ChatGPT

NetworkX 3.6 Comprehensive Technical Reference

Graph Classes and Data Structures

NetworkX 3.6 provides four primary graph classes: **Graph**, **DiGraph**, **MultiGraph**, and **MultiDiGraph** [1] [2]. These classes differ in whether they support directed edges and multiple edges: a `Graph` is undirected (edges have no orientation) and forbids parallel edges, while a `DiGraph` is directed (edge order matters) [3]. `MultiGraph` and `MultiDiGraph` allow multiple edges between the same nodes (distinguished by edge **keys**) at some cost to performance [2]. All graph classes permit self-loop edges (an edge from a node to itself) [4] [5]. Nodes can be any hashable Python object (except `None`), and edges can hold arbitrary data attributes (stored in edge attribute dicts) [6] [7]. Internally, each graph's adjacency structure is a dict-of-dicts: `G._adj` maps each node to an inner dict of neighbors, which in turn maps neighbor nodes to edge data (or to another dict of edge data for MultiGraphs) [8] [9]. This adjacency list representation allows fast additions and lookups in large graphs [8]. Each Graph instance also has a `G.graph` attribute (a dict for graph-level metadata) and `G.nodes` / `G.edges` attribute-views. Node attributes can be set per node (e.g. `G.nodes[n]['color'] = 'red'`) and edge attributes per edge (for MultiGraphs edges are identified by `(u,v,key)`) [10] [11].

Basic Graph Operations: Nodes are added with `G.add_node(n, **attr)` or in bulk with `G.add_nodes_from(iterable, **attr)` [12]. Edges are added with `G.add_edge(u, v, **attr)` (which implicitly adds nodes if not already present) [13], or via `G.add_edges_from([(u,v,attr_dict), ...])` for batches. Removing nodes/edges works similarly (`G.remove_node(n)`, `G.remove_edges_from([...])`, etc.). These methods ignore duplicates or nonexistent elements (no error on adding an existing node, etc.) [14]. The number of nodes/edges is given by `G.number_of_nodes()`, `G.number_of_edges()` or simply `len(G)` for nodes count. Containers for nodes, edges, and neighbors are provided as *views* for efficiency: `G.nodes` yields a dynamic NodeView (iterable and set-like), `G.edges` yields an EdgeView (optionally filterable by nodes) [15], and `G.adj` gives a dict adjacency view. In MultiGraphs, `G.edges(data=True, keys=True)` can list parallel edges with their keys and data. Edge data is accessible via `G[u][v]` (which returns the edge dict for `(u,v)` in simple graphs, or a dict-of-dicts of edge data keyed by edge keys in MultiGraphs). For example, in a MultiGraph if two edges `(1,2)` have keys `"a"` and `"b"`, then `G[1][2]` might look like `{'a': {'weight': 5}, 'b': {'weight': 7}}`.

Graph Attributes and Copying: Each graph has a `G.graph` dict for storing metadata. You can supply graph attributes on creation (`G = nx.Graph(name="MyGraph")`) or modify `G.graph` later [16]. Node attributes can be set during add (`G.add_node(n, **attrs)`) or via dictionary assignments on `G.nodes[n]`. Edge attributes can be set similarly or via `G.add_edge(u,v,**attrs)`. Copying a graph is done with `H = G.copy()` (shallow copy of structure and attributes) or by using the graph class constructor on another graph (which attempts to carry over the graph data) [17]. There are also `G.to_directed()` and `G.to_undirected()` methods to convert or view a graph in the opposite directedness. By default these create a new graph of the appropriate class, but subclasses can override class variables to preserve custom types [18].

Graph Views and Subgraphs

NetworkX supports **graph views** to provide read-only filtered or transformed versions of graphs without copying all data [19] [20]. A view reflects the underlying graph's data in real-time, which is useful for algorithms that need to consider substructures without permanently removing data. For example, `nx.subgraph(G, nbunch)` returns an induced subgraph *view* of `G` on the given node set (by default, as of NetworkX 3.x) rather than copying nodes and edges [21] [22]. Likewise, `nx.edge_subgraph(G, edge_list)` produces a view containing only those edges [23] [24]. Views are essentially read-only – attempts to modify them will raise errors or propagate changes to the underlying graph. Internally, a view stores a reference to the source graph as `view._graph` [25].

Subgraph and Filter Views: For more complex filtering, **restricted views** can be created with `nx.subgraph_view(G, filter_node=..., filter_edge=...)` [26] [27]. For example, one can hide certain nodes or edges by providing filter functions: `filter_node` is a callable `f(n)` that returns True if node `n` should be kept, and similarly for `filter_edge(u,v)` for edges [20] [27]. NetworkX provides helper *filter factory* functions to create these predicates. For instance, `nx.hide_nodes({n1,n2,...})` returns a filter function that excludes a specified set of nodes [28], and `nx.hide_edges([...])` excludes particular undirected edges [29]. There are corresponding `show_nodes` / `show_edges` functions to include only certain elements [30] [31]. Using these, one can quickly produce, say, `H = nx.subgraph_view(G, filter_node=nx.hide_nodes(forbidden_nodes))` to analyze `G` with some nodes "temporarily removed". Similarly, `nx.restricted_view(G, nodes, edges)` is a convenience that hides a given set of nodes and edges in one call [32] [23].

Directed, Reversed, and Other Views: NetworkX also defines specialized views. For any directed graph `D`, `D.reverse(copy=False)` returns a **reverse view** where all edge orientations are flipped [33] (if `copy=True`, it produces a new DiGraph instead). This is useful for traversals on the transpose of a DAG. Treating a directed graph as undirected can be done via `nx.to_undirected(D)` which by default provides a **directed-to-undirected view** (edges become undirected) [34] [35]. **Chains of views:** Since views are graphs, you can nest them (e.g. a subgraph of a subgraph). However, chaining many views can degrade performance; deeply nested views (more than ~15) can lead to slow access [36]. NetworkX internally tries to shortcut common cases (like taking a subgraph of a subgraph) to avoid long chains [36], but in complex scenarios it may be better to make a copy (`G.copy()` or `nx.Graph(view)`) if you need to modify or if performance suffers.

Graph Manipulation and Functional Utilities

Beyond graph object methods, NetworkX provides a rich *functional API* of standalone functions (in `networkx.functions` and similar modules) for common tasks [37] [38]. These functions often mirror graph methods but can be more flexible in functional composition. Key utilities include:

- **Degree and Density:** `nx.degree(G, nbunch)` returns a DegreeView (or single degree) for nodes in `nbunch` [39] [40]. For convenience, `nx.degree_histogram(G)` returns a list where the *i*th element is the number of nodes with degree *i* [41]. `nx.density(G)` computes the graph density = actual edges / possible edges [42] [43].

- **Substructures:** `nx.is_directed(G)` checks directedness. `nx.is_empty(G)` checks if there are zero edges [44]. Functions like `nx.nodes(G)` and `nx.edges(G)` retrieve NodeView or EdgeView objects (same as `G.nodes` and `G.edges`). You can get neighbor information with `nx.all_neighbors(G, node)` which yields all neighbors of a node (in undirected graphs) [45], or use `nx.non_neighbors(G, node)` to find nodes with no edge to the given node [46] [47]. `nx.common_neighbors(G, u, v)` lists the common neighbors of `u` and `v` [46] [47]. For directed graphs, `nx.predecessors(G, n)` and `nx.successors(G, n)` give incoming and outgoing neighbors.

- **Adding predefined structures:** Instead of manual loops, NetworkX has helpers to add certain patterns of edges: `nx.add_path(G, [v1,v2,...], **attrs)` adds a path (v1–v2–...) to graph G [48] [49]; `nx.add_cycle(G, [v1,v2,...], **attrs)` closes a cycle (adding edges v1–v2, v2–v3, ..., last–v1) [50] [51]; `nx.add_star(G, [center, *points], **attrs)` adds a star centered at the first node [52] [53]. These convenience functions ensure the nodes are added and handle attribute assignment uniformly.

- **Attributes and Weights:** To manipulate attributes in bulk, use `nx.set_node_attributes(G, values, name)` to set a node attribute for many nodes at once [54] [55]. Here `values` can be a dict mapping nodes to attribute value or a single value applied to all nodes. Similarly, `nx.set_edge_attributes(G, values, name)` does so for edges [56] [57]. Retrieval is done with `nx.get_node_attributes(G, name)` which returns a dict of node: value [58] [59]. Functions like `nx.is_weighted(G)` check if an edge weight attribute (default `'weight'`) is present on **every** edge [60]. There's also `nx.is_negatively_weighted(G)` to detect negative weights [61] [62] (useful before running algorithms like Dijkstra that require nonnegative weights).

- **Subgraph and Views via functions:** `nx.subgraph(G, nbunch)` returns the induced subgraph on nodes in nbunch (similar to `G.subgraph`) [21], while `nx.induced_subgraph(G, nbunch)` explicitly returns a read-only SubGraph view [63] [64]. `nx.edge_subgraph(G, edge_list)` returns a subgraph view with only those edges (and any nodes incident on them) [23] [24]. To **temporarily remove** nodes or edges, `nx.restricted_view(G, nodes_to_hide, edges_to_hide)` can produce a view with those elements filtered out [32] [65]. If you need to freeze a graph's structure (prevent any additions or removals), use `nx.freeze(G)` [66] [67]. A frozen graph will reject any structural modifications; check with `nx.is_frozen(G)` which returns True if so [68] [69].

- **Graph Operators:** NetworkX supports various binary operations on graphs. **Union**: `nx.union(G, H, rename=('G','H'))` combines two graphs' nodes and edges, renaming node conflicts with optional prefixes [70] [71]. **Intersection**: `nx.intersection(G, H)` keeps only nodes and edges present in both [72]. **Difference**: `nx.difference(G, H)` keeps edges in G that are not in H (on the common nodes) [73] [74]. **Symmetric difference**: `nx.symmetric_difference(G, H)` merges edges that are in either graph but not both [75] [76]. **Compose**: `nx.compose(G, H)` is like union but if both have an edge (u,v) the attributes from H take precedence [77] [70]. There are also multi-graph versions (e.g. `union_all(graphs)` for an iterable of graphs) [78] [79]. In all set operations, graph attributes (like `G.graph` dict) are not automatically merged unless using functions like `compose` which will combine them. Another operation is **complement**: `nx.complement(G)`

produces the complement graph (same nodes, edges where original had none) [80] [81]. For directed graphs, `nx.reverse(D)` returns the graph with all arcs reversed [82].

- **Graph Products:** NetworkX implements standard graph products [83]. For example, `nx.cartesian_product(G, H)` returns the Cartesian product of graphs G and H (nodes are pairs `(u,v)` and adjacency defined by moving in one graph at a time) [83] [84]. Other products: `nx.tensor_product(G, H)` (a.k.a. Kronecker product) where adjacency is the tensor of adjacencies [85] [86], `nx.lexicographic_product(G, H)` where H is substituted for each node of G [83] [84], `nx.strong_product(G, H)` (union of Cartesian, tensor, and lexicographic products) [87], `nx.rooted_product(G, H, root)` attaches a copy of H to each node of G by merging at the specified root node [88] [89], `nx.modular_product(G, H)` (used in graph isomorphism, relates to common subgraph) [90], and `nx.power(G, k)` which returns the kth power of graph G (edges connect nodes with path-length $\leq$k) [91] [92]. These operations generate new graphs (not views) and can be computationally expensive for large input sizes.

Graph Generators

NetworkX 3.6 includes a comprehensive library of graph generator functions to create a wide range of standard graphs, random graphs, and benchmark networks. These are accessible via the `networkx.generators` subpackage or directly as `nx.<name>_graph` functions. Below are major categories of generators:

- **Classic Graphs:** Functions for well-known deterministic graphs. For example, `nx.complete_graph(n)` returns the complete graph K_n on *n* nodes [93], `nx.path_graph(n)` gives a simple path of length *n-1* [94], `nx.cycle_graph(n)` for a cycle C_n [95], `nx.star_graph(n)` for a star with *n* peripheral nodes [96], `nx.wheel_graph(n)` for a wheel graph (cycle with a central hub) [97], `nx.balanced_tree(r, h)` for a perfectly balanced *r*-ary tree of height *h* [98], `nx.lollipop_graph(m, n)` for two complete graphs K_m and K_n connected by a single edge (a "lollipop" shape) [99] [100], `nx.barbell_graph(m1, m2)` for two complete graphs of size m1 joined by a path of length m2 [101], etc. Unless specified with `create_using`, these generators return a simple Graph (undirected) [102] [103]. The **Graph Atlas** is accessible via `nx.graph_atlas_g()` which returns a list of all small graphs up to 7 nodes [104] (and `nx.graph_atlas(i)` for a specific indexed graph) – useful for testing and exhaustive analysis of small graphs.

- **Small Named Graphs:** A collection of famous graphs is available (mostly via `nx.generators.small`). Examples: `nx.petersen_graph()` returns the classic 10-node Petersen graph, `nx.tutte_graph()` returns the Tutte graph [105], `nx.sedgewick_maze_graph()` a specific maze graph [106], `nx.krackhardt_kite_graph()` a small 10-node social network example [107], `nx.chvatal_graph()`, `nx.moebius_kantor_graph()`, `nx.frucht_graph()`, `nx.heawood_graph()`, etc [108] [109]. These are predefined interesting graphs from graph theory literature. Most have no parameters (except some like `nx.generalized_petersen_graph(n, k)` for the family of Petersen generalizations) [110] [111].

- **Expander Graphs:** Under "expanders", NetworkX provides explicit constructions for expander and cage graphs. For example, `nx.margulis_gabber_galil_graph(n)` returns an (n^2)-node expander graph with certain degree [101]; `nx.paley_graph(p)` returns a Paley graph of order *p* (a pseudo-random strongly regular graph) [112]; `nx.chordal_cycle_graph(p)` gives a cycle with chords (an expander example) [113]. There are also functions to test expansion properties (e.g. `nx.is_regular_expander(G, epsilon)` to check if graph is a δ-expander) [114].

- **Lattice Graphs:** For grids and lattices, functions include `nx.grid_2d_graph(m, n)` for an $m \times n$ 2D grid graph [115] [116], `nx.grid_graph(dimensions)` for higher-dimensional grids [117] [118], `nx.triangular_lattice_graph(m, n)` and `nx.hexagonal_lattice_graph(m, n)` for 2D tilings of the plane (triangular, hexagonal lattices) [115] [119]. `nx.hypercube_graph(n)` generates the *n*-dimensional hypercube (2^n nodes) [120].

- **Random Graphs (Erdős–Rényi and variants):** A major category are random graphs. **Erdős–Rényi graphs:** `nx.gnp_random_graph(n, p)` (also aliased as `erdos_renyi_graph`) generates an *n*-node graph where each possible edge is present with probability *p* [121]. `nx.gnm_random_graph(n, m)` generates a graph with exactly *m* edges chosen uniformly at random [122] [123]. There is also `nx.fast_gnp_random_graph(n, p)` which is an optimized implementation for sparse graphs [124] [125]. **Small-world models:** `nx.watts_strogatz_graph(n, k, p)` generates the Watts–Strogatz small-world model (start with ring lattice of degree k, randomly rewire edges with probability p) [126]. `nx.newman_watts_strogatz_graph(n, k, p)` is a variant without edge deletion [127]. `nx.connected_watts_strogatz_graph(n, k, p, tries)` ensures a connected realization [128]. **Barabási–Albert scale-free model:** `nx.barabasi_albert_graph(n, m)` grows a graph by preferential attachment (adding nodes with m edges each) [129]. Variants like `nx.dual_barabasi_albert_graph(n, m1, m2, p)` (two attachment mechanisms) [130] and `nx.powerlaw_cluster_graph(n, m, p)` (Holme–Kim model adding triangles with probability p) combine preferential attachment with clustering [131] [132]. **Random regular graphs:** `nx.random_regular_graph(d, n)` yields a random *d*-regular graph on *n* nodes [133]. **Other models:** `nx.random_lobster(n, p1, p2)` generates a random "lobster" tree (a tree with a backbone and side branches) [134], `nx.random_shell_graph(constructor_list)` creates a random graph with a given shell structure (for social network degree distributions) [135], and `nx.random_powerlaw_tree(n, γ)` produces a tree with degree distribution ~ power-law exponent γ [136]. All these functions accept a `seed` for reproducibility. (See **Randomness** section below for seeding details.)

- **Duplication–Divergence Models:** There are generators for biological network models where nodes duplicate and mutate connections. `nx.duplication_divergence_graph(n, p)` starts from a single node and adds *n-1* nodes, each new node duplicating the connections of a random existing node and retaining each edge with probability *p* [137] [138]. Also `nx.partial_duplication_graph(N, n, p, q)` variant where new node connects to some neighbors of the copied node with different probabilities [139].

- **Degree Sequence Models:** Functions to generate graphs with a given degree sequence: `nx.configuration_model(seq)` creates a multigraph by randomly connecting stubs (the "configuration model") [140] [141], and `nx.havel_hakimi_graph(seq)` creates a simple graph from

a degree sequence using the Havel–Hakimi algorithm [142]. Directed versions (`nx.directed_configuration_model`, `nx.directed_havel_hakimi_graph`) exist for in/out-degree sequences [141] [143]. `nx.expected_degree_graph(w, selfloops=False)` generates a random graph with a given expected degree sequence using the Chung–Lu model [140] [144].

- **Random Clustered Graphs:** `nx.random_clustered_graph(joint_degree_sequence)` constructs a random graph with a given joint degree and triangle degree sequence (i.e. specified numbers of edges and triangles per node) [145]. This allows control over clustering (transitivity) in the generated graph [145].

- **Directed Graph Models:** Additional directed graphs beyond Barabási–Albert extensions include the **GN (growing network)** models: `nx.gn_graph(n)`, `nx.gnr_graph(n, p)` (with redirection probability p), and `nx.gnc_graph(n)` (copying model) for growing digraphs with different attachment rules [146] [147]. Also `nx.random_k_out_graph(n, k, α)` which creates a directed graph where each node chooses k targets with preferential attachment (α is probability of choosing a "new" vs "old" target) [148], and `nx.scale_free_graph(n, α, β, γ)` which implements the directed scale-free graph with given proportion of alpha (preferential attachment), beta (new nodes forming directed edges), gamma (edges added between existing nodes) [149].

- **Geometric Graphs:** For spatial network models, `nx.random_geometric_graph(n, radius, dim=2)` places *n* nodes uniformly at random in a unit square (or higher-dimensional unit cube) and connects nodes within a given distance radius [150] [151]. Variants: `nx.geographical_threshold_graph(n, θ)` places nodes with random weights and connects nodes if product of weights exceeds θ (models distance and weight combined) [152], `nx.waxman_graph(n, β, α)` connects nodes with probability decaying with distance (Waxman model) [153] [154], and "soft" geometric graphs where connection probability is a softened step function: `nx.soft_random_geometric_graph` and `nx.thresholded_random_geometric_graph` [155] [156]. There's also `nx.geometric_edges(G, radius)` to list edges within radius in an existing spatial embedding [157].

- **Line Graphs and Ego Graphs:** `nx.line_graph(G)` produces the line graph L(G) where each node represents an edge of G [158] [159]. In a MultiGraph, each edge with unique key becomes a node in the line graph. `nx.inverse_line_graph(H)` attempts the inverse operation, returning a graph G such that L(G)=H (if H is a line graph) [160]. For ego networks, `nx.ego_graph(G, n, radius=1)` returns the induced subgraph of nodes within a given radius from center node `n` [161] [162].

- **Stochastic Graphs:** `nx.stochastic_graph(G, copy=True)` converts a directed graph into a right-stochastic graph (each node's outneighbors weights sum to 1) [163] [164]. This is useful for Markov chains and random walk simulations.

- **Internet AS Graph:** `nx.random_internet_as_graph(n)` generates a synthetic Internet Autonomous System network topology for *n* nodes using the BA model with some constraints [165] [166].

- **Intersection Graphs:** Functions for random intersection graphs model say users with shared interests. `nx.uniform_random_intersection_graph(n, m, k)` creates a bipartite-like graph

of n "people" and m "attributes" then projects an intersection graph where each pair of people share an edge if they have an attribute in common, with each person having exactly k attributes [167] [168]. `nx.k_random_intersection_graph(n, m, k)` is similar (each of n nodes chooses k of m attributes) [168]. `nx.general_random_intersection_graph(n, m, p)` gives each of n nodes each of m attributes with independent probability p [169] [170]. These functions return the bipartite incidence structure internally; you can then derive the intersection graph as needed.

- **Social Network Datasets:** A few famous small social networks are included: `nx.karate_club_graph()` (Zachary's karate club network, 34 nodes) [171], `nx.davis_southern_women_graph()` (Davis's Southern women bipartite social network) [172], `nx.florentine_families_graph()` (Renaissance Florentine families marriage network) [173], and `nx.les_miserables_graph()` (Les Misérables character co-occurrence network) [174].

- **Community Benchmark Graphs:** Under "Community" generators, NetworkX has planted-partition models. `nx.planted_partition_graph(l, k, p_in, p_out)` creates *l* groups of size *k* each, with intra-group edge probability p_in and inter-group probability p_out [175] [176]. `nx.gaussian_random_partition_graph(n, s, v, p_in, p_out)` divides n nodes into groups with sizes drawn from a normal distribution (mean s, variance v) then connects with given probabilities (useful for testing clustering algorithms) [177]. `nx.relaxed_caveman_graph(l, k, p)` starts with l cliques of size k (caveman graph) and then rewires each edge with probability p to link different cliques [178]. `nx.LFR_benchmark_graph(n, τ1, τ2, μ, ...)` generates a benchmark graph with power-law distributed community sizes and degrees (Lancichinetti–Fortunato–Radicchi model) – this requires additional parameters and may rely on external code if present [179] [180].

- **Spectral Graph Forge:** A recent generator `nx.spectral_graph_forge(G, α)` creates a new graph that approximately shares the spectrum (eigenvalues) of a given graph G [181] [182]. It does so by spectral modifications and is used as a graph anonymization or modeling technique.

- **Trees and Forests:** Besides deterministic trees (like `nx.full_rary_tree` for an *r*-ary tree with a specified number of nodes [183]), NetworkX can sample **uniform random trees**. `nx.random_tree(n)` returns a uniformly random labeled tree on n nodes (via e.g. Prüfer sequence internally) – this is accessible through `nx.random_spanning_tree(G)` as well for spanning trees. More explicitly, in the **Trees** category, `nx.random_labeled_tree(n)` gives a random labeled tree on n nodes (each labeled 0..n-1 equally likely) [184], while `nx.random_unlabeled_tree(n)` gives an *unlabeled* tree drawn from non-isomorphic tree shapes (each unique structure equally likely) [185]. NetworkX distinguishes labeled vs unlabeled random tree sampling, and also offers **rooted** versions: `random_labeled_rooted_tree`, and even **forests** (disjoint union of trees) like `random_unlabeled_rooted_forest` [186] [187]. The difference is subtle (unlabeled draws by isomorphism classes, giving each non-isomorphic tree equal probability) [188] [189]. Also, `nx.prefix_tree(paths)` constructs a directed trie (prefix tree) from a list of sequences [190], treating each sequence as a path from the root.

- **Non-isomorphic Trees Enumeration:** `nx.nonisomorphic_trees(n)` returns an iterator over all non-isomorphic trees of order n (using the WROM algorithm) [191], and

`nx.number_of_nonisomorphic_trees(n)` counts them [192]. This is useful for brute-force checking of small trees.

- **Triads:** `nx.triad_graph(name)` gives the directed triad graph on 3 nodes for a specified triad name (there are 16 possible triad isomorphism classes in directed graphs) [193].

- **Joint Degree Graphs:** To generate a graph with a given joint degree matrix (for specified inter-group degree counts), use `nx.joint_degree_graph(joint_degrees)` which realizes a simple graph with that joint degree dict if possible [194] [195]. Functions `nx.is_valid_joint_degree(joint_degrees)` check graphicality of such sequences [195]. Similarly for directed joint degree (`nx.directed_joint_degree_graph` with in/out joint distribution) [196] [197].

- **Mycielski Graphs:** The Mycielskian is an operation to increase the chromatic number of a graph without creating triangles. `nx.mycielski_graph(n)` gives the *n*th Mycielski graph starting from C_5 (or another base) [198] [199], and `nx.mycielskian(G, iterations)` will iteratively apply the Mycielski transform to graph G a given number of times [200]. These are useful in graph coloring studies.

- **Harary Graphs:** There are two generators for **Harary graphs**, which are extremal connectivity graphs [201]. `nx.hnm_harary_graph(n, m)` gives a graph on *n* nodes with *m* edges that is maximally connected (highest possible node connectivity) [202] [203]. `nx.hkn_harary_graph(k, n)` gives a smallest-edge graph on *n* nodes with connectivity *k* (i.e. minimum edges to be *k*-node-connected) [204] [205].

- **Cographs:** `nx.random_cograph(n)` returns a random cograph on 2^n nodes (cographs are generated by starting with single-node graphs and repeatedly taking disjoint union or complement) [206]. This uses a recursive generation – cographs have no induced P_4.

- **Interval Graphs:** `nx.interval_graph(intervals)` builds an interval graph from a list of intervals (each node corresponds to an interval, and an edge joins overlapping intervals) [207] [208]. Interval graphs are important in scheduling and genomics.

- **Sudoku Graph:** `nx.sudoku_graph(n)` generates the Sudoku graph of order n (n^2 cells, each cell connected to others in its row, column, and subgrid). For the standard 9x9 Sudoku, `nx.sudoku_graph(3)` gives a 81-node graph with degree 20 for each node (which can be used to model Sudoku constraints) [209] [210].

Usage Note: Most generator functions create a new graph from scratch. Many accept a `create_using` parameter if you want a specific graph type (e.g. `nx.complete_graph(n, create_using=nx.DiGraph)` for a directed complete graph) [102]. Some random generators require **NumPy** and/or **SciPy** for performance (NetworkX will use NumPy random routines if available). Always consider the size of generated graphs – some, like `nonisomorphic_trees(n)`, grow super-exponentially. For very large random graphs, specialized libraries may be more efficient, but NetworkX generators are excellent for moderate sizes and rapid prototyping of complex networks.

Graph Matrices and Linear Algebra Tools

NetworkX provides functions to derive matrix representations of graphs and to perform spectral analysis. Many of these require **NumPy** or **SciPy** and are found in `networkx.linalg`. Key functions:

- **Adjacency Matrix:** `nx.adjacency_matrix(G, nodelist=None, weight=None)` returns the adjacency matrix of graph G as a SciPy sparse matrix (by default) [211] [212]. Nodes in `nodelist` order index the matrix (if not provided, order is arbitrary but consistent). For weighted graphs, the edge weights are used (or supply `weight` parameter name). Similarly, `nx.to_numpy_array(G, dtype=float)` produces a dense NumPy array adjacency (with optional nodelist and weight) [213] [214]. There is also `nx.to_pandas_adjacency(G)` for a Pandas DataFrame of the adjacency matrix [215] [216].

- **Incidence Matrix:** `nx.incidence_matrix(G, oriented=True)` gives a node-by-edge incidence matrix (SciPy sparse matrix) [217]. If oriented and G is directed, incidence matrix entries for an edge (u→v) will be +1 for u (source) and -1 for v (target); for undirected or unoriented, entries are 1 for each endpoint.

- **Laplacian Matrices:** The (combinatorial) graph Laplacian $L = D - A$ is obtained by `nx.laplacian_matrix(G)` [218] [212]. For weighted graphs, the diagonal entries are weighted degrees. Normalized Laplacian $L_{norm} = I - D^{-1/2}AD^{-1/2}$ is given by `nx.normalized_laplacian_matrix(G)` [218] [219]. For directed graphs, NetworkX includes *directed Laplacians*: `nx.directed_laplacian_matrix(G, alpha)` which yields the normalized Laplacian for strongly connected directed graphs (based on Hermitian normalization) [220] [221], and `nx.directed_combinatorial_laplacian_matrix(G)` for an analog of $D_{out} - A$ with appropriate shape [220] [222]. The documentation notes that for computing directed Laplacians with in-degree, one can reverse the graph and transpose if needed [223] [218].

- **Bethe Hessian:** `nx.bethe_hessian_matrix(G, r)` returns the Bethe Hessian (or deformed Laplacian) $H_r = (r^2 - 1)I - rA + D$ for parameter r [224] [225]. The Bethe Hessian is used in community detection (related to spin-glass models). If r is not specified, a typical choice is $r = \sqrt{\frac{\sum d}{n}}$. This requires SciPy.

- **Algebraic Connectivity (Fiedler value/vector):** `nx.algebraic_connectivity(G, tol=1e-8)` computes the second-smallest eigenvalue of the Laplacian (Fiedler value) for a connected undirected graph [226] [227]. `nx.fiedler_vector(G)` returns the corresponding eigenvector (as a NumPy array of length n) [228] [229]. These use sparse eigenvalue solvers from SciPy; if G is not connected, algebraic_connectivity will return the smallest nonzero eigenvalue (and warn or raise if appropriate). One can specify `normalized=True` to use the normalized Laplacian. There are also convenience routines: `nx.spectral_ordering(G)` which returns nodes ordered by the Fiedler vector (useful for partitioning or graph drawing) [230] [231], and `nx.spectral_bisection(G)` which actually returns a bipartition of the nodes by the sign of the Fiedler vector (a spectral partition cut) [232] [233].

- **Attribute Matrices:** If your graph has data on nodes or edges that you want in matrix form, use `nx.attr_matrix(G, node_attr=None, edge_attr=None)` [234] [235]. This returns a NumPy

matrix where each entry is an attribute from the edge between the corresponding nodes (or 0 if no edge). You can also stack a node attribute as diagonal if needed. The variant `nx.attr_sparse_matrix(G)` returns a SciPy sparse matrix for large graphs [236] [237]. For example, to get the weight matrix: `W = nx.attr_matrix(G, edge_attr="weight")`. These assume attributes are numeric and fill missing edges with 0.

- **Modularity Matrix:** For community analysis, `nx.modularity_matrix(G)` returns the modularity matrix B of graph G [238] [239]. The modularity matrix is defined as $B_{ij} = A_{ij} - \frac{d_i d_j}{2m}$ for an undirected graph, where d_i is degree of node i and m is number of edges. `nx.directed_modularity_matrix(G)` does a similar for directed graphs (using out-degree * in-degree / m) [239] [240]. These matrices are used in spectral community detection algorithms (like Newman's leading eigenvector method).

- **Spectral Eigenvalues:** You can obtain eigenvalue spectra easily: `nx.adjacency_spectrum(G)` returns the eigenvalues of the adjacency matrix [241] [242], `nx.laplacian_spectrum(G)` for Laplacian eigenvalues [242] [243], `nx.normalized_laplacian_spectrum(G)` for normalized Laplacian eigenvalues, `nx.bethe_hessian_spectrum(G, r)` for Bethe Hessian eigenvalues [244] [245], and `nx.modularity_spectrum(G)` for eigenvalues of the modularity matrix [246]. These return NumPy arrays of eigenvalues (which may be complex for directed graphs or certain matrices). Note that for large graphs, computing the full spectrum is expensive (O(n^3) for dense methods); for few eigenvalues, prefer the functions like `algebraic_connectivity` which use sparse iterative solvers.

Note: Many linear algebra functions in NetworkX require **SciPy**. In fact, as of NetworkX 3.x, some algorithms use SciPy by default for better performance. For example, the PageRank algorithm now uses SciPy's eigenvalue solver when available [247]. If SciPy is not installed, NetworkX may fall back to pure Python (with a warning or reduced performance) or raise an error in some cases. Always ensure SciPy is available for heavy linear algebra tasks with NetworkX.

Graph Algorithms and Analysis

NetworkX offers a vast array of graph algorithms across centrality, connectivity, path-finding, flows, isomorphism, community detection, and more. Functions are typically in `networkx.algorithms`. Below we outline major categories of algorithms, with important functions and usage notes:

Path Traversal and Shortest Paths

- **Depth-First and Breadth-First Search:** `nx.bfs_tree(G, source)` produces a tree (as a Graph) of breadth-first search from the source. `nx.bfs_edges(G, source)` yields edges in BFS order. Similarly, `nx.dfs_tree(G, source)` and `nx.dfs_preorder_nodes(G, source)` (and related `dfs_postorder_nodes`) provide depth-first search traversal. These are useful for exploring reachable nodes. Example: to get connected components one could use BFS/DFS from unexplored nodes.

- **Connected Components:** `nx.connected_components(G)` returns an iterator over sets of nodes, each set being a connected component of an undirected graph.

`nx.number_connected_components(G)` counts them. For directed graphs, use `nx.strongly_connected_components(D)` for strongly connected components (SCCs) and `nx.weakly_connected_components(D)` for weak connectivity. There are convenience booleans `nx.is_connected(G)` (undirected) and `nx.is_strongly_connected(D)` etc. You can get subgraphs: `nx.connected_component_subgraphs(G)` (in 3.x this returns a generator of subgraphs, but note in 3.x it might be removed in favor of list comprehension `[G.subgraph(c) for c in connected_components]`).

- **Single-Source Shortest Path:** For unweighted graphs, `nx.shortest_path(G, source)` returns a dict of target->path for all reachable nodes. `nx.shortest_path_length(G, source)` gives distances. For weighted graphs (nonnegative weights), use Dijkstra's algorithm: `nx.single_source_dijkstra(G, source, weight='weight')` returns (distance dict, predecessor dict). There's also `nx.single_source_bellman_ford(G, source)` for graphs with negative weights (but no negative cycles), and `nx.single_source_shortest_path()` which will choose the appropriate method based on weight parameter. You can retrieve a specific path with `nx.shortest_path(G, source, target)`. Example:

```
length, path = nx.single_source_dijkstra(G, src, target)
print(f"Shortest path length = {length}, path = {path}")
```

If only distance is needed, `nx.shortest_path_length(G, src, tgt)` returns the numeric length directly.

- **All-Pairs Shortest Paths:** `nx.all_pairs_shortest_path(G)` returns an iterator of source->(target->path) for each source. `nx.all_pairs_shortest_path_length(G)` similarly for distances. These run Floyd–Warshall algorithm for weighted graphs (O(n^3) complexity) or repeated Dijkstra (O(n·m log n)). If your graph is weighted and fairly sparse, you may prefer running Dijkstra from each node manually or use Johnson's algorithm: `nx.johnson(G)` which is implemented for all-pairs with potentially negative weights (reweights and runs Dijkstra) [248]. Also, `nx.floyd_warshall(G)` returns a distance matrix (as a dict-of-dicts) for all pairs – simpler but also O(n^3).

- **Heuristic Paths (A*):** `nx.astar_path(G, source, target, heuristic)` finds shortest path using the A* algorithm (directed or undirected). You must supply a `heuristic(u,v)` function (admissible estimate of distance). There's also `nx.astar_path_length`. This is useful in spatial graphs where Euclidean distance is a heuristic, etc.

- **Other Distances:** *Eccentricity* – `nx.eccentricity(G, v)` gives the maximum distance from node v to any other node in its component. `nx.radius(G)` = minimum eccentricity (distance from "center" of graph). `nx.diameter(G)` = maximum eccentricity (longest shortest-path distance in the graph). These require all-pairs distances; NetworkX computes them efficiently by pruning using BFS for unweighted or multi-source for weighted. *Center* – `nx.center(G)` returns the list of nodes with minimum eccentricity. *Periphery* – `nx.periphery(G)` returns nodes with eccentricity equal to diameter.

- **Simple Paths:** To enumerate paths, `nx.all_simple_paths(G, source, target, cutoff)` generates all simple (no repeated nodes) paths between source and target up to a given length. `nx.all_simple_edge_paths` does similarly for edges. Use with caution on graphs that can have exponentially many paths. There's also `nx.has_path(G, u, v)` to quickly check reachability. For *k*-shortest paths, one can use `nx.shortest_simple_paths(G, u, v, weight)` which is a generator of simple paths in increasing order of weight (Yen's algorithm).

Connectivity and Cuts

- **Articulation Points (Cut Vertices):** `nx.articulation_points(G)` yields nodes whose removal increases the number of connected components [249]. This applies to undirected graphs. The algorithm runs in O(n+m) using DFS (Tarjan's algorithm). Example usage:

```
list(nx.articulation_points(G))  # e.g. [Nodes that are articulation
points]
```

  Similarly, `nx.bridges(G)` yields edges whose removal disconnects the graph (these are also called cut edges) – works for undirected graphs [249]. Both are useful for network reliability analysis.

- **Biconnected Components:** `nx.biconnected_components(G)` returns sets of nodes that form maximal biconnected subgraphs (2-vertex-connected components). `nx.biconnected_component_edges(G)` similarly for edges grouping [250]. `nx.is_biconnected(G)` checks if graph is one single biconnected component (no articulation points). There's also `nx.biconnected_component_subgraphs(G)` (which may be removed in 3.x, better to just iterate and do `G.subgraph(c)`). These computations also run in linear time. For directed graphs, analogous concept is strongly connected components (see above in paths/traversal).

- **Node and Edge Connectivity:** *Connectivity* refers to minimum cuts. `nx.node_connectivity(G)` computes the node connectivity of graph G (the minimum number of nodes that must be removed to disconnect G) [251]. This can be computed by max-flow algorithms: by Menger's theorem, node_connectivity can be found by computing max number of disjoint paths between every pair or using specialized routines. NetworkX uses flow-based approximations for efficiency. You can also specify two nodes: `nx.node_connectivity(G, s, t)` (which gives the local connectivity between s and t, i.e. minimum vertex cut separating s,t). Likewise, `nx.edge_connectivity(G)` gives the minimum number of edges whose removal disconnects G. Higher-level, `nx.connectivity.connectivity` module provides methods like `nx.minimum_node_cut(G, s, t)` to actually get the set of nodes that separates s and t (if s and t are in different components, it returns an empty set), and `nx.minimum_edge_cut(G, s, t)` for edges. For global connectivity, `nx.minimum_node_cut(G)` returns a set of nodes whose removal disconnects G (a minimum vertex cut for the whole graph), and `nx.minimum_edge_cut(G)` returns a minimum edge cut set. *Note:* These functions use maximum flow computations under the hood (transforming the problem to a flow network and using algorithms like Edmonds-Karp) [251] [252]. Therefore, they can be expensive for large graphs. However, for moderate size, they provide exact solutions. Also available: `nx.all_pairs_node_connectivity(G)` which returns local connectivity for every unordered

pair of nodes (this computes flows for many pairs, so it's costly) [253]. `nx.average_node_connectivity(G)` computes the average of local node connectivity over all pairs [254].

- **Minimum Cuts and Stoer-Wagner:** For *edge* connectivity, an alternative global min cut algorithm is Stoer-Wagner: `nx.stoer_wagner(G)` which returns the minimum cut weight and the two partitions (this is for weighted or unweighted undirected graphs). The flow-based methods above can also yield min cut sets: e.g. `cut_value, partition = nx.minimum_cut(G, s, t)` returns the cut value and the bipartition (set S, set T) of a minimum (s,t)-cut [255] [256]. If you only need the cut value, `nx.minimum_cut_value(G,s,t)` is slightly faster [256] [257]. For directed, these consider directed cuts. To get all *k*-edge-connected components, you can use `nx.k_edge_components(G, k)` (in connectivity package) to find components that remain connected after removing up to k-1 edges.

- **Flow Connectivity (Edge Disjoint Paths):** `nx.edge_disjoint_paths(G, s, t)` and `nx.node_disjoint_paths(G, s, t)` give the maximum number of pairwise edge-disjoint or node-disjoint paths between s and t (by default uses max flow algorithms). They optionally take a `flow_func` parameter to choose a specific flow algorithm. These are essentially the Menger's theorem implementations.

- **Eulerian Circuits:** An Eulerian circuit/trail is a path that uses every edge exactly once. `nx.is_eulerian(G)` checks if the graph has an Eulerian circuit (true if every node has even degree for undirected, or indegree=outdegree for directed). `nx.eulerian_path(G)` or `nx.eulerian_circuit(G, source)` will return an iterator of edges in an Eulerian circuit if one exists [258] (for path it might return an open trail if no cycle). If the graph is not Eulerian, these may raise `NetworkXError`.

- **Lowest Common Ancestor:** In directed acyclic graphs (DAGs), `nx.lowest_common_ancestor(D, u, v)` finds the lowest common ancestor of nodes u and v in the DAG (if a single one exists). The `algorithms.lowest_common_ancestors` module contains implementations (including Tarjan's offline LCA or binary lifting if preprocessing).

- **Directed Acyclic Graph (DAG) algorithms:** For DAGs, topological sorting is key: `nx.topological_sort(D)` returns an iterator of nodes in topological order (exists if and only if D is acyclic). `nx.is_directed_acyclic_graph(D)` checks for acyclicity. If `topological_sort` is called on a graph with cycles, it raises `NetworkXUnfeasible` (since no linear extension exists) [259]. There is also `nx.all_topological_sorts(D)` to generate all possible topological orderings (which can be exponentially many). Other DAG algorithms: `nx.dag_longest_path(D)` for longest directed path in a DAG (which is efficient since no cycles), and `nx.dag_longest_path_length(D)`. `nx.dag_to_branching(D)` can transform a DAG into a branching (arborescence) by splitting nodes with multiple parents. **Transitive closure and reduction:** `nx.transitive_closure(D)` gives a directed graph where (u,v) is an edge if v is reachable from u in D. `nx.transitive_reduction(D)` gives the minimal directed graph with the same reachability as D (for DAGs only). These are useful in partial order logic. Condensation: `nx.condensation(G)` returns a DAG (the *condensation* of G) where each node represents a

strongly connected component of the original directed graph. This is helpful to analyze high-level structure of a directed graph by collapsing cycles.

Centrality Measures

Centrality algorithms measure the "importance" of nodes or edges in a network. NetworkX implements a wide range:

- **Degree Centrality:** `nx.degree_centrality(G)` returns a dictionary mapping each node to its degree centrality, defined as deg(v) / (n-1) for undirected or out-degree for directed [260]. It's basically the normalized degree (thus in [0,1]). For directed graphs, you can also get `nx.in_degree_centrality` and `nx.out_degree_centrality`. This is O(n+m) to compute.

- **Closeness Centrality:** `nx.closeness_centrality(G, u)` computes $\frac{n-1}{\sum_v d(u,v)}$, the reciprocal of the average shortest path distance from u to all other reachable nodes [260]. By default, it uses the sum of distances in the *inward* direction for directed graphs (you can effectively get out-closeness by running on `G.reverse()` if needed) [261]. Nodes in other components get lower values by considering only their component by default. There's an option `wf_improved=True` which weights by reachable fraction to adjust for disconnected networks. For bipartite graphs, NetworkX has a specialized closeness that only considers distances within the same bipartite set [262] (but generally one can use it normally on each component).

- **Betweenness Centrality:** `nx.betweenness_centrality(G, k=None, normalized=True, weight=None)` computes for each node the fraction of shortest paths between all pairs of other nodes that pass through that node. This can be very slow for large graphs (O(n*m) for unweighted via Brandes' algorithm). The parameter `k` allows approximation by sampling k random target pairs [263]. Edge betweenness: `nx.edge_betweenness_centrality(G)` similarly for edges. By default, results are normalized to [0,1]. There are also group betweenness functions: `nx.group_betweenness_centrality(G, C)` gives the share of shortest paths passing through any* node in a given set C.

- **Eigenvector Centrality:** `nx.eigenvector_centrality(G, max_iter=100, tol=1e-6)` computes the eigenvector centrality of nodes (the entries of the leading eigenvector of the adjacency matrix) via the power iteration method. It gives a measure of influence where a node's centrality is proportional to the sum of centralities of its neighbors. For directed graphs, this computes centrality in the sense of the largest eigenvalue of the *directed* adjacency (which can lead to complex values; NetworkX returns the real component or requires a strongly connected graph). There is `nx.eigenvector_centrality_numpy(G)` which uses NumPy's linear algebra (dense, not recommended for large n) and `nx.eigenvector_centrality(G)` which uses an iterative method (sparse-friendly). If the algorithm fails to converge, a `networkx.exception.PowerIterationFailedConvergence` is raised [264].

- **Katz Centrality:** `nx.katz_centrality(G, alpha=0.1, beta=1.0, max_iter=1000)` generalizes eigenvector centrality by adding a constant offset (beta) – effectively counting all walks, exponentially damped by length. `alpha` must be less than 1/λ_max (the inverse of largest eigenvalue) for convergence. Katz centrality can also be used directed (with separate in/out

handling). If you set beta to 1 for all nodes, it's like an attenuation centrality favoring nodes with many neighbors at distance 2,3,... as well.

- **PageRank:** `nx.pagerank(G, alpha=0.85, max_iter=100, tol=1e-6)` computes the PageRank of each node, which is essentially the stationary distribution of a random walk with damping factor α [265]. For directed graphs this is the standard web PageRank (teleport probability 1-α). It returns a dict of node: rank values that sum to 1. By default NetworkX 3.x uses a SciPy sparse solver for PageRank, and will raise an error if SciPy isn't available [247]. You can also use `nx.pagerank_numpy` (dense eigenvector solve) or `nx.pagerank_scipy`. Optionally, a `personalization` dict can be given to bias the random teleportation to certain nodes (useful for personalized PageRank) [266]. **Google matrix:** `nx.google_matrix(G, alpha)` returns the Google matrix M for the PageRank system [266] – an $n \times n$ matrix if needed for analysis. For undirected graphs, PageRank is essentially degree centrality; for directed, it highlights nodes with many incoming links from influential nodes.

- **HITS (Hubs and Authorities):** `nx.hits(G, max_iter=100, tol=1e-8)` returns two dictionaries `(hubs, authorities)` for HITS algorithm on directed graphs [267]. It treats edges as citations: authorities are nodes pointed to by many hubs; hubs point to many authorities. The algorithm is an eigenvector problem on AA^T for hubs and A^TA for authorities. Like PageRank, it can fail to converge if the graph is not strongly connected; NetworkX by default uses a normalization each iteration and stops after max_iter or when change < tol. `nx.hits_numpy` uses dense eigensolver. Typically one multiplies HITS values by a constant so that sum of squares is 1 or sum is 1.

- **Current-Flow (Information) Centrality:** These are alternative centralities based on electrical network analogies (also known as random-walk centralities). `nx.current_flow_closeness_centrality(G)` treats the graph as a resistor network and computes closeness centrality using effective resistance distance instead of shortest paths [268] [269]. `nx.current_flow_betweenness_centrality(G)` computes betweenness by considering flow passing through edges when one unit of current is injected at one node and extracted at another, averaged over all pairs [270]. These require solving linear equations (Laplacian pseudoinverse), so SciPy is needed; for large graphs, they are expensive (they do an all-pairs effective resistance computation). There are also *approximate* variants (using methods from network science); NetworkX includes `nx.approximate_current_flow_betweenness` which can sample.

- **Communicability and Subgraph Centrality:** Communicability between two nodes counts the number of walks of all lengths between them (more precisely, the sum of $(A^k)_{ij}/k!$ for k=0..∞, which is an entry of the matrix exponential exp(A)) [269]. `nx.communicability(G)` returns a dict of dicts with communicability between every pair of nodes (uses matrix exponential – only feasible for smaller graphs). **Communicability centrality**, also called **subgraph centrality**, counts closed walks at a node (the diagonal of exp(A)) [271] [272]. NetworkX provides `nx.communicability_centrality(G)` for that. Also, `nx.communicability_betweenness_centrality(G)` is defined by Estrada as a betweenness measure using communicability instead of shortest paths [273]. These functions rely on NumPy/SciPy for matrix exponentials and may not work for very large n due to time and memory constraints.

- **Centrality for Groups:** NetworkX can evaluate centrality of groups of nodes. `nx.group_betweenness_centrality(G, C)` gives the fraction of shortest paths that pass through *any* node in a given set C (useful for analyzing a committee or set acting collectively). Similarly `nx.group_closeness_centrality(G, S)` measures how close the group S is to all other nodes (it uses the concept of distance from S to a node as the minimum distance from any member of S to that node) [260] [274]. Group degree centrality is trivial (just sum of degrees inside S, etc.).

- **Other Centralities:** *Load centrality* is similar to betweenness (the load centrality is essentially betweenness without the normalization by number of pairs). *Clustering centrality* or others are not standard in NX. There is *harmonic centrality* (like closeness but uses reciprocal distances without needing reachability normalization) accessible as `nx.harmonic_centrality(G)`. *Entropy centrality*, *boundary centrality* etc. are not built-in.

Example – Betweenness Centrality: To compute and interpret node betweenness:

```
import networkx as nx
G = nx.erdos_renyi_graph(10, 0.4)
b = nx.betweenness_centrality(G, normalized=True)
for node, score in b.items():
    print(f"Node {node} betweenness centrality: {score:.3f}")
```

Nodes with higher scores lie on a larger fraction of shortest paths between others. For edge betweenness, use `nx.edge_betweenness_centrality(G)` similarly.

Link Analysis (Ranking Algorithms)

These algorithms are especially relevant for directed networks, ranking nodes by importance in a network flow sense:

- **PageRank:** As described above, `nx.pagerank(G, α)` returns a dict of node PageRank values [265]. You can adjust `alpha` (damping) from default 0.85. The implementation will iterate until convergence or `max_iter`. Tolerance `tol` is the L1-error target. Dangling nodes (nodes with no outlinks) are automatically handled by distributing their weight uniformly to all nodes each iteration (or using a specified `personalization`). If the algorithm fails to converge within `max_iter`, a `networkx.exception.ExceededMaxIterations` is raised [275]. The sum of PageRank values is 1. For multigraphs, edges are typically treated as separate links (so a node with parallel edges to another counts them with multiplicity in PageRank). The **Google matrix** can be obtained for analysis or custom workflows using `nx.google_matrix(G, alpha)` [266], which gives an $n \times n$ NumPy matrix $M = \alpha P + (1-\alpha)\frac{1}{n}\mathbf{1}\mathbf{1}^T$ where *P* is the row-stochastic transition matrix of the graph (with damping factor).

- **HITS:** `nx.hits(G)` yields hub scores and authority scores for each node [267]. After computation, you might identify "authority" nodes (high authority score) as those that are referenced by many good hubs, and "hub" nodes (high hub score) as those that point to many good authorities. If your

graph is not strongly connected, HITS scores might concentrate on a subset; it's often run on a focused crawl (e.g. a subgraph of interest). Convergence issues: HITS will converge to principal eigenvectors of $A^T A$ and AA^T; sometimes it can oscillate (NetworkX uses a power iteration method which usually converges, but no guarantee if graph has multiple dominant eigenvalues – typically resolved by normalization each iteration).

- **SALSA:** NetworkX does not explicitly have SALSA (Stochastic Approach for Link-Structure Analysis), which is a variant of HITS; but SALSA's hub and authority can be computed via a combination of in-degree and out-degree in bipartite projections.

- **Eigenvector vs PageRank vs HITS:** These three are related. For an undirected graph, PageRank and eigenvector centrality are proportional (PageRank essentially becomes degree centrality in an undirected graph). For directed, PageRank is a variant of eigenvector centrality on the Google matrix (with added teleportation making it well-defined even for networks that are not strongly connected), while HITS gives two sets of scores (hubs and authorities).

Example – PageRank vs HITS:

```
D = nx.DiGraph([(1,2),(1,3),(2,3),(3,1)])
print(nx.pagerank(D, alpha=0.9))
# {1: 0.327, 2: 0.219, 3: 0.454} (example output)
print(nx.hits(D))
# ({1: 0.0, 2: 1.0, 3: 0.0}, {1: 0.0, 2: 0.0, 3: 1.0}) – hubs, authorities
```

In this example, node 3 has the highest PageRank (it has two in-links including one from node1 which itself has input), and HITS identifies node2 as a pure hub (points to 3) and node3 as pure authority. PageRank's ranking often correlates with HITS authority but accounts for link transitive influence differently.

Link Prediction and Similarity

These functions estimate the likelihood of missing edges or similarity between nodes based on network structure:

- **Common Neighbor-based indices:** `nx.jaccard_coefficient(G, ebunch)` computes the Jaccard similarity for each pair in *ebunch* (an iterable of node pairs) [276]. Jaccard coefficient = (number of common neighbors) / (number of total neighbors). It returns a generator of `(u, v, score)` for each pair [277]. Similarly, `nx.adamic_adar_index(G, ebunch)` returns the Adamic–Adar score for each pair – the sum of $1/\log(\deg(w))$ over common neighbors w of u and v [276]. `nx.resource_allocation_index(G, ebunch)` is another metric: sum of $1/\deg(w)$ over common neighbors w (treats each common neighbor as a resource unit divided equally). `nx.preferential_attachment(G, ebunch)` returns for each pair the product of degrees (this isn't really a probability but a propensity measure). These are all implemented as generators that yield scores for specified pairs – if you want all non-edges scored, you have to generate ebunch as `nx.non_edges(G)`. Example:

```
preds = nx.jaccard_coefficient(G)
u,v,p = next(preds)
```

yields a prediction for first pair. Typically you sort these scores to recommend new edges.

- **Katz Score for link prediction:** Katz index between two nodes is like a weighted count of all walks between them. While NetworkX doesn't have a direct `nx.katz_similarity`, one could use `nx.resource_allocation_index` or `nx.simrank_similarity` (if implemented; SimRank might not be in core NX as of 3.6).

- **Graph Distance-based:** `nx.cn_soundarajan_hopcroft(G, ebunch, community='community')` and `nx.ra_index_soundarajan_hopcroft` are variations of common neighbor counts that give more weight if common neighbors are in the same community (using a node attribute to define communities, e.g. `"community"`). These are useful when community metadata is available.

- **Preferential Attachment:** `nx.preferential_attachment(G, ebunch)` simply yields (deg(u)*deg(v)) for each pair [278]. While trivial, it's a baseline for link prediction (often high-degree nodes have higher chance to connect).

- **Accuracy Consideration:** These link prediction measures do not directly tell if an edge *will* appear, but provide scores to rank pairs. One can evaluate their performance by comparing to a test set of known future edges.

Graph Isomorphism and Matching

NetworkX can check graph isomorphism and find subgraph matches:

- **Graph Isomorphism:** `nx.is_isomorphic(G1, G2)` checks if two graphs are isomorphic (ignoring node labels, just structure) and returns True/False. By default it uses the VF2 algorithm, which is implemented in `networkx.algorithms.isomorphism` and is quite efficient for moderate-sized graphs [279]. You can also get an isomorphism mapping: `iso = nx.algorithms.isomorphism.GraphMatcher(G1, G2)` creates a GraphMatcher object, then `iso.is_isomorphic()` returns True/False and `iso.mapping` (or iterate `iso.match()` to get all mappings). The GraphMatcher allows specifying attributes to consider: you can supply a `node_match` function and/or `edge_match` to restrict the isomorphism to consider color labels, etc. For example:

```
from networkx.algorithms import isomorphism
GM = isomorphism.GraphMatcher(G1, G2, node_match=lambda x,y:
x['type']==y['type'])
GM.is_isomorphic()  # True/False
GM.mapping  # dictionary mapping nodes of G1 to nodes of G2
```

Similarly `DiGraphMatcher`, `MultiGraphMatcher` exist for directed and multigraphs. There are specialized isomorphism checks like `nx.is_directed_acyclic_graph(H)` (for structure property, not full iso) and `nx.is_isomorphic(G1, G2, node_match=..., edge_match=...)` directly with match functions as parameters.

- **Subgraph Isomorphism:** The `GraphMatcher` can also find a subgraph isomorphic to another graph. If `G2` has <= nodes of `G1`, `GM.subgraph_is_isomorphic()` will tell if a subgraph of G1 is isomorphic to G2. You can get the matching mapping via `GM.mapping` after calling `GM.subgraph_is_isomorphic()`. To iterate over all subgraph isomorphisms, use `GM.match()` generator.

- **Monomorphism and others:** If you need monomorphism or other variants, NetworkX does not have separate classes but GraphMatcher can be adapted (e.g. by customizing the equality).

- **Isomorphic Graph Classes:** Some graph families have simple invariants: e.g. `nx.is_isomorphic(Tree1, Tree2)` can check for tree isomorphism. For faster tree-specific isomorphism, one might use AHU algorithm externally, but NX uses VF2 general approach. For multigraphs, edge count per pair and degree sequence are quick checks before the actual search.

Example – Finding a subgraph:

```
# Find a triangle (K3) in graph G
from networkx.algorithms import isomorphism
GM = isomorphism.GraphMatcher(G, nx.complete_graph(3))
if GM.subgraph_is_isomorphic():
    print("Triangle found with nodes:", GM.mapping)
```

This will output one mapping of the 3-cycle if present. To get all, iterate `for mapping in GM.subgraph_isomorphisms_iter(): ...`.

- **Graph Hashing:** Although not exactly isomorphism, `nx.weisfeiler_lehman_graph_hash(G)` can produce a graph hash invariant using Weisfeiler-Lehman coloring. If two graphs have different WL hash, they are surely non-isomorphic; if same, they *likely* are isomorphic (not guaranteed, but a quick heuristic).

- **Isomorphism Exceptions:** The isomorphism functions should not raise exceptions normally; they either return False or an empty iterator if no match. The only caution is for multigraph isomorphism with required matching of edge keys or attributes – be sure to supply appropriate edge_match.

Network Flow (Max-Flow and Min-Cut)

NetworkX contains algorithms for maximum flow and minimum cut in networks. These functions reside in `networkx.algorithms.flow`:

- **Maximum Flow:** `nx.maximum_flow(G, s, t, capacity='capacity')` computes the value of the maximum flow from source *s* to sink *t*, and also returns the *flow dict* describing flow on each edge [280] [281]. The return is `(flow_value, flow_dict)`. The flow dict is nested like `flow_dict[u][v] = f` (for each edge u->v in the original graph, indicating how much flow is sent along it). If `G` is undirected, internally it's treated as directed with reciprocal edges. You can choose the algorithm via the `flow_func` parameter; options include Edmonds-Karp (`nx.algorithms.flow.edmonds_karp`), Dinic (`dinitz`), Preflow-Push (`preflow_push`), or the newer Push-Relabel. By default, `maximum_flow` will pick an efficient algorithm (e.g. Edmonds-Karp for dense or small networks) [282]. The **maximum flow value** alone can be obtained by `nx.maximum_flow_value(G, s, t)` more efficiently if you don't need the flow assignment [283].

- **Minimum (s,t)-Cut:** With the max-flow, you automatically get the min-cut by the max-flow min-cut theorem. `nx.minimum_cut(G, s, t)` returns `(cut_value, (reachable, non_reachable))` where `reachable` and `non_reachable` are sets of nodes on the source and sink side of the minimum cut of that value [255] [256]. Edges from reachable to non_reachable constitute the cut-set. `nx.minimum_cut_value(G, s, t)` gives just the cut capacity. For global minimum cut (not specific s,t), use `nx.stoer_wagner(G)` as noted earlier.

- **Flow algorithms detail:** *Edmonds-Karp* is an implementation of Ford-Fulkerson using BFS (O(V * E^2)). *Dinic's algorithm (`dinitz`)* improves to O(min(V^(2/3), E^(1/2)) * E) in theory and usually faster in practice. *Preflow-Push* (`preflow_push`) is another strategy (worst-case O(V^2 * E)). NetworkX also includes *Shortest Augmenting Path* (`shortest_augmenting_path`) which is Edmonds-Karp with some heuristics. Each of these can be invoked directly if needed (they each return a residual network object and flow value). Under the hood, flows are computed on a **residual network** (an internal `networkx.algorithms.flow.ResidualNetwork` class). If you need to inspect that, you can call `R = nx.build_residual_network(G, capacity)` [284] to get the initial residual graph with 0 flow [285]. But typically you just use the results.

- **Minimum Cost Flow:** For flows with costs on edges, NetworkX implements the *network simplex* algorithm. If each edge has a `'capacity'` and a `'weight'` (cost per unit flow) and optionally each node has a supply/demand (`G.nodes[n]['demand']`, default 0), you can compute a min-cost flow. Use `nx.min_cost_flow(G)` which returns a flow dict like max_flow but meeting all demands at minimum cost [286] [287]. If there is no feasible flow, it raises `NetworkXUnfeasible` [259]. The cost of a given flow dict can be calculated with `nx.cost_of_flow(G, flow_dict)` [288] [289]. There's also `nx.min_cost_flow_cost(G)` which directly gives the cost of the minimum cost flow without the flow assignment if you only need that [286] [287]. For max flow min cost (e.g. maximize flow from s to t with minimum cost for that maximum amount), use `nx.max_flow_min_cost(G, s, t)` [290] [291] – it will push as much flow as possible from s to t and among all max flows choose the one with least cost. The algorithm for min-cost flow is a capacity scaling successive shortest path method by default, or one can use `nx.capacity_scaling(G)`

which is an alternative implementation for min cost flow (sometimes faster for dense graphs) [292] [293].

- **Circulation with Demands:** If you have demands on edges (each edge (u,v) requires a certain flow amount), the flow algorithms can accommodate that by adjusting capacities. NetworkX specifically supports node demands (supply >0, demand <0) in min_cost_flow as mentioned. There is also `nx.maximum_flow_with_constraints` if needed for more complex constraints (not in main API, but one can transform constraints into node demands or edge capacities).

- **Example – Max Flow and Min Cut:**

```
capacity = {'capacity': 0}  # default capacity attribute
val, flow = nx.maximum_flow(G, s=1, t=7)
print("Max flow value:", val)
cut_val, (S, T) = nx.minimum_cut(G, 1, 7)
print("Min cut capacity:", cut_val, "Partition S:", S, "T:", T)
```

After this, `flow` might look like `{1: {2: 5, 3: 3}, 2: {4:5}, ...}` mapping each edge's flow. The min cut partitions S and T verify that all edges from S to T are saturated at capacity equal to cut_val. If a flow problem is unbounded (e.g. there is an augmenting path with infinite capacity or a negative cycle in min-cost), NetworkX will raise `NetworkXUnbounded` [294] (for instance in min_cost_flow if a negative-cost cycle is present and no flow bounds, it raises `NetworkXUnbounded` [295]).

Matching and Covering

- **Maximum Matching (Unweighted):** For bipartite graphs, NetworkX uses the Hopcroft–Karp algorithm. `nx.maximum_matching(G)` finds a maximum cardinality matching in a general graph (not necessarily bipartite) using a *general graph* algorithm (it will find any maximal set of edges with no shared endpoints). For bipartite, there's a faster: `nx.algorithms.bipartite.matching.hopcroft_karp_matching(B, top_nodes)` if you know the bipartition. However, `nx.maximum_matching` should handle both cases (it calls a general algorithm for general graph, which for bipartite defaults to Hopcroft–Karp). To specifically ensure bipartite algorithm, you can use `nx.bipartite.maximum_matching(B)`. The output is a dict mapping each node to the node it's matched to (or not present if unmatched). There's also `nx.is_matching(G, matching)` to verify a given set of edges is a valid matching, and `nx.is_maximal_matching(G, matching)` to check if no edge can be added, etc.

- **Maximum Weighted Matching:** `nx.max_weight_matching(G, maxcardinality=False)` uses Edmonds' Blossom algorithm to find a matching that maximizes the sum of weights (for weighted graphs; treats `'weight'` edge attribute as weight, default 1). If `maxcardinality=True`, it finds the maximum-cardinality matching *among* those with maximum weight (only relevant if some weights are equal or zero). This works for general graphs (including non-bipartite). Complexity is roughly O(n^3) but handles a few hundred nodes reasonably.

- **Bipartite Graph Algorithms:** Aside from matching, bipartite graphs have unique routines. `nx.bipartite.sets(B)` will return the two node sets if the graph is bipartite (or raise AmbiguousSolution if not sure which side a node belongs to in a disconnected bipartite graph) [296]. `nx.is_bipartite(G)` simply checks bipartiteness. **Bipartite Projections:** Given a bipartite graph B with node sets U and V, `nx.bipartite.projected_graph(B, U)` returns the one-mode projection on set U (nodes in U connected if they have a common neighbor in V). There is also `nx.bipartite.weighted_projected_graph(B, U)` to count the number of shared neighbors as edge weights.
  For centrality on bipartite networks, one approach is to run centrality on the projected graph. But NX also offers bipartite-specific algorithms: e.g. `nx.bipartite.betweenness_centrality(B, endpoints=True)` will compute betweenness centrality considering paths that stay in the bipartite structure. Similarly, degree centrality etc. can be computed per bipartite set (just restrict to one set or normalize differently). NetworkX's bipartite submodule also has `nx.bipartite.closeness_centrality` which normalizes distances within the same set.

- **Vertex Cover, etc.:** Minimum Vertex Cover can be derived from maximum matching in bipartite graphs (Kőnig's theorem). NetworkX provides `nx.min_weight_vertex_cover(B)` for bipartite graphs given a weight on nodes, or `nx.min_vertex_cover(G)` for general graphs via approximation or ILP (in `approximation.vertex_cover` there's a 2-approx for general). Similarly, edge cover, etc. `nx.maximal_independent_set(G)` gives some maximal (not maximum) independent set via a greedy algorithm, and for bipartite graphs, a minimum vertex cover is complement of a maximum independent set.

- **Travelling Salesman and Matching-based approximations:** In `nx.algorithms.approximation`, you find heuristics like `nx.approximation.traveling_salesman_problem` which uses Christofides algorithm (which requires minimum weight perfect matching as subroutine, solved by Blossom algorithm) – these are advanced but demonstrate interplay of matching algorithms.

Graph Coloring and Labeling

- **Node Coloring:** `nx.coloring.greedy_color(G, strategy='largest_first')` attempts to color the graph's nodes (assign integer colors) with as few colors as possible using a greedy heuristic [297]. Strategies include `'largest_first'` (order by degree), `'smallest_last'`, `'DSATUR'`, etc. It returns a dict node->color. It's not guaranteed optimal but often near-optimal. You can check if a coloring is proper with `nx.coloring.is_valid_coloring(G, coloring)`.

- **Edge Coloring:** There's no built-in edge coloring algorithm (which is a harder problem in general). But for bipartite graphs, a proper edge coloring can be found (it's equivalent to matching partitioning). One could use networkx's `nx.algorithms.bipartite.matching` repeatedly to find a Shannon multicolor, but no out-of-the-box function.

- **Graph Labeling Problems:** *Graph coloring* above is one. Graph isomorphism with labeled nodes can be seen as a labeling problem (GraphMatcher covers attributes). *Graph factorization* (like 1-factorization of a regular graph into perfect matchings) is not explicitly in NX, but one can attempt it via repeated matchings.

Community Detection (Clustering)

Community detection algorithms aim to find groups of nodes with dense internal connections and sparse external ones. NetworkX provides many community algorithms in `networkx.algorithms.community`. Key functions:

- **Modularity-based Greedy:** `nx.greedy_modularity_communities(G)` uses the Clauset-Newman-Moore greedy agglomerative approach to maximize modularity [298] [299]. It returns a list of sets of nodes (the communities). This is an efficient heuristic (O(n^2 log n) or so). There's also `nx.naive_greedy_modularity_communities(G)` which is an alternative implementation [300].

- **Louvain Algorithm:** The Louvain method is a widely used heuristic for community optimization of modularity. `nx.community.louvain_communities(G, weight='weight', resolution=1)` returns a list of communities (each a set of nodes) by running the Louvain algorithm [301] [302]. You can also get the hierarchical structure via `nx.community.louvain_partitions(G)` which yields communities at each level of coarsening [303]. The resolution parameter (>1 finds smaller communities, <1 larger). **Note:** The Louvain functions in NetworkX 3.6 are **dispatchable to backends** – by default, NetworkX does *not* have its own pure-Python Louvain (the function will raise `NetworkXNotImplemented` or require an installed backend) [304] [305]. For example, the `leidenalg` or `cdlib` could be a backend. The doc indicates Leiden is only available via a backend [306]. If you call `nx.algorithms.community.louvain_communities` without a backend, it attempts to use the `python-louvain` package if installed; otherwise it might raise. So ensure to have `python-louvain` installed for this to work in pure Python.

- **Label Propagation:** `nx.asyn_lpa_communities(G, weight=None, seed=None)` implements asynchronous label propagation (Raghavan et al. 2007) [307] [308]. It returns an iterator of sets (each set is a community). This method is fast (linear) but not deterministic. NetworkX also offers a "fast label propagation" variant `nx.fast_label_propagation_communities(G)` which may give different results using a heuristic improvement [309]. Label propagation requires no tuning parameter and often finds reasonably good communities, though results can vary each run (use `seed` for reproducibility).

- **Hierarchical Divisive (Girvan–Newman):** `nx.community.girvan_newman(G)` implements the Girvan–Newman algorithm which recursively removes edges with highest betweenness to split communities [310] [311]. It returns a generator that yields communities at each step of edge removal. Typically you pick the partition with highest modularity from that sequence. Usage:

```
comp_generator = nx.community.girvan_newman(G)
for communities in comp_generator:
    print(tuple(sorted(c) for c in communities))
```

to examine each level. This can be slow for large graphs (complexity O(n*m) for betweenness per step).

- **Spectral Partitioning:** `nx.spectral_modularity_bipartition(G)` splits graph into two communities by computing the leading eigenvector of the modularity matrix (a form of spectral partition) [312]. It returns a tuple of two sets (bipartition). Similarly, `nx.greedy_node_swap_bipartition(G)` performs a two-community partition by greedily swapping nodes to improve modularity [313] [314]. These were introduced in NetworkX 3.6 (spectral bipartition and a node swap heuristic for refining) [315]. They specifically target splitting into *two* communities. To get multiple communities, one could recursively apply bipartitions.

- **K-Clique Communities:** `nx.k_clique_communities(G, k)` finds communities as per the clique percolation method: each community is a union of k-cliques that are adjacent (sharing k-1 nodes) [316] [317]. It returns an iterator of sets of nodes, each being one community. You need to list it: e.g. `list(nx.k_clique_communities(G,3))`. This can identify overlapping communities if a node is part of multiple k-cliques clusters.

- **Edge Partition (Quality metrics):** `nx.community.quality.modularity(G, communities)` computes the modularity value for a given partition (useful to evaluate result) [318] [319]. There's also `nx.partition_quality(G, communities)` which returns (coverage, performance) – coverage is fraction of edges inside communities, performance is fraction of intra-community and inter-community non-edges overall [320] [321]. These metrics help judge community splits.

- **Local Community Detection:** Functions like `nx.greedy_source_expansion(G, source, cutoff, k)` attempt to find the "community" around a given source node using local optimization (expanding or shrinking a set to maximize a local score like modularity density) [322] [323]. This is useful when you care about the community of a specific node rather than partitioning the whole graph.

- **Centrality-based Partitioning:** The Girvan–Newman method falls here (edge betweenness as a centrality to cut). Also, `nx.community.centrality.girvan_newman` is one, and *Leiden* is an improvement over Louvain (not directly in NX without backend).

- **Communities in Directed Graphs:** Most algorithms above assume undirected. Some (like Girvan–Newman) can be applied to directed by considering symmetric edges or considering in/out separately. NetworkX's implementation of modularity treats directed as undirected by default. For directed networks, one can use directed modularity (but then Louvain, etc., require that defined).

Example – using Louvain (with backend):

```
import networkx.algorithms.community as nxcom
comms = nxcom.louvain_communities(G, resolution=1)
print(len(comms), "communities found of sizes:", [len(c) for c in comms])
```

This yields communities maximizing modularity. If `python-louvain` is installed, this will work; otherwise you might need to install a backend like `pip install python-louvain`.

Community Output: Most algorithms return communities as list of sets of nodes. They are not (yet) in a Community object or labeled structure in NX. If you need to assign each node a community label, you can post-process: e.g.

```
part = {node: i for i, comm in enumerate(comms) for node in comm}
```

gives a partition dict node->community_index.

Other Measures and Utilities

- **Clustering Coefficient:** `nx.clustering(G, nodes=None)` gives the clustering coefficient for each node (the fraction of pairs of neighbors of the node that are connected) – essentially, for node v with degree d, C(v) = \frac{2 * \text{# of triangles through v}}{d(d-1)}. Returns a dict for specified nodes or single value if one node. `nx.average_clustering(G)` gives the mean clustering coefficient over all nodes [324]. `nx.transitivity(G)` gives the global clustering coefficient = 3 * (# of triangles) / (# of connected triples). Triangles specifically: `nx.triangles(G, node)` counts the number of triangles node is part of (each triangle counted per node). `nx.triangles(G)` gives a dict for all nodes. These are O(n * d^2) or use efficient vectorization if SciPy is available.

- **Square Clustering:** `nx.square_clustering(G, nodes=None)` measures fraction of possible 4-cycles that exist for each node.

- **Assortativity and Mixing:** `nx.degree_assortativity_coefficient(G)` computes the Pearson correlation of degrees at either ends of edges (degree assortativity). `nx.attribute_assortativity_coefficient(G, 'gender')` would compute assortativity by a categorical node attribute. `nx.mixzing_matrix(G, attribute)` and `nx.degree_mixing_matrix(G)` give full mixing matrices. These help understand homophily.

- **Rich-Club Coefficient:** `nx.rich_club_coefficient(G, k)` returns the rich-club coefficient for degree k (density of subgraph induced by nodes with degree > k). `nx.rich_club_coefficient(G)` returns a dict for all k.

- **Distance Measures:** In addition to diameter, radius (in connectivity section), NetworkX has `nx.average_shortest_path_length(G)` for the average distance between all reachable pairs (graph must be connected for an undirected graph, or strongly connected for directed unless you specify a source subset). There's also `nx.efficiency(G, u, v)` and `nx.global_efficiency(G)` which measure the harmonic mean of distances (useful for network efficiency analysis).

- **Similarity Measures:** Aside from link prediction metrics, `nx.similarity.jaccard_coefficient` etc. are in link prediction. For node similarity beyond structural, you would use domain-specific metrics or embedding techniques outside NetworkX's scope (like Node2Vec, etc., which are not built-in).

- **Graph Metrics Summary:** `nx.info(G)` prints a summary of the graph type, node count, edge count and average degree. For quick stats: `nx.density(G)` for density, `nx.number_of_cliques(G)` for number of cliques etc. `nx.number_of_selfloops(G)`, `nx.selfloop_edges(G)` to list self-loops.

- **Isolates:** `nx.is_isolate(G, n)` checks if node n has no neighbors. `list(nx.isolates(G))` lists all isolates (no degree nodes).

In summary, NetworkX's algorithm suite covers most classical graph computations. For large graphs (thousands of nodes or more), consider the complexity: e.g. centrality measures like betweenness can be slow (but can approximate with `k` parameter), while connectivity and flow rely on heavy algorithms but are usually okay up to maybe 10k nodes depending on density.

Interoperability and Data Conversion

NetworkX graphs can be converted to and from many other formats, enabling integration with libraries like **pandas**, **NumPy/SciPy**, and JSON serialization. Key conversion functions include:

- **To/From Python data structures:** `nx.to_dict_of_dicts(G)` exports the graph adjacency as a dictionary of dictionaries `{u: {v: edge_data,...}, ...}` [325] [326]. `nx.from_dict_of_dicts(d, create_using=Graph)` does the inverse: builds a graph from such an adjacency dict (treating innermost dict as edge attributes) [327] [328]. Similarly, `nx.to_dict_of_lists(G)` gives `{u: [v1,v2,...]}` adjacency [329] and `nx.from_dict_of_lists(d)` creates a graph from that (assuming undirected or outward directed) [330] [331]. These are useful for quick interoperability with pure Python data or for shallow copies.

- **Edge lists:** `nx.to_edgelist(G)` returns a list of (u,v,attrdict) tuples for all edges [332]. `nx.from_edgelist([(u,v), ...], create_using=Graph)` builds a graph from an edge list (no attributes). If you have weighted edges in a list, you can add them via `G.add_weighted_edges_from`. Also, **pandas** DataFrames: `nx.to_pandas_edgelist(G)` returns a DataFrame with columns `['source','target']` (and additional columns for edge attributes) [333]. `nx.from_pandas_edgelist(df, source='src', target='tgt', edge_attr=['w1','w2'], create_using=Graph)` reads a DataFrame to graph, creating one edge for each row, and optional edge_attr list to add those columns as edge data [334] [335]. This is a very handy way to go between NetworkX and pandas. For node data, you might consider joining on node indices.

- **NumPy arrays:** `nx.from_numpy_array(A, create_using=Graph)` builds a graph from a NumPy 2D array or matrix, interpreting it as an adjacency matrix [336] [337]. Optionally, for weighted graphs, the array values are used as weights (zero means no edge). If A is a NumPy matrix, this yields a Graph with nodes 0...n-1. `nx.to_numpy_array(G, dtype=float)` returns the adjacency matrix as a NumPy array [338] [339] (defaults to float, you can specify dtype or order). There's also `nx.to_numpy_matrix(G)` if you specifically want a `numpy.matrix` (though that class is deprecated in favor of ndarray).

- **SciPy sparse matrices:** `nx.to_scipy_sparse_array(G, format='csr')` returns a SciPy sparse adjacency matrix (csr by default) [340] [341]. `nx.from_scipy_sparse_array(A, create_using=Graph)` builds a graph from a SciPy sparse adjacency matrix [342] [343]. This is useful if you want to use SciPy's graph algorithms or pass a graph to other libraries (like for spectral clustering via scikit-learn, etc.). Keep in mind to ensure the matrix is symmetric if you want an undirected graph.

- **pandas adjacency matrix:** `nx.to_pandas_adjacency(G, dtype=float)` returns a Pandas DataFrame indexed by nodes, with entry [i][j] = weight (or 1 if unweighted, 0 if no edge) [216] [334]. `nx.from_pandas_adjacency(df)` creates a graph from a Pandas DataFrame where the index and columns of df are the nodes and values are interpreted as edge weights [335] [344]. This requires df to have identical index and columns.

- **Conversion from other graph libraries:** NetworkX can interoperate with GraphML via `nx.nx_agraph` and `nx.nx_pydot` for Graphviz (see next section), but also has helper `nx.to_networkx_graph(data)` that attempts to guess the input type and convert it [345] [346]. It supports input types like: another NetworkX graph (returns a copy), dict-of-dict adjacency, dict-of-list adjacency, list of edges, NumPy matrix/array, SciPy matrix, or even a PyGraphviz `AGraph` or a pydot `Dot` if those libraries are installed [345] [346]. The NetworkX Graph constructor `nx.Graph(data)` actually calls `to_networkx_graph` internally [347], so you can often do `G = nx.Graph(some_matrix)` and it will do the right thing [348]. Similarly `nx.DiGraph(data)` etc.

- **Relabeling Nodes:** If you need to convert node labels (say from strings to integers or vice versa), use `nx.relabel_nodes(G, mapping)` which returns a new graph with nodes relabeled by the given dict mapping (old->new) [349]. Alternatively, `nx.convert_node_labels_to_integers(G, label_attribute='old_label')` will produce a new graph with nodes 0..n-1 and store the old labels in a node attribute if you specify [350]. This is helpful for algorithms requiring integer labels or for output clarity.

Example – pandas interop:

```
import pandas as pd
edges = pd.DataFrame({'u':[1,2,3], 'v':[2,3,4], 'capacity':[10,20,30]})
G = nx.from_pandas_edgelist(edges, 'u', 'v', edge_attr='capacity')
df = nx.to_pandas_edgelist(G)
print(df)
```

Here, `edges` DataFrame is converted to a graph G with a 'capacity' attribute on each edge, then converted back to a DataFrame `df` (which will include a 'capacity' column). This round-trip demonstrates how convenient DataFrames are for bulk loading or analysis of edges.

Graph Input/Output (File Formats)

NetworkX supports reading from and writing to many graph file formats. These functions are mostly in `networkx.readwrite`. Here are the main formats:

- **Edge List:** Simple text format of one edge per line (node labels separated by space or delimiter).
- Write: `nx.write_edgelist(G, path, delimiter=' ', data=True)` writes each edge `u v key1=val1 key2=val2 ...` if `data=True` for attributes [351] [352]. Without data, it just writes `u v`. For MultiGraphs, it writes one line per edge, possibly repeating node pairs.
- Read: `nx.read_edgelist(path, create_using=Graph, nodetype=int, data=[('weight', float)])` to parse an edgelist. `nodetype` will cast node labels (int in this example). `data=[(...)]` instructs how to parse edge attributes (here expecting a "weight" as float) [353] [351]. There's also `nx.read_weighted_edgelist(path)` specialized if the third column is a weight [352] [354]. These are straightforward; the default delimiter is whitespace.

- Example:

```
nx.write_edgelist(G, "graph.edgelist")
H = nx.read_edgelist("graph.edgelist", nodetype=str)
```

This writes and reads an edgelist (node names will be strings if they were strings, or can enforce type).

- **Adjacency List:** Another simple text: each line "node neighbor1 neighbor2 ...".

- Write: `nx.write_adjlist(G, path)` [355]. By default it will include node and all its neighbors on one line.

- Read: `nx.read_adjlist(path, create_using=Graph, nodetype=None)` [355]. This is less used nowadays than edgelist. There is also a **multiline adjacency list** variant that allows multiple lines per node for readability – use `nx.write_multiline_adjlist` and `nx.read_multiline_adjlist` [356]. The multiline format writes node and neighbors each on separate lines for complex data.

- **Pickled Graph:** Not exactly a standard format but you can use Python pickling: `nx.write_gpickle(G, "file.gpickle")` and `nx.read_gpickle("file.gpickle")`. This preserves Python object structure including attributes. It's only readable by Python/NetworkX but often convenient for quick persistence.

- **GraphML (XML):** GraphML is an XML-based standard for graphs (supports nodes, edges, attributes, hierarchical graphs).

- Write: `nx.write_graphml(G, "graph.graphml", encoding='utf-8', prettyprint=True)` [357]. It will include node and edge attributes. GraphML is good for interoperability (e.g. Gephi, Cytoscape).

- Read: `nx.read_graphml("graph.graphml")` [357].
- Note: GraphML expects string IDs; NetworkX will auto-generate IDs if nodes are not strings or ints. Requires lxml or xml library. GraphML can handle multigraph via `key` elements. NetworkX's GraphML support will throw an error for some data types it cannot handle.

- There are similar XML-based formats: **GEXF** (Graph Exchange XML Format, used by Gephi) and **GML** (Graph Modeling Language, a text format).

- **GEXF:**

- Write: `nx.write_gexf(G, "graph.gexf")` [358] [359].
- Read: `nx.read_gexf("graph.gexf")` [358] [359].
- GEXF supports dynamic graphs and is well-supported by Gephi. NetworkX tries to preserve attributes and uses `label` attribute for node if present. If your node labels are not strings, they might be stored as `id` and an attribute for label.

- Note: Non-ASCII chars in labels might need encoding parameter. GEXF support requires the `networkx.write_gexf` to handle your data types, otherwise you may need to preprocess attributes into int/float/string.

- **GML:**

- Write: `nx.write_gml(G, "graph.gml")` [360].
- Read: `nx.read_gml("graph.gml")` [360].
- GML is a text format with a Lisp-like syntax. NetworkX's GML writer ensures quotes around strings and so on. One issue: GML only supports one kind of quotes and escaping is minimal, so certain special characters in labels can break format.

- GML is good for simple graphs and widely supported (e.g. igraph, Cytoscape).

- **JSON serialization:** NetworkX doesn't directly dump to JSON with one call (because graph JSON can be represented in multiple ways). But it provides *graph adjacency* and *node-link* formats:

- **Node-Link format:** represents graph as dict with node list and edge list. `nx.node_link_data(G)` converts graph to a serializable dict with keys `"nodes"` (list of nodes with their attributes) and `"links"` (list of edges with their attributes, using source/target keys) [361]. This is suitable for JSON output. Then use `json.dumps()` to get JSON string. `nx.node_link_graph(data)` does the inverse: reconstructs a Graph from that dict [361]. There are convenience wrappers: `nx.write_gpickle` (for Python pickling) we mentioned, but for JSON, one can do:

```
import json
data = nx.node_link_data(G)
with open("graph.json","w") as f: json.dump(data, f)
# Later:
with open("graph.json","r") as f: data = json.load(f)
H = nx.node_link_graph(data)
```

NetworkX has shortcuts: `nx.write_gml` etc., but for JSON specifically, use these functions. The module `networkx.readwrite.json_graph` contains `node_link_data`, `node_link_graph`, and also `adjacency_data` etc.

- **Adjacency format (JSON):** `nx.adjacency_data(G)` returns a dict of lists describing adjacency, and `nx.adjacency_graph(data)` to read it [362]. However, node-link is more commonly used as it's straightforward.
- **Cytoscape JSON:** There's `nx.cytoscape_data(G)` which outputs data in Cytoscape.js compatible format (nodes list and edges list with specific key names) [361], and `nx.cytoscape_graph(data)` to read it [362].

- **Tree Data:** If you have a tree, `nx.tree_data(T, root)` can output a JSON-serializable tree format (root with children recursively) [362], and `nx.tree_graph(data)` reads it.

- **Other formats:**

- **DOT (Graphviz):** *Reading:* if you have Graphviz dot files, NetworkX doesn't parse them directly, but if `pygraphviz` or `pydot` is installed, you can do `A = nx.nx_agraph.read_dot("file.dot")` or `A = nx.nx_pydot.read_dot("file.dot")` which returns an AGraph or pydot Graph. Then `nx.Graph(A)` to convert to NX graph. Or use `nx.drawing.nx_pydot.from_pydot(P)` after loading via pydot (see next section for Graphviz interop). *Writing:* `nx.nx_pydot.to_pydot(G)` gives a pydot object, which you can then `write_raw` to DOT file. NetworkX also provides `nx.write_dot(G, path)` via both pygraphviz and pydot interfaces [363] [364] (it will choose one if available). Essentially, to handle DOT, it's easiest to use the Graphviz interface functions discussed in **Visualization**.
- **Sparse Graph6 format:** `nx.write_graph6(G, "file.g6")` and `nx.read_graph6("file.g6")` for the compact *graph6* or *sparse6* formats (good for very small graphs in research). These are ASCII encodings of graph adjacency. They drop node labels (only structure).
- **Matrix Market:** You can export adjacency matrices via SciPy's I/O to MatrixMarket format (`.mtx`). NetworkX doesn't have direct but you can do:

```
from scipy.io import mmwrite, mmread
A = nx.to_scipy_sparse_array(G)
mmwrite("adjacency.mtx", A)
```

Then later `A = mmread("adjacency.mtx")` and `G = nx.from_scipy_sparse_array(A)`.

- **Pajek (NET format):** `nx.write_pajek(G, "file.net")` and `nx.read_pajek("file.net")` can handle Pajek's .net files [365] [366]. Pajek format supports node and edge attributes somewhat and is used in some social network repositories.
- **LEDA (.gw):** `nx.write_leda(G, "file.gw")` and `nx.read_leda("file.gw")` for the old LEDA graph format [367]. Not commonly used now.
- **Others:** For example, `nx.write_sparse6` (like graph6 but for larger graphs). See `networkx.readwrite` docs for exhaustive list.

Important: When writing to file, ensure to use the correct function for the format and **file mode** (e.g., GraphML is XML text, open in text mode; GEXF and GML also text; pickles in binary mode). The

`nx.write_*` functions handle opening file if given a filename. Many formats **do not preserve** Python-specific data types. GraphML, GEXF, GML, etc., typically store numeric types and strings. If you have a complex object as an attribute, consider serializing it to string (e.g., JSON-dumping a dict attribute into a string attribute). NetworkX's JSON node-link can handle basic Python types via JSON.

Finally, remember to choose a format suitable for your needs: For interoperability with other tools (Gephi, Cytoscape) use GEXF or GraphML. For simplest sharing, edgelist or GML. For full fidelity with Python objects, use GPickle or the node-link JSON approach.

Graph Visualization (Drawing)

NetworkX includes basic drawing capabilities and integration with third-party graph visualization tools. The built-in drawing is mainly via **Matplotlib** and is suitable for small to medium graphs for quick visualization. For high-quality or large-scale visualization, one might use Graphviz or other specialized tools, possibly via the provided interfaces.

- **Matplotlib Drawing:** After `import matplotlib.pyplot as plt`, you can use:
- `nx.draw(G, pos=None, **kwargs)` – Draw the graph *G* on a Matplotlib Axes [368] [369]. If no `pos` (position dict mapping nodes to (x,y) coordinates) is provided, a spring layout is computed by default (Fruchterman-Reingold force-directed) which may not be optimal for very large graphs. Keyword args can control node size, color, labels, etc. For quick usage:

```
nx.draw(G, with_labels=True, node_color='lightblue', edge_color='gray')
plt.show()
```

  This will plot with simple circular layout if small or spring if larger, drawing node labels.
- There are more specific versions: `nx.draw_networkx(G, pos=pos, **kwargs)` [369] [370] which draws nodes, edges, and labels in one go (like `draw` but with `with_labels=True` logic). Or you can manually do `nx.draw_networkx_nodes(G, pos, nodelist=[...], node_size=..., node_color=...)` [371], `nx.draw_networkx_edges(G, pos, edgelist=[...], style='dashed')` [372], and `nx.draw_networkx_labels(G, pos, labels=label_dict)` [373] to have fine control. These functions return Matplotlib artists for further tweaking. For directed graphs, `nx.draw_networkx_edges(..., arrows=True)` will attempt to draw arrowheads (using Matplotlib's ArrowStyle; for better arrowheads, consider using `nx.draw_networkx_edges(..., connectionstyle='arc3, rad=0.1')` for curved edges if needed).
- **Layout functions:** NetworkX provides many layout algorithms to compute `pos` (a dict mapping node->(x,y)). For example:
  - `pos = nx.spring_layout(G, k=None, iterations=50, seed=None)` – Fruchterman-Reingold force-directed layout [374] [375]. Parameter `k` is optimal distance between nodes (defaults to ~1/√n), and `iterations` for the solver. It's the most used layout for general graphs.
  - `nx.kamada_kawai_layout(G)` – another force-based algorithm using spring forces based on graph distances (often yields visually pleasing layouts for smaller graphs) [376].
  - `nx.circular_layout(G)` – places nodes on a circle [377].

- `nx.shell_layout(G, nlist=[list1, list2, ...])` – places nodes in concentric circles (you provide the layering of nodes) [378].
- `nx.spectral_layout(G)` – uses the first two eigenvectors of the Laplacian (good for certain structured graphs) [379].
- `nx.planar_layout(G)` – for planar graphs, finds an embedding (if G is planar) [380].
- `nx.random_layout(G)` – random positions in [0,1] box [381].
- `nx.bipartite_layout(B, nodesU)` – positions bipartite graph with nodesU on left and others on right (vertical alignment) [382].
- `nx.spiral_layout(G)` – nodes on an Archimedean spiral [383].
- `nx.multipartite_layout(G, subset_key='depth')` – for layered DAGs (requires nodes have attribute giving subset id) [384].
- `nx.rescale_layout(pos, scale=1.0)` can rescale positions to fit in a certain range [385] (and `rescale_layout_dict` to get dict form) [386].
- **Third-party**: If you use 3D layouts or geospatial, you might create pos accordingly (NetworkX layout are 2D by default except spectral can do higher dims if `dim` param changed).

- **Matplotlib styling:** All `draw_*` functions accept Matplotlib styling keywords: e.g., `node_size`, `node_color`, `node_shape` (like 'o' for circle, '^' triangle, etc.), `alpha` for transparency, `linewidths`, `font_size`, `font_color`, `edge_color`, `style` ('solid','dashed'), `width` (edge line width). If you pass a list to node_color of the same length as nodelist, it will colormap accordingly. You can also map values via `nx.draw_networkx_nodes(..., cmap=plt.cm.Blues, node_color=[values], vmin=..., vmax=...)`. For edges, there's `edge_cmap`, etc. If drawing parallel edges in MultiGraph, Matplotlib will overlap them by default; you might want to use curved edges (`connectionstyle`) or jitter the positions slightly.

- **Interactive Matplotlib Plotting:** If working in Jupyter notebook, `%matplotlib inline` or `%matplotlib notebook` helps. There's also an experimental widget 3d drawing (not official in NX, but via libraries like `networkx-igraph` or others).

- **Graphviz Integration (via PyGraphviz or pydot):** Graphviz produces high-quality layouts (dot, neato, etc.). NetworkX provides two interfaces:

- **PyGraphviz (nx_agraph):** If `pygraphviz` is installed, you can use `A = nx.nx_agraph.to_agraph(G)` to convert NetworkX graph to a PyGraphviz AGraph (Graphviz graph object) [387] [388]. Then you can call AGraph methods, e.g. `A.layout(prog='dot')` and `A.draw("output.png")` to render. Conversely, `G = nx.nx_agraph.from_agraph(A)` gives back a NetworkX graph [389]. For simple use, `nx.nx_agraph.graphviz_layout(G, prog='dot')` computes positions using Graphviz and returns a pos dict [390] [391]. `prog` can be 'dot' (hierarchical), 'neato' (force-directed), 'twopi' (circular), 'spring', 'fdp', 'circo', etc. This requires pygraphviz installed (which in turn needs Graphviz system library).
  - Example:

```
pos = nx.nx_agraph.graphviz_layout(G, prog='neato')
nx.draw(G, pos=pos)
```

This will use Graphviz's Neato layout but draw via Matplotlib. For small graphs that's fine; for large, one might directly output via Graphviz as below.

- To write DOT files: `nx.nx_agraph.write_dot(G, "file.dot")` [392] [393]. Or use AGraph conversion and A.write.
- Reading DOT: `A = nx.nx_agraph.read_dot("file.dot")` returns an AGraph; do `nx.Graph(A)` to get NetworkX Graph (this loses some Graphviz-specific info but structure and attributes are preserved).

- **Pydot (nx_pydot):** If `pydot` is installed (which uses Graphviz as well), similar functions exist: `P = nx.nx_pydot.to_pydot(G)` [394] [394] returns a pydot `Dot` object; `nx.nx_pydot.from_pydot(P)` to get NX graph [389]. `nx.nx_pydot.graphviz_layout(G, prog='dot')` works analogous to agraph version, but uses pydot internally [391] [395]. Pydot is pure Python and sometimes easier to install.
- **Selecting backend:** If both pygraphviz and pydot are present, either interface can be used. The functions in `nx.drawing.nx_pydot` and `nx.drawing.nx_agraph` are separate but achieve similar ends.
- **Output images via Graphviz:** Instead of Matplotlib, one can let Graphviz output directly. For example, with PyGraphviz:

```
A = nx.nx_agraph.to_agraph(G)
A.layout('dot')
A.draw('graph.png')
```

This will produce a PNG using Graphviz's dot layout. Or `A.draw('graph.svg')` for SVG.

- **Graphviz from Python without external libs:** If neither pygraphviz nor pydot is installed, you can still produce a DOT file via `nx.write_dot` (if pydotplus is there or via a fallback) and then call Graphviz from command line. But it's simplest to install one of the two libraries for full integration.

- **3D and Interactive Graph Drawing:** NetworkX doesn't natively do 3D, but you can use the positions from `nx.spring_layout(G, dim=3)` to get 3D coords and then plot via other tools (matplotlib's 3D axes or plotly). For interactive visualization (like in notebooks or web apps), one can convert NetworkX graphs to other libraries: e.g., to **Bokeh** using `from_networkx` function (in Bokeh's networkx integration), or to **Plotly** by extracting edge traces. There's also `pyvis` library which takes a NetworkX graph and shows it in the browser with physics (Vis.js). Or use `nx.readwrite.json_graph` to get node-link JSON and feed into D3.js.

- **TikZ/LaTeX Export:** NetworkX 3.6 introduced LaTeX export for graphs using the TikZ library. `nx.to_latex(G, caption="Caption", label="Graph")` returns a LaTeX string for a figure with the graph drawn using the TikZ **graphdrawing** library [396] [397]. It will include `\begin{figure}` environment, `\begin{tikzpicture}` with nodes and edges placed. Positions can be computed via Graphviz algorithms if you have `pygraphviz` or use spring by default. `nx.write_latex(G, "graph.tex", as_document=False, **kwargs)` writes it to file [398]. If `as_document=True`, it outputs a standalone LaTeX document. One can also produce subfigures for multiple graphs using a list. This feature requires a LaTeX installation with TikZ and possibly the `graphdrawing` TikZ library for advanced layout (the docs mention intentions to interface TikZ's layout in the future). Options: you can supply node attributes `pos` as LaTeX coordinates or styling

options via node attributes, which `to_latex` will incorporate [399]. For example, if a node has `G.nodes[n]['style'] = 'red, draw=black'`, that will be applied. Similarly, edge attributes can carry TikZ options or labels. The LaTeX output is advanced usage but useful for embedding graphs in academic papers with full control.

Graph Drawing Performance: Matplotlib drawing is fine up to a few thousand nodes, but beyond that it can be slow or cluttered. Graphviz can handle tens of thousands of nodes but layout might be slow (neato or sfdp are better for large). For interactive exploration of large networks, one might use external tools (Gephi, Cytoscape, etc.) by exporting via GEXF/GraphML.

Performance and Backend Support

NetworkX 3.6 introduces a **plugin architecture for dispatching certain algorithms to faster backend implementations** [400] [401]. This means you can leverage external high-performance graph libraries (like *GraphBLAS*, *CuGraph*, *NetworkX-Primitives for parallel*, *Rapids*, etc.) without changing your NetworkX code, by **configuring a backend**.

What are Dispatchable Functions? Many NetworkX functions (especially in algorithms and generators) are decorated with `@nx._dispatch` internally. Such a function will, when called, check if a backend is set and if that backend provides its own implementation of that function. For example, `nx.pagerank` is dispatchable – if you have configured a backend that has a specialized PageRank (like CuGraph for GPU), NetworkX will call that instead of its pure Python version [402] [403].

Available Backends: Backends are Python packages that register entry points `networkx.backends` and implement certain API. As of 3.6, known backends include: - **GraphBLAS** (`networkx-graphblas` package): uses sparse linear algebra (GraphBLAS via suitesparse or similar) to speed up algorithms like centrality, BFS, etc. Especially good for very large sparse graphs (thousands to millions of nodes) on CPU. - **CuGraph** (`cugraph` from RAPIDS.ai): GPU-accelerated algorithms for common analytics (PageRank, BFS, SSSP, etc.). Requires Nvidia GPU and RAPIDS stack. - **Parallel** (`nx-parallel`): a hypothetical backend for multi-core parallel execution of algorithms (e.g., parallel betweenness). It might use joblib or multiprocessing under the hood. NetworkX mentions a "parallel" backend with get_chunks parameter for betweenness [404]. - **Neptune** (`nx-neptune`): Possibly a backend hooking into some distributed graph system or an AWS Neptune DB. The config snippet shows a “neptune” backend with many parameters (likely for a graph database) [405].

NetworkX does not bundle these; you install them separately. Once installed, they declare to NetworkX what functions they implement.

Using a Backend Explicitly: You can request a specific backend on a per-call basis by passing the `backend` keyword:

```
import networkx as nx
val = nx.betweenness_centrality(G, k=10, backend="parallel")
```

This will attempt to use the “parallel” backend’s betweenness if available [406] [407]. Or:

```
H = nx.cugraph.DiGraph(G)  # create a CuGraph graph from NetworkX graph
pr = nx.pagerank(H)  # CuGraph backend will compute this on GPU
```

Alternatively, you can explicitly create a graph of a backend type: e.g. `G = nx.Graph(backend="graphblas")` to get a Graph implemented by the backend (if supported) [408] [409]. Passing a backend graph into a dispatchable function will automatically trigger that backend's function if available [410] [411].

Automatic Backend Dispatch: Instead of specifying each call, you can configure NetworkX to always try certain backends first. This is done via the global config:

```
nx.config.backend_priority = ["graphblas", "cugraph"]
```

This list sets a priority order for **algorithm functions** [412]. With this, whenever you call a dispatchable algorithm (like `nx.pagerank` or `nx.connected_components` etc.), NetworkX will check each backend in the list to see if it implements that function [412] [413]. The first backend that can handle it will be used. If none can, it falls back to NetworkX's implementation [414] [411]. Similarly, there is `nx.config.backend_priority.generators` for dispatching **graph creation** functions that return a graph [415]. For example, `nx.empty_graph(n)` could directly create a backend graph without first making a NetworkX graph then converting. And `nx.config.backend_priority.classes` to have `nx.Graph()` actually return a backend graph class instead of NetworkX Graph [416] [417].

By default, these priorities are empty (so no auto-dispatch). You can set environment variables for convenience: - `NETWORKX_BACKEND_PRIORITY="graphblas,cugraph"` (comma-separated) sets algorithm priority [412]. - `NETWORKX_BACKEND_PRIORITY_GENERATORS="graphblas"` for generator functions [415]. - `NETWORKX_BACKEND_PRIORITY_CLASSES="graphblas"` so that `nx.Graph()` yields a GraphBLAS-based graph type [418]. - `NETWORKX_FALLBACK_TO_NX=True` to allow mixing backends (explained below) [419] [420].

Graph Conversion and Caching: When dispatching, if the input is a NetworkX graph and backend needs its own type, NetworkX will **convert** the graph. For instance, calling `nx.pagerank(G)` with backend "cugraph" will under the hood do `G_conv = cugraph.Graph(G)`, compute on `G_conv`, then possibly convert result back [421]. To avoid repeated conversions, NetworkX caches the converted graph by storing it in `G.__networkx_cache__` [422]. Subsequent calls on the same original graph can reuse the cached backend graph (assuming it wasn't mutated in a way that breaks cache). The config `nx.config.cache_converted_graphs` (default True) controls this caching [422]. If you manually mutate a graph, you should clear its cache via `G.__networkx_cache__.clear()` to be safe (NetworkX will auto-clear on structural changes done via its API, like add_edge, but direct dict access could bypass it) [423]. Caching uses more memory but saves time on multiple calls.

Mixing Workflows: If you obtain a backend graph (say a CuGraph Graph) as a result of a generator or by directly using backend API, you can pass it to NetworkX algorithms. If that algorithm isn't implemented in backend, by default it will error (because it doesn't know how to handle that type). You can allow a *fallback to NetworkX* by setting `nx.config.fallback_to_nx = True` [424] [425]. Then if a backend graph is passed

to a function not supported by backend, NetworkX will auto-convert it back to a NetworkX Graph and use its own implementation [419] [426]. This adds overhead of conversion but permits seamless use of combinations. If the backend's graph class is designed to duck-type NetworkX (i.e., has same methods), the backend might choose to just run NX code on it without converting.

Selecting/Inspecting Backends: To see which backend was actually used in a call, you can enable logging. The NetworkX dispatcher logs at level INFO/DEBUG on logger `"networkx"` about backend selection [427] [428]. For example:

```
import logging; logging.getLogger('networkx').setLevel(logging.INFO)
logging.basicConfig()
nx.pagerank(G)  # will log which backend ran or if fell back to NX
```

This can help verify that, say, CuGraph was indeed used (and not silently fell back due to an unsupported parameter).

Backend Implementation Note: For developers, to make a new backend, one defines Python functions with same names and signatures as NetworkX functions and registers them via entry_points. They should ideally handle the backend's graph type and possibly allow conversion from NetworkX graph. Backends also can provide `can_run` and `should_run` methods to tell NetworkX if they can or prefer to handle certain inputs (e.g., maybe skip small graphs as overhead might not pay off) [429].

Example – using CuGraph backend:

```
import cugraph
nx.config.backend_priority = ["cugraph"]  # prefer CuGraph for algorithms
Gnx = nx.fast_gnp_random_graph(1000, 0.01)
pr = nx.pagerank(Gnx)  # this will dispatch to cugraph.pagerank
```

Here, `nx.fast_gnp_random_graph` might return a NetworkX Graph (since generator dispatch can also be configured), but pagerank will notice `backend_priority` and use CuGraph. CuGraph will convert the NetworkX Graph to a CuGraph Graph (on GPU) and compute quickly, then return a dict of ranks. If `fallback_to_nx=True` and you called something not in CuGraph (like `nx.communicability_centrality(Gnx)`), it would detect CuGraph doesn't implement it and run NetworkX's version (possibly converting a CuGraph graph to NX if you passed one).

Important Config Options: Summarizing, in code or env: - `nx.config.backend_priority` (or env `NETWORKX_BACKEND_PRIORITY`) – list of backend names for algorithms [412]. - `nx.config.backend_priority.generators` (`NETWORKX_BACKEND_PRIORITY_GENERATORS`) – list for generator functions (that return graphs) [415]. - `nx.config.backend_priority.classes` (`NETWORKX_BACKEND_PRIORITY_CLASSES`) – list for graph constructors [416] [417]. - `nx.config.fallback_to_nx` (`NETWORKX_FALLBACK_TO_NX`) – bool, default False. If True, allow converting backend graphs to NX if needed [424] [425]. - `nx.config.cache_converted_graphs` (`NETWORKX_CACHE_CONVERTED_GRAPHS`) – bool, default True. Controls caching of conversions [422].

Using backends can dramatically speed up computations: e.g., GraphBLAS backend can solve centrality via matrix ops in Python with order-of-magnitude speedups, CuGraph can do PageRank or BFS on millions of nodes in seconds on a GPU.

Caveats: Not every NetworkX function is dispatchable – primarily the commonly used ones (graph creation like `empty_graph`, most algorithms like shortest path, components, centrality, etc.). Check NetworkX documentation or source: such functions have `@nx._dispatch` decorator in code. Also, ensure that backends produce correct results; for very large graphs, slight float differences or needing to set `nx.config.fallback_to_nx=True` for unsupported scenarios might be necessary.

Random Number Generation and Seeding

Randomness in NetworkX appears in random graph generators, random walks, random layout initializations, etc. To ensure **reproducible results**, it's important to control random seeds for both Python's random module and NumPy's random module because NetworkX may use either. By default, NetworkX will use whichever RNG is natural for the task (often Python's `random` for pure Python algorithms, or NumPy's `numpy.random` for matrix-based ones) [430] [431]. Both use Mersenne Twister under the hood, but they are separate instances.

In general, **to get reproducible pseudorandom results, set the seed**:

```
import random, numpy as np
random.seed(1234)
np.random.seed(1234)
```

This ensures both global RNGs are seeded [432] [433]. For most use, you can alternatively use the NetworkX seed mechanism:

Many NetworkX functions accept a `seed` parameter (commonly in generators and algorithms that use randomness). The `seed` parameter is flexible [434] [435]: - If `seed=None` (default), the function uses the global RNG state as-is (so results will differ each run unless you globally fixed it) [436] [435]. - If `seed` is an integer, NetworkX will **create a local random number generator** with that seed just for this function call [436] [435]. That ensures the outcome is the same each time for that seed, and it does not advance the global RNG state. Under the hood, it might use `random.Random(seed)` for functions originally using Python's random, or `numpy.random.default_rng(seed)` for those using NumPy (NetworkX chooses one consistent with the function's design). - If `seed` is `numpy.random` (the module), NetworkX will force using NumPy's global RNG for even algorithms that normally use `random` [437] [438]. Conversely, if `seed` is `random` (the module), it might try to use Python's global RNG. However, **not all combinations are guaranteed**; the primary design is to allow always using NumPy's RNG if desired (since NumPy's RNG has advantages like independent streams). - If `seed` is a `numpy.random.RandomState` or a new `numpy.random.Generator` instance, NetworkX will use that as the source of randomness [437] [438]. This allows you to manage state externally or use NumPy's new Generator with a specific bit generator.

This design allows fine control: for instance, you might want all random processes in your project to use a single `numpy.random.Generator` for compatibility with other code. You could do:

```
rng = np.random.default_rng(42)
G = nx.gnp_random_graph(100, 0.1, seed=rng)  # uses rng for randomness
```

Now all randomness (edge presence decisions in this E-R graph) come from that `rng`, and you can continue to use `rng` for other random needs, keeping one stream [439] [440].

If you pass an older `numpy.random.RandomState` object, it will be used similarly (though `Generator` is recommended for new code). If you pass an instance of Python's `random.Random` (with its own state), it's *accepted* for some functions, but caution: the documentation notes they can't guarantee full support of a `random.Random` because NumPy's RNG has features that Python's doesn't (like drawing from distributions efficiently) [441]. In practice, stick to either int seeds or NumPy Generator/RandomState for advanced control.

Summary of best practices: - For *simple reproducibility*, pass an `int` as seed to each function or set both global seeds at the top of your script. - For *parallel or multiple experiments*, you can create separate RNG objects to avoid interfering state. E.g., `nx.gnp_random_graph(n, p, seed=42)` in one experiment and `seed=43` in another ensures independent graphs. If using one global, calling `random.seed(42)` once is fine but then any other usage of `random` will advance it. - NetworkX ensures that if you provide a seed, results are deterministic across runs and also *across RNG implementations*. For example, `nx.fast_gnp_random_graph` uses Python's `random.random()` by default, but if you do `seed=np.random`, it will use NumPy under the hood to generate uniform randoms - NetworkX tries to produce identical sequence given the same seed even if using numpy vs random (though this is subtle and mostly, if you want reproducibility, just consistently use one approach).

Seeding example:

```
# Reproducible random graphs example
G1 = nx.gnp_random_graph(10, 0.3, seed=42)
G2 = nx.gnp_random_graph(10, 0.3, seed=42)
assert nx.is_isomorphic(G1, G2) and G1.number_of_edges()==G2.number_of_edges()
```

Both G1 and G2 will be identical because the same seed was used. If no seed, they'd differ.

Mixing RNG types: If a function strictly uses one RNG (say Python's), and you pass a NumPy Generator, NetworkX has a compatibility layer: it can generate random numbers from the provided RNG even if the algorithm expects the other. For instance, if an algorithm uses `random.random()` in a loop, and you pass `seed=np.random`, NetworkX will create a local RandomState and draw from it for each needed random number, ensuring distribution matches. The docs hint that using a `random.Random` for a NumPy-based function might not be fully supported due to missing methods (so better avoid that direction) [441].

Global Impact: If you rely on NetworkX's global RNG usage, be aware that some parts use `random` (e.g., `nx.gnp_random_graph` if seed is None) and some use `numpy.random` (e.g., maybe `nx.random_layout` uses numpy to generate coordinates). The safer route for reproducibility is always to pass a seed or set both random seeds globally.

One RNG for all: If you prefer NumPy's RNG always, you can do global:

```
nx.utils.create_py_random_state = nx.utils.create_random_state  # not typical
usage
```

Actually, better is: pass `seed=np.random` for everything, which instructs to use numpy's global RNG for all that support it [436] [435]. Or create a `Generator` and consistently pass it.

In summary, **NetworkX's seed logic ensures that whether an algorithm was implemented with `random` or `numpy.random`, you can force it to use a particular source** [434] [442]. The default global approach, however, means you might need to seed both for full control.

Exceptions in NetworkX

NetworkX defines custom exception classes for various error conditions, all inheriting from `NetworkXException` [443]. Understanding these can help with error handling:

- **NetworkXException:** Base class for all NetworkX-specific exceptions [443] [444]. A catch-all for "NetworkX went wrong" that isn't a standard Python error.

- **NetworkXError:** General exception for serious errors in NetworkX (often input errors) [445]. For example, if you call `nx.shortest_path(G, u, v)` on an unconnected graph and no path exists, it might raise `NetworkXNoPath` (a more specific one). But if you pass nodes that don't exist in G, it will raise `NetworkXError` saying "Node X not in graph".

- **NetworkXPointlessConcept:** Raised when an operation is not defined because the concept is "pointless" on the given input [446] [447]. Specifically, NetworkX uses this if you try to analyze a null graph (0 nodes) in a context that doesn't make sense. For example, asking for the degree distribution of a null graph – there's no sensible answer. The name comes from a paper titled "Is the Null Graph a Pointless Concept?" [448]. This is rarely encountered unless dealing with 0-node graphs.

- **NetworkXAlgorithmError:** Used when an algorithm encounters an unexpected condition that prevents it from proceeding [449]. E.g., if a connectivity algorithm assumed something and found it violated mid-run. It's a general "algorithm failed" error, not common in normal usage (more an internal issue usually).

- **NetworkXUnfeasible:** Means no feasible solution exists for the algorithm's problem [259] [259]. For example, `nx.topological_sort(D)` will raise `NetworkXUnfeasible` if the digraph D has a cycle (so a topological order is impossible) [450]. Another example: bipartite matching if something is unsolvable with given constraints, etc.

- **NetworkXNoPath:** Raised when a path is required but none exists [451] [452]. For instance, `nx.shortest_path(G, u, v)` if u and v are disconnected raises `NetworkXNoPath` with a message "No path between u and v." Similarly used in flow if asking for path in residual and none

found (but in flows they usually return 0 flow rather than raise). Also used by functions like `nx.bidirectional_dijkstra` or connectivity when target not reachable.

- **NetworkXNoCycle:** Similarly, raised when an algorithm expects to find a cycle but cannot [453] [454]. For example, `nx.find_cycle(G)` on an acyclic graph raises `NetworkXNoCycle` [455]. Or if you request an Eulerian cycle on a graph that isn't Eulerian.

- **NodeNotFound:** Raised when a specified node isn't in the graph [456] [457]. Many functions will raise this instead of KeyError. For example, `G.nodes[5]` (NodeView) would KeyError, but `nx.shortest_path(G, 1, 42)` if 42 not in G raises NodeNotFound. Check error message to see which node was not found. It inherits NetworkXError likely.

- **HasACycle:** Raised when an algorithm that requires an acyclic graph encounters a cycle [450] [458]. E.g., `nx.dag_longest_path` if graph is not acyclic might raise this or `NetworkXUnfeasible`. Actually, `nx.is_directed_acyclic_graph` returns False, but some algorithms like `nx.topological_sort` raise `NetworkXUnfeasible` for cycle. `HasACycle` is used in algorithms like DAG checking if expecting no cycle but found one mid-process (like a certificate of cycle presence). Specifically, it's mentioned to be raised e.g. by algorithms for feedback arc sets or others expecting DAG.

- **NetworkXUnbounded:** Raised when a problem is unbounded – typically in flow or optimization contexts [294] [459]. For instance, in a max flow problem if there's an infinite capacity path, `nx.maximum_flow` might raise this, or in a min-cost flow if a negative cycle exists and no constraints, cost is unbounded (the code snippet from a test shows `NetworkXUnbounded: Negative cycle detected` in Bellman-Ford context [295]). So, `NetworkXUnbounded` usually implies some algorithm like Bellman-Ford found a negative cycle (unbounded negative cost path) [295] or a max flow had no capacity limits leading to infinite flow (which in a real network context won't happen unless you allow inf capacity edges).

- **NetworkXNotImplemented:** Raised when an algorithm is not applicable to the graph type or not implemented for that type [460] [461]. For example, trying to run a bipartite-specific algorithm on a non-bipartite graph might raise this. Or some algorithms might be only for directed graphs – calling on an undirected might raise NotImplemented. Also if a function stub exists that hasn't been implemented. This is often used with the `@not_implemented_for` decorator in NetworkX, which checks graph type and raises accordingly (e.g., some flow algorithms require directed graph and raise if undirected is passed).

- **AmbiguousSolution:** Raised when an algorithm finds more than one equally valid result where it expected a unique solution [296] [462]. The example given in docs: determining bipartite sets in a disconnected bipartite graph – there's an ambiguity of which side to put one component's nodes (since flipping all sides in a component is also a valid bipartition) [463]. NetworkX's `nx.bipartite.sets` will raise AmbiguousSolution if the graph is disconnected (and you didn't specify which node is in which set) [464]. It basically says "I refuse to guess in an ambiguous scenario." You need to provide additional info or pick one arbitrarily yourself.

- **ExceededMaxIterations:** Raised when an iterative algorithm exceeds a preset iteration limit without converging [465] [466]. For example, power iteration methods for eigenvector centrality or hitting time might not converge to tolerance in given iterations. PageRank will raise this if `max_iter` is reached without reaching error < tol [467]. Also, some approximation algorithms may raise if they loop too long. The exception message often includes how many iterations ran. You can catch this and perhaps try again with higher max_iter or adjust parameters.

- **PowerIterationFailedConvergence:** A subclass of ExceededMaxIterations specifically for power-iteration eigenvector algorithms (e.g. eigenvector_centrality) [264] [468]. It includes the attribute `num_iterations` so you can see how many iterations were done [264]. You might handle this by using a different algorithm or increasing iterations. For eigenvector centrality, this means maybe the graph is not connected or has some issue (if graph is directed not strongly connected, eigenvector might not converge).

When writing robust code with NetworkX, you may want to catch these exceptions. For example:

```
try:
    path = nx.shortest_path(G, u, v)
except nx.NetworkXNoPath:
    print("No path between %s and %s" % (u, v))
```

Or catching `NetworkXUnfeasible` for DAG routines to handle cycles gracefully. Generally, it's better to catch specific exceptions than `NetworkXException` unless you want to handle all.

Finally, note that exceptions like NodeNotFound or NetworkXError might indicate usage issues (like wrong input), whereas NetworkXNoPath or NoCycle indicate a legitimate result of analysis (path not exists, etc.). Use these to differentiate error vs a result like "not found".

References: The exceptions are documented in the reference [469] [259] [459]. For example, the note on `NetworkXPointlessConcept` clarifies null vs empty graph difference [448], on `AmbiguousSolution` clarifies bipartite confusion [296], etc. These help to understand why some error arises.

This concludes the comprehensive overview of NetworkX 3.6 features, organized for quick lookup by expert developers and AI agents. With this reference, one can utilize NetworkX's full power – from graph construction and algorithms to advanced backends and configuration – to analyze complex networks effectively. [1] [465] (References and documentation for deeper details on specific functions and behaviors.)

---

[1] [2] [3] [6] [7] [8] [9] [37] [38] Introduction — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/introduction.html

[4] [5] [19] [20] [25] [26] [27] [28] [29] [30] [31] [33] [36] Graph types — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/classes/index.html

10 11 12 13 14 15 16 17 18 Graph—Undirected graphs with self loops — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/classes/graph.html

21 22 23 24 32 34 35 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 Functions — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/functions.html

70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 Operators — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/algorithms/operators.html

93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 Graph generators — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/generators.html

211 212 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 Linear algebra — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/linalg.html

213 214 215 216 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 Converting to and from other data formats — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/convert.html

247 Migration guide from 2.X to 3.0 — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/release/migration_guide_from_2.x_to_3.0.html

248 Index — NetworkX 3.6 documentation
https://networkx.org/documentation/stable/genindex.html

249 Components — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/algorithms/component.html

250 biconnected_component_edges — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.components.biconnected_component_edges.html

251 252 Connectivity — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/algorithms/connectivity.html

253 build_auxiliary_node_connectivity — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.connectivity.utils.build_auxiliary_node_connectivity.html

254 average_node_connectivity — NetworkX 3.6 documentation
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.connectivity.connectivity.average_node_connectivity.html

255 256 257 280 281 282 283 284 285 286 287 288 289 290 291 292 293 Flows — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/algorithms/flow.html

258 Reference — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/index.html

259 264 275 294 296 443 444 445 446 447 448 449 450 451 452 453 454 456 457 458 459 460 461 462 463 464 465 466 468 469 Exceptions — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/exceptions.html

260 274 Centrality — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/algorithms/centrality.html

261 closeness_centrality — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.closeness_centrality.html

262 closeness_centrality — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.bipartite.centrality.closeness_centrality.html

263 400 401 402 403 404 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 424 425 426 427 428 429 Backends — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/backends.html

265 467 pagerank — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.link_analysis.pagerank_alg.pagerank.html

266 Link Analysis — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/algorithms/link_analysis.html

267 278 279 297 Algorithms — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/algorithms/index.html

268 current_flow_closeness_centrality — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.current_flow_closeness_centrality.html

269 communicability — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.communicability_alg.communicability.html

270 273 communicability_betweenness_c...
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.communicability_betweenness_centrality.html

271 272 communicability_centrality — NetworkX 1.9 documentation
https://networkx.org/documentation/networkx-1.9/reference/generated/networkx.algorithms.centrality.communicability_centrality.html

276 Link Prediction — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/algorithms/link_prediction.html

277 jaccard_coefficient — NetworkX 3.6 documentation
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.link_prediction.jaccard_coefficient.html

295 goldberg_radzik — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.shortest_paths.weighted.goldberg_radzik.html

298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 316 317 318 319 320 321 322 323 Communities — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/algorithms/community.html

315 NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/release/release_3.6.1.html

324 communicability_exp — NetworkX 3.5 documentation
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.communicability_alg.communicability_exp.html

349 350 Relabeling nodes — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/relabel.html

351 352 353 354 355 356 357 358 359 360 361 362 365 366 367 Reading and writing graphs — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/readwrite/index.html

363 364 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 394 395 396 397 398 399 Drawing — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/drawing.html

392 393 DOT — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/readwrite/dot.html

405 422 423 Configs — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/configs.html

430 431 432 433 434 435 436 437 438 439 440 441 442 Randomness — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/randomness.html

455 find_cycle — NetworkX 3.6.1 documentation
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.cycles.find_cycle.html