
extended-networkx-tools

Documentation

Release 0.16.0.rc1

Johan Niklasson, Oskar Hahr

May 07, 2019

Contents:

1 Indices and tables	7
Python Module Index	9

class Analytics.Analytics

Bases: object

static convergence_rate (*nxg*: *networkx.classes.graph.Graph* = *None*, *stochastic_neighbour_matrix*: *List[List[float]]* = *None*) → float
 Function to retrieve the 2nd largest eigenvalue in the adjacency matrix of a graph

Parameters

- **nxg** (*nx.Graph*) – networkx bi-directional graph object
- **stochastic_neighbour_matrix** (*List[List[float]]*) – The stochastic neighbour matrix of the given graph.

Returns The 2nd largest eigenvalue of the adjacency matrix

Return type float

static convergence_rate2 (*nxg*: *networkx.classes.graph.Graph*) → float
 Function to retrieve convergence rate based on an alternate approach.

Parameters **nxg** (*nx.Graph*) – networkx bi-directional graph object

Returns Alternate convergence rate

Return type float

convergence_rate_cuda

static get_adjacency_matrix (*nxg*: *networkx.classes.graph.Graph*, *self_assignment=False*) → *List[List[int]]*

Creates a neighbour matrix for a specified graph: *g*, each row represents a node in the graph where the values in each column represents if there is an edge or not between those nodes.

Parameters

- **nxg** (*bool*) – networkx bi-directional graph object.
- **self_assignment** – Whether or not to use self assignment in the graph. Used for convergence rate.

Return A List of rows, representing the adjacency matrix.

Return type *List[List[float]]*

static get_average_eccentricity (*nxg*: *networkx.classes.graph.Graph*) → float
 Calculates the average eccentricity from the given graph.

Return type float

Parameters **nxg** – The graph to get the average eccentricity from.

Returns The average eccentricity from the graph.

static get_degree_matrix (*nxg*: *networkx.classes.graph.Graph*) → *List[List[int]]*

static get_distance_distribution (*nxg*: *networkx.classes.graph.Graph*) → *Dict[int, int]*
 Makes a list representing the distribution of longest shortest paths between every node in the graph.

Return type *Dict[int, int]*

Parameters **nxg** – A given graph with edges.

Returns A dict with a distribution of the longest shortest paths between nodes.

static get_eccentricity_distribution (*nxg*: *networkx.classes.graph.Graph*) → *Dict[int, int]*

Makes a list representing the distribution of longest shortest paths between every node in the graph.

Return type Dict[int, int]

Parameters `nxg` – A given graph with edges.

Returns A dict with a distribution of the longest shortest paths between nodes.

static `get_edge_dict` (*nxg: networkx.classes.graph.Graph*) → Dict[int, List[int]]

Converts a networkx object to a dict with edges and their neighbours. Can be used to recreate a new graph with `Creator.from_dict()`.

Return type Dict[int, List[int]]

Parameters `nxg` – The graph to get the edges from.

Returns A neighbour list for all nodes.

static `get_eigenvalues` (*mx: List[List[float]]*, *symmetrical: bool = False*) → numpy.ndarray

Simple function to retrieve the eigenvalues of a matrix.

Parameters

- `mx` – A matrix made up of nested lists.
- `symmetrical` – Whether or not the matrix is symmetrical. If true it can make faster computations.

Returns List of eigenvalues of the provided matrix.

Return type List[float]

static `get_laplacian_matrix` (*nxg: networkx.classes.graph.Graph*) → List[List[int]]

Calculates the laplacian matrix based on a given graph.

Parameters `nxg` – The graph to get the laplacian matrix from.

Returns The laplacian matrix, such as $L = D - A$ where D = Degree matrix and A = Adjacency matrix

static `get_neighbour_matrix` (*nxg: networkx.classes.graph.Graph*)

static `get_node_dict` (*nxg: networkx.classes.graph.Graph*) → Dict[int, Tuple[int, int]]

Converts a networkx object to a dict with nodes and their positions. Can be used to recreate a new graph with `Creator.from_dict()`.

Return type Dict[int, Tuple[int, int]]

Parameters `nxg` – The graph to get the nodes from.

Returns A dict of nodes with their corresponding positions.

static `get_stochastic_neighbour_matrix` (*nxg: networkx.classes.graph.Graph = None*,
adjacency_matrix: List[List[int]] = None) → List[List[float]]

Creates a stochastic adjacency matrix for a specified graph: `g`, each row represents a node in the graph where the values in each column represents if there is an edge or not between those nodes. The values for each neighbour is represented by $1/(\text{number of neighbours})$, if no edge exists this value is 0.

Parameters

- `nxg` (*nx.Graph*) – Networkx bi-directional graph object.
- `adjacency_matrix` (*List[List[int]]*) – Self assigned adjacency matrix.

Return A List of rows, representing the adjacency matrix.

Return type List[List[float]]

static hypothetical_max_edge_cost (*nxg: networkx.classes.graph.Graph*) → float
 Calculates the hypothetical total edge cost if the graph were to be complete.

Return type float

Parameters **nxg** – The graph to calculate the hypothetical edge cost of.

Returns The total edge cost if the graph were complete.

static is_graph_connected (*laplacian_matrix: List[List[int]]*)

Checks whether a given graph is connected based on its laplacian matrix.

Parameters **laplacian_matrix** – The laplacian matrix, representing the graph.

Returns Whether it's connected or not.

static is_nodes_connected (*nxg: networkx.classes.graph.Graph, origin: int, destination: int*)
 → bool

Checks if two nodes are connected with each other using a BFS approach.

Parameters

- **nxg** – The graph that contains the two nodes.
- **origin** – The origin node id to check from.
- **destination** – The destination node to check the connectivity to.

Returns True if there's a connection between the nodes, otherwise False.

is_nodes_connected_cuda

static second_largest (*numbers: List[float], sorted_list: bool = False*) → float

Simple function to return the 2nd largest number in a list of numbers.

Parameters

- **numbers** – A list of numbers
- **sorted_list** – If the list is sorted or not

Returns The 2nd largest number in the list numbers

Return type float

second_largest_cuda

Simple function to return the 2nd largest number in a list of numbers.

Parameters **numbers** – A list of numbers

Returns The 2nd largest number in the list numbers

Return type float

static second_smallest (*numbers: List[float], sorted_list: bool = False*) → float

Simple function to return the 2nd smallest number in a list of numbers.

Parameters

- **numbers** – A list of numbers
- **sorted_list** – If the list is sorted or not

Returns The 2nd smallest number in the list numbers

Return type float

static total_edge_cost (*nxg: networkx.classes.graph.Graph*) → int

Calculates the total cost of all edges in the given graph

Parameters `nxg` (*nx.Graph*) – A networkx object with nodes and edges.

Returns The total cost of all edges in the graph.

Return type float

class `Creator.Creator`

Bases: object

Static class that works with creating graph objects from given specifications. Can either create a random unassigned graph with given nodes or a graph with edges from given parameters.

static `add_weighted_edge` (*nxg: networkx.classes.graph.Graph, origin: int, destination: int, ignore_validity: bool = False*) → bool

Adds a bidirectional edge between 2 nodes with weight corresponding to the distance between the nodes squared.

Parameters

- `nxg` – The graph to add an edge to.
- `origin` – First node id to add the edge from
- `destination` – Second node id to add the edge to.
- `ignore_validity` – Whether to skip the validity check when adding the edge

Returns True if the edge was added, otherwise false if the edge already existed.

static `from_random` (*node_count: int, area_dimension: int = None*) → *networkx.classes.graph.Graph*

Creates an unassigned graph with nodes of random position. The work area corresponds to the node count squared.

Return type *networkx.Graph*

Parameters

- `node_count` – The number of nodes to create a graph from.
- `area_dimension` – The size of the area to put nodes in. Defaults to the node count.

Returns An unassigned graph with nodes with random position.

static `from_spec` (*v: Dict[int, Tuple[int, int]], e: Dict[int, List[int]]*) → *networkx.classes.graph.Graph*

Creates a graph from given parameters, that also assigns weighted edges based on a neighbour list.

Parameters

- `v` – Nodes in the graph. Should be a dict with the format { node_1: (x, y), node_2: (x, y)... }
- `e` – Edges that connects the nodes. Should be a dict with the format { node_1: [dest_1, dest_2, ...], node_2: [dest_3, dest_4, ...] }

Returns A graph with assigned nodes and weighted edges.

Return type *networkx.Graph*

class `Solver.Solver`

Bases: object

Class to add edges to given networkx graphs taken from simple Graph Theory, such as path, cycle and complete graph.

static complete (*nix: networkx.classes.graph.Graph*) → networkx.classes.graph.Graph
 Makes a graph a complete graph, such as all nodes are connected to each other with one edge.

Return type networkx.Graph

Parameters **nixg** – A graph with nodes containing coordinates.

Returns A complete graph.

static cycle (*nix: networkx.classes.graph.Graph*) → networkx.classes.graph.Graph
 Adds edges to a given graph as a path, such as the following: (0, 1), (1, 2), ... (n-1, n), (n, 0)

Return type networkx.Graph

Parameters **nixg** – A graph with nodes containing coordinates.

Returns A graph with connected nodes such as they form a cycle.

static path (*nix: networkx.classes.graph.Graph*) → networkx.classes.graph.Graph
 Adds edges to a given graph as a path, such as the following: (0, 1), (1, 2), ... (n-1, n)

Return type networkx.Graph

Parameters **nixg** – A graph with nodes containing coordinates.

Returns A graph with connected nodes such as they form a path.

class Visual.**Visual**

Bases: object

Static class that only helps in visualising graph information.

static draw (*nx_graph*)

Takes a networkx graph and prints the nodes with given edges in the fixed positions.

Parameters **nx_graph** (*networkx.Graph*) – The networkx object to show the graph from.

static save (*nx_graph, filename*)

Takes a networkx graph and save graph with given edges in the fixed positions to a PNG-image.

Parameters **nx_graph** (*networkx.Graph*) – The networkx object to show the graph from.

class AnalyticsGraph.**AnalyticsGraph** (*nix: networkx.classes.graph.Graph*)

Bases: object

add_edge (*origin, destination*)

get_adjacency_matrix_sa ()

get_convergence_rate () → float

Calculates the convergence rate for the current graph.

Returns

get_dimension ()

get_edge_cost () → float

Calculates the edge cost for the current graph.

Returns

get_laplacian_matrix ()

graph () → networkx.classes.graph.Graph

Returns the graph instance that the class has been working on.

Returns The current networkx graph instance.

has_edge (*origin, destination*)

Checks whether the graph has an edge by looking up directly in a adjacency matrix.

Parameters

- **origin** –
- **destination** –

Returns

is_connected () → bool

Checks whether the graph is connected or not.

Returns

move_edge (*origin, old_destination, new_destination*)

remove_edge (*origin, destination*)

reset_stage_actions ()

revert ()

CHAPTER 1

Indices and tables

- [genindex](#)
- [modindex](#)
- [search](#)

a

Analytics, 1
AnalyticsGraph, 5

c

Creator, 4

s

Solver, 4

v

Visual, 5

A

add_edge() (*AnalyticsGraph.AnalyticsGraph method*), 5
 add_weighted_edge() (*Creator.Creator static method*), 4
 Analytics (*class in Analytics*), 1
 Analytics (*module*), 1
 AnalyticsGraph (*class in AnalyticsGraph*), 5
 AnalyticsGraph (*module*), 5

C

complete() (*Solver.Solver static method*), 4
 convergence_rate() (*Analytics.Analytics static method*), 1
 convergence_rate2() (*Analytics.Analytics static method*), 1
 convergence_rate_cuda (*Analytics.Analytics attribute*), 1
 Creator (*class in Creator*), 4
 Creator (*module*), 4
 cycle() (*Solver.Solver static method*), 5

D

draw() (*Visual.Visual static method*), 5

F

from_random() (*Creator.Creator static method*), 4
 from_spec() (*Creator.Creator static method*), 4

G

get_adjacency_matrix() (*Analytics.Analytics static method*), 1
 get_adjacency_matrix_sa() (*Analytics-Graph.AnalyticsGraph method*), 5
 get_average_eccentricity() (*Analytics.Analytics static method*), 1
 get_convergence_rate() (*Analytics-Graph.AnalyticsGraph method*), 5

get_degree_matrix() (*Analytics.Analytics static method*), 1
 get_dimension() (*AnalyticsGraph.AnalyticsGraph method*), 5
 get_distance_distribution() (*Analytics.Analytics static method*), 1
 get_eccentricity_distribution() (*Analytics.Analytics static method*), 1
 get_edge_cost() (*AnalyticsGraph.AnalyticsGraph method*), 5
 get_edge_dict() (*Analytics.Analytics static method*), 2
 get_eigenvalues() (*Analytics.Analytics static method*), 2
 get_laplacian_matrix() (*Analytics.Analytics static method*), 2
 get_laplacian_matrix() (*Analytics-Graph.AnalyticsGraph method*), 5
 get_neighbour_matrix() (*Analytics.Analytics static method*), 2
 get_node_dict() (*Analytics.Analytics static method*), 2
 get_stochastic_neighbour_matrix() (*Analytics.Analytics static method*), 2
 graph() (*AnalyticsGraph.AnalyticsGraph method*), 5

H

has_edge() (*AnalyticsGraph.AnalyticsGraph method*), 5
 hypothetical_max_edge_cost() (*Analytics.Analytics static method*), 2

I

is_connected() (*AnalyticsGraph.AnalyticsGraph method*), 6
 is_graph_connected() (*Analytics.Analytics static method*), 3
 is_nodes_connected() (*Analytics.Analytics static method*), 3

`is_nodes_connected_cuda` (*Analytics.Analytics attribute*), 3

M

`move_edge()` (*AnalyticsGraph.AnalyticsGraph method*), 6

P

`path()` (*Solver.Solver static method*), 5

R

`remove_edge()` (*AnalyticsGraph.AnalyticsGraph method*), 6

`reset_stage_actions()` (*AnalyticsGraph.AnalyticsGraph method*), 6

`revert()` (*AnalyticsGraph.AnalyticsGraph method*), 6

S

`save()` (*Visual.Visual static method*), 5

`second_largest()` (*Analytics.Analytics static method*), 3

`second_largest_cuda` (*Analytics.Analytics attribute*), 3

`second_smallest()` (*Analytics.Analytics static method*), 3

`Solver` (*class in Solver*), 4

`Solver` (*module*), 4

T

`total_edge_cost()` (*Analytics.Analytics static method*), 3

V

`Visual` (*class in Visual*), 5

`Visual` (*module*), 5