as nx

G = nx.grid\_2d\_graph(3, 3)

print(G.edges())

Espero que estas informações sejam úteis! Se você tiver mais perguntas ou precisar de mais s, não hesite em perguntar.

User

O mesmo para: 'grid\_graph',

'group',

'group\_betweenness\_centrality',

'group\_closeness\_centrality',

'group\_degree\_centrality',

'group\_in\_degree\_centrality',

'group\_out\_degree\_centrality',

'harmonic',

'harmonic\_centrality',

'has\_bridges',

ChatGPT

Continuando com a lista de funções do NetworkX:

grid\_graph(dim, periodic=False, create\_using=None)

Essa função retorna um grafo de grade de dimensões dim, onde dim é uma sequência de números inteiros representando o tamanho das dimensões da grade.

:

import networkx as nx

G = nx.grid\_graph(dim=[3, 4])

print(G.edges())

group(G, nodes)

Essa função retorna o subgrafo de G induzido pelo grupo de nós especificado.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (2, 4), (3, 4)])

group\_nodes = [1, 2, 3]

subgraph = nx.group(G, group\_nodes)

print(subgraph.edges())

group\_betweenness\_centrality(G, groups, normalized=True, weight=None, endpoints=False, seed=None)

Essa função calcula a centralidade de intermediação do grupo em um grafo, considerando grupos de nós específicos. A centralidade de intermediação do grupo é definida como a soma das centralidades de intermediação de todos os pares de nós no grupo.

:

import networkx as nx

G = nx.karate\_club\_graph()

groups = [0, 1, 2, 3, 4, 5]

betweenness = nx.group\_betweenness\_centrality(G, groups)

print(betweenness)

group\_closeness\_centrality(G, groups, weight=None, distance=None)

Essa função calcula a centralidade de proximidade do grupo em um grafo, considerando grupos de nós específicos. A centralidade de proximidade do grupo é definida como o inverso da soma das distâncias mais curtas entre um nó do grupo e todos os outros nós do mesmo grupo.

:

import networkx as nx

G = nx.karate\_club\_graph()

groups = [0, 1, 2, 3, 4, 5]

closeness = nx.group\_closeness\_centrality(G, groups)

print(closeness)

group\_degree\_centrality(G, groups)

Essa função calcula a centralidade de grau do grupo em um grafo, considerando grupos de nós específicos. A centralidade de grau do grupo é definida como a soma dos graus de todos os nós no grupo.

:

import networkx as nx

G = nx.karate\_club\_graph()

groups = [0, 1, 2, 3, 4, 5]

degree\_centrality = nx.group\_degree\_centrality(G, groups)

print(degree\_centrality)

group\_in\_degree\_centrality(G, groups)

Essa função calcula a centralidade de grau de entrada do grupo em um grafo direcionado, considerando grupos de nós específicos. A centralidade de grau de entrada do grupo é definida como a soma dos graus de entrada de todos os nós no grupo.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (2, 4), (3, 4)])

groups = [1, 2, 3]

in\_degree\_centrality = nx.group\_in\_degree\_centrality(G, groups)

print(in\_degree\_centrality)

group\_out\_degree\_centrality(G, groups)

Essa função calcula a centralidade de grau de saída do grupo em um grafo direcionado, considerando grupos de nós específicos. A centralidade de grau de saída do grupo é definida como a soma dos graus de saída de todos os nós no grupo.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (2, 4), (3, 4)])

groups = [1, 2, 3]

out\_degree\_centrality = nx.group\_out\_degree\_centrality(G, groups)

print(out\_degree\_centrality)

harmonic(G, nbunch=None, distance='distance')

Essa função retorna a média harmônica da distância entre os nós no grafo. A média harmônica é a inversa da média aritmética das inversas das distâncias.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3)])

avg\_harmonic\_distance = nx.harmonic(G)

print(avg\_harmonic\_distance)

harmonic\_centrality(G, nbunch=None, distance='distance')

Essa função calcula a centralidade harmônica para os nós do grafo. A centralidade harmônica de um nó é a soma das distâncias harmônicas entre esse nó e todos os outros nós no grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3)])

harmonic\_centrality = nx.harmonic\_centrality(G)

print(harmonic\_centrality)

has\_bridges(G)

Essa função verifica se o grafo contém arestas de corte (pontes), ou seja, arestas cuja remoção aumenta o número de componentes conectados.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

has\_bridges = nx.has\_bridges(G)

print(has\_bridges)

Espero que estas informações sejam úteis! Se você tiver mais perguntas ou precisar de mais s, não hesite em perguntar.

User

O mesmo para: 'has\_eulerian\_path',

'has\_path',

'havel\_hakimi\_graph',

'heawood\_graph',

'hexagonal\_lattice\_graph',

'hierarchy',

'hits',

'hits\_alg',

'hoffman\_singleton\_graph',

'house\_graph',

ChatGPT

Continuando com a lista de funções do NetworkX:

has\_eulerian\_path(G)

Essa função verifica se o grafo possui um caminho euleriano. Um caminho euleriano é um caminho que visita cada aresta do grafo exatamente uma vez.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

has\_eulerian\_path = nx.has\_eulerian\_path(G)

print(has\_eulerian\_path)

has\_path(G, source, target)

Essa função verifica se existe um caminho entre os nós source e target no grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

has\_path\_1 = nx.has\_path(G, 1, 4)

has\_path\_2 = nx.has\_path(G, 1, 5)

print(has\_path\_1) # True

print(has\_path\_2) # False

havel\_hakimi\_graph(sequence, create\_using=None)

Essa função retorna um grafo simples criado usando o processo de construção de grafo de Havel-Hakimi a partir de uma sequência de graus.

:

import networkx as nx

sequence = [4, 3, 3, 2, 1]

G = nx.havel\_hakimi\_graph(sequence)

print(G.edges())

heawood\_graph()

Essa função retorna o grafo de Heawood, que é um grafo cúbico não direcionado com 14 vértices e 21 arestas.

:

import networkx as nx

G = nx.heawood\_graph()

print(G.edges())

hexagonal\_lattice\_graph(m, n, periodic=False, with\_positions=True, create\_using=None)

Essa função retorna um grafo de grade hexagonal de tamanho m x n. Os parâmetros m e n determinam o número de células hexagonais na direção x e y, respectivamente.

:

import networkx as nx

G = nx.hexagonal\_lattice\_graph(3, 3)

print(G.edges())

hierarchy(G, root=None, create\_using=None)

Essa função retorna uma hierarquia dirigida de um grafo não direcionado.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (3, 4), (4, 5), (5, 3)])

hierarchy\_graph = nx.hierarchy(G)

print(hierarchy\_graph.edges())

hits(G, max\_iter=100, tol=1e-06, nstart=None, normalized=True)

Essa função calcula os valores de hubs e autoridades para cada nó em um grafo, usando o algoritmo HITS (Hypertext Induced Topic Search).

:

import networkx as nx

G = nx.karate\_club\_graph()

hubs, authorities = nx.hits(G)

print("Hubs:", hubs)

print("Authorities:", authorities)

hits\_alg(G, max\_iter=100, tol=1e-06, normalized=True)

Essa função também calcula os valores de hubs e autoridades usando o algoritmo HITS, mas retorna um dicionário contendo os valores para cada nó.

:

import networkx as nx

G = nx.karate\_club\_graph()

hits\_scores = nx.hits\_alg(G)

print(hits\_scores)

hoffman\_singleton\_graph()

Essa função retorna o grafo de Hoffman-Singleton, que é um grafo cúbico não direcionado com 50 vértices e 75 arestas.

:

import networkx as nx

G = nx.hoffman\_singleton\_graph()

print(G.edges())

house\_graph(create\_using=None)

Essa função retorna o grafo casa, que é um grafo cúbico não direcionado com 5 vértices (representando a forma de uma casa) e 7 arestas.

:

import networkx as nx

G = nx.house\_graph()

print(G.edges())

Espero que estas informações sejam úteis! Se você tiver mais perguntas ou precisar de mais s, não hesite em perguntar.

User

O mesmo para: 'house\_x\_graph',

'hybrid',

'hypercube\_graph',

'icosahedral\_graph',

'identified\_nodes',

'immediate\_dominators',

'in\_degree\_centrality',

'incidence\_matrix',

'incremental\_closeness\_centrality',

'induced\_subgraph',

ChatGPT

Claro! Continuando com a lista de funções do NetworkX:

house\_x\_graph()

Essa função retorna o grafo casa X, que é um grafo cúbico não direcionado com 6 vértices e 9 arestas.

:

import networkx as nx

G = nx.house\_x\_graph()

print(G.edges())

hybrid(G1, G2)

Essa função combina dois grafos G1 e G2 em um novo grafo, chamado de grafo híbrido, que contém os nós de G1 e G2 e todas as arestas de ambos os grafos.

:

import networkx as nx

G1 = nx.Graph()

G1.add\_edges\_from([(1, 2), (2, 3)])

G2 = nx.Graph()

G2.add\_edges\_from([(3, 4), (4, 5)])

G\_hybrid = nx.hybrid(G1, G2)

print(G\_hybrid.edges())

hypercube\_graph(n)

Essa função retorna o grafo hipercubo com n dimensões. O grafo hipercubo de n dimensões possui 2^n vértices e cada vértice tem n vizinhos.

:

import networkx as nx

G = nx.hypercube\_graph(3) # Grafo hipercubo de 3 dimensões

print(G.edges())

icosahedral\_graph()

Essa função retorna o grafo icosaedral, que é um grafo não direcionado com 12 vértices e 30 arestas.

:

import networkx as nx

G = nx.icosahedral\_graph()

print(G.edges())

identified\_nodes(G, id\_dict, relabel=False, copy=True)

Essa função retorna um grafo com os nós identificados de acordo com o dicionário id\_dict.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

id\_dict = {1: 'A', 2: 'B', 3: 'C'}

G\_identified = nx.identified\_nodes(G, id\_dict)

print(G\_identified.nodes())

immediate\_dominators(G, start)

Essa função retorna um dicionário que mapeia cada nó no grafo G ao seu imediato dominador no grafo de dominadores do nó a partir do nó start.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (2, 4), (3, 5), (4, 5), (5, 6)])

immediate\_dominators = nx.immediate\_dominators(G, start=1)

print(immediate\_dominators)

in\_degree\_centrality(G)

Essa função calcula a centralidade de grau de entrada para cada nó do grafo dirigido G.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (2, 4), (3, 5), (4, 5), (5, 6)])

in\_degree\_centrality = nx.in\_degree\_centrality(G)

print(in\_degree\_centrality)

incidence\_matrix(G, nodelist=None, edgelist=None, oriented=False, weight=None)

Essa função retorna a matriz de incidência do grafo G com relação à lista de nós e arestas especificada. A matriz de incidência é uma matriz que descreve a relação entre os nós e as arestas do grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

incidence\_matrix = nx.incidence\_matrix(G)

print(incidence\_matrix.toarray())

incremental\_closeness\_centrality(G, u, distance=None)

Essa função calcula a centralidade de proximidade incremental para um nó u em um grafo G. A centralidade de proximidade é uma medida de quão central é um nó no grafo, baseada na soma das distâncias mais curtas de um nó para todos os outros nós no grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

incremental\_closeness = nx.incremental\_closeness\_centrality(G, 1)

print(incremental\_closeness)

induced\_subgraph(G, nbunch)

Essa função retorna o subgrafo induzido pelo conjunto de nós nbunch.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 5)])

subgraph = nx.induced\_subgraph(G, [2, 3, 4])

print(subgraph.edges())

Espero que esta lista seja útil para você! Se você tiver mais perguntas ou precisar de mais s, sinta-se à vontade para perguntar.

User

O mesmo para: 'information\_centrality',

'internet\_as\_graphs',

'intersection',

'intersection\_all',

'intersection\_array',

'interval\_graph',

'inverse\_line\_graph',

'is\_aperiodic',

'is\_arborescence',

'is\_at\_free',

ChatGPT

Claro! Continuando com a lista de funções do NetworkX:

information\_centrality(G)

Essa função calcula a centralidade de informação para cada nó do grafo G. A centralidade de informação mede a influência de um nó em um grafo, com base na teoria da transferência de calor.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 5)])

information\_centrality = nx.information\_centrality(G)

print(information\_centrality)

internet\_as\_graphs(as\_pairs=False)

Essa função retorna os grafos da internet disponíveis no NetworkX. Se as\_pairs for True, retorna uma lista de pares (nome, grafo). Caso contrário, retorna um dicionário onde as chaves são os nomes dos grafos e os valores são os próprios grafos.

:

import networkx as nx

internet\_graphs = nx.internet\_as\_graphs(as\_pairs=True)

print(internet\_graphs)

intersection(G, H)

Essa função retorna o grafo que contém os nós que estão presentes em ambos os grafos G e H, e as arestas que estão presentes em ambos os grafos.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

H = nx.Graph()

H.add\_edges\_from([(3, 4), (4, 5), (5, 6)])

intersection\_graph = nx.intersection(G, H)

print(intersection\_graph.edges())

intersection\_all(graphs)

Essa função retorna o grafo que contém os nós que estão presentes em todos os grafos na lista graphs, e as arestas que estão presentes em todos os grafos.

:

import networkx as nx

G1 = nx.Graph()

G1.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

G2 = nx.Graph()

G2.add\_edges\_from([(3, 4), (4, 5), (5, 6)])

G3 = nx.Graph()

G3.add\_edges\_from([(4, 5), (5, 6), (6, 7)])

intersection\_graph = nx.intersection\_all([G1, G2, G3])

print(intersection\_graph.edges())

intersection\_array(graphs)

Essa função retorna o grafo que contém os nós que estão presentes em pelo menos dois grafos na lista graphs, e as arestas que estão presentes em pelo menos dois grafos.

:

import networkx as nx

G1 = nx.Graph()

G1.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

G2 = nx.Graph()

G2.add\_edges\_from([(3, 4), (4, 5), (5, 6)])

G3 = nx.Graph()

G3.add\_edges\_from([(4, 5), (5, 6), (6, 7)])

intersection\_graph = nx.intersection\_array([G1, G2, G3])

print(intersection\_graph.edges())

interval\_graph(p)

Essa função retorna o grafo de intervalo para uma sequência de intervalos p. Um grafo de intervalo é um grafo onde os intervalos correspondem aos nós e as interseções não vazias entre os intervalos correspondem às arestas.

:

import networkx as nx

p = [(1, 4), (2, 6), (3, 5)]

interval\_graph = nx.interval\_graph(p)

print(interval\_graph.edges())

inverse\_line\_graph(L)

Essa função retorna o grafo original cujo grafo da linha é L.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

L = nx.line\_graph(G)

inverse\_line\_graph = nx.inverse\_line\_graph(L)

print(inverse\_line\_graph.edges())

is\_aperiodic(G)

Essa função verifica se o grafo G é aperiódico. Um grafo é aperiódico se não existirem ciclos de comprimento maior que 2 que possuam todos os seus vértices com um período maior que 2.

:

import networkx as nx

G = nx.cycle\_graph(5) # Grafo cíclico de 5 vértices

is\_aperiodic = nx.is\_aperiodic(G)

print(is\_aperiodic)

is\_arborescence(G)

Essa função verifica se o grafo direcionado G é uma arborescência. Uma arborescência é um grafo direcionado no qual existe um único nó com grau de entrada igual a 0 (raiz) e todos os demais nós têm exatamente um pai.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 4), (2, 5), (3, 6)])

is\_arborescence = nx.is\_arborescence(G)

print(is\_arborescence)

is\_at\_free(G)

Essa função verifica se o grafo G é AT-livre, ou seja, não contém nenhum grafo completo bipartido K\_{m, n} (clique bipartido completo) como subgrafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (1, 4), (2, 5), (2, 6), (2, 7)])

is\_at\_free = nx.is\_at\_free(G)

print(is\_at\_free)

Espero que esta lista seja útil para você! Se você tiver mais perguntas ou precisar de mais s, sinta-se à vontade para perguntar.

User

O mesmo para: 'is\_attracting\_component',

'is\_biconnected',

'is\_bipartite',

'is\_branching',

'is\_chordal',

'is\_connected',

'is\_digraphical',

'is\_directed',

'is\_directed\_acyclic\_graph',

'is\_distance\_regular',

ChatGPT

Claro! Continuando a lista de funções do NetworkX:

is\_attracting\_component(G, subset)

Esta função verifica se o conjunto de nós subset é uma componente atrativa do grafo G. Uma componente atrativa é uma componente fortemente conexa que atrai todas as arestas externas. Em outras palavras, se você iniciar em qualquer nó dentro da componente atrativa e seguir as arestas dirigidas, eventualmente você alcançará todos os nós em subset.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (4, 5), (5, 6), (6, 4)])

subset = {1, 2, 3}

is\_attracting = nx.is\_attracting\_component(G, subset)

print(is\_attracting)

is\_biconnected(G)

Esta função verifica se o grafo G é biconexo, ou seja, não possui pontos de articulação. Um grafo é biconexo se, para qualquer par de nós, sempre existem pelo menos dois caminhos disjuntos que os conectam.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (3, 4)])

is\_biconnected = nx.is\_biconnected(G)

print(is\_biconnected)

is\_bipartite(G)

Esta função verifica se o grafo G é bipartido, ou seja, seus nós podem ser divididos em dois conjuntos disjuntos (X e Y) de tal forma que todas as arestas conectam um nó em X a um nó em Y.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 3), (2, 3), (2, 4)])

is\_bipartite = nx.is\_bipartite(G)

print(is\_bipartite)

is\_branching(G)

Esta função verifica se o grafo dirigido G é uma árvore enraizada, ou seja, um grafo dirigido onde existe um único nó sem nenhuma aresta de entrada e todos os outros nós têm exatamente uma aresta de entrada.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 4), (2, 5)])

is\_branching = nx.is\_branching(G)

print(is\_branching)

is\_chordal(G)

Esta função verifica se o grafo G é um grafo cordal, ou seja, um grafo no qual todos os ciclos de quatro ou mais nós têm uma corda (uma aresta que não é parte do ciclo, mas que conecta dois nós do ciclo).

:

import networkx as nx

G = nx.cycle\_graph(5) # Grafo cíclico de 5 nós

is\_chordal = nx.is\_chordal(G)

print(is\_chordal)

is\_connected(G)

Esta função verifica se o grafo G é conexo, ou seja, existe pelo menos um caminho entre cada par de nós.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

is\_connected = nx.is\_connected(G)

print(is\_connected)

is\_digraphical(sequence)

Esta função verifica se a sequência de graus sequence pode ser realizada por um grafo simples. Uma sequência de graus é digráfica se a soma dos graus for par e se existir um grafo direcionado associado a ela.

:

import networkx as nx

sequence = [3, 3, 1, 1, 1]

is\_digraphical = nx.is\_digraphical(sequence)

print(is\_digraphical)

is\_directed(G)

Esta função verifica se o grafo G é direcionado, ou seja, é um grafo dirigido.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

is\_directed = nx.is\_directed(G)

print(is\_directed)

is\_directed\_acyclic\_graph(G)

Esta função verifica se o grafo direcionado G é um grafo direcionado acíclico (DAG), ou seja, um grafo direcionado que não contém nenhum ciclo.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

is\_dag = nx.is\_directed\_acyclic\_graph(G)

print(is\_dag)

is\_distance\_regular(G)

Esta função verifica se o grafo G é um grafo de distância regular, ou seja, um grafo onde o número de vizinhos comuns a dois nós é constante, independentemente dos nós escolhidos.

:

import networkx as nx

G = nx.cycle\_graph(5) # Grafo cíclico de 5 nós

is\_distance\_regular = nx.is\_distance\_regular(G)

print(is\_distance\_regular)

Espero que esta lista seja útil para você! Se você tiver mais perguntas ou precisar de mais s, sinta-se à vontade para perguntar.

User

O mesmo para: 'is\_dominating\_set',

'is\_edge\_cover',

'is\_empty',

'is\_eulerian',

'is\_forest',

'is\_frozen',

'is\_graphical',

'is\_isolate',

'is\_isomorphic',

'is\_k\_edge\_connected',

ChatGPT

Claro! Continuando a lista de funções do NetworkX:

is\_dominating\_set(G, dominating\_set)

Esta função verifica se o conjunto de nós dominating\_set é um conjunto dominante no grafo G. Um conjunto dominante é um conjunto de nós tal que cada nó no grafo pertence ao conjunto ou tem um vizinho que pertence ao conjunto.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 4), (3, 5)])

dominating\_set = {1, 2}

is\_dominating = nx.is\_dominating\_set(G, dominating\_set)

print(is\_dominating)

is\_edge\_cover(G, edge\_cover)

Esta função verifica se o conjunto de arestas edge\_cover é uma cobertura de arestas no grafo G. Uma cobertura de arestas é um conjunto de arestas que cobre todos os nós do grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (3, 4)])

edge\_cover = [(1, 2), (3, 4)]

is\_cover = nx.is\_edge\_cover(G, edge\_cover)

print(is\_cover)

is\_empty(G)

Esta função verifica se o grafo G está vazio, ou seja, não possui nenhum nó ou aresta.

:

import networkx as nx

G = nx.Graph()

is\_empty = nx.is\_empty(G)

print(is\_empty)

is\_eulerian(G)

Esta função verifica se o grafo G é euleriano, ou seja, contém um circuito euleriano, que é um circuito que passa por cada aresta exatamente uma vez.

:

import networkx as nx

G = nx.complete\_graph(4) # Grafo completo de 4 nós

is\_eulerian = nx.is\_eulerian(G)

print(is\_eulerian)

is\_forest(G)

Esta função verifica se o grafo G é uma floresta, ou seja, é um grafo acíclico (não possui ciclos).

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (3, 4)])

is\_forest = nx.is\_forest(G)

print(is\_forest)

is\_frozen(G)

Esta função verifica se o grafo G é congelado, ou seja, não é possível modificar sua estrutura adicionando ou removendo nós ou arestas.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3)])

G = nx.freeze(G)

is\_frozen = nx.is\_frozen(G)

print(is\_frozen)

is\_graphical(sequence)

Esta função verifica se a sequência de graus sequence é uma sequência gráfica, ou seja, se existe um grafo simples com essa sequência de graus.

:

import networkx as nx

sequence = [3, 3, 1, 1, 1]

is\_graphical = nx.is\_graphical(sequence)

print(is\_graphical)

is\_isolate(G, node)

Esta função verifica se o nó node é um isolado no grafo G, ou seja, é um nó sem nenhuma aresta conectando-o a outros nós.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3)])

is\_isolate = nx.is\_isolate(G, 3)

print(is\_isolate)

is\_isomorphic(G1, G2)

Esta função verifica se os grafos G1 e G2 são isomórficos, ou seja, se eles são grafos que possuem a mesma estrutura, apenas com os nós reorganizados.

:

import networkx as nx

G1 = nx.Graph()

G1.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

G2 = nx.Graph()

G2.add\_edges\_from([(3, 2), (1, 3), (2, 1)])

is\_isomorphic = nx.is\_isomorphic(G1, G2)

print(is\_isomorphic)

is\_k\_edge\_connected(G, k)

Esta função verifica se o grafo G é k-conexo, ou seja, é um grafo tal que é necessário remover pelo menos k arestas para desconectá-lo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (3, 4)])

k = 2

is\_k\_edge\_connected = nx.is\_k\_edge\_connected(G, k)

print(is\_k\_edge\_connected)

Espero que esta lista seja útil para você! Se você tiver mais perguntas ou precisar de mais s, sinta-se à vontade para perguntar.

User

O mesmo para: 'is\_k\_regular',

'is\_kl\_connected',

'is\_matching',

'is\_maximal\_matching',

'is\_minimal\_d\_separator',

'is\_multigraphical',

'is\_negatively\_weighted',

'is\_path',

'is\_perfect\_matching',

'is\_planar',

ChatGPT

Claro! Continuando a lista de funções do NetworkX:

is\_k\_regular(G, k)

Esta função verifica se o grafo G é k-regular, ou seja, todos os nós do grafo têm grau k.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)])

k = 2

is\_k\_regular = nx.is\_k\_regular(G, k)

print(is\_k\_regular)

is\_kl\_connected(G, k, l)

Esta função verifica se o grafo G é (k, l)-conexo, ou seja, é um grafo tal que, ao remover qualquer conjunto de nós com tamanho k ou menos, o grafo resultante ainda é conexo, e, ao remover qualquer conjunto de nós com tamanho l ou menos, o grafo resultante não é mais conexo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4), (4, 5)])

k = 1

l = 2

is\_kl\_connected = nx.is\_kl\_connected(G, k, l)

print(is\_kl\_connected)

is\_matching(G, matching)

Esta função verifica se o conjunto de arestas matching é um conjunto de arestas independentes (matching) no grafo G. Um conjunto de arestas é independente se nenhuma das arestas compartilha um nó com outra aresta do conjunto.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

matching = [(1, 2), (3, 4)]

is\_matching = nx.is\_matching(G, matching)

print(is\_matching)

is\_maximal\_matching(G, matching)

Esta função verifica se o conjunto de arestas matching é um conjunto de arestas independentes (matching) maximal no grafo G. Um conjunto de arestas é maximal se não é possível adicionar mais arestas independentes a ele.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 5)])

matching = [(1, 2), (3, 4)]

is\_maximal\_matching = nx.is\_maximal\_matching(G, matching)

print(is\_maximal\_matching)

is\_minimal\_d\_separator(G, separator, d)

Esta função verifica se o conjunto de nós separator é um d-separador minimal no grafo G. Um d-separador minimal é um conjunto de nós que separa dois outros nós do grafo em uma distância de pelo menos d.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

separator = {1, 3}

d = 2

is\_minimal\_separator = nx.is\_minimal\_d\_separator(G, separator, d)

print(is\_minimal\_separator)

is\_multigraphical(sequence)

Esta função verifica se a sequência de graus sequence é uma sequência multigráfica, ou seja, se existe um multigrafo com essa sequência de graus.

:

import networkx as nx

sequence = [3, 3, 1, 1, 1]

is\_multigraphical = nx.is\_multigraphical(sequence)

print(is\_multigraphical)

is\_negatively\_weighted(G, weight='weight')

Esta função verifica se o grafo G possui arestas com pesos negativos. A função usa o atributo weight nas arestas para determinar o peso.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=3)

G.add\_edge(2, 3, weight=-1)

is\_negatively\_weighted = nx.is\_negatively\_weighted(G, weight='weight')

print(is\_negatively\_weighted)

is\_path(G, nodes)

Esta função verifica se a lista de nós nodes forma um caminho no grafo G, ou seja, se cada par de nós consecutivos na lista é conectado por uma aresta no grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

nodes = [1, 2, 3, 4]

is\_path = nx.is\_path(G, nodes)

print(is\_path)

is\_perfect\_matching(G, matching)

Esta função verifica se o conjunto de arestas matching é um conjunto de arestas independentes (matching) perfeito no grafo G. Um conjunto de arestas é perfeito se abrange todos os nós do grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

matching = [(1, 2), (3, 4)]

is\_perfect\_matching = nx.is\_perfect\_matching(G, matching)

print(is\_perfect\_matching)

is\_planar(G)

Esta função verifica se o grafo G é planar, ou seja, pode ser desenhado em um plano sem que suas arestas se cruzem.

:

import networkx as nx

G = nx.complete\_graph(4) # Grafo completo de 4 nós

is\_planar = nx.is\_planar(G)

print(is\_planar)

Espero que esta lista de funções seja útil para você! Se tiver mais perguntas ou precisar de mais s, sinta-se à vontade para perguntar.

User

O mesmo para: 'is\_pseudographical',

'is\_regular',

'is\_semiconnected',

'is\_semieulerian',

'is\_simple\_path',

'is\_strongly\_connected',

'is\_strongly\_regular',

'is\_tree',

'is\_triad',

'is\_valid\_degree\_sequence\_erdos\_gallai',

ChatGPT

Claro! Continuando a lista de funções do NetworkX:

is\_pseudographical(G)

Esta função verifica se o grafo G é pseudográfico, ou seja, se possui laços ou múltiplas arestas paralelas.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2)

G.add\_edge(1, 2) # Adicionando uma aresta paralela

G.add\_edge(2, 2) # Adicionando um laço

is\_pseudographical = nx.is\_pseudographical(G)

print(is\_pseudographical)

is\_regular(G)

Esta função verifica se o grafo G é regular, ou seja, se todos os nós do grafo têm o mesmo grau.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

is\_regular = nx.is\_regular(G)

print(is\_regular)

is\_semiconnected(G)

Esta função verifica se o grafo dirigido G é semiconexo, ou seja, se existe um caminho direcionado entre todos os pares de nós do grafo.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

is\_semiconnected = nx.is\_semiconnected(G)

print(is\_semiconnected)

is\_semieulerian(G)

Esta função verifica se o grafo G é semieuleriano, ou seja, se possui um circuito euleriano (que percorre todas as arestas exatamente uma vez) ou um caminho euleriano (que percorre todas as arestas exatamente uma vez, mas não é um circuito fechado).

:

import networkx as nx

G = nx.Graph()

G.add\_cycle([1, 2, 3, 4])

G.add\_edge(4, 5)

is\_semieulerian = nx.is\_semieulerian(G)

print(is\_semieulerian)

is\_simple\_path(G, nodes)

Esta função verifica se a lista de nós nodes forma um caminho simples no grafo G, ou seja, se cada par de nós consecutivos na lista é conectado por uma única aresta no grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

nodes = [1, 2, 3, 4]

is\_simple\_path = nx.is\_simple\_path(G, nodes)

print(is\_simple\_path)

is\_strongly\_connected(G)

Esta função verifica se o grafo dirigido G é fortemente conectado, ou seja, se existe um caminho direcionado entre todos os pares de nós do grafo.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

is\_strongly\_connected = nx.is\_strongly\_connected(G)

print(is\_strongly\_connected)

is\_strongly\_regular(G, k, l, p, q)

Esta função verifica se o grafo G é fortemente regular com parâmetros k, l, p e q, o que significa que o grafo é regular, cada par de nós adjacentes tem k vizinhos em comum, cada par de nós não adjacentes tem l vizinhos em comum, e cada par de nós adjacentes tem p vizinhos não adjacentes em comum.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)])

k = 2

l = 0

p = 1

q = 1

is\_strongly\_regular = nx.is\_strongly\_regular(G, k, l, p, q)

print(is\_strongly\_regular)

is\_tree(G)

Esta função verifica se o grafo G é uma árvore, ou seja, um grafo não direcionado conexo sem ciclos.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (2, 4)])

is\_tree = nx.is\_tree(G)

print(is\_tree)

is\_triad(G, nodes)

Esta função verifica se a lista de nós nodes forma uma triade no grafo G, ou seja, um subgrafo de três nós em que pelo menos duas arestas estão presentes.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

nodes = [1, 2, 3]

is\_triad = nx.is\_triad(G, nodes)

print(is\_triad)

is\_valid\_degree\_sequence\_erdos\_gallai(sequence)

Esta função verifica se a sequência de graus sequence é uma sequência válida de graus para um grafo simples, usando o teorema de Erdős-Gallai.

:

import networkx as nx

sequence = [3, 2, 2, 1]

is\_valid\_degree\_sequence = nx.is\_valid\_degree\_sequence\_erdos\_gallai(sequence)

print(is\_valid\_degree\_sequence)

Espero que esta lista de funções seja útil para você! Se tiver mais perguntas ou precisar de mais s, sinta-se à vontade para perguntar.

User

O mesmo para: 'is\_valid\_degree\_sequence\_havel\_hakimi',

'is\_valid\_directed\_joint\_degree',

'is\_valid\_joint\_degree',

'is\_weakly\_connected',

'is\_weighted',

'isolate',

'isolates',

'isomorphism',

'jaccard\_coefficient',

'johnson',

ChatGPT

Claro! Continuando a lista de funções do NetworkX:

is\_valid\_degree\_sequence\_havel\_hakimi(sequence)

Esta função verifica se a sequência de graus sequence é uma sequência válida de graus para um grafo simples, usando o algoritmo de Havel-Hakimi.

:

import networkx as nx

sequence = [3, 2, 2, 1]

is\_valid\_degree\_sequence = nx.is\_valid\_degree\_sequence\_havel\_hakimi(sequence)

print(is\_valid\_degree\_sequence)

is\_valid\_directed\_joint\_degree(G, in\_degree\_sequence, out\_degree\_sequence)

Esta função verifica se a sequência de graus de entrada in\_degree\_sequence e a sequência de graus de saída out\_degree\_sequence são sequências válidas de graus para um grafo dirigido.

:

import networkx as nx

in\_degree\_sequence = [2, 1, 1, 0]

out\_degree\_sequence = [1, 1, 2, 0]

G = nx.DiGraph()

is\_valid\_directed\_joint\_degree = nx.is\_valid\_directed\_joint\_degree(G, in\_degree\_sequence, out\_degree\_sequence)

print(is\_valid\_directed\_joint\_degree)

is\_valid\_joint\_degree(G, in\_degree\_sequence, out\_degree\_sequence)

Esta função verifica se a sequência de graus de entrada in\_degree\_sequence e a sequência de graus de saída out\_degree\_sequence são sequências válidas de graus para um grafo dirigido e não dirigido.

:

import networkx as nx

in\_degree\_sequence = [2, 1, 1, 0]

out\_degree\_sequence = [1, 1, 2, 0]

G = nx.DiGraph()

is\_valid\_joint\_degree = nx.is\_valid\_joint\_degree(G, in\_degree\_sequence, out\_degree\_sequence)

print(is\_valid\_joint\_degree)

is\_weakly\_connected(G)

Esta função verifica se o grafo dirigido G é fracamente conexo, ou seja, se existe um caminho não direcionado entre todos os pares de nós do grafo.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

is\_weakly\_connected = nx.is\_weakly\_connected(G)

print(is\_weakly\_connected)

is\_weighted(G)

Esta função verifica se o grafo G possui atributos de peso nas arestas.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=3)

G.add\_edge(2, 3)

is\_weighted = nx.is\_weighted(G)

print(is\_weighted)

isolate(node)

Esta função cria um nó isolado (sem vizinhos) com o valor node.

:

import networkx as nx

G = nx.Graph()

G.add\_node(1)

G.add\_edge(2, 3)

nx.isolate(G, 4) # Adiciona o nó 4 como um nó isolado

print(G.nodes())

isolates(G)

Esta função retorna uma lista com todos os nós isolados (sem vizinhos) do grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2)

G.add\_node(3)

isolates = nx.isolates(G)

print(isolates)

isomorphism(G1, G2)

Esta função verifica se os grafos G1 e G2 são isomórficos, ou seja, se há uma correspondência biunívoca entre os nós dos grafos que preserva a estrutura das arestas.

:

import networkx as nx

G1 = nx.Graph()

G2 = nx.Graph()

G1.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

G2.add\_edges\_from([(11, 12), (12, 13), (13, 11)])

is\_isomorphic = nx.is\_isomorphic(G1, G2)

print(is\_isomorphic)

jaccard\_coefficient(G, ebunch=None)

Esta função calcula o coeficiente de Jaccard para todas as arestas ou uma lista de arestas ebunch em um grafo não direcionado ponderado.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (1, 3)])

coefficients = nx.jaccard\_coefficient(G)

for u, v, coef in coefficients:

print(f"({u}, {v}): {coef}")

johnson(G, weight='weight')

Esta função calcula o algoritmo de Johnson para encontrar os caminhos mais curtos entre todos os pares de nós em um grafo dirigido ou não dirigido ponderado.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (1, 3, {'weight': 5})])

shortest\_paths = nx.johnson(G)

print(shortest\_paths)

Espero que esta lista de funções seja útil para você! Se tiver mais perguntas ou precisar de mais s, sinta-se à vontade para perguntar.

User

O mesmo para: 'join',

'joint\_degree\_graph',

'joint\_degree\_seq',

'json\_graph',

'junction\_tree',

'k\_components',

'k\_core',

'k\_corona',

'k\_crust',

'k\_edge\_augmentation',

ChatGPT

Claro! Continuando a lista de funções do NetworkX:

join(G, H, rename=(1, 2))

Esta função junta os grafos G e H, adicionando todas as arestas de H a G. A junção é feita criando uma cópia de H e relabelando seus nós para que não haja conflitos de rótulos com os nós de G.

:

import networkx as nx

G = nx.Graph()

H = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3)])

H.add\_edges\_from([(10, 11), (11, 12)])

J = nx.join(G, H)

print(list(J.edges()))

joint\_degree\_graph(sequence)

Esta função retorna um grafo não direcionado com a sequência de graus dada em sequence como graus conjuntos (graus combinados) dos nós.

:

import networkx as nx

sequence = [3, 2, 2, 1]

G = nx.joint\_degree\_graph(sequence)

print(list(G.edges()))

joint\_degree\_seq(G)

Esta função retorna uma sequência de graus conjuntos (graus combinados) para cada nó do grafo não direcionado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

degree\_sequence = nx.joint\_degree\_seq(G)

print(degree\_sequence)

json\_graph(data, directed=False, multigraph=False, attrs=None)

Esta função converte um dicionário ou uma string JSON em um grafo NetworkX.

:

import networkx as nx

data = {

"nodes": [{"id": 1, "label": "Node 1"}, {"id": 2, "label": "Node 2"}],

"edges": [{"source": 1, "target": 2, "weight": 3}],

}

G = nx.json\_graph.json\_graph(data)

print(list(G.edges()))

junction\_tree(G)

Esta função retorna uma árvore de junção (junction tree) de um grafo não direcionado conectado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (1, 3)])

junction\_tree = nx.junction\_tree(G)

print(junction\_tree.nodes())

k\_components(G)

Esta função retorna todas as componentes k-core do grafo não direcionado G, onde uma k-core é um subgrafo em que cada nó tem grau pelo menos k.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (3, 4), (4, 5), (5, 3)])

k\_components = nx.k\_components(G)

print(k\_components)

k\_core(G, k=None, core\_number=None)

Esta função retorna a k-core do grafo não direcionado G, onde uma k-core é um subgrafo em que cada nó tem grau pelo menos k. Se k não for fornecido, a função tenta calcular o valor máximo de k possível.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (3, 4), (4, 5), (5, 3)])

k\_core = nx.k\_core(G)

print(k\_core.nodes())

k\_corona(G, k)

Esta função retorna o grafo corona k do grafo não direcionado G. O grafo corona k é obtido conectando cada nó não vizinho de G a k cópias de seus vizinhos mais próximos.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

k\_corona = nx.k\_corona(G, 2)

print(k\_corona.nodes())

k\_crust(G, k)

Esta função retorna o grafo crust k do grafo não direcionado G. O grafo crust k é obtido removendo todas as arestas que estão presentes em todos os ciclos de tamanho k ou maior.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (3, 4), (4, 5), (5, 3)])

k\_crust = nx.k\_crust(G, 3)

print(k\_crust.edges())

k\_edge\_augmentation(G, k)

Esta função retorna uma cópia do grafo não direcionado G, aumentado com novos nós e arestas para formar um grafo k-aresta conectado.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3)])

k\_edge\_augmentation = nx.k\_edge\_augmentation(G, 1)

print(list(k\_edge\_augmentation.edges()))

Espero que esta lista de funções seja útil para você! Se tiver mais perguntas ou precisar de mais s, sinta-se à vontade para perguntar.

User

O mesmo para: 'k\_edge\_components',

'k\_edge\_subgraphs',

'k\_factor',

'k\_random\_intersection\_graph',

'k\_shell',

'k\_truss',

'kamada\_kawai\_layout',

'karate\_club\_graph',

'katz',

'katz\_centrality',

ChatGPT

Claro! Continuando a lista de funções do NetworkX:

k\_edge\_components(G, k)

Esta função retorna todas as componentes k-edge do grafo não direcionado G, onde uma k-edge componente é um subgrafo em que cada aresta pertence a pelo menos k caminhos independentes.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (3, 4), (4, 5), (5, 3)])

k\_edge\_components = nx.k\_edge\_components(G, 2)

print(k\_edge\_components)

k\_edge\_subgraphs(G, k)

Esta função retorna uma lista de subgrafos k-edge conectados do grafo não direcionado G, onde um grafo k-edge conectado é um subgrafo em que cada nó é incidente a pelo menos k arestas.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (3, 4), (4, 5), (5, 3)])

k\_edge\_subgraphs = nx.k\_edge\_subgraphs(G, 2)

for subgraph in k\_edge\_subgraphs:

print(list(subgraph.edges()))

k\_factor(G, k)

Esta função retorna o fator k do grafo não direcionado G, que é um subgrafo em que cada nó tem grau exatamente k.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (3, 4), (4, 5), (5, 3)])

k\_factor = nx.k\_factor(G, 2)

print(k\_factor.edges())

k\_random\_intersection\_graph(n, m, k, p=0.5, seed=None)

Esta função retorna um grafo de interseção aleatório k-uniforme de n nós, onde cada nó possui exatamente k vizinhos, escolhidos aleatoriamente dentre os outros nós. A probabilidade de escolher uma aresta é controlada pelo parâmetro p.

:

import networkx as nx

G = nx.k\_random\_intersection\_graph(10, 15, 3, p=0.3, seed=42)

print(list(G.edges()))

k\_shell(G, k=None, core\_number=None)

Esta função retorna o grafo k-shell do grafo não direcionado G, que é o subgrafo de maior grau tal que todos os nós têm grau pelo menos k. Se k não for fornecido, a função tenta calcular o valor máximo de k possível.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (3, 4), (4, 5), (5, 3)])

k\_shell = nx.k\_shell(G)

print(k\_shell.nodes())

k\_truss(G, k)

Esta função retorna o grafo k-truss do grafo não direcionado G, que é o subgrafo de maior grau tal que todos os nós têm pelo menos grau k e cada triângulo contém pelo menos k-2 arestas do k-truss.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (3, 4), (4, 5), (5, 3)])

k\_truss = nx.k\_truss(G, 2)

print(k\_truss.edges())

kamada\_kawai\_layout(G, dist=None, pos=None, weight='weight', scale=1, center=None, dim=2)

Esta função retorna as posições dos nós do grafo não direcionado G usando o layout de Kamada-Kawai, que é um algoritmo de posicionamento para grafos que tenta posicionar os nós de tal forma que as arestas tenham comprimentos ideais.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

pos = nx.kamada\_kawai\_layout(G)

print(pos)

karate\_club\_graph()

Esta função retorna o famoso grafo "Karate Club", um grafo não direcionado representando amizades entre 34 membros de um clube de karatê, coletado pelo antropólogo Wayne Zachary.

:

import networkx as nx

G = nx.karate\_club\_graph()

print(G.number\_of\_nodes(), G.number\_of\_edges())

katz(G, alpha=0.1, beta=1.0, max\_iter=1000, tol=1e-06, nstart=None, normalized=True)

Esta função calcula as pontuações de centralidade de Katz para os nós do grafo não direcionado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

katz\_centrality = nx.katz\_centrality(G)

print(katz\_centrality)

Espero que esta lista de funções seja útil para você! Se tiver mais perguntas ou precisar de mais s, sinta-se à vontade para perguntar.

User

O mesmo para: 'katz\_centrality\_numpy',

'kl\_connected\_subgraph',

'kosaraju\_strongly\_connected\_components',

'krackhardt\_kite\_graph',

'ladder\_graph',

'laplacian',

'laplacian\_centrality',

'laplacian\_matrix',

'laplacian\_spectrum',

'laplacianmatrix',

ChatGPT

Claro! Continuando a lista de funções do NetworkX:

katz\_centrality\_numpy(G, alpha=0.1, beta=1.0, normalized=True, weight='weight')

Esta função calcula as pontuações de centralidade de Katz para os nós do grafo não direcionado G usando uma implementação mais rápida baseada em numpy.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

katz\_centrality = nx.katz\_centrality\_numpy(G)

print(katz\_centrality)

kl\_connected\_subgraph(G, k, l)

Esta função retorna o subgrafo l-conectado mais denso que é k-conectado do grafo não direcionado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

kl\_connected\_subgraph = nx.kl\_connected\_subgraph(G, 2, 3)

print(kl\_connected\_subgraph.edges())

kosaraju\_strongly\_connected\_components(G)

Esta função retorna uma lista de componentes fortemente conectados do grafo direcionado G usando o algoritmo de Kosaraju.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (3, 4), (4, 5), (5, 3)])

strongly\_connected\_components = nx.kosaraju\_strongly\_connected\_components(G)

print(strongly\_connected\_components)

krackhardt\_kite\_graph()

Esta função retorna o grafo de Krackhardt Kite, um grafo não direcionado com 10 nós e 18 arestas.

:

import networkx as nx

G = nx.krackhardt\_kite\_graph()

print(G.number\_of\_nodes(), G.number\_of\_edges())

ladder\_graph(n, create\_using=None)

Esta função retorna uma escada (ladder) de n degraus (nós) como um grafo não direcionado.

:

import networkx as nx

G = nx.ladder\_graph(5)

print(G.number\_of\_nodes(), G.number\_of\_edges())

laplacian(G, nodelist=None, weight='weight')

Esta função retorna a matriz Laplaciana do grafo não direcionado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

laplacian\_matrix = nx.laplacian(G)

print(laplacian\_matrix)

laplacian\_centrality(G, weight='weight', v=None)

Esta função calcula a centralidade laplaciana para todos os nós do grafo não direcionado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

laplacian\_centrality = nx.laplacian\_centrality(G)

print(laplacian\_centrality)

laplacian\_matrix(G, nodelist=None, weight='weight')

Esta função retorna a matriz Laplaciana do grafo não direcionado G. (Nota: Esta função é obsoleta, é recomendado usar laplacian() em vez disso.)

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

laplacian\_matrix = nx.laplacian\_matrix(G)

print(laplacian\_matrix)

laplacian\_spectrum(G, weight='weight')

Esta função retorna o espectro da matriz Laplaciana do grafo não direcionado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

laplacian\_spectrum = nx.laplacian\_spectrum(G)

print(laplacian\_spectrum)

Espero que esta lista de funções seja útil! Se tiver mais perguntas ou precisar de mais s, sinta-se à vontade para perguntar.

User

O mesmo para: 'laplacianmatrix',

'lattice',

'lattice\_reference',

'layout',

'lazy\_imports',

'leda',

'les\_miserables\_graph',

'lexicographic\_product',

'lexicographical\_topological\_sort',

'linalg',

ChatGPT

Continuando com a lista de funções do NetworkX:

laplacianmatrix(G, nodelist=None, weight='weight')

Esta função retorna a matriz Laplaciana do grafo não direcionado G. (Nota: Esta função é obsoleta, é recomendado usar laplacian() em vez disso.)

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

laplacian\_matrix = nx.laplacianmatrix(G)

print(laplacian\_matrix)

lattice(m, n, periodic=False, create\_using=None)

Esta função retorna uma grade 2D de m x n nós como um grafo não direcionado.

:

import networkx as nx

G = nx.lattice(3, 3)

print(G.number\_of\_nodes(), G.number\_of\_edges())

lattice\_reference(m, n, periodic=False)

Esta função retorna a referência de uma grade 2D de m x n nós como um dicionário.

:

import networkx as nx

lattice\_ref = nx.lattice\_reference(3, 3)

print(lattice\_ref)

layout(G, k=None, seed=None, \*\*kwargs)

Esta função calcula um layout gráfico para os nós em G.

:

import networkx as nx

import matplotlib.pyplot as plt

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

pos = nx.layout(G, k=0.3)

nx.draw(G, pos, with\_labels=True)

plt.show()

lazy\_imports

Esta função retorna uma lista de módulos que são importados preguiçosamente (lazy) pelo NetworkX.

:

import networkx as nx

lazy\_imports\_list = nx.lazy\_imports()

print(lazy\_imports\_list)

leda

Esta função retorna um grafo não direcionado da base de dados LEDA.

:

import networkx as nx

G = nx.leda()

print(G.number\_of\_nodes(), G.number\_of\_edges())

les\_miserables\_graph()

Esta função retorna um grafo não direcionado com base nos personagens do livro "Les Misérables" de Victor Hugo.

:

import networkx as nx

G = nx.les\_miserables\_graph()

print(G.number\_of\_nodes(), G.number\_of\_edges())

lexicographic\_product(G, H, create\_using=None)

Esta função retorna o produto lexicográfico de G e H.

:

import networkx as nx

G = nx.complete\_graph(3)

H = nx.path\_graph(4)

lexicographic\_product = nx.lexicographic\_product(G, H)

print(lexicographic\_product.edges())

lexicographical\_topological\_sort(G)

Esta função retorna uma lista de nós de G em uma ordem lexicográfica topológica.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3)])

topological\_order = nx.lexicographical\_topological\_sort(G)

print(topological\_order)

linalg

Esta função retorna o módulo do NumPy numpy.linalg.

:

import networkx as nx

linalg\_module = nx.linalg

print(linalg\_module)

Espero que esta lista seja útil! Se você tiver mais perguntas ou precisar de mais s, fique à vontade para perguntar.

User

O mesmo para: 'line',

'line\_graph',

'link\_analysis',

'link\_prediction',

'load',

'load\_centrality',

'local\_bridges',

'local\_constraint',

'local\_efficiency',

'local\_reaching\_centrality',

ChatGPT

Claro, aqui está a continuação da lista de funções do NetworkX:

line(n, create\_using=None)

Esta função retorna um grafo da linha com n nós.

:

import networkx as nx

G = nx.line(5)

print(G.number\_of\_nodes(), G.number\_of\_edges())

line\_graph(G, create\_using=None)

Esta função retorna o grafo da linha de G.

:

import networkx as nx

G = nx.complete\_graph(3)

line\_graph = nx.line\_graph(G)

print(line\_graph.edges())

link\_analysis

Este módulo oferece funções relacionadas à análise de links em grafos, como PageRank, HITS, etc.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

pagerank = nx.link\_analysis.pagerank(G)

print(pagerank)

link\_prediction

Este módulo oferece funções relacionadas à predição de links em grafos, como Adamic-Adar, Common Neighbors, etc.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

common\_neighbors = nx.link\_prediction.common\_neighbors(G, ebunch=[(1, 4)])

print(common\_neighbors)

load(path, format=None, delimiter=None, comments='#', encoding='utf-8')

Esta função carrega um grafo de um arquivo.

:

import networkx as nx

G = nx.load("example\_graph.txt", delimiter="\t")

print(G.number\_of\_nodes(), G.number\_of\_edges())

load\_centrality(G, v=None, cutoff=None, normalized=True, weight=None)

Esta função retorna a centralidade de carga para todos os nós em G ou apenas para o nó v.

:

import networkx as nx

G = nx.path\_graph(5)

load\_centrality = nx.load\_centrality(G)

print(load\_centrality)

local\_bridges(G, weight='weight')

Esta função retorna um gerador de todas as arestas de "ponte" locais no grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1), (4, 5)])

local\_bridges = nx.local\_bridges(G)

print(list(local\_bridges))

local\_constraint(G, nodes=None, weight=None)

Esta função retorna o coeficiente de restrição local para todos os nós no grafo G.

:

import networkx as nx

G = nx.cycle\_graph(4)

local\_constraint = nx.local\_constraint(G)

print(local\_constraint)

local\_efficiency(G, nodes=None, weight=None)

Esta função retorna a eficiência local para todos os nós no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (3, 4)])

local\_efficiency = nx.local\_efficiency(G)

print(local\_efficiency)

local\_reaching\_centrality(G, normalized=True, weight=None)

Esta função retorna a centralidade de alcance local para todos os nós no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (3, 4)])

reaching\_centrality = nx.local\_reaching\_centrality(G)

print(reaching\_centrality)

Espero que esta lista seja útil! Se você tiver mais perguntas ou precisar de mais s, fique à vontade para perguntar.

User

O mesmo para: 'lollipop\_graph',

'lowest\_common\_ancestor',

'lowest\_common\_ancestors',

'make\_clique\_bipartite',

'make\_max\_clique\_graph',

'margulis\_gabber\_galil\_graph',

'matching',

'max\_flow\_min\_cost',

'max\_weight\_clique',

'max\_weight\_matching',

ChatGPT

Claro, continuando a lista de funções do NetworkX:

lollipop\_graph(m, n, create\_using=None)

Esta função retorna um grafo lollipop, que é a concatenação de um grafo completo com m nós e um caminho com n nós.

:

import networkx as nx

G = nx.lollipop\_graph(3, 4)

print(G.number\_of\_nodes(), G.number\_of\_edges())

lowest\_common\_ancestor(T, sources, target)

Esta função retorna o ancestral comum mais baixo de um conjunto de nós de origem sources em uma árvore direcionada T com relação ao nó alvo target.

:

import networkx as nx

T = nx.DiGraph()

T.add\_edges\_from([(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7)])

sources = [4, 5]

target = 6

lowest\_common\_ancestor = nx.lowest\_common\_ancestor(T, sources, target)

print(lowest\_common\_ancestor)

lowest\_common\_ancestors(T, sources, root=None)

Esta função retorna um dicionário contendo os ancestrais comuns mais baixos para cada nó de origem em sources em uma árvore direcionada T com relação à raiz especificada root. Se a raiz não for fornecida, a função usará o nó de origem mais comum como raiz.

:

import networkx as nx

T = nx.DiGraph()

T.add\_edges\_from([(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7)])

sources = [4, 5]

root = 1

lowest\_common\_ancestors = nx.lowest\_common\_ancestors(T, sources, root=root)

print(lowest\_common\_ancestors)

make\_clique\_bipartite(G, top\_nodes=None, container=None)

Esta função transforma um grafo de clique em um grafo bipartido em que os nós de clique se tornam os nós de uma partição e os outros nós se tornam os nós de outra partição.

:

import networkx as nx

G = nx.complete\_graph(4)

clique\_bipartite = nx.make\_clique\_bipartite(G)

print(clique\_bipartite.number\_of\_nodes(), clique\_bipartite.number\_of\_edges())

make\_max\_clique\_graph(G, create\_using=None)

Esta função retorna um grafo cujos nós são cliques do grafo G, e duas cliques são conectadas por uma aresta se e somente se elas têm interseção vazia.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (3, 4)])

max\_clique\_graph = nx.make\_max\_clique\_graph(G)

print(max\_clique\_graph.number\_of\_nodes(), max\_clique\_graph.number\_of\_edges())

margulis\_gabber\_galil\_graph(k, seed=None, data=False)

Esta função retorna um grafo de Margulis-Gabber-Galil com o parâmetro k, que é uma família de grafos ramificados aleatoriamente que são expansores.

:

import networkx as nx

k = 3

margulis\_graph = nx.margulis\_gabber\_galil\_graph(k)

print(margulis\_graph.number\_of\_nodes(), margulis\_graph.number\_of\_edges())

matching(G, maxcardinality=False, weight='weight', top\_nodes=None)

Esta função retorna um conjunto de arestas que formam uma cobertura por vértices de peso máximo ou uma correspondência de peso máximo no grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 1}), (1, 3, {'weight': 3})])

matching = nx.matching(G, maxcardinality=True, weight='weight')

print(matching)

max\_flow\_min\_cost(G, s, t, capacity='capacity', weight='weight', \*\*kwargs)

Esta função retorna o fluxo máximo de custo mínimo entre os nós s e t em um grafo ponderado G.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edge(1, 2, capacity=10, weight=2)

G.add\_edge(1, 3, capacity=5, weight=1)

G.add\_edge(2, 3, capacity=2, weight=2)

flow, cost = nx.max\_flow\_min\_cost(G, 1, 3, capacity='capacity', weight='weight')

print(flow, cost)

max\_weight\_clique(G, weight='weight')

Esta função retorna a clique de peso máximo em um grafo ponderado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 5}), (2, 3, {'weight': 10}), (1, 3, {'weight': 8})])

max\_weight\_clique = nx.max\_weight\_clique(G, weight='weight')

print(max\_weight\_clique)

max\_weight\_matching(G, maxcardinality=False, weight='weight', top\_nodes=None)

Esta função retorna um conjunto de arestas que formam uma correspondência de peso máximo no grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 1}), (1, 3, {'weight': 3})])

max\_weight\_matching = nx.max\_weight\_matching(G, weight='weight')

print(max\_weight\_matching)

Espero que esta lista seja útil! Se você tiver mais perguntas ou precisar de mais s, fique à vontade para perguntar.

User

O mesmo para: 'maximal\_independent\_set',

'maximal\_matching',

'maximum\_branching',

'maximum\_flow',

'maximum\_flow\_value',

'maximum\_spanning\_arborescence',

'maximum\_spanning\_edges',

'maximum\_spanning\_tree',

'min\_cost\_flow',

'min\_cost\_flow\_cost',

ChatGPT

Claro! Continuando com a lista de funções do NetworkX:

maximal\_independent\_set(G)

Esta função retorna um conjunto de nós que formam um conjunto independente máximo (maximal independent set) no grafo G. Um conjunto independente máximo é um conjunto de nós que não possuem arestas em comum.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

maximal\_independent\_set = nx.maximal\_independent\_set(G)

print(maximal\_independent\_set)

maximal\_matching(G, weight=None)

Esta função retorna um conjunto de arestas que forma um emparelhamento máximo (maximal matching) no grafo G. Um emparelhamento é um conjunto de arestas que não compartilham nós em comum.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

maximal\_matching = nx.maximal\_matching(G)

print(maximal\_matching)

maximum\_branching(G, weight='weight')

Esta função retorna uma árvore de abrangência máxima (maximum branching) em um grafo direcionado ponderado G.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edge(1, 2, weight=3)

G.add\_edge(1, 3, weight=2)

G.add\_edge(2, 3, weight=1)

maximum\_branching = nx.maximum\_branching(G, weight='weight')

print(maximum\_branching.edges())

maximum\_flow(G, s, t, capacity='capacity', flow\_func=None, \*\*kwargs)

Esta função retorna o fluxo máximo entre os nós s e t em um grafo direcionado ponderado G.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edge(1, 2, capacity=10)

G.add\_edge(1, 3, capacity=5)

G.add\_edge(2, 3, capacity=2)

flow\_dict = nx.maximum\_flow(G, 1, 3, capacity='capacity')

print(flow\_dict[1][3])

maximum\_flow\_value(G, s, t, capacity='capacity', flow\_func=None, \*\*kwargs)

Esta função retorna o valor do fluxo máximo entre os nós s e t em um grafo direcionado ponderado G.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edge(1, 2, capacity=10)

G.add\_edge(1, 3, capacity=5)

G.add\_edge(2, 3, capacity=2)

max\_flow\_value = nx.maximum\_flow\_value(G, 1, 3, capacity='capacity')

print(max\_flow\_value)

maximum\_spanning\_arborescence(G, attr='weight')

Esta função retorna uma arborescência de abrangência máxima (maximum spanning arborescence) em um grafo direcionado ponderado G.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edge(1, 2, weight=3)

G.add\_edge(1, 3, weight=2)

G.add\_edge(2, 3, weight=1)

max\_spanning\_arborescence = nx.maximum\_spanning\_arborescence(G, attr='weight')

print(max\_spanning\_arborescence.edges())

maximum\_spanning\_edges(G, data=True, weight='weight')

Esta função retorna um conjunto de arestas que formam uma árvore de abrangência máxima (maximum spanning tree) em um grafo não direcionado ponderado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=3)

G.add\_edge(1, 3, weight=2)

G.add\_edge(2, 3, weight=1)

max\_spanning\_edges = nx.maximum\_spanning\_edges(G, data=True, weight='weight')

print(list(max\_spanning\_edges))

maximum\_spanning\_tree(G, weight='weight')

Esta função retorna uma árvore de abrangência máxima (maximum spanning tree) em um grafo não direcionado ponderado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=3)

G.add\_edge(1, 3, weight=2)

G.add\_edge(2, 3, weight=1)

max\_spanning\_tree = nx.maximum\_spanning\_tree(G, weight='weight')

print(max\_spanning\_tree.edges())

min\_cost\_flow(G, demand='demand', capacity='capacity', weight='weight',...

User

O mesmo para: 'min\_edge\_cover',

'min\_weight\_matching',

'minimal\_d\_separator',

'minimum\_branching',

'minimum\_cut',

'minimum\_cut\_value',

'minimum\_cycle\_basis',

'minimum\_edge\_cut',

'minimum\_node\_cut',

'minimum\_spanning\_arborescence',

ChatGPT

Continuando com a lista de funções do NetworkX:

min\_edge\_cover(G, maxcardinality=False)

Esta função retorna um conjunto de arestas que forma uma cobertura de arestas mínima (minimum edge cover) no grafo G. Uma cobertura de arestas é um conjunto de arestas que cobre todos os nós do grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2)

G.add\_edge(1, 3)

G.add\_edge(2, 3)

min\_edge\_cover = nx.min\_edge\_cover(G)

print(min\_edge\_cover)

min\_weight\_matching(G, weight='weight', maxcardinality=False)

Esta função retorna um conjunto de arestas que forma um emparelhamento de peso mínimo (minimum weight matching) no grafo não direcionado ponderado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=3)

G.add\_edge(1, 3, weight=2)

G.add\_edge(2, 3, weight=1)

min\_weight\_matching = nx.min\_weight\_matching(G, weight='weight')

print(min\_weight\_matching)

minimal\_d\_separator(G, s, t)

Esta função retorna um conjunto de nós que forma um D-separador mínimo (minimal D-separator) entre os nós s e t em um grafo direcionado G. Um D-separador é um conjunto de nós que bloqueia todos os caminhos direcionados entre s e t.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

minimal\_d\_separator = nx.minimal\_d\_separator(G, 1, 4)

print(minimal\_d\_separator)

minimum\_branching(G, root, weight='weight')

Esta função retorna uma árvore de ramificação mínima (minimum branching) em um grafo direcionado ponderado G, tendo o nó root como raiz.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edge(1, 2, weight=3)

G.add\_edge(1, 3, weight=2)

G.add\_edge(2, 3, weight=1)

minimum\_branching = nx.minimum\_branching(G, root=1, weight='weight')

print(minimum\_branching.edges())

minimum\_cut(G, s, t, capacity='capacity', flow\_func=None, \*\*kwargs)

Esta função retorna um corte mínimo (minimum cut) entre os nós s e t em um grafo direcionado ponderado G. Um corte mínimo é um conjunto de arestas de capacidade mínima que desconecta s e t.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edge(1, 2, capacity=10)

G.add\_edge(1, 3, capacity=5)

G.add\_edge(2, 3, capacity=2)

min\_cut = nx.minimum\_cut(G, 1, 3, capacity='capacity')

print(min\_cut)

minimum\_cut\_value(G, s, t, capacity='capacity', flow\_func=None, \*\*kwargs)

Esta função retorna o valor do corte mínimo (minimum cut) entre os nós s e t em um grafo direcionado ponderado G. O valor do corte mínimo é a capacidade total das arestas no corte mínimo.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edge(1, 2, capacity=10)

G.add\_edge(1, 3, capacity=5)

G.add\_edge(2, 3, capacity=2)

min\_cut\_value = nx.minimum\_cut\_value(G, 1, 3, capacity='capacity')

print(min\_cut\_value)

minimum\_cycle\_basis(G)

Esta função retorna um conjunto de ciclos que forma uma base de ciclos mínima (minimum cycle basis) em um grafo não direcionado G. Uma base de ciclos é um conjunto de ciclos que forma todos os ciclos simples do grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_cycle([1, 2, 3])

G.add\_cycle([3, 4, 5])

minimum\_cycle\_basis = nx.minimum\_cycle\_basis(G)

print(minimum\_cycle\_basis)

minimum\_edge\_cut(G, s, t, flow\_func=None)

Esta função retorna um conjunto de arestas que forma um corte de arestas mínimo (minimum edge cut) entre os nós s e t em um grafo não direcionado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2)

G.add\_edge(1, 3)

G.add\_edge(2, 3)

G.add\_edge(3, 4)

min\_edge\_cut = nx.minimum\_edge\_cut(G, 1, 4)

print(min\_edge\_cut)

minimum\_node\_cut(G, s, t, flow\_func=None)

Esta função retorna um conjunto de nós que forma um corte de nós mínimo (minimum node cut) entre os nós s e t em um grafo não direcionado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2)

G.add\_edge(1, 3)

G.add\_edge(2, 3)

G.add\_edge(3, 4)

min\_node\_cut = nx.minimum\_node\_cut(G, 1, 4)

print(min\_node\_cut)

minimum\_spanning\_arborescence(G, attr='weight')

Esta função retorna uma arborescência de abrangência mínima (minimum spanning arborescence) em um grafo direcionado ponderado G.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edge(1, 2, weight=3)

G.add\_edge(1, 3, weight=2)

G.add\_edge(2, 3, weight=1)

min\_spanning\_arborescence = nx.minimum\_spanning\_arborescence(G, attr='weight')

print(min\_spanning\_arborescence.edges())

Com isso, completamos a lista de funções do NetworkX. Lembre-se de que o NetworkX é uma biblioteca rica e poderosa para manipulação e análise de redes e grafos em . Cada uma dessas funções possui diversos parâmetros e opções adicionais que permitem ajustar seu comportamento conforme necessário.

User

O mesmo para: 'minimum\_spanning\_edges',

'minimum\_spanning\_tree',

'minors',

'mis',

'mixing',

'mixing\_dict',

'mixing\_expansion',

'modularity\_matrix',

'modularity\_spectrum',

'modularitymatrix',

ChatGPT

Continuando com a lista de funções do NetworkX:

minimum\_spanning\_edges(G, weight='weight', data=True)

Esta função retorna um gerador de tuplas que representa as arestas de uma árvore de abrangência mínima (minimum spanning tree) em um grafo não direcionado ponderado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=3)

G.add\_edge(1, 3, weight=2)

G.add\_edge(2, 3, weight=1)

mst\_edges = nx.minimum\_spanning\_edges(G, weight='weight', data=True)

print(list(mst\_edges))

minimum\_spanning\_tree(G, weight='weight', algorithm='kruskal', ignore\_nan=False)

Esta função retorna a árvore de abrangência mínima (minimum spanning tree) de um grafo não direcionado ponderado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=3)

G.add\_edge(1, 3, weight=2)

G.add\_edge(2, 3, weight=1)

mst = nx.minimum\_spanning\_tree(G, weight='weight')

print(mst.edges())

minors(G, nodes)

Esta função retorna um dicionário contendo cópias de um grafo G, mas com os nós especificados em nodes removidos.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

nodes\_to\_remove = [2, 4]

minors\_graphs = nx.minors(G, nodes\_to\_remove)

for node in nodes\_to\_remove:

print(f"Graph after removing node {node}:")

print(minors\_graphs[node].edges())

print()

mis(G)

Esta função retorna um conjunto independente máximo (maximum independent set) em um grafo não direcionado G. Um conjunto independente é um conjunto de nós que não têm arestas entre si.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

max\_independent\_set = nx.mis(G)

print(max\_independent\_set)

mixing(G, nodes, weight='weight')

Esta função retorna uma medida de mistura (mixing) de nós em um grafo não direcionado ponderado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 0.5}), (1, 3, {'weight': 0.2}), (2, 3, {'weight': 0.8})])

nodes = [1, 2, 3]

mixing\_measure = nx.mixing(G, nodes, weight='weight')

print(mixing\_measure)

mixing\_dict(G, weight='weight')

Esta função retorna um dicionário contendo a medida de mistura (mixing) de cada nó em um grafo não direcionado ponderado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 0.5}), (1, 3, {'weight': 0.2}), (2, 3, {'weight': 0.8})])

mixing\_dict = nx.mixing\_dict(G, weight='weight')

print(mixing\_dict)

mixing\_expansion(G, nodes, weight='weight')

Esta função retorna uma medida de expansão (expansion) de nós em um grafo não direcionado ponderado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 0.5}), (1, 3, {'weight': 0.2}), (2, 3, {'weight': 0.8})])

nodes = [1, 2, 3]

expansion\_measure = nx.mixing\_expansion(G, nodes, weight='weight')

print(expansion\_measure)

modularity\_matrix(G, nodelist=None, weight='weight')

Esta função retorna a matriz de modularidade (modularity matrix) de um grafo não direcionado ponderado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 0.5}), (1, 3, {'weight': 0.2}), (2, 3, {'weight': 0.8})])

modularity\_matrix = nx.modularity\_matrix(G, weight='weight')

print(modularity\_matrix)

modularity\_spectrum(G, weight='weight')

Esta função retorna o espectro de modularidade (modularity spectrum) de um grafo não direcionado ponderado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 0.5}), (1, 3, {'weight': 0.2}), (2, 3, {'weight': 0.8})])

modularity\_spectrum = nx.modularity\_spectrum(G, weight='weight')

print(modularity\_spectrum)

Com isso, concluímos a lista de funções do NetworkX. Essas funções oferecem várias ferramentas poderosas para análise, manipulação e visualização de grafos e redes em . Lembre-se de consultar a documentação oficial do NetworkX para obter mais informações detalhadas sobre cada uma delas.

User

O mesmo para: 'moebius\_kantor\_graph',

'moral',

'moral\_graph',

'multi\_source\_dijkstra',

'multi\_source\_dijkstra\_path',

'multi\_source\_dijkstra\_path\_length',

'multidigraph',

'multigraph',

'multiline\_adjlist',

'multipartite\_layout',

ChatGPT

Continuando com a lista de funções do NetworkX:

moebius\_kantor\_graph(n1, n2)

Esta função retorna o grafo de Moebius-Kantor, que é um grafo bipartido completo com n1 nós em um conjunto e n2 nós no outro conjunto.

:

import networkx as nx

n1, n2 = 3, 4

G = nx.moebius\_kantor\_graph(n1, n2)

print(G.edges())

moral(graph)

Esta função retorna o grafo moral de um grafo direcionado acíclico (DAG).

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

moral\_graph = nx.moral(G)

print(moral\_graph.edges())

moral\_graph(digraph)

Esta função retorna o grafo moral de um grafo direcionado acíclico (DAG) usando a abordagem do algoritmo.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

moral\_graph = nx.moral\_graph(G)

print(moral\_graph.edges())

multi\_source\_dijkstra(G, sources, target, weight='weight')

Esta função retorna as distâncias mínimas e os caminhos entre várias fontes e um alvo em um grafo ponderado G.

:

import networkx as nx

G = nx.Graph()

G.add\_weighted\_edges\_from([(1, 2, 1), (1, 3, 2), (2, 3, 1), (3, 4, 3)])

sources = [1, 2]

target = 4

distances, paths = nx.multi\_source\_dijkstra(G, sources, target, weight='weight')

print("Distances:", distances)

print("Paths:", paths)

multi\_source\_dijkstra\_path(G, sources, target, weight='weight')

Esta função retorna o caminho mínimo entre várias fontes e um alvo em um grafo ponderado G.

:

import networkx as nx

G = nx.Graph()

G.add\_weighted\_edges\_from([(1, 2, 1), (1, 3, 2), (2, 3, 1), (3, 4, 3)])

sources = [1, 2]

target = 4

shortest\_path = nx.multi\_source\_dijkstra\_path(G, sources, target, weight='weight')

print(shortest\_path)

multi\_source\_dijkstra\_path\_length(G, sources, target, weight='weight')

Esta função retorna o comprimento do caminho mínimo entre várias fontes e um alvo em um grafo ponderado G.

:

import networkx as nx

G = nx.Graph()

G.add\_weighted\_edges\_from([(1, 2, 1), (1, 3, 2), (2, 3, 1), (3, 4, 3)])

sources = [1, 2]

target = 4

shortest\_path\_length = nx.multi\_source\_dijkstra\_path\_length(G, sources, target, weight='weight')

print(shortest\_path\_length)

multidigraph()

Esta função retorna um grafo direcionado sem pesos, permitindo múltiplas arestas entre os mesmos nós.

:

import networkx as nx

G = nx.multidigraph()

G.add\_edges\_from([(1, 2), (1, 2), (2, 3)])

print(G.edges())

multigraph()

Esta função retorna um grafo não direcionado sem pesos, permitindo múltiplas arestas entre os mesmos nós.

:

import networkx as nx

G = nx.multigraph()

G.add\_edges\_from([(1, 2), (1, 2), (2, 3)])

print(G.edges())

multiline\_adjlist(lines, comments='#', delimiter=None)

Esta função lê uma sequência de linhas de texto e retorna um grafo do tipo MultiGraph ou MultiDiGraph com base nas informações fornecidas.

:

import networkx as nx

lines = [

'1 2 3',

'1 3 2',

'2 3 1',

'3 4 3',

]

G = nx.multiline\_adjlist(lines, delimiter=' ')

print(G.edges(data=True))

multipartite\_layout(G, subset\_key='subset', align='vertical', scale=1, center=None, aspect\_ratio=4/3)

Esta função retorna um layout para um grafo multipartido, onde os nós de cada partição são colocados em uma linha ou coluna, dependendo da opção de alinhamento.

:

import networkx as nx

import matplotlib.pyplot as plt

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 4), (2, 5), (3, 5), (3, 6)])

# Definir atributos 'subset' para cada nó indicando a que partição ele pertence

nx.set\_node\_attributes(G, {1: 0, 2: 0, 3: 0, 4: 1, 5: 1, 6: 1}, 'subset')

pos = nx.multipartite\_layout(G, subset\_key='subset')

nx.draw(G, pos, with\_labels=True, node\_color='lightblue', node\_size=500, font\_size=10)

plt.show()

Com isso, finalizamos a lista de funções do NetworkX. Essas funções oferecem várias ferramentas para trabalhar com grafos e redes em , permitindo análises complexas, visualizações e manipulações de redes. Lembre-se de consultar a documentação oficial do NetworkX para obter mais detalhes e s.

User

O mesmo para: 'mycielski',

'mycielski\_graph',

'mycielskian',

'navigable\_small\_world\_graph',

'negative\_edge\_cycle',

'neighbor\_degree',

'neighbors',

'network\_simplex',

'newman\_watts\_strogatz\_graph',

'node\_attribute\_xy',

ChatGPT

Continuando com a lista de funções do NetworkX:

mycielski(k)

Esta função retorna o grafo de Mycielski de ordem k.

:

import networkx as nx

k = 3

G = nx.mycielski(k)

print(G.edges())

mycielski\_graph(k)

Esta função retorna o grafo de Mycielski de ordem k. Ela é equivalente à função mycielski(k).

:

import networkx as nx

k = 3

G = nx.mycielski\_graph(k)

print(G.edges())

mycielskian(G, k)

Esta função retorna o grafo Mycielskian de um grafo G de ordem k.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

k = 2

M = nx.mycielskian(G, k)

print(M.edges())

navigable\_small\_world\_graph(n, p, q, r, dim=2, seed=None)

Esta função retorna um grafo de mundo pequeno navegável de dimensão dim, com n nós, p probabilidade de re-conexão de longo alcance, q probabilidade de re-conexão de curto alcance e r parâmetro de atração.

:

import networkx as nx

n, p, q, r = 10, 0.2, 0.1, 0.5

G = nx.navigable\_small\_world\_graph(n, p, q, r, dim=2)

print(G.edges())

negative\_edge\_cycle(G, weight='weight')

Esta função encontra e retorna um ciclo de peso negativo em um grafo ponderado G.

:

import networkx as nx

G = nx.DiGraph()

G.add\_weighted\_edges\_from([(1, 2, -1), (2, 3, 2), (3, 1, -4), (3, 4, 3)])

cycle = nx.negative\_edge\_cycle(G, weight='weight')

print(cycle)

neighbor\_degree(G, nodes=None, weight=None)

Esta função retorna um dicionário com o grau médio dos vizinhos de cada nó em nodes para o grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

neighbor\_degrees = nx.neighbor\_degree(G)

print(neighbor\_degrees)

neighbors(G, node)

Esta função retorna uma lista dos vizinhos do nó node no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

neighbors\_of\_node\_3 = nx.neighbors(G, 3)

print(neighbors\_of\_node\_3)

network\_simplex(G, demand='demand', capacity='capacity', weight='weight')

Esta função encontra um fluxo máximo no grafo G sujeito a demandas e capacidades em cada nó e atributos de peso nas arestas.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edge(1, 2, capacity=10, weight=1)

G.add\_edge(1, 3, capacity=20, weight=2)

G.add\_edge(2, 3, capacity=5, weight=3)

G.add\_edge(2, 4, capacity=15, weight=4)

G.add\_edge(3, 4, capacity=10, weight=5)

demand = {1: -5, 4: 5}

flow\_cost, flow\_dict = nx.network\_simplex(G, demand=demand, capacity='capacity', weight='weight')

print("Flow cost:", flow\_cost)

print("Flow dictionary:", flow\_dict)

newman\_watts\_strogatz\_graph(n, k, p, seed=None)

Esta função retorna um grafo de mundo pequeno baseado no modelo de Newman-Watts-Strogatz com n nós, cada nó é conectado a k vizinhos mais próximos em um anel e a probabilidade p de re-conexão.

:

import networkx as nx

n, k, p = 10, 2, 0.3

G = nx.newman\_watts\_strogatz\_graph(n, k, p)

print(G.edges())

node\_attribute\_xy(G, attribute, scale=1)

Esta função atribui coordenadas x, y aos nós de um grafo com base em um atributo numérico especificado.

:

import networkx as nx

import matplotlib.pyplot as plt

G = nx.Graph()

G.add\_nodes\_from([1, 2, 3, 4])

nx.set\_node\_attributes(G, {1: 5, 2: 10, 3: 20, 4: 25}, 'attribute')

pos = nx.node\_attribute\_xy(G, 'attribute')

nx.draw(G, pos, with\_labels=True, node\_color='lightblue', node\_size=500, font\_size=10)

plt.show()

Com isso, concluímos a lista de funções do NetworkX. Essas funções oferecem várias ferramentas para trabalhar com grafos e redes em , permitindo análises complexas, visualizações e manipulações de redes. Lembre-se de consultar a documentação oficial do NetworkX para obter mais detalhes e s.

User

O mesmo para: 'node\_boundary',

'node\_classification',

'node\_clique\_number',

'node\_connected\_component',

'node\_connectivity',

'node\_degree\_xy',

'node\_disjoint\_paths',

'node\_expansion',

'node\_link',

'node\_link\_data',

ChatGPT

Continuando a lista de funções do NetworkX:

node\_boundary(G, nbunch=None)

Esta função retorna um conjunto dos nós de fronteira de um conjunto de nós nbunch em um grafo G. Os nós de fronteira são os nós que têm pelo menos um vizinho em nbunch e pelo menos um vizinho fora de nbunch.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

boundary\_nodes = nx.node\_boundary(G, [2, 3])

print(boundary\_nodes)

node\_classification(G, clf\_name='modularity\_max')

Esta função executa a classificação dos nós em um grafo G usando um algoritmo especificado por clf\_name.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

classification = nx.node\_classification(G, clf\_name='modularity\_max')

print(classification)

node\_clique\_number(G, nodes=None)

Esta função retorna o número de cliques máximos aos quais um conjunto de nós nodes pertence.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

clique\_number = nx.node\_clique\_number(G, [2, 3])

print(clique\_number)

node\_connected\_component(G, node)

Esta função retorna o componente conexo ao qual um nó node pertence em um grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

component = nx.node\_connected\_component(G, 3)

print(component)

node\_connectivity(G, s=None, t=None, flow\_func=None)

Esta função retorna a conectividade entre um nó s e um nó t em um grafo G, ou a conectividade global se nenhum nó s ou t for especificado.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

connectivity = nx.node\_connectivity(G, s=1, t=4)

print(connectivity)

node\_degree\_xy(G, pos, centered=True, scale=1)

Esta função retorna um dicionário com os graus normalizados dos nós em G, de acordo com suas coordenadas pos em um gráfico.

:

import networkx as nx

import matplotlib.pyplot as plt

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

pos = nx.spring\_layout(G)

degree\_dict = nx.node\_degree\_xy(G, pos, centered=True, scale=100)

nx.draw(G, pos, with\_labels=True, node\_color='lightblue', node\_size=[v for v in degree\_dict.values()], font\_size=10)

plt.show()

node\_disjoint\_paths(G, source, target, cutoff=None)

Esta função retorna a quantidade de caminhos disjuntos entre um nó source e um nó target em um grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

num\_disjoint\_paths = nx.node\_disjoint\_paths(G, 1, 4)

print(num\_disjoint\_paths)

node\_expansion(G, nbunch=None, level=1)

Esta função retorna o conjunto de nós alcançáveis em até level etapas a partir dos nós em nbunch no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

expansion = nx.node\_expansion(G, [2, 3], level=2)

print(expansion)

node\_link(G, id, attrs=None)

Esta função retorna o nó ou aresta associado a um id único em um gráfico.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

node = nx.node\_link(G, 1)

print(node)

node\_link\_data(G, attrs=None)

Esta função retorna a representação em formato de dicionário de um gráfico G compatível com o formato JSON.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

data = nx.node\_link\_data(G)

print(data)

Com isso, concluímos a lista de funções do NetworkX. Lembre-se de consultar a documentação oficial do NetworkX para obter mais detalhes e s.

User

O mesmo para: 'node\_link\_graph',

'nodes',

'nodes\_with\_selfloops',

'non\_edges',

'non\_neighbors',

'non\_randomness',

'nonisomorphic\_trees',

'normalized\_cut\_size',

'normalized\_laplacian\_matrix',

'normalized\_laplacian\_spectrum',

ChatGPT

Continuando a lista de funções do NetworkX:

node\_link\_graph(data, directed=False, multigraph=False)

Esta função retorna um gráfico criado a partir de uma representação em formato de dicionário data, compatível com o formato JSON.

:

import networkx as nx

data = {

"nodes": [{"id": 1}, {"id": 2}, {"id": 3}],

"links": [{"source": 1, "target": 2}, {"source": 2, "target": 3}]

}

G = nx.node\_link\_graph(data)

print(G.edges())

nodes(G)

Esta função retorna uma visualização dos nós em um gráfico G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

nodes = nx.nodes(G)

print(nodes)

nodes\_with\_selfloops(G)

Esta função retorna um conjunto dos nós que têm laços de autoconexão em um gráfico G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 2), (3, 3), (3, 4)])

nodes\_with\_selfloops = nx.nodes\_with\_selfloops(G)

print(nodes\_with\_selfloops)

non\_edges(G)

Esta função retorna um gerador de todas as arestas que não estão presentes em um gráfico G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (3, 4)])

non\_edges = nx.non\_edges(G)

print(list(non\_edges))

non\_neighbors(G, n)

Esta função retorna um gerador de todos os nós que não são vizinhos do nó n em um gráfico G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (3, 4)])

non\_neighbors = nx.non\_neighbors(G, 1)

print(list(non\_neighbors))

non\_randomness(G, zscore\_threshold=2)

Esta função retorna um dicionário com a não aleatoriedade do grafo G para cada triângulo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

non\_randomness = nx.non\_randomness(G)

print(non\_randomness)

nonisomorphic\_trees(n)

Esta função retorna um gerador de todos os grafos não isomórficos com n nós que são árvores.

:

import networkx as nx

nonisomorphic\_trees = nx.nonisomorphic\_trees(4)

for tree in nonisomorphic\_trees:

print(tree.edges())

normalized\_cut\_size(G, S, T)

Esta função retorna a medida de corte normalizado entre dois conjuntos de nós S e T em um grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

cut\_size = nx.normalized\_cut\_size(G, {1, 2}, {3, 4})

print(cut\_size)

normalized\_laplacian\_matrix(G, nodelist=None, weight='weight')

Esta função retorna a matriz do Laplaciano normalizada de um gráfico G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (1, 3, {'weight': 3}), (2, 3, {'weight': 1}), (3, 4, {'weight': 4})])

laplacian\_matrix = nx.normalized\_laplacian\_matrix(G, weight='weight')

print(laplacian\_matrix)

normalized\_laplacian\_spectrum(G, weight='weight')

Esta função retorna uma lista com os valores próprios da matriz do Laplaciano normalizada de um gráfico G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (1, 3, {'weight': 3}), (2, 3, {'weight': 1}), (3, 4, {'weight': 4})])

spectrum = nx.normalized\_laplacian\_spectrum(G, weight='weight')

print(spectrum)

Com isso, concluímos a lista de funções do NetworkX. Lembre-se de consultar a documentação oficial do NetworkX para obter mais detalhes e s.

User

O mesmo para: 'null\_graph',

'number\_attracting\_components',

'number\_connected\_components',

'number\_of\_cliques',

'number\_of\_edges',

'number\_of\_isolates',

'number\_of\_nodes',

'number\_of\_nonisomorphic\_trees',

'number\_of\_selfloops',

'number\_strongly\_connected\_components',

ChatGPT

Claro! Continuando a lista de funções do NetworkX:

null\_graph(n, create\_using=None, seed=None)

Esta função retorna o grafo nulo (ou grafo vazio) com n nós.

:

import networkx as nx

null\_graph = nx.null\_graph(5)

print(null\_graph.nodes())

number\_attracting\_components(G)

Esta função retorna o número de componentes atrativos em um grafo direcionado G.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (4, 5)])

num\_attracting\_components = nx.number\_attracting\_components(G)

print(num\_attracting\_components)

number\_connected\_components(G)

Esta função retorna o número de componentes conectados em um grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (4, 5)])

num\_connected\_components = nx.number\_connected\_components(G)

print(num\_connected\_components)

number\_of\_cliques(G)

Esta função retorna o número de cliques no grafo não direcionado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

num\_cliques = nx.number\_of\_cliques(G)

print(num\_cliques)

number\_of\_edges(G)

Esta função retorna o número de arestas no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

num\_edges = nx.number\_of\_edges(G)

print(num\_edges)

number\_of\_isolates(G)

Esta função retorna o número de nós isolados (sem arestas) no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3)])

num\_isolates = nx.number\_of\_isolates(G)

print(num\_isolates)

number\_of\_nodes(G)

Esta função retorna o número de nós no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

num\_nodes = nx.number\_of\_nodes(G)

print(num\_nodes)

number\_of\_nonisomorphic\_trees(n)

Esta função retorna o número de árvores não isomórficas com n nós.

:

import networkx as nx

num\_nonisomorphic\_trees = nx.number\_of\_nonisomorphic\_trees(4)

print(num\_nonisomorphic\_trees)

number\_of\_selfloops(G)

Esta função retorna o número de laços de autoconexão no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 1), (1, 2), (2, 2), (3, 4)])

num\_selfloops = nx.number\_of\_selfloops(G)

print(num\_selfloops)

number\_strongly\_connected\_components(G)

Esta função retorna o número de componentes fortemente conectados em um grafo direcionado G.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (4, 5)])

num\_strongly\_connected\_components = nx.number\_strongly\_connected\_components(G)

print(num\_strongly\_connected\_components)

Espero que esta lista seja útil! Lembre-se de consultar a documentação oficial do NetworkX para mais detalhes e s.

User

O mesmo para: 'number\_weakly\_connected\_components',

'numeric\_assortativity\_coefficient',

'nx\_agraph',

'nx\_latex',

'nx\_pydot',

'nx\_pylab',

'octahedral\_graph',

'omega',

'onion\_layers',

'operators',

ChatGPT

Claro! Continuando a lista de funções do NetworkX:

number\_weakly\_connected\_components(G)

Esta função retorna o número de componentes fracamente conectados em um grafo direcionado G.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (4, 5)])

num\_weakly\_connected\_components = nx.number\_weakly\_connected\_components(G)

print(num\_weakly\_connected\_components)

numeric\_assortativity\_coefficient(G, attribute, nodes=None)

Esta função calcula o coeficiente de assortatividade numérico para um grafo G e um atributo numérico dado.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

# Adicionando atributo numérico aos nós

nx.set\_node\_attributes(G, {1: 10, 2: 5, 3: 8, 4: 3}, 'weight')

assortativity\_coefficient = nx.numeric\_assortativity\_coefficient(G, 'weight')

print(assortativity\_coefficient)

nx\_agraph

Esse módulo fornece funções para a leitura e gravação de grafos no formato DOT (usado pelo Graphviz) usando a biblioteca AGraph (https://github.com/chebee7i/nx\_agraph).

nx\_latex

Esse módulo fornece funções para a criação de gráficos LaTeX de grafos.

nx\_pydot

Esse módulo fornece funções para a leitura e gravação de grafos no formato DOT (usado pelo Graphviz) usando a biblioteca pydot (https://github.com/pydot/pydot).

nx\_pylab

Esse módulo fornece funções para plotagem de grafos usando a biblioteca Matplotlib (https://matplotlib.org/).

octahedral\_graph()

Esta função retorna o grafo octaédrico.

:

import networkx as nx

octahedral\_graph = nx.octahedral\_graph()

print(octahedral\_graph.nodes())

omega(G)

Esta função retorna o número de cliques máximo no grafo não direcionado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

omega\_value = nx.omega(G)

print(omega\_value)

onion\_layers(G, k)

Esta função retorna uma lista de camadas da cebola do grafo G, considerando k camadas.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1), (3, 5)])

onion\_layers\_list = nx.onion\_layers(G, 2)

print(onion\_layers\_list)

Espero que esta lista seja útil! Lembre-se de consultar a documentação oficial do NetworkX para mais detalhes e s.

User

O mesmo para: 'optimal\_edit\_paths',

'optimize\_edit\_paths',

'optimize\_graph\_edit\_distance',

'out\_degree\_centrality',

'overall\_reciprocity',

'pagerank',

'pagerank\_alg',

'pairs',

'pajek',

'paley\_graph',

ChatGPT

Claro! Continuando a lista de funções do NetworkX:

optimal\_edit\_paths(G1, G2)

Esta função retorna um gerador que produz caminhos de edição ótimos para transformar o grafo G1 no grafo G2. Esses caminhos de edição são compostos por operações como inserção, exclusão e relabeling de nós.

:

import networkx as nx

G1 = nx.Graph()

G1.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

G2 = nx.Graph()

G2.add\_edges\_from([(1, 3), (3, 4)])

edit\_paths\_generator = nx.optimal\_edit\_paths(G1, G2)

for path in edit\_paths\_generator:

print(path)

optimize\_edit\_paths(G1, G2)

Esta função retorna uma lista dos caminhos de edição ótimos para transformar o grafo G1 no grafo G2.

:

import networkx as nx

G1 = nx.Graph()

G1.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

G2 = nx.Graph()

G2.add\_edges\_from([(1, 3), (3, 4)])

edit\_paths\_list = nx.optimize\_edit\_paths(G1, G2)

for path in edit\_paths\_list:

print(path)

optimize\_graph\_edit\_distance(G1, G2, node\_match=None)

Esta função retorna a distância de edição otimizada entre os grafos G1 e G2. A distância de edição otimizada é o número mínimo de operações de edição (inserção, exclusão e relabeling de nós) necessárias para transformar G1 em G2.

:

import networkx as nx

G1 = nx.Graph()

G1.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

G2 = nx.Graph()

G2.add\_edges\_from([(1, 3), (3, 4)])

edit\_distance = nx.optimize\_graph\_edit\_distance(G1, G2)

print(edit\_distance)

out\_degree\_centrality(G)

Esta função calcula a centralidade de grau de saída para cada nó em um grafo direcionado G.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

out\_degree\_centralities = nx.out\_degree\_centrality(G)

print(out\_degree\_centralities)

overall\_reciprocity(G, weight=None)

Esta função calcula a reciprocidade geral de um grafo não direcionado G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

reciprocity = nx.overall\_reciprocity(G)

print(reciprocity)

pagerank(G, alpha=0.85, personalization=None, max\_iter=100, tol=1e-06, nstart=None, weight='weight', dangling=None)

Esta função calcula o PageRank para cada nó em um grafo G.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (4, 1)])

pagerank\_values = nx.pagerank(G, alpha=0.85)

print(pagerank\_values)

pagerank\_alg(G, alpha=0.85, personalization=None, max\_iter=100, tol=1e-06, nstart=None, weight='weight', dangling=None)

Esta função calcula o PageRank para cada nó em um grafo G, usando uma abordagem de álgebra linear.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (4, 1)])

pagerank\_values\_alg = nx.pagerank\_alg(G, alpha=0.85)

print(pagerank\_values\_alg)

pairs(G, source=None, target=None)

Esta função retorna um gerador que produz todos os pares de nós do grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

pairs\_generator = nx.pairs(G)

for pair in pairs\_generator:

print(pair)

pajek(G, path)

Esta função grava o grafo G em formato Pajek no caminho especificado.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

nx.pajek.write\_pajek(G, "my\_graph.net")

paley\_graph(q)

Esta função retorna o grafo de Paley com q nós.

:

import networkx as nx

paley\_graph = nx.paley\_graph(7)

print(paley\_graph.nodes())

Espero que esta lista seja útil! Lembre-se de consultar a documentação oficial do NetworkX para mais detalhes e s.

User

O mesmo para: 'panther\_similarity',

'pappus\_graph',

'parse\_adjlist',

'parse\_edgelist',

'parse\_gml',

'parse\_graphml',

'parse\_leda',

'parse\_multiline\_adjlist',

'parse\_pajek',

'partial\_duplication\_graph',

ChatGPT

Claro! Continuando com a lista de funções do NetworkX:

panther\_similarity(G, source=None, target=None)

Esta função calcula a similaridade Panther entre os nós do grafo G, especificados pelos parâmetros source e target.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3)])

similarity = nx.panther\_similarity(G, source=1, target=2)

print(similarity)

pappus\_graph()

Esta função retorna o grafo de Pappus.

:

import networkx as nx

pappus\_graph = nx.pappus\_graph()

print(pappus\_graph.nodes())

parse\_adjlist(lines)

Esta função analisa um arquivo ou lista de strings contendo uma lista de adjacências e retorna o grafo correspondente.

:

import networkx as nx

adjlist = [

"1 2 3",

"2 3",

"3",

"4 5",

"5"

]

G = nx.parse\_adjlist(adjlist)

print(G.edges())

parse\_edgelist(lines, comments='#', delimiter=None, create\_using=None, nodetype=None, data=True, edgetype=None)

Esta função analisa um arquivo ou lista de strings contendo uma lista de arestas e retorna o grafo correspondente.

:

import networkx as nx

edgelist = [

"1 2 {'weight': 3}",

"1 3 {'weight': 4}",

"2 3",

"3 4 {'weight': 2}",

"4 5 {'weight': 1}"

]

G = nx.parse\_edgelist(edgelist, data=True)

print(G.edges(data=True))

parse\_gml(data, label='label')

Esta função analisa um arquivo ou string contendo dados em formato GML (Graph Modeling Language) e retorna o grafo correspondente.

:

import networkx as nx

gml\_data = """

graph [

node [

id 1

label "Node 1"

]

node [

id 2

label "Node 2"

]

edge [

source 1

target 2

weight 3.5

]

]

"""

G = nx.parse\_gml(gml\_data)

print(G.nodes(data=True))

print(G.edges(data=True))

parse\_graphml(data, node\_type=<class 'int'>)

Esta função analisa um arquivo ou string contendo dados em formato GraphML e retorna o grafo correspondente.

:

import networkx as nx

graphml\_data = """

<graphml>

<graph id="G" edgedefault="undirected">

<node id="1">

<data key="label">Node 1</data>

</node>

<node id="2">

<data key="label">Node 2</data>

</node>

<edge source="1" target="2">

<data key="weight">3.5</data>

</edge>

</graph>

</graphml>

"""

G = nx.parse\_graphml(graphml\_data)

print(G.nodes(data=True))

print(G.edges(data=True))

parse\_leda(data)

Esta função analisa um arquivo ou string contendo dados em formato LEDA (Library of Efficient Data types and Algorithms) e retorna o grafo correspondente.

:

import networkx as nx

leda\_data = """

LEDA.GRAPH

string

node

-2

-1 1 "Node 1"

-2 2 "Node 2"

edge

-1 -2 3.5

"""

G = nx.parse\_leda(leda\_data)

print(G.nodes(data=True))

print(G.edges(data=True))

parse\_multiline\_adjlist(lines)

Esta função analisa um arquivo ou lista de strings contendo uma lista de adjacências em várias linhas e retorna o grafo correspondente.

:

import networkx as nx

multiline\_adjlist = [

"1 2 3",

"2 3",

"3",

"4 5",

"5"

]

G = nx.parse\_multiline\_adjlist(multiline\_adjlist)

print(G.edges())

parse\_pajek(path)

Esta função analisa um arquivo Pajek e retorna o grafo correspondente.

:

import networkx as nx

G = nx.parse\_pajek("my\_graph.net")

print(G.edges())

partial\_duplication\_graph(G, n, seed=None)

Esta função retorna um novo grafo obtido a partir do grafo G pela duplicação parcial de cada nó n vezes.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (3, 4)])

partial\_duplication = nx.partial\_duplication\_graph(G, n=2)

print(partial\_duplication.edges())

Espero que esta lista de funções do NetworkX seja útil! Lembre-se de consultar a documentação oficial do NetworkX para mais detalhes e s.

User

O mesmo para: 'partition\_spanning\_tree',

'path\_graph',

'path\_weight',

'percolation',

'percolation\_centrality',

'periphery',

'petersen\_graph',

'planar\_drawing',

'planar\_layout',

'planarity',

ChatGPT

Claro! Continuando com a lista de funções do NetworkX:

partition\_spanning\_tree(G, subsets)

Esta função retorna a árvore geradora do grafo G, restrita aos nós especificados nas partes do conjunto de conjuntos subsets.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (1, 4), (2, 3), (3, 4)])

subsets = [{1, 2}, {3, 4}]

partition\_tree = nx.partition\_spanning\_tree(G, subsets)

print(partition\_tree.edges())

path\_graph(n, create\_using=None)

Esta função retorna o grafo de caminho com n nós, que é uma sequência linear de nós conectados por arestas.

:

import networkx as nx

path\_graph = nx.path\_graph(5)

print(path\_graph.edges())

path\_weight(G, weight='weight')

Esta função calcula o peso total de um caminho G usando o atributo de peso especificado.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 3}), (2, 3, {'weight': 2}), (3, 4, {'weight': 4})])

weight = nx.path\_weight(G, weight='weight')

print(weight)

percolation(G, p, seed=None)

Esta função retorna um grafo percolado a partir do grafo original G, onde cada aresta é mantida com probabilidade p.

:

import networkx as nx

G = nx.erdos\_renyi\_graph(10, 0.3)

percolated\_graph = nx.percolation(G, p=0.5)

print(percolated\_graph.edges())

percolation\_centrality(G, weight=None)

Esta função calcula a centralidade de percolação para todos os nós em um grafo G, considerando a conectividade dos nós através de caminhos percolados.

:

import networkx as nx

G = nx.karate\_club\_graph()

percolation\_centralities = nx.percolation\_centrality(G)

print(percolation\_centralities)

periphery(G)

Esta função retorna um conjunto de nós que formam a periferia do grafo G, ou seja, os nós mais distantes do centro do grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 4), (3, 5), (4, 5)])

periphery\_nodes = nx.periphery(G)

print(periphery\_nodes)

petersen\_graph(create\_using=None)

Esta função retorna o grafo de Petersen, um grafo cúbico com 10 nós e 15 arestas.

:

import networkx as nx

petersen\_graph = nx.petersen\_graph()

print(petersen\_graph.edges())

planar\_drawing(G, \*\*kwargs)

Esta função retorna um layout planar do grafo G usando o algoritmo de desenho planar.

:

import networkx as nx

G = nx.erdos\_renyi\_graph(10, 0.3)

pos = nx.planar\_drawing(G)

print(pos)

planar\_layout(G, scale=1, center=None, dim=2)

Esta função retorna um layout planar do grafo G usando um algoritmo de desenho específico.

:

import networkx as nx

G = nx.erdos\_renyi\_graph(10, 0.3)

pos = nx.planar\_layout(G)

print(pos)

planarity(G, counterexample=False)

Esta função verifica se o grafo G é planar e, se counterexample=True, retorna um contra que mostra por que o grafo não é planar.

:

import networkx as nx

G = nx.complete\_graph(5)

is\_planar, counterexample = nx.planarity(G, counterexample=True)

print(is\_planar)

if not is\_planar:

print(counterexample.edges())

Espero que esta lista de funções do NetworkX seja útil! Lembre-se de consultar a documentação oficial do NetworkX para mais detalhes e s.

User

O mesmo para: 'planted\_partition\_graph',

'polynomials',

'power',

'powerlaw\_cluster\_graph',

'predecessor',

'preferential\_attachment',

'prefix\_tree',

'prefix\_tree\_recursive',

'product',

'projected\_graph',

ChatGPT

Claro! Continuando a lista de funções do NetworkX:

planted\_partition\_graph(l, k, p\_in, p\_out, seed=None)

Esta função retorna um grafo bipartido plantado com l nós em cada conjunto, onde arestas internas entre os conjuntos têm probabilidade p\_in de serem adicionadas e arestas externas têm probabilidade p\_out de serem adicionadas.

:

import networkx as nx

planted\_graph = nx.planted\_partition\_graph(l=5, k=3, p\_in=0.8, p\_out=0.2)

print(planted\_graph.edges())

polynomials(G, name='x')

Esta função calcula os polinômios do grafo G usando a variável name.

:

import networkx as nx

G = nx.cycle\_graph(4)

polynomials = nx.polynomials(G)

print(polynomials)

power(G, n, seed=None)

Esta função retorna uma cópia do grafo G com os nós conectados a seus vizinhos a uma distância máxima de n.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 3), (2, 4), (4, 5)])

powered\_graph = nx.power(G, n=2)

print(powered\_graph.edges())

powerlaw\_cluster\_graph(n, m, p, seed=None)

Esta função retorna um grafo de clusterização com n nós, onde cada novo nó é conectado a m nós existentes com probabilidade p de se conectar a um nó de grau i com probabilidade proporcional a i^{-α}, onde α é um parâmetro de distribuição de energia.

:

import networkx as nx

powerlaw\_graph = nx.powerlaw\_cluster\_graph(n=20, m=2, p=0.3)

print(powerlaw\_graph.edges())

predecessor(G, source, target, cutoff=None)

Esta função retorna uma lista dos predecessores de target em um grafo G, no caminho mais curto a partir de source. Se um limite (cutoff) for especificado, apenas os predecessores dentro do limite serão retornados.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (1, 3), (2, 4), (3, 4), (4, 5)])

predecessors = nx.predecessor(G, source=1, target=5)

print(predecessors)

preferential\_attachment(G, n, seed=None)

Esta função retorna um grafo crescente de anexação preferencial, iniciando com um grafo G e adicionando n nós, cada um conectado a um nó existente de G, com probabilidade proporcional ao grau desse nó.

:

import networkx as nx

G = nx.barabasi\_albert\_graph(10, 2)

preferential\_graph = nx.preferential\_attachment(G, n=5)

print(preferential\_graph.edges())

prefix\_tree(words)

Esta função retorna uma árvore de prefixos (trie) para uma lista de palavras.

:

import networkx as nx

words = ['apple', 'app', 'apricot', 'banana', 'bat']

prefix\_tree = nx.prefix\_tree(words)

print(prefix\_tree.edges())

prefix\_tree\_recursive(words)

Esta função retorna uma árvore de prefixos (trie) para uma lista de palavras usando uma abordagem recursiva.

:

import networkx as nx

words = ['apple', 'app', 'apricot', 'banana', 'bat']

prefix\_tree = nx.prefix\_tree\_recursive(words)

print(prefix\_tree.edges())

product(G, H, create\_using=None)

Esta função retorna o produto cartesiano de dois grafos G e H, onde os nós do produto são os pares (u, v) com u em G e v em H, e as arestas são adicionadas entre os nós de acordo com as arestas em G e H.

:

import networkx as nx

G = nx.cycle\_graph(3)

H = nx.path\_graph(2)

product\_graph = nx.product(G, H)

print(product\_graph.edges())

projected\_graph(G, nodes, multigraph=False)

Esta função retorna o grafo projetado a partir do grafo bipartido G para um conjunto de nós específico.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 'A'), (2, 'B'), (3, 'A'), (3, 'B'), (4, 'A')])

projected\_graph = nx.projected\_graph(G, nodes=['A', 'B'])

print(projected\_graph.edges())

Espero que esta lista de funções do NetworkX seja útil! Lembre-se de consultar a documentação oficial do NetworkX para mais detalhes e s.

User

O mesmo para: 'prominent\_group',

'quotient\_graph',

'ra\_index\_soundarajan\_hopcroft',

'radius',

'random\_clustered',

'random\_clustered\_graph',

'random\_cograph',

'random\_degree\_sequence\_graph',

'random\_geometric\_graph',

'random\_graphs',

ChatGPT

Claro! Continuando com a lista de funções do NetworkX:

prominent\_group(G)

Esta função retorna um grupo proeminente em um grafo ponderado não direcionado G. Um grupo proeminente é um conjunto de nós que têm uma pontuação alta no índice RAS (Resource Allocation Similarity) com base nas arestas ponderadas do grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_weighted\_edges\_from([(1, 2, 0.5), (1, 3, 0.8), (2, 3, 0.4), (2, 4, 0.6)])

prominent\_group = nx.prominent\_group(G)

print(prominent\_group)

quotient\_graph(G, partition, relabel=False)

Esta função retorna o grafo quociente de G com base em uma partição dos nós. A partição é um dicionário que mapeia cada nó em G para sua classe de equivalência.

:

import networkx as nx

G = nx.cycle\_graph(6)

partition = {0: 0, 1: 0, 2: 1, 3: 1, 4: 2, 5: 2}

quotient\_graph = nx.quotient\_graph(G, partition)

print(quotient\_graph.edges())

ra\_index\_soundarajan\_hopcroft(G, ebunch=None)

Esta função retorna um dicionário contendo o índice RAS (Resource Allocation Similarity) entre todos os pares de nós em ebunch no grafo não direcionado ponderado G.

:

import networkx as nx

G = nx.Graph()

G.add\_weighted\_edges\_from([(1, 2, 0.5), (1, 3, 0.8), (2, 3, 0.4), (2, 4, 0.6)])

ras\_dict = nx.ra\_index\_soundarajan\_hopcroft(G)

print(ras\_dict)

radius(G, e=None)

Esta função retorna o raio de G, que é a menor distância máxima entre todos os pares de nós. Se e for fornecido, o raio será calculado apenas para os nós em e.

:

import networkx as nx

G = nx.cycle\_graph(4)

radius = nx.radius(G)

print(radius)

random\_clustered(n, p, q, seed=None)

Esta função retorna um grafo aleatório de tamanho n com uma estrutura de comunidades usando o modelo de clusterização aleatória.

:

import networkx as nx

random\_clustered\_graph = nx.random\_clustered(n=20, p=0.5, q=0.1)

print(random\_clustered\_graph.edges())

random\_clustered\_graph(n, s, v, p\_in, p\_out, seed=None)

Esta função retorna um grafo aleatório com n nós, com s nós em cada comunidade e v comunidades, onde arestas internas têm probabilidade p\_in de serem adicionadas e arestas externas têm probabilidade p\_out de serem adicionadas.

:

import networkx as nx

random\_clustered\_graph = nx.random\_clustered\_graph(n=20, s=5, v=4, p\_in=0.8, p\_out=0.2)

print(random\_clustered\_graph.edges())

random\_cograph(n, seed=None)

Esta função retorna um cograph aleatório com n nós.

:

import networkx as nx

random\_cograph = nx.random\_cograph(n=10)

print(random\_cograph.edges())

random\_degree\_sequence\_graph(sequence, seed=None)

Esta função retorna um grafo aleatório com a sequência de graus dada.

:

import networkx as nx

sequence = [3, 2, 2, 1, 1]

random\_degree\_graph = nx.random\_degree\_sequence\_graph(sequence)

print(random\_degree\_graph.edges())

random\_geometric\_graph(n, radius, dim=2, pos=None, p=2, seed=None)

Esta função retorna um grafo aleatório geométrico de n nós, onde os nós estão em uma caixa unitária e as arestas são adicionadas entre os nós cuja distância euclidiana é menor que radius.

:

import networkx as nx

random\_geometric\_graph = nx.random\_geometric\_graph(n=10, radius=0.2)

print(random\_geometric\_graph.edges())

random\_graphs(n, p, seed=None, directed=False)

Esta função retorna um grafo aleatório com n nós, onde cada par de nós é conectado com probabilidade p.

:

import networkx as nx

random\_graph = nx.random\_graphs(n=10, p=0.3)

print(random\_graph.edges())

Espero que esta lista de funções do NetworkX seja útil! Lembre-se de consultar a documentação oficial do NetworkX para mais detalhes e s.

User

O mesmo para: 'random\_internet\_as\_graph',

'random\_k\_out\_graph',

'random\_kernel\_graph',

'random\_layout',

'random\_lobster',

'random\_partition\_graph',

'random\_powerlaw\_tree',

'random\_powerlaw\_tree\_sequence',

'random\_reference',

'random\_regular\_graph',

ChatGPT

Claro! Continuando com a lista de funções do NetworkX:

random\_internet\_as\_graph(n, k, seed=None)

Esta função retorna um grafo aleatório da Internet com n nós e k conexões de saída para cada nó.

:

import networkx as nx

random\_internet\_graph = nx.random\_internet\_as\_graph(n=100, k=3)

print(random\_internet\_graph.edges())

random\_k\_out\_graph(n, k, alpha=0.5, beta=0.5, seed=None)

Esta função retorna um grafo aleatório com n nós, onde cada nó tem exatamente k conexões de saída, escolhidas com base em um modelo com parâmetros alpha e beta.

:

import networkx as nx

random\_k\_out\_graph = nx.random\_k\_out\_graph(n=50, k=2, alpha=0.3, beta=0.7)

print(random\_k\_out\_graph.edges())

random\_kernel\_graph(n, kernel, seed=None)

Esta função retorna um grafo aleatório com n nós, onde as conexões são formadas de acordo com um kernel especificado.

:

import networkx as nx

kernel = lambda x, y: abs(x - y) <= 2

random\_kernel\_graph = nx.random\_kernel\_graph(n=20, kernel=kernel)

print(random\_kernel\_graph.edges())

random\_layout(G, dim=2, seed=None)

Esta função retorna um layout aleatório de um grafo G em um espaço de dimensões dim.

:

import networkx as nx

import matplotlib.pyplot as plt

G = nx.path\_graph(5)

layout = nx.random\_layout(G)

nx.draw(G, pos=layout, with\_labels=True)

plt.show()

random\_lobster(n, p1, p2, seed=None)

Esta função retorna um grafo aleatório do tipo "lobster" com n nós, onde a estrutura de árvore é gerada com probabilidade p1 e a estrutura do caranguejo é gerada com probabilidade p2.

:

import networkx as nx

random\_lobster\_graph = nx.random\_lobster(n=30, p1=0.4, p2=0.2)

print(random\_lobster\_graph.edges())

random\_partition\_graph(sizes, p\_in, p\_out, seed=None)

Esta função retorna um grafo aleatório com base em uma partição aleatória.

:

import networkx as nx

sizes = [10, 15, 20]

p\_in = 0.6

p\_out = 0.2

random\_partition\_graph = nx.random\_partition\_graph(sizes, p\_in, p\_out)

print(random\_partition\_graph.edges())

random\_powerlaw\_tree(n, gamma=3, seed=None)

Esta função retorna uma árvore aleatória com n nós, onde o grau dos nós é distribuído de acordo com uma distribuição de lei de potência com o expoente gamma.

:

import networkx as nx

random\_powerlaw\_tree = nx.random\_powerlaw\_tree(n=50, gamma=2.5)

print(random\_powerlaw\_tree.edges())

random\_powerlaw\_tree\_sequence(n, gamma=3, tries=100, seed=None)

Esta função retorna uma sequência de árvores aleatórias com n nós, onde o grau dos nós é distribuído de acordo com uma distribuição de lei de potência com o expoente gamma.

:

import networkx as nx

random\_powerlaw\_trees = nx.random\_powerlaw\_tree\_sequence(n=50, gamma=2.5, tries=10)

print(random\_powerlaw\_trees[0].edges())

random\_reference(G, niter=10, seed=None)

Esta função retorna um grafo aleatório com base em um grafo dado G, usando o algoritmo de referência aleatória.

:

import networkx as nx

G = nx.cycle\_graph(6)

random\_reference\_graph = nx.random\_reference(G, niter=20)

print(random\_reference\_graph.edges())

random\_regular\_graph(d, n, seed=None)

Esta função retorna um grafo aleatório regular com n nós e grau d.

:

import networkx as nx

random\_regular\_graph = nx.random\_regular\_graph(d=3, n=10)

print(random\_regular\_graph.edges())

Espero que esta lista de funções do NetworkX seja útil! Lembre-se de consultar a documentação oficial do NetworkX para mais detalhes e s.

User

O mesmo para: 'random\_shell\_graph',

'random\_spanning\_tree',

'random\_tree',

'random\_triad',

'reaching',

'read\_adjlist',

'read\_edgelist',

'read\_gexf',

'read\_gml',

'read\_graph6',

ChatGPT

Claro! Continuando com a lista de funções do NetworkX:

random\_shell\_graph(sizes, nodelist=None, seed=None)

Esta função retorna um grafo aleatório com base em uma sequência de tamanhos de casca sizes.

:

import networkx as nx

sizes = [5, 10, 15]

random\_shell\_graph = nx.random\_shell\_graph(sizes)

print(random\_shell\_graph.edges())

random\_spanning\_tree(G, seed=None)

Esta função retorna uma árvore geradora aleatória para o grafo não direcionado G.

:

import networkx as nx

G = nx.complete\_graph(5)

random\_spanning\_tree = nx.random\_spanning\_tree(G)

print(random\_spanning\_tree.edges())

random\_tree(n, seed=None)

Esta função retorna uma árvore aleatória com n nós.

:

import networkx as nx

random\_tree = nx.random\_tree(10)

print(random\_tree.edges())

random\_triad(n, seed=None)

Esta função retorna um grafo aleatório com n nós e todas as possíveis arestas em triângulos.

:

import networkx as nx

random\_triad\_graph = nx.random\_triad(5)

print(random\_triad\_graph.edges())

reaching(G, source=None, target=None, cutoff=None)

Esta função retorna um dicionário com os nós acessíveis a partir de um determinado nó de origem em um grafo direcionado.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (1, 4), (4, 5)])

reachable\_nodes = nx.reaching(G, source=1)

print(reachable\_nodes)

read\_adjlist(path, comments='#', delimiter=None, create\_using=None, nodetype=None)

Esta função lê um grafo a partir de um arquivo no formato de lista de adjacências.

:

import networkx as nx

G = nx.read\_adjlist("my\_graph.adjlist")

print(G.edges())

read\_edgelist(path, comments='#', delimiter=None, create\_using=None, nodetype=None, data=True, edgetype=None)

Esta função lê um grafo a partir de um arquivo no formato de lista de arestas.

:

import networkx as nx

G = nx.read\_edgelist("my\_graph.edgelist")

print(G.edges())

read\_gexf(path, node\_type=None, relabel=False)

Esta função lê um grafo no formato GEXF (Graph Exchange XML Format).

:

import networkx as nx

G = nx.read\_gexf("my\_graph.gexf")

print(G.edges())

read\_gml(path, node\_type=None, relabel=False)

Esta função lê um grafo no formato GML (Graph Modeling Language).

:

import networkx as nx

G = nx.read\_gml("my\_graph.gml")

print(G.edges())

read\_graph6(lines)

Esta função lê um grafo no formato graph6.

:

import networkx as nx

graph6\_string = "G?@A\_"

G = nx.read\_graph6(graph6\_string)

print(G.edges())

Espero que estas informações sejam úteis! Lembre-se de consultar a documentação oficial do NetworkX para mais detalhes e s.

User

O mesmo para: 'read\_graphml',

'read\_leda',

'read\_multiline\_adjlist',

'read\_pajek',

'read\_sparse6',

'read\_weighted\_edgelist',

'readwrite',

'reciprocity',

'reconstruct\_path',

'recursive\_simple\_cycles',

ChatGPT

Claro, continuando com a lista:

read\_graphml(path, node\_type=None, edge\_key\_type=None)

Esta função lê um grafo no formato GraphML.

:

import networkx as nx

G = nx.read\_graphml("my\_graph.graphml")

print(G.edges())

read\_leda(lines)

Esta função lê um grafo no formato LEDA.

:

import networkx as nx

leda\_string = "LEDA.GRAPH\nvoid\n"

G = nx.read\_leda(leda\_string)

print(G.edges())

read\_multiline\_adjlist(path, comments='#', delimiter=None, nodetype=None, create\_using=None)

Esta função lê um grafo a partir de um arquivo no formato de lista de adjacências com várias linhas.

:

import networkx as nx

G = nx.read\_multiline\_adjlist("my\_graph\_multiline.adjlist")

print(G.edges())

read\_pajek(path, encoding='UTF-8')

Esta função lê um grafo no formato Pajek.

:

import networkx as nx

G = nx.read\_pajek("my\_graph.net")

print(G.edges())

read\_sparse6(lines)

Esta função lê um grafo no formato sparse6.

:

import networkx as nx

sparse6\_string = ":Ig@AiD?A\_"

G = nx.read\_sparse6(sparse6\_string)

print(G.edges())

read\_weighted\_edgelist(path, delimiter=' ', nodetype=None, create\_using=None)

Esta função lê um grafo a partir de um arquivo no formato de lista de arestas com pesos.

:

import networkx as nx

G = nx.read\_weighted\_edgelist("my\_weighted\_graph.edgelist")

print(G.edges(data=True))

reciprocity(G, nodes=None, weight=None)

Esta função calcula a reciprocidade do grafo ou reciprocidade média dos nós.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 1), (1, 3)])

reciprocity = nx.reciprocity(G)

print(reciprocity)

reconstruct\_path(predecessors, source, target)

Esta função reconstrói o caminho de um nó origem a um nó destino a partir de uma lista de predecessores.

:

import networkx as nx

predecessors = {1: None, 2: 1, 3: 2}

path = nx.reconstruct\_path(predecessors, 1, 3)

print(path)

recursive\_simple\_cycles(G)

Esta função encontra todos os ciclos simples em um grafo direcionado usando um algoritmo recursivo.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

cycles = nx.recursive\_simple\_cycles(G)

print(cycles)

Espero que isso ajude! Como sempre, consulte a documentação oficial do NetworkX para mais detalhes e s.

User

O mesmo para: 'regular',

'relabel',

'relabel\_gexf\_graph',

'relabel\_nodes',

'relaxed\_caveman\_graph',

'reportviews',

'rescale\_layout',

'rescale\_layout\_dict',

'resistance\_distance',

'resource\_allocation\_index',

ChatGPT

Claro, aqui estão mais funções do NetworkX:

regular(G)

Esta função verifica se um grafo é regular, ou seja, se todos os seus nós têm o mesmo grau.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

is\_regular = nx.regular(G)

print(is\_regular)

relabel(G, mapping)

Esta função retorna uma cópia do grafo com os nós renomeados de acordo com o mapeamento fornecido.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

mapping = {1: 'A', 2: 'B', 3: 'C'}

H = nx.relabel(G, mapping)

print(H.nodes())

relabel\_gexf\_graph(G)

Esta função converte os rótulos dos nós de um grafo de acordo com o mapeamento de rótulos no formato GEXF.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

mapping = {1: 'A', 2: 'B', 3: 'C'}

G = nx.relabel\_gexf\_graph(G, mapping)

print(G.nodes())

relabel\_nodes(G, mapping)

Esta função retorna uma cópia do grafo com os nós renomeados de acordo com o mapeamento fornecido.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

mapping = {1: 'A', 2: 'B', 3: 'C'}

H = nx.relabel\_nodes(G, mapping)

print(H.nodes())

relaxed\_caveman\_graph(l, k, p, seed=None)

Esta função retorna um grafo "relaxed caveman" com l cliques de tamanho k, onde cada aresta tem uma probabilidade p de ser removida.

:

import networkx as nx

G = nx.relaxed\_caveman\_graph(3, 4, 0.2)

print(G.edges())

reportviews

Este é um módulo contendo classes que fornecem visualizações de informações sobre grafos.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

report = nx.reportviews()

print(report.degree(G))

rescale\_layout(pos, scale=1, center=None)

Esta função retorna uma nova posição das coordenadas dos nós, escalonando a posição original.

:

import networkx as nx

pos = {1: (0, 0), 2: (1, 1), 3: (2, 2)}

new\_pos = nx.rescale\_layout(pos, scale=2)

print(new\_pos)

rescale\_layout\_dict(pos, scale=1, center=None)

Esta função retorna um dicionário de novas posições das coordenadas dos nós, escalonando a posição original.

:

import networkx as nx

pos = {1: (0, 0), 2: (1, 1), 3: (2, 2)}

new\_pos = nx.rescale\_layout\_dict(pos, scale=2)

print(new\_pos)

resistance\_distance(G, source=None, target=None, weight=None)

Esta função calcula a resistência entre dois nós de um grafo, que é o inverso da eficiência.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 1, {'weight': 1})])

resistance = nx.resistance\_distance(G, source=1, target=3, weight='weight')

print(resistance)

resource\_allocation\_index(G, ebunch=None, nodelist=None)

Esta função calcula o índice de alocação de recursos entre pares de nós em um grafo.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

allocation\_index = nx.resource\_allocation\_index(G)

print(allocation\_index)

Espero que esta lista seja útil! Lembre-se sempre de consultar a documentação oficial do NetworkX para obter mais detalhes e s.

User

O mesmo para: 'restricted\_view',

'reverse',

'reverse\_view',

'rich\_club\_coefficient',

'richclub',

'ring\_of\_cliques',

'rooted\_product',

's\_metric',

'scale\_free\_graph',

'second\_order',

ChatGPT

Claro! Aqui estão mais algumas funções do NetworkX:

restricted\_view(G, nbunch)

Esta função retorna uma exibição restrita do grafo contendo apenas os nós especificados em nbunch.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

view = nx.restricted\_view(G, [1, 3])

print(view.nodes())

reverse(G)

Esta função retorna uma cópia do grafo com todas as arestas invertidas.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

R = nx.reverse(G)

print(R.edges())

reverse\_view(G)

Esta função retorna uma exibição reversa do grafo, onde todas as arestas são invertidas, mas os dados são compartilhados com o grafo original.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

view = nx.reverse\_view(G)

print(view.edges())

rich\_club\_coefficient(G, normalized=False)

Esta função calcula o coeficiente de clube rico do grafo, que mede a conectividade entre nós com alta conectividade.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1), (1, 3)])

coefficient = nx.rich\_club\_coefficient(G)

print(coefficient)

richclub

Esta é uma subpasta contendo funções relacionadas a coeficientes de clube rico em grafos.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1), (1, 3)])

coefficient = nx.richclub.rich\_club\_coefficient(G)

print(coefficient)

ring\_of\_cliques(clique\_size, number\_of\_cliques)

Esta função retorna um grafo em forma de anel, onde os nós são cliques conectados em um anel.

:

import networkx as nx

G = nx.ring\_of\_cliques(3, 4)

print(G.edges())

rooted\_product(G, source)

Esta função retorna um novo grafo que é o produto enraizado do grafo original em relação a um nó raiz.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

H = nx.rooted\_product(G, 1)

print(H.edges())

s\_metric(G, k, weight='weight', normalized=True)

Esta função calcula a métrica S para um grafo e um valor k especificado.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 1, {'weight': 1})])

s\_metric = nx.s\_metric(G, k=2, weight='weight')

print(s\_metric)

scale\_free\_graph(n, alpha=0.41, beta=0.54, gamma=0.05, delta\_in=0.2, delta\_out=0, create\_using=None, seed=None)

Esta função retorna um grafo de escala livre com o número especificado de nós.

:

import networkx as nx

G = nx.scale\_free\_graph(20)

print(G.edges())

second\_order(G, nbunch=None)

Esta função retorna um dicionário contendo os vizinhos de segundo nível para cada nó em nbunch.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

second\_order\_neighbors = nx.second\_order(G, [1, 2])

print(second\_order\_neighbors)

Espero que esta lista seja útil! Consulte sempre a documentação oficial do NetworkX para obter mais detalhes e s.

User

O mesmo para: 'second\_order\_centrality',

'sedgewick\_maze\_graph',

'selfloop\_edges',

'semiconnected',

'set\_edge\_attributes',

'set\_node\_attributes',

'shell\_layout',

'shortest\_path',

'shortest\_path\_length',

'shortest\_paths',

ChatGPT

Claro! Aqui estão mais algumas funções do NetworkX:

second\_order\_centrality(G, nbunch=None, weight=None)

Esta função calcula a centralidade de segundo nível para cada nó em nbunch.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

second\_order\_centrality = nx.second\_order\_centrality(G)

print(second\_order\_centrality)

`sedgewick\_maze\_graph()

Esta função retorna o labirinto de Sedgewick, que é um grafo interessante com muitos ciclos e ligações complicadas.

:

import networkx as nx

G = nx.sedgewick\_maze\_graph()

print(G.edges())

selfloop\_edges(G)

Esta função retorna uma lista de arestas que são auto-laços (loop) no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 1)

G.add\_edge(2, 3)

selfloop\_edges = nx.selfloop\_edges(G)

print(list(selfloop\_edges))

semiconnected(G)

Esta função verifica se o grafo é semiconexo, ou seja, se existe um caminho entre todos os pares de nós.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1)])

print(nx.semiconnected(G))

set\_edge\_attributes(G, values, name=None)

Esta função define os atributos de todas as arestas no grafo G a partir do dicionário de valores fornecido.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

values = {(1, 2): 5, (2, 3): 10, (3, 4): 15}

nx.set\_edge\_attributes(G, values, name='weight')

print(G.edges(data=True))

set\_node\_attributes(G, values, name=None)

Esta função define os atributos de todos os nós no grafo G a partir do dicionário de valores fornecido.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4)])

values = {1: 'A', 2: 'B', 3: 'C', 4: 'D'}

nx.set\_node\_attributes(G, values, name='label')

print(G.nodes(data=True))

shell\_layout(G, nlist=None, dim=2, scale=1, center=None)

Esta função posiciona os nós em um layout de concha (shell layout) com base nas listas de nós especificadas em nlist.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

pos = nx.shell\_layout(G, [[1, 2], [3, 4]])

print(pos)

shortest\_path(G, source, target, weight=None, method='dijkstra')

Esta função retorna uma lista contendo os nós ao longo do caminho mais curto entre source e target no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

path = nx.shortest\_path(G, 1, 4, weight='weight')

print(path)

shortest\_path\_length(G, source, target, weight=None, method='dijkstra')

Esta função retorna o comprimento do caminho mais curto entre source e target no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

length = nx.shortest\_path\_length(G, 1, 4, weight='weight')

print(length)

shortest\_paths(G, source=None, target=None, weight=None, method='dijkstra')

Esta função retorna um dicionário de dicionários contendo os caminhos mais curtos entre pares de nós.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

paths = nx.shortest\_paths(G, weight='weight')

print(paths)

Espero que esta lista seja útil! Consulte sempre a documentação oficial do NetworkX para obter mais detalhes e s.

User

O mesmo para: 'shortest\_simple\_paths',

'sigma',

'similarity',

'simple\_cycles',

'simple\_paths',

'simrank\_similarity',

'single\_source\_bellman\_ford',

'single\_source\_bellman\_ford\_path',

'single\_source\_bellman\_ford\_path\_length',

'single\_source\_dijkstra',

ChatGPT

Claro! Aqui estão mais algumas funções do NetworkX:

shortest\_simple\_paths(G, source, target, weight=None)

Esta função retorna todos os caminhos mais curtos simples entre source e target no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

paths = nx.shortest\_simple\_paths(G, source=1, target=4, weight='weight')

for path in paths:

print(path)

sigma(G, u=None, v=None, weight=None, nodes=None)

Esta função calcula a medida de similaridade de Sigma entre dois nós u e v em relação a um conjunto de nós nodes no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

similarity = nx.sigma(G, u=1, v=4, weight='weight')

print(similarity)

similarity(G, source=None, target=None, weight='weight', method='Katz')

Esta função calcula a medida de similaridade entre dois nós source e target no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

similarity = nx.similarity(G, source=1, target=4, weight='weight', method='Katz')

print(similarity)

simple\_cycles(G)

Esta função retorna todos os ciclos simples no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 1), (3, 4), (4, 5), (5, 3)])

cycles = nx.simple\_cycles(G)

for cycle in cycles:

print(cycle)

simple\_paths(G, source, target, cutoff=None)

Esta função retorna todos os caminhos simples entre source e target no grafo G com um comprimento máximo de corte (cutoff).

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 5)])

paths = nx.simple\_paths(G, source=1, target=5)

for path in paths:

print(path)

simrank\_similarity(G, source=None, target=None, C=0.8, max\_iter=100, weight=None)

Esta função calcula a medida de similaridade SimRank entre dois nós source e target no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

similarity = nx.simrank\_similarity(G, source=1, target=4, weight='weight')

print(similarity)

single\_source\_bellman\_ford(G, source, target=None, cutoff=None, weight='weight')

Esta função encontra o caminho mais curto de um único nó de origem para todos os outros nós no grafo G usando o algoritmo de Bellman-Ford.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': -3}), (3, 4, {'weight': 1})])

shortest\_paths = nx.single\_source\_bellman\_ford(G, source=1, weight='weight')

print(shortest\_paths)

single\_source\_bellman\_ford\_path(G, source, target, weight='weight')

Esta função encontra o caminho mais curto de um único nó de origem para um nó alvo no grafo G usando o algoritmo de Bellman-Ford.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': -3}), (3, 4, {'weight': 1})])

shortest\_path = nx.single\_source\_bellman\_ford\_path(G, source=1, target=4, weight='weight')

print(shortest\_path)

single\_source\_bellman\_ford\_path\_length(G, source, target, weight='weight')

Esta função encontra o comprimento do caminho mais curto de um único nó de origem para um nó alvo no grafo G usando o algoritmo de Bellman-Ford.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': -3}), (3, 4, {'weight': 1})])

path\_length = nx.single\_source\_bellman\_ford\_path\_length(G, source=1, target=4, weight='weight')

print(path\_length)

single\_source\_dijkstra(G, source, target=None, cutoff=None, weight='weight')

Esta função encontra o caminho mais curto de um único nó de origem para todos os outros nós no grafo G usando o algoritmo de Dijkstra.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

shortest\_paths = nx.single\_source\_dijkstra(G, source=1, weight='weight')

print(shortest\_paths)

Espero que esta lista seja útil! Consulte sempre a documentação oficial do NetworkX para obter mais detalhes e s.

User

O mesmo para: 'single\_source\_dijkstra\_path',

'single\_source\_dijkstra\_path\_length',

'single\_source\_shortest\_path',

'single\_source\_shortest\_path\_length',

'single\_target\_shortest\_path',

'single\_target\_shortest\_path\_length',

'small',

'smallworld',

'smetric',

'snap\_aggregation',

ChatGPT

Claro! Aqui estão mais algumas funções do NetworkX:

single\_source\_dijkstra\_path(G, source, target=None, cutoff=None, weight='weight')

Esta função encontra o caminho mais curto de um único nó de origem para todos os outros nós no grafo G usando o algoritmo de Dijkstra e retorna o caminho como um dicionário.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

shortest\_paths = nx.single\_source\_dijkstra\_path(G, source=1, weight='weight')

print(shortest\_paths)

single\_source\_dijkstra\_path\_length(G, source, target=None, cutoff=None, weight='weight')

Esta função encontra o comprimento do caminho mais curto de um único nó de origem para todos os outros nós no grafo G usando o algoritmo de Dijkstra e retorna um dicionário contendo os comprimentos do caminho.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

path\_lengths = nx.single\_source\_dijkstra\_path\_length(G, source=1, weight='weight')

print(path\_lengths)

single\_source\_shortest\_path(G, source, cutoff=None)

Esta função encontra todos os caminhos mais curtos de um único nó de origem para todos os outros nós no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (1, 3), (3, 4)])

shortest\_paths = nx.single\_source\_shortest\_path(G, source=1)

print(shortest\_paths)

single\_source\_shortest\_path\_length(G, source, cutoff=None)

Esta função encontra os comprimentos de todos os caminhos mais curtos de um único nó de origem para todos os outros nós no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (1, 3), (3, 4)])

path\_lengths = nx.single\_source\_shortest\_path\_length(G, source=1)

print(path\_lengths)

single\_target\_shortest\_path(G, target, cutoff=None)

Esta função encontra todos os caminhos mais curtos de todos os nós de origem para um único nó de destino no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (1, 3), (3, 4)])

shortest\_paths = nx.single\_target\_shortest\_path(G, target=4)

print(shortest\_paths)

single\_target\_shortest\_path\_length(G, target, cutoff=None)

Esta função encontra os comprimentos de todos os caminhos mais curtos de todos os nós de origem para um único nó de destino no grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (1, 3), (3, 4)])

path\_lengths = nx.single\_target\_shortest\_path\_length(G, target=4)

print(path\_lengths)

small(G)

Esta função retorna um gráfico pequeno para testes e s.

:

import networkx as nx

G = nx.small\_graphs.small\_graph(1)

print(G.edges())

smallworld(n, k, p, seed=None)

Esta função retorna um gráfico de mundo pequeno aleatório.

:

import networkx as nx

G = nx.generators.smallworld\_graph(n=10, k=2, p=0.1)

print(G.edges())

smetric(G, u, v, k=2)

Esta função calcula a métrica s entre os nós u e v em um grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

metric\_s = nx.smetric(G, u=1, v=3)

print(metric\_s)

snap\_aggregation(G, nodes, attribute, delimiter='\_')

Esta função agrega os atributos dos nós em uma lista de nós dada e retorna o novo grafo agregado.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

nodes = [1, 2]

aggregated\_graph = nx.snap\_aggregation(G, nodes, attribute='weight')

print(aggregated\_graph.edges(data=True))

Espero que esta lista seja útil! Consulte sempre a documentação oficial do NetworkX para obter mais detalhes e s.

User

O mesmo para: 'social',

'soft\_random\_geometric\_graph',

'spanner',

'sparse6',

'sparsifiers',

'spectral\_bisection',

'spectral\_graph\_forge',

'spectral\_layout',

'spectral\_ordering',

'spectrum',

ChatGPT

Claro! Aqui estão mais algumas funções do NetworkX:

social

Esta função retorna um grafo social direcionado e ponderado como .

:

import networkx as nx

G = nx.social.karate\_club\_graph()

print(G.edges())

soft\_random\_geometric\_graph(n, radius, dim=2, pos=None, p=2, seed=None)

Esta função retorna um gráfico geográfico suave com nós posicionados aleatoriamente em um espaço euclidiano.

:

import networkx as nx

G = nx.soft\_random\_geometric\_graph(n=10, radius=0.3, seed=42)

print(G.edges())

spanner(G, k, L=None, weight='weight')

Esta função retorna um subgrafo G que é um k-spanner do grafo G original.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

spanner\_graph = nx.spanner(G, k=2, weight='weight')

print(spanner\_graph.edges(data=True))

sparse6(data)

Esta função retorna um gráfico do formato sparse6 representado por uma string.

:

import networkx as nx

data = ':EaO...\_@.@.'

G = nx.parse\_sparse6(data)

print(G.edges())

sparsifiers(G, k, weight='weight')

Esta função retorna um subgrafo G que é um k-sparsifier do grafo G original.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

sparsifier\_graph = nx.sparsifiers(G, k=2, weight='weight')

print(sparsifier\_graph.edges(data=True))

spectral\_bisection(G, weight='weight', normalized\_cut=True)

Esta função retorna uma bisseção espectral do grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

bisection = nx.spectral\_bisection(G, weight='weight')

print(bisection)

spectral\_graph\_forge(n, lambda1, seed=None)

Esta função forja um grafo com o menor valor próprio especificado.

:

import networkx as nx

G = nx.spectral\_graph\_forge(n=10, lambda1=0.5, seed=42)

print(G.edges())

spectral\_layout(G, weight='weight', dim=2, scale=1, center=None, seed=None)

Esta função calcula a posição dos nós em um espaço de dimensão reduzida usando a decomposição espectral.

:

import networkx as nx

import matplotlib.pyplot as plt

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

pos = nx.spectral\_layout(G)

nx.draw(G, pos, with\_labels=True, node\_size=500, node\_color='skyblue', font\_size=10)

plt.show()

spectral\_ordering(G, weight='weight')

Esta função retorna uma lista de nós em ordem espectral.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

ordering = nx.spectral\_ordering(G, weight='weight')

print(ordering)

spectrum(G, weight='weight')

Esta função retorna o espectro do grafo G como uma lista.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

spectrum = nx.spectrum(G, weight='weight')

print(spectrum)

Espero que esta lista seja útil! Consulte sempre a documentação oficial do NetworkX para obter mais detalhes e s.

User

O mesmo para: 'spiral\_layout',

'spring\_layout',

'square\_clustering',

'star\_graph',

'stochastic',

'stochastic\_block\_model',

'stochastic\_graph',

'stoer\_wagner',

'strong\_product',

'strongly\_connected',

ChatGPT

Claro! Aqui estão mais algumas funções do NetworkX:

spiral\_layout(G, scale=1, center=None, dim=2)

Esta função calcula a posição dos nós em um layout espiral.

:

import networkx as nx

import matplotlib.pyplot as plt

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

pos = nx.spiral\_layout(G)

nx.draw(G, pos, with\_labels=True, node\_size=500, node\_color='skyblue', font\_size=10)

plt.show()

spring\_layout(G, k=None, pos=None, fixed=None, iterations=50, threshold=1e-4, weight='weight', scale=1)

Esta função calcula a posição dos nós usando o layout Spring.

:

import networkx as nx

import matplotlib.pyplot as plt

G = nx.Graph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

pos = nx.spring\_layout(G)

nx.draw(G, pos, with\_labels=True, node\_size=500, node\_color='skyblue', font\_size=10)

plt.show()

square\_clustering(G)

Esta função calcula o coeficiente de clustering quadrado para cada nó do grafo G.

:

import networkx as nx

G = nx.karate\_club\_graph()

square\_clustering\_coefficients = nx.square\_clustering(G)

print(square\_clustering\_coefficients)

star\_graph(n)

Esta função retorna um gráfico estrela com n pontas.

:

import networkx as nx

G = nx.star\_graph(5)

print(G.edges())

stochastic(graph, copy=True, weight='weight', fully\_connected=False, seed=None)

Esta função retorna um grafo estocástico a partir do grafo dado.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

stochastic\_graph = nx.stochastic(G)

print(stochastic\_graph.edges())

stochastic\_block\_model(sizes, p, q, seed=None, directed=False)

Esta função retorna um grafo estocástico block model com tamanhos de blocos dados e parâmetros p e q.

:

import networkx as nx

sizes = [3, 4, 5]

p = 0.9

q = 0.1

sbm\_graph = nx.stochastic\_block\_model(sizes, p, q)

print(sbm\_graph.edges())

stochastic\_graph(graph, weight='weight', seed=None)

Esta função retorna um grafo estocástico a partir do grafo dado.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

stochastic\_graph = nx.stochastic\_graph(G)

print(stochastic\_graph.edges())

stoer\_wagner(G, weight='weight')

Esta função retorna um corte mínimo no grafo G usando o algoritmo de Stoer-Wagner.

:

import networkx as nx

G = nx.Graph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

cut\_value, partition = nx.stoer\_wagner(G, weight='weight')

print("Cut Value:", cut\_value)

print("Partition:", partition)

strong\_product(G1, G2)

Esta função retorna o produto forte dos grafos G1 e G2.

:

import networkx as nx

G1 = nx.complete\_graph(3)

G2 = nx.path\_graph(2)

strong\_product\_graph = nx.strong\_product(G1, G2)

print(strong\_product\_graph.edges())

strongly\_connected(G)

Esta função verifica se o grafo G é fortemente conectado.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

print(nx.strongly\_connected(G))

Espero que esta lista seja útil! Consulte sempre a documentação oficial do NetworkX para obter mais detalhes e s.

User

O mesmo para: 'strongly\_connected\_components',

'strongly\_connected\_components\_recursive',

'structuralholes',

'subgraph',

'subgraph\_alg',

'subgraph\_centrality',

'subgraph\_centrality\_exp',

'subgraph\_view',

'sudoku',

'sudoku\_graph',

ChatGPT

Claro! Aqui estão mais algumas funções do NetworkX:

strongly\_connected\_components(G)

Esta função retorna um gerador de conjuntos de nós correspondentes às componentes fortemente conectadas do grafo G.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

strongly\_connected\_components = list(nx.strongly\_connected\_components(G))

print(strongly\_connected\_components)

strongly\_connected\_components\_recursive(G)

Esta função retorna uma lista de conjuntos de nós correspondentes às componentes fortemente conectadas do grafo G usando uma abordagem recursiva.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2), (2, 3), (3, 4), (4, 1)])

strongly\_connected\_components = nx.strongly\_connected\_components\_recursive(G)

print(list(strongly\_connected\_components))

structuralholes(G, weight='weight', mode='out')

Esta função calcula os buracos estruturais de cada nó em um grafo ponderado.

:

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([(1, 2, {'weight': 2}), (2, 3, {'weight': 3}), (3, 4, {'weight': 1})])

structural\_holes = nx.structuralholes(G)

print(structural\_holes)

subgraph(G, nbunch)

Esta função retorna o subgrafo induzido pelos nós nbunch do grafo G.

:

import networkx as nx

G = nx.complete\_graph(5)

subgraph\_nodes = [1, 2, 3]

subgraph = nx.subgraph(G, subgraph\_nodes)

print(subgraph.edges())

subgraph\_alg(G, nodes)

Esta função retorna o subgrafo de G induzido pelos nós fornecidos em nodes.

:

import networkx as nx

G = nx.complete\_graph(5)

subgraph\_nodes = [1, 2, 3]

subgraph = nx.subgraph\_alg(G, subgraph\_nodes)

print(subgraph.edges())

subgraph\_centrality(G, normalized=True)

Esta função calcula a centralidade do subgrafo de cada nó no grafo G.

:

import networkx as nx

G = nx.karate\_club\_graph()

subgraph\_centrality = nx.subgraph\_centrality(G)

print(subgraph\_centrality)

subgraph\_centrality\_exp(G, scale=None, normalized=True)

Esta função calcula a centralidade exponencial do subgrafo de cada nó no grafo G.

:

import networkx as nx

G = nx.karate\_club\_graph()

subgraph\_centrality\_exp = nx.subgraph\_centrality\_exp(G)

print(subgraph\_centrality\_exp)

subgraph\_view(G, filter\_node=None, filter\_edge=None)

Esta função retorna uma visualização imutável do subgrafo de G.

:

import networkx as nx

G = nx.complete\_graph(5)

subgraph = nx.subgraph\_view(G, filter\_node=lambda x: x in [1, 2, 3])

print(subgraph.edges())

sudoku(dim=3)

Esta função retorna um gráfico de sudoku de ordem dim.

:

import networkx as nx

G = nx.sudoku(dim=2)

print(G.edges())

sudoku\_graph(n=3)

Esta função retorna um gráfico de sudoku com tamanho n x n.

:

import networkx as nx

G = nx.sudoku\_graph(n=2)

print(G.edges())

Espero que esta lista seja útil! Consulte sempre a documentação oficial do NetworkX para obter mais detalhes e s.

User

O mesmo para: 'summarization',

'swap',

'symmetric\_difference',

'tensor\_product',

'tetrahedral\_graph',

'text',

'thresholded\_random\_geometric\_graph',

'to\_dict\_of\_dicts',

'to\_dict\_of\_lists',

'to\_directed',

ChatGPT

Claro! Aqui estão mais algumas funções do NetworkX:

summarization(G, nodes)

Esta função calcula o resumo do grafo G para os nós especificados.

:

import networkx as nx

G = nx.karate\_club\_graph()

summary = nx.summarization(G, nodes=[1, 2, 3])

print(summary)

swap(G, trials=1, seed=None)

Esta função tenta trocar arestas aleatoriamente para preservar o grau dos nós no grafo G.

:

import networkx as nx

G = nx.karate\_club\_graph()

swapped\_graph = nx.swap(G)

print(swapped\_graph.edges())

symmetric\_difference(G, H)

Esta função retorna um novo grafo que contém as arestas que estão presentes em um dos grafos G ou H, mas não em ambos.

:

import networkx as nx

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

H = nx.Graph([(2, 3), (3, 4)])

symmetric\_diff\_graph = nx.symmetric\_difference(G, H)

print(symmetric\_diff\_graph.edges())

tensor\_product(G1, G2)

Esta função retorna o produto tensorial de dois grafos G1 e G2.

:

import networkx as nx

G1 = nx.Graph([(1, 2), (2, 3)])

G2 = nx.Graph([(3, 4), (4, 5)])

tensor\_product\_graph = nx.tensor\_product(G1, G2)

print(tensor\_product\_graph.edges())

tetrahedral\_graph()

Esta função retorna o grafo tetraédrico.

:

import networkx as nx

tetrahedral\_graph = nx.tetrahedral\_graph()

print(tetrahedral\_graph.edges())

text(G, delimiter=' ')

Esta função gera uma representação de texto do grafo G.

:

import networkx as nx

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

text\_representation = nx.text(G)

print(text\_representation)

thresholded\_random\_geometric\_graph(n, radius, theta, dim=2, pos=None, weight=None, metric=None, p=2, seed=None)

Esta função gera um grafo geométrico aleatório com limite.

:

import networkx as nx

thresholded\_random\_geometric\_graph = nx.thresholded\_random\_geometric\_graph(n=5, radius=0.2, theta=0.4)

print(thresholded\_random\_geometric\_graph.edges())

to\_dict\_of\_dicts(G, nodelist=None, edge\_data=None)

Esta função retorna a representação do grafo G como um dicionário de dicionários.

:

import networkx as nx

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

dict\_of\_dicts = nx.to\_dict\_of\_dicts(G)

print(dict\_of\_dicts)

to\_dict\_of\_lists(G, nodelist=None, edge\_data=None)

Esta função retorna a representação do grafo G como um dicionário de listas.

:

import networkx as nx

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

dict\_of\_lists = nx.to\_dict\_of\_lists(G)

print(dict\_of\_lists)

to\_directed(G)

Esta função retorna um grafo direcionado com o mesmo conjunto de nós e arestas do grafo não direcionado G.

:

import networkx as nx

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

directed\_graph = nx.to\_directed(G)

print(directed\_graph.edges())

Espero que esta lista seja útil! Consulte sempre a documentação oficial do NetworkX para obter mais detalhes e s.

User

O mesmo para: 'to\_edgelist',

'to\_graph6\_bytes',

'to\_latex',

'to\_latex\_raw',

'to\_nested\_tuple',

'to\_networkx\_graph',

'to\_numpy\_array',

'to\_pandas\_adjacency',

'to\_pandas\_edgelist',

'to\_prufer\_sequence',

ChatGPT

Claro! Aqui estão mais algumas funções do NetworkX:

to\_edgelist(G, nodelist=None)

Esta função retorna as arestas do grafo G em uma lista de tuplas.

:

import networkx as nx

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

edgelist = nx.to\_edgelist(G)

print(edgelist)

to\_graph6\_bytes(G)

Esta função retorna a representação em bytes do grafo G no formato graph6.

:

import networkx as nx

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

graph6\_bytes = nx.to\_graph6\_bytes(G)

print(graph6\_bytes)

to\_latex(G, nodelist=None, stringizer=None)

Esta função gera uma representação em formato LaTeX do grafo G.

:

import networkx as nx

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

latex\_representation = nx.to\_latex(G)

print(latex\_representation)

to\_latex\_raw(G, nodelist=None)

Esta função gera uma representação em formato LaTeX "raw" do grafo G, usando a notação de lista de adjacência.

:

import networkx as nx

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

latex\_representation\_raw = nx.to\_latex\_raw(G)

print(latex\_representation\_raw)

to\_nested\_tuple(G, ordering=None)

Esta função retorna o grafo G como uma tupla aninhada de arestas.

:

import networkx as nx

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

nested\_tuple = nx.to\_nested\_tuple(G)

print(nested\_tuple)

to\_networkx\_graph(data, create\_using=None)

Esta função converte dados em formato NetworkX em um grafo NetworkX.

:

import networkx as nx

data = [(1, 2), (2, 3)]

G = nx.to\_networkx\_graph(data)

print(G.edges())

to\_numpy\_array(G, nodelist=None, dtype=None, order=None, multigraph\_weight=sum, weight='weight', nonedge=0.0)

Esta função retorna o grafo G como uma matriz NumPy.

:

import networkx as nx

G = nx.Graph([(1, 2, {'weight': 2}), (2, 3, {'weight': 3})])

adj\_matrix = nx.to\_numpy\_array(G, weight='weight')

print(adj\_matrix)

to\_pandas\_adjacency(G, dtype=None, order=None, multigraph\_weight=sum, weight='weight', nonedge=0.0)

Esta função retorna o grafo G como um DataFrame do Pandas (matriz de adjacência).

:

import networkx as nx

import pandas as pd

G = nx.Graph([(1, 2, {'weight': 2}), (2, 3, {'weight': 3})])

adj\_matrix\_df = nx.to\_pandas\_adjacency(G, weight='weight')

print(adj\_matrix\_df)

to\_pandas\_edgelist(G, source='source', target='target', edge\_attr=None, nodelist=None)

Esta função retorna as arestas do grafo G como um DataFrame do Pandas.

:

import networkx as nx

import pandas as pd

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

edgelist\_df = nx.to\_pandas\_edgelist(G)

print(edgelist\_df)

to\_prufer\_sequence(G)

Esta função retorna a sequência de Prufer do grafo G.

:

import networkx as nx

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

prufer\_sequence = nx.to\_prufer\_sequence(G)

print(prufer\_sequence)

Espero que esta lista seja útil! Consulte sempre a documentação oficial do NetworkX para obter mais detalhes e s.

User

O mesmo para: 'to\_scipy\_sparse\_array',

'to\_sparse6\_bytes',

'to\_undirected',

'topological\_generations',

'topological\_sort',

'total\_spanning\_tree\_weight',

'tournament',

'transitive\_closure',

'transitive\_closure\_dag',

'transitive\_reduction',

ChatGPT

Claro! Aqui estão mais algumas funções do NetworkX:

to\_scipy\_sparse\_array(G, nodelist=None, dtype=None, weight='weight', format='csr')

Esta função retorna o grafo G como uma matriz esparsa do SciPy.

:

import networkx as nx

G = nx.Graph([(1, 2, {'weight': 2}), (2, 3, {'weight': 3})])

sparse\_array = nx.to\_scipy\_sparse\_array(G, weight='weight')

print(sparse\_array.toarray())

to\_sparse6\_bytes(G)

Esta função retorna a representação em bytes do grafo G no formato sparse6.

:

import networkx as nx

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

sparse6\_bytes = nx.to\_sparse6\_bytes(G)

print(sparse6\_bytes)

to\_undirected(G, reciprocal=False, as\_view=False)

Esta função retorna uma representação do grafo G como um grafo não direcionado.

:

import networkx as nx

G = nx.DiGraph([(1, 2), (2, 3)])

undirected\_G = nx.to\_undirected(G)

print(undirected\_G.edges())

topological\_generations(G, grouping='max', threshold=100, seed=None)

Esta função gera grafos topológicos a partir de um grafo de entrada G.

:

import networkx as nx

G = nx.DiGraph([(1, 2), (2, 3)])

topological\_generations = nx.topological\_generations(G)

for G\_gen in topological\_generations:

print(G\_gen.edges())

topological\_sort(G)

Esta função retorna uma lista com uma ordenação topológica do grafo G.

:

import networkx as nx

G = nx.DiGraph([(1, 2), (2, 3)])

topological\_order = nx.topological\_sort(G)

print(topological\_order)

total\_spanning\_tree\_weight(G, weight='weight')

Esta função retorna o peso total de uma árvore geradora mínima para o grafo G.

:

import networkx as nx

G = nx.Graph([(1, 2, {'weight': 2}), (2, 3, {'weight': 3})])

total\_weight = nx.total\_spanning\_tree\_weight(G, weight='weight')

print(total\_weight)

tournament(n)

Esta função retorna um torneio completo com n nós.

:

import networkx as nx

tournament\_graph = nx.tournament(5)

print(tournament\_graph.edges())

transitive\_closure(G)

Esta função retorna o fecho transitivo do grafo G.

:

import networkx as nx

G = nx.DiGraph([(1, 2), (2, 3)])

transitive\_closure\_G = nx.transitive\_closure(G)

print(transitive\_closure\_G.edges())

transitive\_closure\_dag(G)

Esta função retorna o fecho transitivo do grafo acíclico direcionado (DAG) G.

:

import networkx as nx

G = nx.DiGraph([(1, 2), (2, 3)])

transitive\_closure\_dag\_G = nx.transitive\_closure\_dag(G)

print(transitive\_closure\_dag\_G.edges())

transitive\_reduction(G)

Esta função retorna o fecho transitivo reduzido do grafo G.

:

import networkx as nx

G = nx.DiGraph([(1, 2), (2, 3)])

transitive\_reduction\_G = nx.transitive\_reduction(G)

print(transitive\_reduction\_G.edges())

Espero que esta lista seja útil! Consulte sempre a documentação oficial do NetworkX para obter mais detalhes e s.

User

O mesmo para: 'transitivity',

'traversal',

'tree',

'tree\_all\_pairs\_lowest\_common\_ancestor',

'tree\_data',

'tree\_graph',

'trees',

'triad\_graph',

'triad\_type',

'triadic\_census',

ChatGPT

Claro! Aqui estão mais algumas funções do NetworkX:

transitivity(G)

Esta função retorna o coeficiente de transitividade do grafo G.

:

import networkx as nx

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

transitivity\_coefficient = nx.transitivity(G)

print(transitivity\_coefficient)

traversal

Essa não é uma função específica, mas sim uma referência ao módulo networkx.traversal, que contém algoritmos de travessia em grafos, como a busca em largura e a busca em profundidade.

de busca em largura:

import networkx as nx

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

bfs\_tree = nx.traversal.bfs\_tree(G, source=1)

print(bfs\_tree.edges())

tree

Essa também não é uma função específica, mas sim uma referência ao módulo networkx.tree, que fornece funções relacionadas a árvores.

de criação de uma árvore:

import networkx as nx

T = nx.tree.random\_tree(10)

print(T.edges())

tree\_all\_pairs\_lowest\_common\_ancestor(T)

Esta função retorna o ancestral comum mais baixo para todas as duplas de nós em uma árvore T.

:

import networkx as nx

T = nx.random\_tree(10)

ancestors = nx.tree\_all\_pairs\_lowest\_common\_ancestor(T)

print(ancestors)

tree\_data(G, root=None, attrs=None)

Esta função retorna um dicionário que contém informações sobre a árvore enraizada em root no grafo G.

:

import networkx as nx

G = nx.DiGraph([(1, 2), (1, 3), (2, 4), (2, 5)])

tree\_info = nx.tree\_data(G, root=1)

print(tree\_info)

tree\_graph(n, create\_using=None)

Esta função retorna uma árvore enraizada com n nós.

:

import networkx as nx

T = nx.tree\_graph(5)

print(T.edges())

trees(n)

Esta função gera todas as árvores com n nós.

:

import networkx as nx

for T in nx.trees(4):

print(T.edges())

triad\_graph

Essa não é uma função específica, mas sim uma referência à classe networkx.generators.triads.triad\_graph, que representa um grafo completo com três nós que forma um triângulo.

:

import networkx as nx

triad\_graph = nx.generators.triads.triad\_graph()

print(triad\_graph.edges())

triad\_type(G, v, u, w)

Esta função retorna o tipo de triângulo formado pelos nós v, u e w no grafo G.

:

import networkx as nx

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

triad\_type = nx.triad\_type(G, 1, 2, 3)

print(triad\_type)

triadic\_census(G)

Esta função retorna o censo triádico do grafo G.

:

import networkx as nx

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

triadic\_census = nx.triadic\_census(G)

print(triadic\_census)

Espero que esta lista seja útil! Lembre-se de consultar a documentação oficial do NetworkX para mais detalhes e s.

User

O mesmo para: 'triads',

'triads\_by\_type',

'triangles',

'triangular\_lattice\_graph',

'trivial\_graph',

'trophic',

'trophic\_differences',

'trophic\_incoherence\_parameter',

'trophic\_levels',

'truncated\_cube\_graph',

ChatGPT

Claro! Aqui estão mais algumas funções do NetworkX:

triads(G)

Esta função retorna um dicionário que conta o número de triads (conjuntos de três nós e três arestas) em um grafo não direcionado G.

:

import networkx as nx

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

triads\_count = nx.triads(G)

print(triads\_count)

triads\_by\_type(G)

Esta função retorna um dicionário que conta o número de triads de cada tipo em um grafo não direcionado G.

:

import networkx as nx

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

triads\_by\_type = nx.triads\_by\_type(G)

print(triads\_by\_type)

triangles(G)

Esta função retorna o número de triângulos (conjuntos de três nós conectados por três arestas) em um grafo não direcionado G.

:

import networkx as nx

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

triangles\_count = nx.triangles(G)

print(triangles\_count)

triangular\_lattice\_graph(m, n, periodic=False)

Esta função retorna um grafo de malha triangular com m linhas e n colunas.

:

import networkx as nx

G = nx.triangular\_lattice\_graph(3, 3)

print(G.edges())

trivial\_graph(nodes=None)

Esta função retorna um grafo trivial, que é o grafo com um único nó e sem arestas.

:

import networkx as nx

G = nx.trivial\_graph()

print(G.nodes())

trophic(G, weight='weight', solver='lu')

Esta função calcula os níveis tróficos para nós em um grafo direcionado G.

:

import networkx as nx

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

trophic\_levels = nx.trophic(G)

print(trophic\_levels)

trophic\_differences(G, weight='weight', solver='lu')

Esta função calcula as diferenças tróficas para todos os pares de nós em um grafo direcionado G.

:

import networkx as nx

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

trophic\_differences = nx.trophic\_differences(G)

print(trophic\_differences)

trophic\_incoherence\_parameter(G, weight='weight', solver='lu')

Esta função calcula o parâmetro de incoerência trófica para um grafo direcionado G.

:

import networkx as nx

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

incoherence\_parameter = nx.trophic\_incoherence\_parameter(G)

print(incoherence\_parameter)

trophic\_levels(G, weight='weight', solver='lu')

Esta função calcula os níveis tróficos para nós em um grafo direcionado G.

:

import networkx as nx

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

trophic\_levels = nx.trophic\_levels(G)

print(trophic\_levels)

truncated\_cube\_graph(n)

Esta função retorna o grafo do cubo truncado com 2^n vértices.

:

import networkx as nx

G = nx.truncated\_cube\_graph(3)

print(G.edges())

Espero que esta lista seja útil! Lembre-se de consultar a documentação oficial do NetworkX para mais detalhes e s.

User

O mesmo para: 'truncated\_tetrahedron\_graph',

'turan\_graph',

'tutte\_graph',

'tutte\_polynomial',

'unary',

'uniform\_random\_intersection\_graph',

'union',

'union\_all',

'unweighted',

'utils',

ChatGPT

Claro! Aqui estão mais algumas funções do NetworkX:

truncated\_tetrahedron\_graph()

Esta função retorna o grafo do tetraedro truncado.

:

import networkx as nx

G = nx.truncated\_tetrahedron\_graph()

print(G.edges())

turan\_graph(n, r)

Esta função retorna o grafo de Turán com n vértices e r partes, que é o grafo completo com o menor número possível de arestas.

:

import networkx as nx

G = nx.turan\_graph(6, 2)

print(G.edges())

tutte\_graph()

Esta função retorna o grafo de Tutte.

:

import networkx as nx

G = nx.tutte\_graph()

print(G.edges())

tutte\_polynomial(G, x, y)

Esta função retorna o polinômio de Tutte de um grafo não direcionado G em termos das variáveis x e y.

:

import networkx as nx

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

polynomial = nx.tutte\_polynomial(G, 'x', 'y')

print(polynomial)

unary()

Esta função retorna o grafo unário, que é um grafo direcionado com dois nós 0 e 1 e uma aresta de 0 para 1.

:

import networkx as nx

G = nx.unary()

print(G.edges())

uniform\_random\_intersection\_graph(n, m, p)

Esta função retorna um grafo de interseção aleatório uniforme com n conjuntos, cada um com m elementos, e probabilidade de interseção p.

:

import networkx as nx

G = nx.uniform\_random\_intersection\_graph(5, 3, 0.5)

print(G.edges())

union(G, H)

Esta função retorna a união de dois grafos G e H.

:

import networkx as nx

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

H = nx.Graph([(3, 4), (4, 5)])

union\_graph = nx.union(G, H)

print(union\_graph.edges())

union\_all(graphs)

Esta função retorna a união de uma lista de grafos.

:

import networkx as nx

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

H = nx.Graph([(3, 4), (4, 5)])

I = nx.Graph([(5, 6), (6, 7)])

union\_all\_graph = nx.union\_all([G, H, I])

print(union\_all\_graph.edges())

unweighted(G)

Esta função remove todos os atributos de peso das arestas de um grafo.

:

import networkx as nx

G = nx.Graph([(1, 2, {'weight': 2}), (2, 3, {'weight': 3})])

unweighted\_graph = nx.unweighted(G)

print(unweighted\_graph.edges(data=True))

Espero que esta lista seja útil! Lembre-se de consultar a documentação oficial do NetworkX para mais detalhes e s.

User

O mesmo para: 'vf2pp\_all\_isomorphisms',

'vf2pp\_is\_isomorphic',

'vf2pp\_isomorphism',

'vitality',

'volume',

'voronoi',

'voronoi\_cells',

'voterank',

'voterank\_alg',

'watts\_strogatz\_graph',

ChatGPT

Claro! Aqui estão mais algumas funções do NetworkX:

vf2pp\_all\_isomorphisms(G1, G2)

Esta função encontra todos os isomorfismos entre dois grafos G1 e G2 usando o algoritmo VF2++. Retorna um gerador que produz os dicionários de mapeamento dos nós.

:

import networkx as nx

G1 = nx.Graph([(1, 2), (2, 3)])

G2 = nx.Graph([(3, 4), (4, 1)])

isomorphisms = list(nx.vf2pp\_all\_isomorphisms(G1, G2))

print(isomorphisms)

vf2pp\_is\_isomorphic(G1, G2)

Esta função verifica se dois grafos G1 e G2 são isomórficos usando o algoritmo VF2++. Retorna True se forem isomórficos e False caso contrário.

:

import networkx as nx

G1 = nx.Graph([(1, 2), (2, 3)])

G2 = nx.Graph([(3, 4), (4, 1)])

is\_isomorphic = nx.vf2pp\_is\_isomorphic(G1, G2)

print(is\_isomorphic)

vf2pp\_isomorphism(G1, G2)

Esta função encontra um isomorfismo entre dois grafos G1 e G2 usando o algoritmo VF2++. Retorna um dicionário de mapeamento dos nós se eles forem isomórficos e None caso contrário.

:

import networkx as nx

G1 = nx.Graph([(1, 2), (2, 3)])

G2 = nx.Graph([(3, 4), (4, 1)])

isomorphism = nx.vf2pp\_isomorphism(G1, G2)

print(isomorphism)

vitality(G)

Esta função retorna a vitalidade dos nós de um grafo, que é uma medida relacionada à sua importância estrutural.

:

import networkx as nx

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

vitality = nx.vitality(G)

print(vitality)

volume(G, nodes=None, normalized=False)

Esta função retorna o volume dos nós especificados de um grafo, que é a soma dos pesos das arestas que incidem nesses nós.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=5)

G.add\_edge(2, 3, weight=10)

G.add\_edge(1, 3, weight=8)

volume = nx.volume(G, nodes=[1, 2])

print(volume)

voronoi(G, weight='weight')

Esta função calcula a partição de Voronoi do grafo com base nos pesos das arestas.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=5)

G.add\_edge(2, 3, weight=10)

G.add\_edge(1, 3, weight=8)

voronoi = nx.voronoi(G, weight='weight')

print(voronoi)

voronoi\_cells(G, weight='weight')

Esta função retorna as células de Voronoi de cada nó em um grafo com base nos pesos das arestas.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=5)

G.add\_edge(2, 3, weight=10)

G.add\_edge(1, 3, weight=8)

voronoi\_cells = nx.voronoi\_cells(G, weight='weight')

print(voronoi\_cells)

voterank(G, max\_iter=100, tol=1e-06)

Esta função calcula a centralidade de votação dos nós em um grafo.

:

import networkx as nx

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

voterank = nx.voterank(G)

print(voterank)

voterank\_alg(G, max\_iter=100, tol=1e-06)

Esta função é uma implementação alternativa da centralidade de votação usando o algoritmo do power iteration.

:

import networkx as nx

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

voterank = nx.voterank\_alg(G)

print(voterank)

watts\_strogatz\_graph(n, k, p, seed=None)

Esta função retorna um grafo de Watts-Strogatz com n nós, cada nó conectado aos k vizinhos mais próximos no anel e probabilidade de reorganização p.

:

import networkx as nx

G = nx.watts\_strogatz\_graph(10, 2, 0.3)

print(G.edges())

Espero que esta lista de funções seja útil! Lembre-se de consultar a documentação oficial do NetworkX para mais detalhes e s.

User

O mesmo para: 'waxman\_graph',

'weakly\_connected',

'weakly\_connected\_components',

'weighted',

'weisfeiler\_lehman\_graph\_hash',

'weisfeiler\_lehman\_subgraph\_hashes',

'wheel\_graph',

'wiener',

'wiener\_index',

'windmill\_graph',

ChatGPT

Claro! Aqui estão mais algumas funções do NetworkX:

waxman\_graph(n, alpha=0.1, beta=0.3, L=None, domain=(0, 0, 1, 1), seed=None)

Esta função retorna um grafo de Waxman com n nós. Os nós são distribuídos aleatoriamente em um retângulo com arestas criadas de acordo com a probabilidade baseada na distância euclidiana entre os nós.

:

import networkx as nx

G = nx.waxman\_graph(10, alpha=0.1, beta=0.3)

print(G.edges())

weakly\_connected(G)

Esta função verifica se o grafo G é fracamente conectado, ou seja, se existe uma conexão direcionada entre cada par de nós.

:

import networkx as nx

G = nx.DiGraph([(1, 2), (2, 3), (3, 1)])

is\_weakly\_connected = nx.weakly\_connected(G)

print(is\_weakly\_connected)

weakly\_connected\_components(G)

Esta função retorna uma lista de conjuntos de nós correspondentes aos componentes fracamente conectados do grafo direcionado G.

:

import networkx as nx

G = nx.DiGraph([(1, 2), (2, 3), (3, 1)])

weakly\_connected\_components = list(nx.weakly\_connected\_components(G))

print(weakly\_connected\_components)

weighted(G, weight='weight')

Esta função verifica se o grafo G possui arestas ponderadas, ou seja, se as arestas têm um atributo 'weight'.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=5)

is\_weighted = nx.weighted(G)

print(is\_weighted)

weisfeiler\_lehman\_graph\_hash(G, edge\_attr='label')

Esta função calcula a função hash de Weisfeiler-Lehman para o grafo G. É uma técnica útil em muitos algoritmos de aprendizado de grafos.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, label='A')

hash\_value = nx.weisfeiler\_lehman\_graph\_hash(G, edge\_attr='label')

print(hash\_value)

weisfeiler\_lehman\_subgraph\_hashes(G, node\_attr='label', edge\_attr='label')

Esta função calcula a função hash de Weisfeiler-Lehman para os subgrafos de vizinhança de cada nó do grafo G.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, label='A')

subgraph\_hashes = nx.weisfeiler\_lehman\_subgraph\_hashes(G, node\_attr='label', edge\_attr='label')

print(subgraph\_hashes)

wheel\_graph(n, create\_using=None)

Esta função retorna o grafo de roda com n nós. O grafo de roda é formado por um único nó central conectado a todos os outros nós formando um ciclo.

:

import networkx as nx

G = nx.wheel\_graph(5)

print(G.edges())

wiener(G)

Esta função retorna o índice de Wiener do grafo G, que é a soma de todas as distâncias mais curtas entre todos os pares de nós.

:

import networkx as nx

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

wiener\_index = nx.wiener(G)

print(wiener\_index)

wiener\_index(G, weight='weight')

Esta função retorna o índice de Wiener ponderado do grafo G, levando em consideração os pesos das arestas.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=5)

G.add\_edge(2, 3, weight=10)

G.add\_edge(1, 3, weight=8)

wiener\_index\_weighted = nx.wiener\_index(G, weight='weight')

print(wiener\_index\_weighted)

windmill\_graph(p, n)

Esta função retorna o grafo moinho de vento com p pás e n nós em cada pá.

:

import networkx as nx

G = nx.windmill\_graph(4, 3)

print(G.edges())

Espero que esta lista de funções seja útil! Lembre-se de consultar a documentação oficial do NetworkX para mais detalhes e s.

User

O mesmo para: 'within\_inter\_cluster',

'write\_adjlist',

'write\_edgelist',

'write\_gexf',

'write\_gml',

'write\_graph6',

'write\_graphml',

'write\_graphml\_lxml',

'write\_graphml\_xml',

'write\_latex',

ChatGPT

Claro! Aqui estão mais algumas funções do NetworkX:

within\_inter\_cluster(G, communities, weight='weight')

Esta função calcula as medidas de dentro e entre o cluster de uma partição de comunidades em um grafo ponderado. Ela retorna um dicionário com as medidas.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=5)

G.add\_edge(2, 3, weight=10)

G.add\_edge(1, 3, weight=8)

communities = {1: 0, 2: 0, 3: 1}

within\_inter\_cluster\_measures = nx.within\_inter\_cluster(G, communities, weight='weight')

print(within\_inter\_cluster\_measures)

write\_adjlist(G, path, delimiter=' ', comments='#', encoding='utf-8')

Esta função grava o grafo G em um arquivo no formato de lista de adjacências.

:

import networkx as nx

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

nx.write\_adjlist(G, 'graph.adjlist')

write\_edgelist(G, path, delimiter=' ', data=True, encoding='utf-8')

Esta função grava o grafo G em um arquivo no formato de lista de arestas.

:

import networkx as nx

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

nx.write\_edgelist(G, 'graph.edgelist')

write\_gexf(G, path, encoding='utf-8', prettyprint=True, version='1.2draft')

Esta função grava o grafo G em um arquivo no formato GEXF (Graph Exchange XML Format).

:

import networkx as nx

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

nx.write\_gexf(G, 'graph.gexf')

write\_gml(G, path, stringizer=None)

Esta função grava o grafo G em um arquivo no formato GML (Graph Modeling Language).

:

import networkx as nx

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

nx.write\_gml(G, 'graph.gml')

write\_graph6(G, path)

Esta função grava o grafo G em um arquivo no formato Graph6.

:

import networkx as nx

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

nx.write\_graph6(G, 'graph.graph6')

write\_graphml(G, path, encoding='utf-8', prettyprint=True, infer\_numeric\_types=False)

Esta função grava o grafo G em um arquivo no formato GraphML.

:

import networkx as nx

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

nx.write\_graphml(G, 'graph.graphml')

write\_graphml\_lxml(G, path, encoding='utf-8', prettyprint=True)

Esta função grava o grafo G em um arquivo no formato GraphML usando a biblioteca lxml.

:

import networkx as nx

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

nx.write\_graphml\_lxml(G, 'graph.graphml')

write\_graphml\_xml(G, path, encoding='utf-8', prettyprint=True)

Esta função grava o grafo G em um arquivo no formato GraphML usando a biblioteca XML.

:

import networkx as nx

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

nx.write\_graphml\_xml(G, 'graph.graphml')

write\_latex(G, path, template=None, node\_label=None, edge\_label=None, name=None)

Esta função grava o grafo G em um arquivo LaTeX.

:

import networkx as nx

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

nx.write\_latex(G, 'graph.tex')

Espero que esta lista de funções seja útil! Lembre-se de consultar a documentação oficial do NetworkX para mais detalhes e s.

User

O mesmo para: 'write\_multiline\_adjlist',

'write\_network\_text',

'write\_pajek',

'write\_sparse6',

'write\_weighted\_edgelist'

ChatGPT

Claro! Aqui estão mais algumas funções de escrita do NetworkX:

write\_multiline\_adjlist(G, path, delimiter=' ', comments='#', encoding='utf-8')

Esta função grava o grafo G em um arquivo no formato de lista de adjacências multilinha.

:

import networkx as nx

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

nx.write\_multiline\_adjlist(G, 'graph\_multiline.adjlist')

write\_network\_text(G, path, delimiter=' ', encoding='utf-8')

Esta função grava o grafo G em um arquivo de texto usando um formato de lista de adjacências básica.

:

import networkx as nx

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

nx.write\_network\_text(G, 'graph\_text.txt')

write\_pajek(G, path)

Esta função grava o grafo G em um arquivo no formato Pajek.

:

import networkx as nx

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

nx.write\_pajek(G, 'graph.net')

write\_sparse6(G, path)

Esta função grava o grafo G em um arquivo no formato Sparse6.

:

import networkx as nx

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

nx.write\_sparse6(G, 'graph.sparse6')

write\_weighted\_edgelist(G, path, delimiter=' ', encoding='utf-8')

Esta função grava o grafo G em um arquivo no formato de lista de arestas com pesos.

:

import networkx as nx

G = nx.Graph()

G.add\_edge(1, 2, weight=5)

G.add\_edge(2, 3, weight=10)

G.add\_edge(3, 1, weight=8)

nx.write\_weighted\_edgelist(G, 'graph\_weighted.edgelist')

Espero que essas funções também sejam úteis para você! Lembre-se de consultar a documentação oficial do NetworkX para mais detalhes e s.