Metodi NetworkX

GRAFO SEMPLICE → nx.Graph()

Reporting nodes edges and neighbors

`Graph.nodes`	A NodeView of the Graph as G.nodes or G.nodes().
`Graph.__iter__`()	Iterate over the nodes.
`Graph.has_node`(n)	Returns True if the graph contains the node n.
`Graph.__contains__`(n)	Returns True if n is a node, False otherwise.
`Graph.edges`	An EdgeView of the Graph as G.edges or G.edges().
`Graph.has_edge`(u, v)	Returns True if the edge (u, v) is in the graph.
`Graph.get_edge_data`(u, v[, default])	Returns the attribute dictionary associated with edge (u, v).
`Graph.neighbors`(n)	Returns an iterator over all neighbors of node n.
`Graph.adj`	Graph adjacency object holding the neighbors of each node.
`Graph.__getitem__`(n)	Returns a dict of neighbors of node n.
`Graph.adjacency`()	Returns an iterator over (node, adjacency dict) tuples for all nodes.
`Graph.nbunch_iter`([nbunch])	Returns an iterator over nodes contained in nbunch that are also in the graph.

Counting nodes edges and neighbors

`Graph.order`()	Returns the number of nodes in the graph.
`Graph.number_of_nodes`()	Returns the number of nodes in the graph.
`Graph.__len__`()	Returns the number of nodes in the graph.
`Graph.degree`	A DegreeView for the Graph as G.degree or G.degree().
`Graph.size`([weight])	Returns the number of edges or total of all edge weights.
`Graph.number_of_edges`([u, v])	Returns the number of edges between two nodes.

GRAFO ORIENTATO → nx.DiGraph()

Adding and removing nodes and edges

DiGraph.__init__([incoming_graph_data])	Initialize a graph with edges, name, or graph attributes.
DiGraph.add_node(node_for_adding, **attr)	Add a single node `node_for_adding` and update node attributes.
DiGraph.add_nodes_from(nodes_for_adding, **attr)	Add multiple nodes.
DiGraph.remove_node(n)	Remove node n.
DiGraph.remove_nodes_from(nodes)	Remove multiple nodes.
DiGraph.add_edge(u_of_edge, v_of_edge, **attr)	Add an edge between u and v.
DiGraph.add_edges_from(ebunch_to_add, **attr)	Add all the edges in ebunch_to_add.
DiGraph.add_weighted_edges_from(ebunch_to_add)	Add weighted edges in `ebunch_to_add` with specified weight attr
DiGraph.remove_edge(u, v)	Remove the edge between u and v.
DiGraph.remove_edges_from(ebunch)	Remove all edges specified in ebunch.
DiGraph.update([edges, nodes])	Update the graph using nodes/edges/graphs as input.
DiGraph.clear()	Remove all nodes and edges from the graph.
DiGraph.clear_edges()	Remove all edges from the graph without altering nodes.

Reporting nodes edges and neighbors

DiGraph.nodes	A NodeView of the Graph as G.nodes or G.nodes().
DiGraph.__iter__()	Iterate over the nodes.
DiGraph.has_node(n)	Returns True if the graph contains the node n.
DiGraph.__contains__(n)	Returns True if n is a node, False otherwise.
DiGraph.edges	An OutEdgeView of the DiGraph as G.edges or G.edges().
DiGraph.out_edges	An OutEdgeView of the DiGraph as G.edges or G.edges().
DiGraph.in_edges	A view of the in edges of the graph as G.in_edges or G.in_edges().
DiGraph.has_edge(u, v)	Returns True if the edge (u, v) is in the graph.
DiGraph.get_edge_data(u, v[, default])	Returns the attribute dictionary associated with edge (u, v).

DiGraph.neighbors(n)	Returns an iterator over successor nodes of n.
DiGraph.adj	Graph adjacency object holding the neighbors of each node.
DiGraph.__getitem__(n)	Returns a dict of neighbors of node n.
DiGraph.successors(n)	Returns an iterator over successor nodes of n.
DiGraph.succ	Graph adjacency object holding the successors of each node.
DiGraph.predecessors(n)	Returns an iterator over predecessor nodes of n.
DiGraph.pred	Graph adjacency object holding the predecessors of each node.
DiGraph.adjacency()	Returns an iterator over (node, adjacency dict) tuples for all nodes.
DiGraph.nbunch_iter([nbunch])	Returns an iterator over nodes contained in nbunch that are also in the graph.

Counting nodes edges and neighbors

DiGraph.order()	Returns the number of nodes in the graph.
DiGraph.number_of_nodes()	Returns the number of nodes in the graph.
DiGraph.__len__()	Returns the number of nodes in the graph.
DiGraph.degree	A DegreeView for the Graph as G.degree or G.degree().
DiGraph.in_degree	An InDegreeView for (node, in_degree) or in_degree for single node.
DiGraph.out_degree	An OutDegreeView for (node, out_degree)
DiGraph.size([weight])	Returns the number of edges or total of all edge weights.
DiGraph.number_of_edges([u, v])	Returns the number of edges between two nodes.

METODI GRAFI COMPONENTI CONNESSE, FORTEMENTE E DEBOLMENTE

Connectivity

`is_connected`(G)	Returns True if the graph is connected, False otherwise.
`number_connected_components`(G)	Returns the number of connected components.
`connected_components`(G)	Generate connected components.
`node_connected_component`(G, n)	Returns the set of nodes in the component of graph containing node n.

Strong connectivity

`is_strongly_connected`(G)	Test directed graph for strong connectivity.
`number_strongly_connected_components`(G)	Returns number of strongly connected components in graph.
`strongly_connected_components`(G)	Generate nodes in strongly connected components of graph.
`kosaraju_strongly_connected_components`(G[, ...])	Generate nodes in strongly connected components of graph.
`condensation`(G[, scc])	Returns the condensation of G.

Weak connectivity

`is_weakly_connected`(G)	Test directed graph for weak connectivity.
`number_weakly_connected_components`(G)	Returns the number of weakly connected components in G.
`weakly_connected_components`(G)	Generate weakly connected components of G.

RICERCHE CAMMINI DFS, BFS

Depth First Search

Basic algorithms for depth-first searching the nodes of a graph.

`dfs_edges`(G[, source, depth_limit, ...])	Iterate over edges in a depth-first-search (DFS).
`dfs_tree`(G[, source, depth_limit, ...])	Returns oriented tree constructed from a depth-first-search from source.
`dfs_predecessors`(G[, source, depth_limit, ...])	Returns dictionary of predecessors in depth-first-search from source.
`dfs_successors`(G[, source, depth_limit, ...])	Returns dictionary of successors in depth-first-search from source.
`dfs_preorder_nodes`(G[, source, depth_limit, ...])	Generate nodes in a depth-first-search pre-ordering starting at source.
`dfs_postorder_nodes`(G[, source, ...])	Generate nodes in a depth-first-search post-ordering starting at source.
`dfs_labeled_edges`(G[, source, depth_limit, ...])	Iterate over edges in a depth-first-search (DFS) labeled by type.

Breadth First Search

Basic algorithms for breadth-first searching the nodes of a graph.

`bfs_edges`(G, source[, reverse, depth_limit, ...])	Iterate over edges in a breadth-first-search starting at source.
`bfs_layers`(G, sources)	Returns an iterator of all the layers in breadth-first search traversal.
`bfs_tree`(G, source[, reverse, depth_limit, ...])	Returns an oriented tree constructed from of a breadth-first-search starting at source.
`bfs_predecessors`(G, source[, depth_limit, ...])	Returns an iterator of predecessors in breadth-first-search from source.
`bfs_successors`(G, source[, depth_limit, ...])	Returns an iterator of successors in breadth-first-search from source.
`descendants_at_distance`(G, source, distance)	Returns all nodes at a fixed `distance` from `source` in `G`.
`generic_bfs_edges`(G, source[, neighbors, ...])	Iterate over edges in a breadth-first search.

METODI PER CICLI SU GRAFI

Cycles

`cycle_basis` (G[, root])	Returns a list of cycles which form a basis for cycles of G.
`simple_cycles` (G[, length_bound])	Find simple cycles (elementary circuits) of a graph.
`recursive_simple_cycles` (G)	Find simple cycles (elementary circuits) of a directed graph.
`find_cycle` (G[, source, orientation])	Returns a cycle found via depth-first traversal.
`minimum_cycle_basis` (G[, weight])	Returns a minimum weight cycle basis for G
`chordless_cycles` (G[, length_bound])	Find simple chordless cycles of a graph.
`girth` (G)	Returns the girth of the graph.

`dag_longest_path` (G[, weight, ...])	Returns the longest path in a directed acyclic graph (DAG).
`dag_longest_path_length` (G[, weight, ...])	Returns the longest path length in a DAG

CAMMINI EURELIANI

`is_eulerian` (G)	Returns True if and only if `G` is Eulerian.
`eulerian_circuit` (G[, source, keys])	Returns an iterator over the edges of an Eulerian circuit in `G`.
`eulerize` (G)	Transforms a graph into an Eulerian graph.
`is_semieulerian` (G)	Return True iff `G` is semi-Eulerian.
`has_eulerian_path` (G[, source])	Return True iff `G` has an Eulerian path.
`eulerian_path` (G[, source, keys])	Return an iterator over the edges of an Eulerian path in `G`.

OPERATORI SU GRAFI

Operators

Unary operations on graphs

`complement` (G)	Returns the graph complement of G.
`reverse` (G[, copy])	Returns the reverse directed graph of G.

Operations on graphs including union, intersection, difference.

`compose` (G, H)	Compose graph G with H by combining nodes and edges into a single graph.
`union` (G, H[, rename])	Combine graphs G and H.
`disjoint_union` (G, H)	Combine graphs G and H.
`intersection` (G, H)	Returns a new graph that contains only the nodes and the edges that exist in both G and H.
`difference` (G, H)	Returns a new graph that contains the edges that exist in G but not in H.
`symmetric_difference` (G, H)	Returns new graph with edges that exist in either G or H but not both.
`full_join` (G, H[, rename])	Returns the full join of graphs G and H.

Operations on many graphs.

`compose_all` (graphs)	Returns the composition of all graphs.
`union_all` (graphs[, rename])	Returns the union of all graphs.
`disjoint_union_all` (graphs)	Returns the disjoint union of all graphs.
`intersection_all` (graphs)	Returns a new graph that contains only the nodes and the edges that exist in all graphs.

CAMMINI MINIMI

Shortest Paths

Compute the shortest paths and path lengths between nodes in the graph.

These algorithms work with undirected and directed graphs.

`shortest_path` (G[, source, target, weight, ...])	Compute shortest paths in the graph.
`all_shortest_paths` (G, source, target[, ...])	Compute all shortest simple paths in the graph.
`all_pairs_all_shortest_paths` (G[, weight, method])	Compute all shortest paths between all nodes.
`single_source_all_shortest_paths` (G, source)	Compute all shortest simple paths from the given source in the graph.
`shortest_path_length` (G[, source, target, ...])	Compute shortest path lengths in the graph.
`average_shortest_path_length` (G[, weight, method])	Returns the average shortest path length.
`has_path` (G, source, target)	Returns *True* if *G* has a path from *source* to *target*.

Shortest path algorithms for weighted graphs.

`dijkstra_predecessor_and_distance` (G, source)	Compute weighted shortest path length and predecessors.
`dijkstra_path` (G, source, target[, weight])	Returns the shortest weighted path from source to target in G.
`dijkstra_path_length` (G, source, target[, weight])	Returns the shortest weighted path length in G from source to target.
`single_source_dijkstra` (G, source[, target, ...])	Find shortest weighted paths and lengths from a source node.
`single_source_dijkstra_path` (G, source[, ...])	Find shortest weighted paths in G from a source node.
`single_source_dijkstra_path_length` (G, source)	Find shortest weighted path lengths in G from a source node.
`multi_source_dijkstra` (G, sources[, target, ...])	Find shortest weighted paths and lengths from a given set of source nodes.
`multi_source_dijkstra_path` (G, sources[, ...])	Find shortest weighted paths in G from a given set of source nodes.
`multi_source_dijkstra_path_length` (G, sources)	Find shortest weighted path lengths in G from a given set of source nodes.