DyNetx Documentation

Release 0.1.0

Giulio Rossetti

Oct 02, 2018

Contents

DyNetx is a Python software package that extends `networkx` with dynamic network models and algorithms.

Date	Python Versions	Main Author	GitHub	pypl
August 8, 2017	2.7.x/3.x	Giulio Rossetti	Source	Distribution

Contents:

CHAPTER 1

Overview

DyNetx is a Python language software package for describing, model, and study dynamic complex netowrks.

1.1 Who uses DyNetx?

The potential audience for DyNetx includes mathematicians, physicists, biologists, computer scientists, and social scientists.

1.2 Goals

DyNetx is built upon the NetworkX python library and is intended to provide:

- tools for the study dynamic social, biological, and infrastructure networks,
- a rapid development environment for collaborative, multidisciplinary, projects.

1.3 The Python DyNetx library

DyNetx is a powerful Python package that allows simple and flexible modelling of dynamic networks.

Most importantly, DyNetx, as well as the Python programming language, is free, well-supported, and a joy to use.

1.4 Free software

DyNetx is free software; you can redistribute it and/or modify it under the terms of the BSD License. We welcome contributions from the community.

CHAPTER 2

Download

2.1 Software

Source and binary releases: https://pypi.python.org/pypi/dynetx

Github (latest development): https://github.com/GiulioRossetti/dynetx

2.2 Documentation

CHAPTER 3

Installing

Before installing DyNetx, you need to have setuptools installed.

3.1 Note

In case of misaligned versions between pypl and GitHub, the documentation will refer to the GitHub version.

3.1.1 Quick install

Get DyNetx from the Python Package Index at pypl.

or install it with

```
pip install dynetx
```

and an attempt will be made to find and install an appropriate version that matches your operating system and Python version.

You can install the development version with

```
pip install git://github.com/GiulioRossetti/dynetx.git
```

3.1.2 Installing from source

You can install from source by downloading a source archive file (tar.gz or zip) or by checking out the source files from the GitHub source code repository.

DyNetx is a pure Python package; you don't need a compiler to build or install it.

Source archive file

Download the source (tar.gz or zip file) from pypl or get the latest development version from GitHub

Unpack and change directory to the source directory (it should have the files README.txt and setup.py).

Run python setup.py install to build and install

GitHub

Clone the DyNetx repostitory (see GitHub for options)

```
git clone https://github.com/GiulioRossetti/dynetx.git
```

Change directory to ndlib

Run python setup.py install to build and install

If you don't have permission to install software on your system, you can install into another directory using the –user, –prefix, or –home flags to setup.py.

For example

```
python setup.py install --prefix=/home/username/python
```

or

```
python setup.py install --home=~
```

or

```
python setup.py install --user
```

If you didn't install in the standard Python site-packages directory you will need to set your PYTHONPATH variable to the alternate location. See http://docs.python.org/2/install/index.html#search-path for further details.

3.1.3 Requirements

Python

To use DyNetx you need Python 2.7, 3.2 or later.

The easiest way to get Python and most optional packages is to install the Enthought Python distribution "Canopy" or using Anaconda.

There are several other distributions that contain the key packages you need for scientific computing.

Required packages

The following are packages required by `DyNetx`.

NetworkX

`DyNetx` extends the networkx python library adding dynamic network facilities.

Download: http://networkx.github.io/download.html

CHAPTER 4

DyNetx Tutorial

DyNetx is built upon networkx and is designed to configure, model and analyze dynamic networks.

In this tutorial we will introduce the `DynGraph` object that can be used to describe undirected, temporal graphs.

4.1 Creating a graph

Create an empty dynamic graph with no nodes and no edges.

```
import dynetx as dn
g = dn.DynGraph(edge_removal=True)
```

During the construction phase the `edge_removal` parameter allows to specify if the dynamic graph will allow edge removal or not.

4.1.1 Interactions

`G` can be grown by adding one interaction at a time. Every interaction is univocally defined by its endpoints, `u` and `v`, as well as its timestamp `t`.

```
g.add_interaction(u=1, v=2, t=0)
```

Moreover, also interaction duration can be specified at creation time:

```
g.add_interaction(u=1, v=2, t=0, e=3)
```

In the above example the interaction `(1, 2)` appear at time `0` and vanish at time `3`, thus being present in `[0, 2]`.

Interaction list can also be added: in such scenario all the interactions in the list will have a same timestamp (i.e. they will belong to a same network *snapshot*)

```
g.add_interactions_from([(1, 2), (2, 3), (3, 1)], t=2)
```

The same method can be used to add any `ebunch` of interaction. An *ebunch* is any iterable container of interaction-tuples.

```
g.add_interactions_from(H.edges(), t=2)
```

4.1.2 Nodes

Flattened node degree can be computed via the usual `degree` method exposed by `networkx` graph objects. In order to get the time dependent degree a parameter `t`, identifying the desired snapshot, must be specified.

Similarly, the `neighbors` method has been extended with a similar optional filtering parameter `t`.

4.2 Read graph from file

`DyNetx` allows to read/write networks from files in two formats:

- snapshot graph (extended edgelist)
- interaction graph (extended edgelist)

The former format describes the dynamic graph one edge per row as a 3-tuple

```
n1 n2 t1
```

where

- `n1` and `n2` are nodes
- `t1` is the timestamp of interaction appearance

The latter format describes the dynamic graph one interaction per row as a 4-tuple

```
n1 n2 op t1
```

where

- `n1` and `n2` are nodes
- `t1` is the timestamp of interaction appearance
- `op` identify either the insertion, `+`, or deletion, `-` of the edge

4.2.1 Snapshot Graph

In order to read a snapshot graph file

```
g = dn.read_snapshots(graph_filename, nodetype=int, timestamptype=int)
```

in order to save a graph in the same format

```
dn.write_snapshots(graph, graph_filename)
```

4.2.2 Interaction Graph

In order to read an interaction graph file

```
g = dn.read_interactions(graph_filename, nodetype=int, timestamptype=int)
```

in order to save a graph in the same format

```
dn.write_interactions(graph, graph_filename)
```

4.3 Snapshots and Interactions

The timestamps associated to graph edges can be retrieved through

```
g.temporal_snapshots_ids()
```

Similarly, the number of interactions in a given snapshot can be obtained via

```
g.number_of_interactions(t=snapshot_id)
```

if the parameter `t` is not specified a dictionary snapshot->edges number will be returned.

4.4 Slicing a Dynamic Network

Once loaded a graph it is possible to extract from it a time slice, i.e., a time-span graph

```
s = g.time_slice(t_from=2, t_to=3)
```

the resulting `DynGraph` stored in `s` will be composed by nodes and interactions existing within the time span `[2, 3]`.

4.5 Obtain the Interaction Stream

A dynamic network can be also described as stream of interactions, a chronologically ordered list of interactions

```
for e in g.stream_interactions():
        print e
```

the `stream_interactions` method returns a generator that streams the interactions in `g`, where `e` is a 4-tuple `(u, v, op, t)`

- `u, v` are nodes
- `op` is a edge creation or deletion event (respectively +, −)
- `t` is the interactions timestamp

CHAPTER 5

Reference

In this section are introduced the components that constitute `DyNetx`, namely

- The implemented dynamic graph models
- The implemented algorithms

In `DyNetx` are implemented the following **Dynamic Graph** models:

5.1 Graph types

DyNetx provides data structures and methods for storing graphs.

The choice of graph class depends on the structure of the graph you want to represent.

5.1.1 Which graph class should I use?

Dynamic Graph Type	DyNetx Class
Undirected	DynGraph
Directed	DynDiGraph

5.1.2 Basic graph types

Undirected Dynamic Graphs

Overview

class dynetx.**DynGraph**(*data=None*, *edge_removal=True*, ***attr*)

Base class for undirected dynamic graphs.

A DynGraph stores nodes and timestamped interaction.

DynGraph hold undirected interaction. Self loops are allowed.

Nodes can be arbitrary (hashable) Python objects with optional key/value attributes.

Parameters

- **data** (*input graph*) – Data to initialize graph. If data=None (default) an empty graph is created. The data can be an interaction list, or any NetworkX graph object.
- **attr** (*keyword arguments, optional (default= no attributes)*) – Attributes to add to graph as key=value pairs.
- **edge_removal** (*bool, optional (default=True)*) – Specify if the dynamic graph instance should allows edge removal or not.

See also:

DynDiGraph

Examples

Create an empty graph structure (a "null graph") with no nodes and no interactions.

```
>>> G = dn.DynGraph()
```

G can be grown in several ways.

Nodes:

Add one node at a time:

```
>>> G.add_node(1)
```

Add the nodes from any container (a list, dict, set or even the lines from a file or the nodes from another graph).

```
>>> G.add_nodes_from([2,3])
>>> G.add_nodes_from(range(100,110))
>>> H=dn.DynGraph()
>>> H.add_path([0,1,2,3,4,5,6,7,8,9], t=0)
>>> G.add_nodes_from(H)
```

In addition to strings and integers any hashable Python object (except None) can represent a node.

```
>>> G.add_node(H)
```

Edges:

G can also be grown by adding interaction and specifying their timestamp.

Add one interaction,

```
>>> G.add_interaction(1, 2, t=0)
```

a list of interaction

```
>>> G.add_interactions_from([(3, 2), (1,3)], t=1)
```

If some interaction connect nodes not yet in the graph, the nodes are added automatically.

To traverse all interactions of a graph a time t use the interactions(t) method.

```
>>> G.interactions(t=1)
[(3, 2), (1, 3)]
```

Adding and removing nodes and edges

DynGraph.__init__([data, edge_removal])	Initialize a graph with interaction, name, graph attributes.
DynGraph.add_interaction(u, v[, t, e])	Add an interaction between u and v at time t vanishing (optional) at time e.
DynGraph.add_interactions_from(ebunch[, t, e])	Add all the interaction in ebunch at time t.
DynGraph.add_star(nodes[, t])	Add a star at time t.
DynGraph.add_path(nodes[, t])	Add a path at time t.
DynGraph.add_cycle(nodes[, t])	Add a cycle at time t.

dynetx.DynGraph.__init__

DynGraph.**__init__**(*data=None*, *edge_removal=True*, ***attr*)

Initialize a graph with interaction, name, graph attributes.

Parameters

- **data** (*input graph*) – Data to initialize graph. If data=None (default) an empty graph is created. The data can be an interaction list, or any NetworkX/DyNetx graph object. If the corresponding optional Python packages are installed the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.
- **edge_removal** (*bool, optional (default=True)*) – Specify if the dynamic graph instance should allows edge removal or not.
- **attr** (*keyword arguments, optional (default= no attributes)*) – Attributes to add to graph as key=value pairs.

Examples

```
>>> G = dn.DynGraph()
>>> G = dn.DynGraph(edge_removal=True)
```

dynetx.DynGraph.add_interaction

DynGraph.**add_interaction**(*u*, *v*, *t=None*, *e=None*)

Add an interaction between u and v at time t vanishing (optional) at time e.

The nodes u and v will be automatically added if they are not already in the graph.

Parameters

- **v** (*u,*) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.
- **t** (*appearance snapshot id, mandatory*) –

- **e** (*vanishing snapshot id, optional (default=None)*) –

See also:

add_edges_from() add a collection of interaction at time t

Notes

Adding an interaction that already exists but with different snapshot id updates the interaction data.

Examples

The following all add the interaction e=(1,2, 0) to graph G:

```
>>> G = dn.DynGraph()
>>> G.add_interaction(1, 2, 0)           # explicit two-node form
>>> G.add_interaction( [(1,2)], t=0 ) # add interaction from iterable container
```

Specify the vanishing of the interaction

>>>> G.add_interaction(1, 3, t=1, e=10)

will produce an interaction present in snapshots [0, 9]

dynetx.DynGraph.add_interactions_from

DynGraph.**add_interactions_from**(*ebunch*, *t=None*, *e=None*)

Add all the interaction in ebunch at time t.

Parameters

- **ebunch** (*container of interaction*) – Each interaction given in the container will be added to the graph. The interaction must be given as as 2-tuples (u,v) or 3-tuples (u,v,d) where d is a dictionary containing interaction data.
- **t** (*appearance snapshot id, mandatory*) –
- **e** (*vanishing snapshot id, optional*) –

See also:

add_edge() add a single interaction

Examples

```
>>> G = dn.DynGraph()
>>> G.add_edges_from([(0,1),(1,2)], t=0)
```

dynetx.DynGraph.add_star

DynGraph.**add_star**(*nodes*, *t=None*)

Add a star at time t.

The first node in nodes is the middle of the star. It is connected to all other nodes.

Parameters

- **nodes** (*iterable container*) – A container of nodes.
- **t** (*snapshot id (default=None)*) –

See also:

add_path(), *add_cycle()*

Examples

```
>>> G = dn.DynGraph()
>>> G.add_star([0,1,2,3], t=0)
```

dynetx.DynGraph.add_path

DynGraph.**add_path**(*nodes*, *t=None*)

Add a path at time t.

Parameters

- **nodes** (*iterable container*) – A container of nodes.
- **t** (*snapshot id (default=None)*) –

See also:

add_path(), *add_cycle()*

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2,3], t=0)
```

dynetx.DynGraph.add_cycle

DynGraph.**add_cycle**(*nodes*, *t=None*)

Add a cycle at time t.

Parameters

- **nodes** (*iterable container*) – A container of nodes.
- **t** (*snapshot id (default=None)*) –

See also:

add_path(), *add_cycle()*

Examples

```
>>> G = dn.DynGraph()
>>> G.add_cycle([0,1,2,3], t=0)
```

Iterating over nodes and edges

DynGraph.interactions([nbunch, t])	Return the list of interaction present in a given snapshot.
DynGraph.interactions_iter([nbunch, t])	Return an iterator over the interaction present in a given snapshot.
DynGraph.neighbors(n[, t])	Return a list of the nodes connected to the node n at time t.
DynGraph.neighbors_iter(n[, t])	Return an iterator over all neighbors of node n at time t.
DynGraph.nodes([t, data])	Return a list of the nodes in the graph at a given snapshot.
DynGraph.nodes_iter([t, data])	Return an iterator over the nodes with respect to a given temporal snapshot.

dynetx.DynGraph.interactions

DynGraph.**interactions**(*nbunch=None*, *t=None*)

Return the list of interaction present in a given snapshot.

Edges are returned as tuples in the order (node, neighbor).

Parameters

- **nbunch** (*iterable container, optional (default= all nodes)*) – A container of nodes. The container will be iterated through once.
- **t** (*snapshot id (default=None)*) – If None the the method returns all the edges of the flattened graph.

Returns interaction_list – Interactions that are adjacent to any node in nbunch, or a list of all interactions if nbunch is not specified.

Return type list of interaction tuples

See also:

edges_iter() return an iterator over the interactions

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-interaction.

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2], t=0)
>>> G.add_edge(2,3, t=1)
>>> G.interactions(t=0)
[(0, 1), (1, 2)]
>>> G.interactions()
[(0, 1), (1, 2), (2, 3)]
>>> G.interactions([0,3], t=0)
[(0, 1)]
```

dynetx.DynGraph.interactions_iter

DynGraph.**interactions_iter**(*nbunch=None*, *t=None*)

Return an iterator over the interaction present in a given snapshot.

Edges are returned as tuples in the order (node, neighbor).

Parameters

- **nbunch** (*iterable container, optional (default= all nodes)*) – A container of nodes. The container will be iterated through once.
- **t** (*snapshot id (default=None)*) – If None the the method returns an iterator over the edges of the flattened graph.

Returns **edge_iter** – An iterator of (u,v) tuples of interaction.

Return type iterator

See also:

interaction() return a list of interaction

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-interaction.

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2], 0)
>>> G.add_interaction(2,3,1)
>>> [e for e in G.interactions_iter(t=0)]
[(0, 1), (1, 2)]
>>> list(G.interactions_iter())
[(0, 1), (1, 2), (2, 3)]
```

dynetx.DynGraph.neighbors

DynGraph.**neighbors**(*n*, *t=None*)

Return a list of the nodes connected to the node n at time t.

Parameters

- **n** (*node*) – A node in the graph
- **t** (*snapshot id (default=None)*) – If None will be returned the neighbors of the node on the flattened graph.

Returns **nlist** – A list of nodes that are adjacent to n.

Return type list

Raises NetworkXError – If the node n is not in the graph.

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2,3], t=0)
>>> G.neighbors(0, t=0)
[1]
>>> G.neighbors(0, t=1)
[]
```

dynetx.DynGraph.neighbors_iter

DynGraph.**neighbors_iter**(*n*, *t=None*)

Return an iterator over all neighbors of node n at time t.

Parameters

- **n** (*node*) – A node in the graph
- **t** (*snapshot id (default=None)*) – If None will be returned an iterator over the neighbors of the node on the flattened graph.

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2,3], t=0)
>>> [n for n in G.neighbors_iter(0, t=0)]
[1]
```

dynetx.DynGraph.nodes

DynGraph.**nodes**(*t=None*, *data=False*)

Return a list of the nodes in the graph at a given snapshot.

Parameters

- **t** (*snapshot id (default=None)*) – If None the the method returns all the nodes of the flattened graph.
- **data** (*boolean, optional (default=False)*) – If False return a list of nodes. If True return a two-tuple of node and node data dictionary

Returns **nlist** – A list of nodes. If data=True a list of two-tuples containing (node, node data dictionary).

Return type list

Examples

```
>>> G = dn.DynGraph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2], 0)
>>> G.nodes(t=0)
[0, 1, 2]
```

(continues on next page)

(continued from previous page)

```
>>> G.add_edge(1, 4, t=1)
>>> G.nodes(t=0)
[0, 1, 2]
```

dynetx.DynGraph.nodes_iter

DynGraph.**nodes_iter**(*t=None*, *data=False*)

Return an iterator over the nodes with respect to a given temporal snapshot.

Parameters

- **t** (*snapshot id (default=None)*) – If None the iterator returns all the nodes of the flattened graph.
- **data** (*boolean, optional (default=False)*) – If False the iterator returns nodes. If True return a two-tuple of node and node data dictionary

Returns **niter** – An iterator over nodes. If data=True the iterator gives two-tuples containing (node, node data, dictionary)

Return type iterator

Examples

```
>>> G = dn.DynGraph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2], 0)
```

```
>>> [n for n, d in G.nodes_iter(t=0)]
[0, 1, 2]
```

Information about graph structure

DynGraph.has_interaction(u, v[, t])	Return True if the interaction (u,v) is in the graph at time t.
DynGraph.number_of_interactions([u, v, t])	Return the number of interaction between two nodes at time t.
DynGraph.degree([nbunch, t])	Return the degree of a node or nodes at time t.
DynGraph.degree_iter([nbunch, t])	Return an iterator for (node, degree) at time t.
DynGraph.size([t])	Return the number of edges at time t.
DynGraph.order([t])	Return the number of nodes in the t snpashot of a dynamic graph.
DynGraph.has_node(n[, t])	Return True if the graph, at time t, contains the node n.
DynGraph.number_of_nodes([t])	Return the number of nodes in the t snpashot of a dynamic graph.
DynGraph.to_directed()	Return a directed representation of the graph.

dynetx.DynGraph.has_interaction

DynGraph.**has_interaction**(*u*, *v*, *t=None*)

Return True if the interaction (u,v) is in the graph at time t.

Parameters

- **v** (*u,*) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.
- **t** (*snapshot id (default=None)*) – If None will be returned the presence of the interaction on the flattened graph.

Returns **edge_ind** – True if interaction is in the graph, False otherwise.

Return type bool

Examples

Can be called either using two nodes u,v or interaction tuple (u,v)

```
>>> G = nx.Graph()
>>> G.add_path([0,1,2,3], t=0)
>>> G.has_interaction(0,1, t=0)
True
>>> G.has_interaction(0,1, t=1)
False
```

dynetx.DynGraph.number_of_interactions

DynGraph.**number_of_interactions**(*u=None*, *v=None*, *t=None*)

Return the number of interaction between two nodes at time t.

Parameters

- **v** (*u,*) – If u and v are specified, return the number of interaction between u and v. Otherwise return the total number of all interaction.
- **t** (*snapshot id (default=None)*) – If None will be returned the number of edges on the flattened graph.

Returns **nedges** – The number of interaction in the graph. If nodes u and v are specified return the number of interaction between those nodes. If a single node is specified return None.

Return type int

See also:

size()

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2,3], t=0)
>>> G.number_of_interactions()
3
```

(continues on next page)

(continued from previous page)

```
>>> G.number_of_interactions(0,1, t=0)
1
>>> G.add_edge(3, 4, t=1)
>>> G.number_of_interactions()
4
```

dynetx.DynGraph.degree

DynGraph.**degree**(*nbunch=None*, *t=None*)

Return the degree of a node or nodes at time t.

The node degree is the number of interaction adjacent to that node in a given time frame.

Parameters

- **nbunch** (*iterable container, optional (default=all nodes)*) – A container of nodes. The container will be iterated through once.
- **t** (*snapshot id (default=None)*) – If None will be returned the degree of nodes on the flattened graph.

Returns **nd** – A dictionary with nodes as keys and degree as values or a number if a single node is specified.

Return type dictionary, or number

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2,3], t=0)
>>> G.degree(0, t=0)
1
>>> G.degree([0,1], t=1)
{0: 0, 1: 0}
>>> list(G.degree([0,1], t=0).values())
[1, 2]
```

dynetx.DynGraph.degree_iter

DynGraph.**degree_iter**(*nbunch=None*, *t=None*)

Return an iterator for (node, degree) at time t.

The node degree is the number of edges adjacent to the node in a given timeframe.

Parameters

- **nbunch** (*iterable container, optional (default=all nodes)*) – A container of nodes. The container will be iterated through once.
- **t** (*snapshot id (default=None)*) – If None will be returned an iterator over the degree of nodes on the flattened graph.

Returns **nd_iter** – The iterator returns two-tuples of (node, degree).

Return type an iterator

See also:

degree()

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2,3], t=0)
>>> list(G.degree_iter(0, t=0))
[(0, 1)]
>>> list(G.degree_iter([0,1], t=0))
[(0, 1), (1, 2)]
```

dynetx.DynGraph.size

DynGraph.**size**(*t=None*)

Return the number of edges at time t.

Parameters **t** (*snapshot id (default=None)*) – If None will be returned the size of the flattened graph.

Returns **nedges** – The number of edges

Return type int

See also:

number_of_edges()

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2,3], t=0)
>>> G.size(t=0)
3
```

dynetx.DynGraph.order

DynGraph.**order**(*t=None*)

Return the number of nodes in the t snpashot of a dynamic graph.

Parameters **t** (*snapshot id (default=None)*) – If None return the number of nodes in the flattened graph.

Returns **nnodes** – The number of nodes in the graph.

Return type int

See also:

number_of_nodes()

Examples

```
>>> G = dn.DynGraph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2], t=0)
>>> G.order(0)
3
```

dynetx.DynGraph.has_node

DynGraph.**has_node**(*n*, *t=None*)

Return True if the graph, at time t, contains the node n.

Parameters

- **n** (*node*) –
- **t** (*snapshot id (default None)*) – If None return the presence of the node in the flattened graph.

Examples

```
>>> G = dn.DynGraph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2], t=0)
>>> G.has_node(0, t=0)
True
```

It is more readable and simpler to use

```
>>> 0 in G
True
```

dynetx.DynGraph.number_of_nodes

DynGraph.**number_of_nodes**(*t=None*)

Return the number of nodes in the t snpashot of a dynamic graph.

Parameters **t** (*snapshot id (default=None)*) – If None return the number of nodes in the flattened graph.

Returns **nnodes** – The number of nodes in the graph.

Return type int

See also:

order()

Examples

```
>>> G = dn.DynGraph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2], t=0)
>>> G.number_of_nodes(0)
3
```

dynetx.DynGraph.to_directed

DynGraph.**to_directed**()

Return a directed representation of the graph.

Returns **G** – A dynamic directed graph with the same name, same nodes, and with each edge (u,v,data) replaced by two directed edges (u,v,data) and (v,u,data).

Return type *DynDiGraph*

Notes

This returns a "deepcopy" of the edge, node, and graph attributes which attempts to completely copy all of the data and references.

This is in contrast to the similar D=DynDiGraph(G) which returns a shallow copy of the data.

See the Python copy module for more information on shallow and deep copies, http://docs.python.org/library/copy.html.

Warning: If you have subclassed Graph to use dict-like objects in the data structure, those changes do not transfer to the DynDiGraph created by this method.

Examples

```
>>> G = dn.DynGraph()    # or MultiGraph, etc
>>> G.add_path([0,1])
>>> H = G.to_directed()
>>> H.edges()
[(0, 1), (1, 0)]
```

If already directed, return a (deep) copy

```
>>> G = dn.DynDiGraph()    # or MultiDiGraph, etc
>>> G.add_path([0,1])
>>> H = G.to_directed()
>>> H.edges()
[(0, 1)]
```

Dynamic Representation: Access Snapshots and Interactions

DynGraph.stream_interactions()	Generate a temporal ordered stream of interactions.
DynGraph.time_slice(t_from[, t_to])	Return an new graph containing nodes and interactions present in [t_from, t_to].
DynGraph.temporal_snapshots_ids()	Return the ordered list of snapshot ids present in the dynamic graph.
DynGraph.interactions_per_snapshots([t])	Return the number of interactions within snapshot t.
DynGraph.inter_event_time_distribution([t, v])	Return the distribution of inter event time.

dynetx.DynGraph.stream_interactions

DynGraph.**stream_interactions**()

Generate a temporal ordered stream of interactions.

Returns **nd_iter** – The iterator returns a 4-tuples of (node, node, op, timestamp).

Return type an iterator

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2,3], t=0)
>>> G.add_path([3,4,5,6], t=1)
>>> list(G.stream_interactions())
[(0, 1, '+', 0), (1, 2, '+', 0), (2, 3, '+', 0), (3, 4, '+', 1), (4, 5, '+', 1),␣
↪(5, 6, '+', 1)]
```

dynetx.DynGraph.time_slice

DynGraph.**time_slice**(*t_from*, *t_to=None*)

Return an new graph containing nodes and interactions present in [t_from, t_to].

Parameters

- **t_from** (*snapshot id, mandatory*) –
- **t_to** (*snapshot id, optional (default=None)*) – If None t_to will be set equal to t_from

Returns **H** – the graph described by interactions in [t_from, t_to]

Return type a DynGraph object

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2,3], t=0)
>>> G.add_path([0,4,5,6], t=1)
>>> G.add_path([7,1,2,3], t=2)
>>> H = G.time_slice(0)
>>> H.interactions()
[(0, 1), (1, 2), (1, 3)]
>>> H = G.time_slice(0, 1)
>>> H.interactions()
[(0, 1), (1, 2), (1, 3), (0, 4), (4, 5), (5, 6)]
```

dynetx.DynGraph.temporal_snapshots_ids

DynGraph.**temporal_snapshots_ids**()

Return the ordered list of snapshot ids present in the dynamic graph.

Returns **nd** – a list of snapshot ids

Return type list

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2,3], t=0)
>>> G.add_path([0,4,5,6], t=1)
>>> G.add_path([7,1,2,3], t=2)
>>> G.temporal_snapshots_ids()
[0, 1, 2]
```

dynetx.DynGraph.interactions_per_snapshots

DynGraph.**interactions_per_snapshots**(*t=None*)

Return the number of interactions within snapshot t.

Parameters **t** (*snapshot id (default=None)*) – If None will be returned total number of interactions across all snapshots

Returns **nd** – A dictionary with snapshot ids as keys and interaction count as values or a number if a single snapshot id is specified.

Return type dictionary, or number

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2,3], t=0)
>>> G.add_path([0,4,5,6], t=1)
>>> G.add_path([7,1,2,3], t=2)
>>> G.interactions_per_snapshots(t=0)
3
>>> G.interactions_per_snapshots()
{0: 3, 1: 3, 2: 3}
```

dynetx.DynGraph.inter_event_time_distribution

DynGraph.**inter_event_time_distribution**(*u=None*, *v=None*)

Return the distribution of inter event time. If u and v are None the dynamic graph intere event distribution is returned. If u is specified the inter event time distribution of interactions involving u is returned. If u and v are specified the inter event time distribution of (u, v) interactions is returned

Parameters

- **u** (*node id*) –
- **v** (*node id*) –

Returns **nd** – A dictionary from inter event time to number of occurrences

Return type dictionary

Directed Dynamic Graphs

Overview

class dynetx.**DynDiGraph**(*data=None*, *edge_removal=True*, ***attr*)

Base class for directed dynamic graphs.

DynDiGraph hold directed interactions. Self loops are allowed.

Nodes can be arbitrary (hashable) Python objects with optional key/value attributes.

Interactions are represented as links between nodes.

Parameters

- **data** (*input graph*) – Data to initialize graph. If data=None (default) an empty graph is created. The data can be an interaction list, or any NetworkX graph object.
- **attr** (*keyword arguments, optional (default= no attributes)*) – Attributes to add to graph as key=value pairs.
- **edge_removal** (*bool, optional (default=True)*) – Specify if the dynamic graph instance should allows interactions removal or not.

See also:

DynGraph

Examples

Create an empty graph structure (a "null graph") with no nodes and no interactions.

```
>>> G = dn.DynDiGraph()
```

G can be grown in several ways.

Nodes:

Add one node at a time:

```
>>> G.add_node(1)
```

Add the nodes from any container (a list, dict, set or even the lines from a file or the nodes from another graph).

```
>>> G.add_nodes_from([2,3])
>>> G.add_nodes_from(range(100,110))
>>> H=dn.DynDiGraph()
>>> H.add_path([0,1,2,3,4,5,6,7,8,9], t=0)
>>> G.add_nodes_from(H)
```

In addition to strings and integers any hashable Python object (except None) can represent a node.

```
>>> G.add_node(H)
```

Edges:

G can also be grown by adding interaction and specifying their timestamp.

Add one interaction,

```
>>> G.add_interaction(1, 2, t=0)
```

a list of interaction

```
>>> G.add_interactions_from([(3, 2), (1,3)], t=1)
```

If some interaction connect nodes not yet in the graph, the nodes are added automatically.

To traverse all interactions of a graph a time t use the interactions(t) method.

```
>>> G.interactions(t=1)
[(3, 2), (1, 3)]
```

Adding and removing nodes and edges

DynDiGraph.__init__([data, edge_removal])	Initialize a directed graph with interaction, name, graph attributes.
DynDiGraph.add_interaction(u, v[, t, e])	Add an interaction between u and v at time t vanishing (optional) at time e.
DynDiGraph.add_interactions_from(ebunch[, t, e])	Add all the interaction in ebunch at time t.

dynetx.DynDiGraph.__init__

DynDiGraph.**__init__**(*data=None*, *edge_removal=True*, ***attr*)

Initialize a directed graph with interaction, name, graph attributes.

Parameters

- **data** (*input graph*) – Data to initialize graph. If data=None (default) an empty graph is created. The data can be an interaction list, or any NetworkX/DyNetx graph object. If the corresponding optional Python packages are installed the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.
- **edge_removal** (*bool, optional (default=True)*) – Specify if the dynamic graph instance should allows edge removal or not.
- **attr** (*keyword arguments, optional (default= no attributes)*) – Attributes to add to graph as key=value pairs.

Examples

```
>>> G = dn.DynDiGraph()
>>> G = dn.DynDiGraph(edge_removal=True)
```

dynetx.DynDiGraph.add_interaction

DynDiGraph.**add_interaction**(*u*, *v*, *t=None*, *e=None*)

Add an interaction between u and v at time t vanishing (optional) at time e.

The nodes u and v will be automatically added if they are not already in the graph.

Parameters

- **v** (*u,*) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.
- **t** (*appearance snapshot id, mandatory*) –
- **e** (*vanishing snapshot id, optional (default=None)*) –

See also:

add_edges_from() add a collection of interaction at time t

Notes

Adding an interaction that already exists but with different snapshot id updates the interaction data.

Examples

The following all add the interaction e=(1,2, 0) to graph G:

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(1, 2, 0)
>>> G.add_interaction( [(1,2)], t=0 )
```

Specify the vanishing of the interaction

>>>> G.add_interaction(1, 3, t=1, e=10)

will produce an interaction present in snapshots [0, 9]

dynetx.DynDiGraph.add_interactions_from

DynDiGraph.**add_interactions_from**(*ebunch*, *t=None*, *e=None*)
Add all the interaction in ebunch at time t.

Parameters

- **ebunch** (*container of interaction*) – Each interaction given in the container will be added to the graph. The interaction must be given as as 2-tuples (u,v) or 3-tuples (u,v,d) where d is a dictionary containing interaction data.
- **t** (*appearance snapshot id, mandatory*) –
- **e** (*vanishing snapshot id, optional*) –

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interactions_from([(0,1),(1,2)], t=0)
```

Iterating over nodes and edges

DynDiGraph.interactions([nbunch, t])	Return the list of interaction present in a given snapshot.
DynDiGraph.interactions_iter([nbunch, t])	Return an iterator over the interaction present in a given snapshot.
DynDiGraph.in_interactions([nbunch, t])	Return the list of incoming interaction present in a given snapshot.
DynDiGraph.in_interactions_iter([nbunch, t])	Return an iterator over the in interactions present in a given snapshot.
DynDiGraph.out_interactions([nbunch, t])	Return the list of out interaction present in a given snapshot.
DynDiGraph.out_interactions_iter([nbunch, t])	Return an iterator over the out interactions present in a given snapshot.
DynDiGraph.neighbors(n[, t])	Return a list of successor nodes of n at time t (optional).
DynDiGraph.neighbors_iter(n[, t])	Return an iterator over successor nodes of n at time t (optional).
DynDiGraph.successors(n[, t])	Return a list of successor nodes of n at time t (optional).
DynDiGraph.successors_iter(n[, t])	Return an iterator over successor nodes of n at time t (optional).
DynDiGraph.predecessors(n[, t])	Return a list of predecessor nodes of n at time t (optional).
DynDiGraph.predecessors_iter(n[, t])	Return an iterator over predecessors nodes of n at time t (optional).
DynDiGraph.nodes([t, data])	Return a list of the nodes in the graph at a given snapshot.
DynDiGraph.nodes_iter([t, data])	Return an iterator over the nodes with respect to a given temporal snapshot.

dynetx.DynDiGraph.interactions

DynDiGraph.**interactions**(*nbunch=None*, *t=None*)

Return the list of interaction present in a given snapshot.

Edges are returned as tuples in the order (node, neighbor).

Parameters

- **nbunch** (*iterable container, optional (default= all nodes)*) – A container of nodes. The container will be iterated through once.
- **t** (*snapshot id (default=None)*) – If None the the method returns all the edges of the flattened graph.

Returns interaction_list – Interactions that are adjacent to any node in nbunch, or a list of all interactions if nbunch is not specified.

Return type list of interaction tuples

See also:

edges_iter() return an iterator over the interactions

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-interaction.

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(0, 1, t=0)
>>> G.add_interaction(2, 3, t=1)
>>> G.interactions(t=0)
[(0, 1)]
>>> G.interactions()
[(0, 1), (2, 3)]
>>> G.interactions([0,3], t=0)
[(0, 1)]
```

dynetx.DynDiGraph.interactions_iter

DynDiGraph.**interactions_iter**(*nbunch=None*, *t=None*)

Return an iterator over the interaction present in a given snapshot.

Edges are returned as tuples in the order (node, neighbor).

Parameters

- **nbunch** (*iterable container, optional (default= all nodes)*) – A container of nodes. The container will be iterated through once.
- **t** (*snapshot id (default=None)*) – If None the the method returns an iterator over the edges of the flattened graph.

Returns **edge_iter** – An iterator of (u,v) tuples of interaction.

Return type iterator

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-interaction.

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(0,1, 0)
>>> G.add_interaction(1,2, 0)
>>> G.add_interaction(2,3,1)
>>> [e for e in G.interactions_iter(t=0)]
[(0, 1), (1, 2)]
>>> list(G.interactions_iter())
[(0, 1), (1, 2), (2, 3)]
```

dynetx.DynDiGraph.in_interactions

DynDiGraph.**in_interactions**(*nbunch=None*, *t=None*)

Return the list of incoming interaction present in a given snapshot.

Edges are returned as tuples in the order (node, neighbor).

Parameters

- **nbunch** (*iterable container, optional (default= all nodes)*) – A container of nodes. The container will be iterated through once.
- **t** (*snapshot id (default=None)*) – If None the the method returns all the edges of the flattened graph.

Returns **interaction_list** – Interactions that are adjacent to any node in nbunch, or a list of all interactions if nbunch is not specified.

Return type list of interaction tuples

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-interaction.

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(0, 1, t=0)
>>> G.add_interaction(2, 3, t=1)
>>> G.in_interactions(t=0)
[(1, 0)]
>>> G.in_interactions()
[(1, 0), (3, 2)]
>>> G.in_interactions([0,3], t=0)
[(3, 2)]
```

dynetx.DynDiGraph.in_interactions_iter

DynDiGraph.**in_interactions_iter**(*nbunch=None*, *t=None*)

Return an iterator over the in interactions present in a given snapshot.

Edges are returned as tuples in the order (node, neighbor).

Parameters

- **nbunch** (*iterable container, optional (default= all nodes)*) – A container of nodes. The container will be iterated through once.
- **t** (*snapshot id (default=None)*) – If None the the method returns an iterator over the edges of the flattened graph.

Returns **edge_iter** – An iterator of (u,v) tuples of interaction.

Return type iterator

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-interaction.

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(0,1, 0)
>>> G.add_interaction(1,2, 0)
>>> G.add_interaction(2,3,1)
>>> [e for e in G.in_interactions_iter(t=0)]
[(0, 1), (1, 2)]
>>> list(G.in_interactions_iter())
[(0, 1), (1, 2), (2, 3)]
```

dynetx.DynDiGraph.out_interactions

DynDiGraph.**out_interactions**(*nbunch=None*, *t=None*)

Return the list of out interaction present in a given snapshot.

Edges are returned as tuples in the order (node, neighbor).

Parameters

- **nbunch** (*iterable container, optional (default= all nodes)*) – A container of nodes. The container will be iterated through once.
- **t** (*snapshot id (default=None)*) – If None the the method returns all the edges of the flattened graph.

Returns **interaction_list** – Interactions that are adjacent to any node in nbunch, or a list of all interactions if nbunch is not specified.

Return type list of interaction tuples

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-interaction.

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(0, 1, t=0)
>>> G.add_interaction(2, 3, t=1)
>>> G.out_interactions(t=0)
[(0, 1)]
>>> G.out_interactions()
[(0, 1), (2, 3)]
>>> G.out_interactions([0,3], t=0)
[(0, 1)]
```

dynetx.DynDiGraph.out_interactions_iter

DynDiGraph.**out_interactions_iter**(*nbunch=None*, *t=None*)

Return an iterator over the out interactions present in a given snapshot.

Edges are returned as tuples in the order (node, neighbor).

Parameters

- **nbunch** (*iterable container, optional (default= all nodes)*) – A container of nodes. The container will be iterated through once.
- **t** (*snapshot id (default=None)*) – If None the the method returns an iterator over the edges of the flattened graph.

Returns **edge_iter** – An iterator of (u,v) tuples of interaction.

Return type iterator

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-interaction.

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(0,1, 0)
>>> G.add_interaction(1,2, 0)
>>> G.add_interaction(2,3,1)
>>> [e for e in G.out_interactions_iter(t=0)]
[(0, 1), (1, 2)]
>>> list(G.out_interactions_iter())
[(0, 1), (1, 2), (2, 3)]
```

dynetx.DynDiGraph.neighbors

DynDiGraph.**neighbors**(*n*, *t=None*)

Return a list of successor nodes of n at time t (optional).

Parameters

- **n** (*node*) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.
- **t** (*snapshot id (default=None)*) – If None will be returned the presence of the interaction on the flattened graph.

dynetx.DynDiGraph.neighbors_iter

DynDiGraph.**neighbors_iter**(*n*, *t=None*)

Return an iterator over successor nodes of n at time t (optional). :param n: Nodes can be, for example, strings or numbers.

Nodes must be hashable (and not None) Python objects.

Parameters **t** (*snapshot id (default=None)*) – If None will be returned the presence of the interaction on the flattened graph.

dynetx.DynDiGraph.successors

DynDiGraph.**successors**(*n*, *t=None*)
Return a list of successor nodes of n at time t (optional).

Parameters

- **n** (*node*) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.
- **t** (*snapshot id (default=None)*) – If None will be returned the presence of the interaction on the flattened graph.

dynetx.DynDiGraph.successors_iter

DynDiGraph.**successors_iter**(*n*, *t=None*)
Return an iterator over successor nodes of n at time t (optional). :param n: Nodes can be, for example, strings or numbers.

Nodes must be hashable (and not None) Python objects.

Parameters **t** (*snapshot id (default=None)*) – If None will be returned the presence of the interaction on the flattened graph.

dynetx.DynDiGraph.predecessors

DynDiGraph.**predecessors**(*n*, *t=None*)
Return a list of predecessor nodes of n at time t (optional).

Parameters

- **n** (*node*) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.
- **t** (*snapshot id (default=None)*) – If None will be returned the presence of the interaction on the flattened graph.

dynetx.DynDiGraph.predecessors_iter

DynDiGraph.**predecessors_iter**(*n*, *t=None*)
Return an iterator over predecessors nodes of n at time t (optional).

Parameters

- **n** (*node*) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.
- **t** (*snapshot id (default=None)*) – If None will be returned the presence of the interaction on the flattened graph.

dynetx.DynDiGraph.nodes

DynDiGraph.**nodes**(*t=None*, *data=False*)
Return a list of the nodes in the graph at a given snapshot.

Parameters

- **t** (*snapshot id (default=None)*) – If None the the method returns all the nodes of the flattened graph.
- **data** (*boolean, optional (default=False)*) – If False return a list of nodes. If True return a two-tuple of node and node data dictionary

Returns **nlist** – A list of nodes. If data=True a list of two-tuples containing (node, node data dictionary).

Return type list

Examples

```
>>> G = dn.DynDiGraph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_interaction(0, 1, 0)
>>> G.nodes(t=0)
[0, 1]
>>> G.add_interaction(1, 4, t=1)
>>> G.nodes(t=0)
[0, 1]
```

dynetx.DynDiGraph.nodes_iter

DynDiGraph.**nodes_iter**(*t=None*, *data=False*)

Return an iterator over the nodes with respect to a given temporal snapshot.

Parameters

- **t** (*snapshot id (default=None)*) – If None the iterator returns all the nodes of the flattened graph.
- **data** (*boolean, optional (default=False)*) – If False the iterator returns nodes. If True return a two-tuple of node and node data dictionary

Returns **niter** – An iterator over nodes. If data=True the iterator gives two-tuples containing (node, node data, dictionary)

Return type iterator

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(0,1, 0)
>>> G.add_interaction(1,2, 0)
```

```
>>> [n for n, d in G.nodes_iter(t=0)]
[0, 1, 2]
```

Information about graph structure

DynDiGraph.has_interaction(u, v[, t])	Return True if the interaction (u,v) is in the graph at time t.
DynDiGraph.has_successor(u, v[, t])	Return True if node u has successor v at time t (optional).
DynDiGraph.has_predecessor(u, v[, t])	Return True if node u has predecessor v at time t (optional).
DynDiGraph.number_of_interactions([u, v, t])	Return the number of interaction between two nodes at time t.
DynDiGraph.degree([nbunch, t])	Return the degree of a node or nodes at time t.
DynDiGraph.degree_iter([nbunch, t])	Return an iterator for (node, degree) at time t.
DynDiGraph.in_degree([nbunch, t])	Return the in degree of a node or nodes at time t.
DynDiGraph.in_degree_iter([nbunch, t])	Return an iterator for (node, in_degree) at time t.
DynDiGraph.out_degree([nbunch, t])	Return the out degree of a node or nodes at time t.
DynDiGraph.out_degree_iter([nbunch, t])	Return an iterator for (node, out_degree) at time t.
DynDiGraph.size([t])	Return the number of edges at time t.
DynDiGraph.order()	Return the number of nodes in the graph.
DynDiGraph.has_node(n[, t])	Return True if the graph, at time t, contains the node n.
DynDiGraph.number_of_nodes([t])	Return the number of nodes in the t snapshot of a dynamic graph.
DynGraph.to_undirected([as_view])	Return an undirected copy of the graph.

dynetx.DynDiGraph.has_interaction

DynDiGraph.**has_interaction**(*u*, *v*, *t=None*)

Return True if the interaction (u,v) is in the graph at time t.

Parameters

- **v** (*u,*) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.
- **t** (*snapshot id (default=None)*) – If None will be returned the presence of the interaction on the flattened graph.

Returns **edge_ind** – True if interaction is in the graph, False otherwise.

Return type bool

Examples

Can be called either using two nodes u,v or interaction tuple (u,v)

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(0,1, t=0)
>>> G.add_interaction(1,2, t=0)
>>> G.add_interaction(2,3, t=0)
>>> G.has_interaction(0,1, t=0)
True
>>> G.has_interaction(0,1, t=1)
False
```

dynetx.DynDiGraph.has_successor

DynDiGraph.**has_successor**(*u*, *v*, *t=None*)

Return True if node u has successor v at time t (optional).

This is true if graph has the edge u->v.

Parameters

- **v** (*u,*) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.
- **t** (*snapshot id (default=None)*) – If None will be returned the presence of the interaction on the flattened graph.

dynetx.DynDiGraph.has_predecessor

DynDiGraph.**has_predecessor**(*u*, *v*, *t=None*)

Return True if node u has predecessor v at time t (optional).

This is true if graph has the edge u<-v.

Parameters

- **v** (*u,*) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.
- **t** (*snapshot id (default=None)*) – If None will be returned the presence of the interaction on the flattened graph.

dynetx.DynDiGraph.number_of_interactions

DynDiGraph.**number_of_interactions**(*u=None*, *v=None*, *t=None*)

Return the number of interaction between two nodes at time t.

Parameters

- **v** (*u,*) – If u and v are specified, return the number of interaction between u and v. Otherwise return the total number of all interaction.
- **t** (*snapshot id (default=None)*) – If None will be returned the number of edges on the flattened graph.

Returns **nedges** – The number of interaction in the graph. If nodes u and v are specified return the number of interaction between those nodes. If a single node is specified return None.

Return type int

See also:

size()

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_path([0,1,2,3], t=0)
>>> G.number_of_interactions()
```

(continues on next page)

(continued from previous page)

```
3
>>> G.number_of_interactions(0,1, t=0)
1
>>> G.add_edge(3, 4, t=1)
>>> G.number_of_interactions()
4
```

dynetx.DynDiGraph.degree

DynDiGraph.**degree**(*nbunch=None*, *t=None*)

Return the degree of a node or nodes at time t.

The node degree is the number of interaction adjacent to that node in a given time frame.

Parameters

- **nbunch** (*iterable container, optional (default=all nodes)*) – A container of nodes. The container will be iterated through once.
- **t** (*snapshot id (default=None)*) – If None will be returned the degree of nodes on the flattened graph.

Returns **nd** – A dictionary with nodes as keys and degree as values or a number if a single node is specified.

Return type dictionary, or number

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(0,1, t=0)
>>> G.add_interaction(1,2, t=0)
>>> G.add_interaction(2,3, t=0)
>>> G.degree(0, t=0)
1
>>> G.degree([0,1], t=1)
{0: 0, 1: 0}
>>> list(G.degree([0,1], t=0).values())
[1, 2]
```

dynetx.DynDiGraph.degree_iter

DynDiGraph.**degree_iter**(*nbunch=None*, *t=None*)

Return an iterator for (node, degree) at time t.

The node degree is the number of edges adjacent to the node in a given timeframe.

Parameters

- **nbunch** (*iterable container, optional (default=all nodes)*) – A container of nodes. The container will be iterated through once.
- **t** (*snapshot id (default=None)*) – If None will be returned an iterator over the degree of nodes on the flattened graph.

Returns **nd_iter** – The iterator returns two-tuples of (node, degree).

Return type an iterator

See also:

degree()

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(0, 1, t=0)
>>> list(G.degree_iter(0, t=0))
[(0, 1)]
>>> list(G.degree_iter([0,1], t=0))
[(0, 1), (1, 1)]
```

dynetx.DynDiGraph.in_degree

DynDiGraph.**in_degree**(*nbunch=None*, *t=None*)

Return the in degree of a node or nodes at time t.

The node in degree is the number of incoming interaction to that node in a given time frame.

Parameters

- **nbunch** (*iterable container, optional (default=all nodes)*) – A container of nodes. The container will be iterated through once.
- **t** (*snapshot id (default=None)*) – If None will be returned the degree of nodes on the flattened graph.

Returns **nd** – A dictionary with nodes as keys and degree as values or a number if a single node is specified.

Return type dictionary, or number

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(0,1, t=0)
>>> G.add_interaction(1,2, t=0)
>>> G.add_interaction(2,3, t=0)
>>> G.in_degree(0, t=0)
1
>>> G.in_degree([0,1], t=1)
{0: 0, 1: 0}
>>> list(G.in_degree([0,1], t=0).values())
[1, 2]
```

dynetx.DynDiGraph.in_degree_iter

DynDiGraph.**in_degree_iter**(*nbunch=None*, *t=None*)

Return an iterator for (node, in_degree) at time t.

The node degree is the number of edges incoming to the node in a given timeframe.

Parameters

- **nbunch** (*iterable container, optional (default=all nodes)*) – A container of nodes. The container will be iterated through once.
- **t** (*snapshot id (default=None)*) – If None will be returned an iterator over the degree of nodes on the flattened graph.

Returns **nd_iter** – The iterator returns two-tuples of (node, degree).

Return type an iterator

See also:

degree()

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(0, 1, t=0)
>>> list(G.in_degree_iter(0, t=0))
[(0, 0)]
>>> list(G.in_degree_iter([0,1], t=0))
[(0, 0), (1, 1)]
```

dynetx.DynDiGraph.out_degree

DynDiGraph.**out_degree**(*nbunch=None*, *t=None*)

Return the out degree of a node or nodes at time t.

The node degree is the number of interaction outgoing from that node in a given time frame.

Parameters

- **nbunch** (*iterable container, optional (default=all nodes)*) – A container of nodes. The container will be iterated through once.
- **t** (*snapshot id (default=None)*) – If None will be returned the degree of nodes on the flattened graph.

Returns **nd** – A dictionary with nodes as keys and degree as values or a number if a single node is specified.

Return type dictionary, or number

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interactions(0,1, t=0)
>>> G.add_interactions(1,2, t=0)
>>> G.add_interactions(2,3, t=0)
>>> G.out_degree(0, t=0)
1
>>> G.out_degree([0,1], t=1)
{0: 0, 1: 0}
```

(continues on next page)

(continued from previous page)

```
>>> list(G.out_degree([0,1], t=0).values())
[1, 2]
```

dynetx.DynDiGraph.out_degree_iter

DynDiGraph.**out_degree_iter**(*nbunch=None*, *t=None*)

Return an iterator for (node, out_degree) at time t.

The node out degree is the number of interactions outgoing from the node in a given timeframe.

Parameters

- **nbunch** (*iterable container, optional (default=all nodes)*) – A container of nodes. The container will be iterated through once.
- **t** (*snapshot id (default=None)*) – If None will be returned an iterator over the degree of nodes on the flattened graph.

Returns **nd_iter** – The iterator returns two-tuples of (node, degree).

Return type an iterator

See also:

degree()

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(0, 1, t=0)
>>> list(G.out_degree_iter(0, t=0))
[(0, 1)]
>>> list(G.out_degree_iter([0,1], t=0))
[(0, 1)]
```

dynetx.DynDiGraph.size

DynDiGraph.**size**(*t=None*)

Return the number of edges at time t.

Parameters **t** (*snapshot id (default=None)*) – If None will be returned the size of the flattened graph.

Returns **nedges** – The number of edges

Return type int

Examples

```
>>> G = dn.DynDinGraph()
>>> G.add_interaction(0,1, t=0)
>>> G.add_interaction(1,2, t=0)
>>> G.add_interaction(2,3, t=0)
```

(continues on next page)

(continued from previous page)

```
>>> G.size(t=0)
3
```

dynetx.DynDiGraph.order

DynDiGraph.**order**()

Return the number of nodes in the graph.

Returns **nnodes** – The number of nodes in the graph.

Return type int

See also:

number_of_nodes(), __len__()

dynetx.DynDiGraph.has_node

DynDiGraph.**has_node**(*n*, *t=None*)

Return True if the graph, at time t, contains the node n.

Parameters

- **n** (*node*) –
- **t** (*snapshot id (default None)*) – If None return the presence of the node in the flattened graph.

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(0, 1, t=0)
>>> G.has_node(0, t=0)
True
```

dynetx.DynDiGraph.number_of_nodes

DynDiGraph.**number_of_nodes**(*t=None*)

Return the number of nodes in the t snapshot of a dynamic graph.

Parameters **t** (*snapshot id (default=None)*) – If None return the number of nodes in the flattened graph.

Returns **nnodes** – The number of nodes in the graph.

Return type int

See also:

order()

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(0, 1, t=0)
>>> G.number_of_nodes(0)
2
```

dynetx.DynGraph.to_undirected

DynGraph.**to_undirected**(*as_view=False*)

Return an undirected copy of the graph.

Parameters **as_view** (*bool (optional, default=False)*) – If True return a view of the original undirected graph.

Returns **G** – A deepcopy of the graph.

Return type Graph/MultiGraph

See also:

`Graph(), copy(), add_edge(), add_edges_from()`

Notes

This returns a "deepcopy" of the edge, node, and graph attributes which attempts to completely copy all of the data and references.

This is in contrast to the similar *G = nx.DiGraph(D)* which returns a shallow copy of the data.

See the Python copy module for more information on shallow and deep copies, https://docs.python.org/2/library/copy.html.

Warning: If you have subclassed DiGraph to use dict-like objects in the data structure, those changes do not transfer to the Graph created by this method.

Examples

```
>>> G = nx.path_graph(2)    # or MultiGraph, etc
>>> H = G.to_directed()
>>> list(H.edges)
[(0, 1), (1, 0)]
>>> G2 = H.to_undirected()
>>> list(G2.edges)
[(0, 1)]
```

Dynamic Representation: Access Snapshots and Interactions

DynDiGraph.stream_interactions()	Generate a temporal ordered stream of interactions.
DynDiGraph.time_slice(t_from[, t_to])	Return an new graph containing nodes and interactions present in [t_from, t_to].

Continued on next page

Table 8 – continued from previous page

DynDiGraph.temporal_snapshots_ids()	Return the ordered list of snapshot ids present in the dynamic graph.
DynDiGraph.interactions_per_snapshots([t])	Return the number of interactions within snapshot t.
DynDiGraph.inter_event_time_distribution([u, v])	Return the distribution of inter event time.
DynDiGraph.inter_in_event_time_distribution([u, v])	Return the distribution of inter event time for in interactions.
DynDiGraph.inter_out_event_time_distribution([u, v])	Return the distribution of inter event time for out interactions.

dynetx.DynDiGraph.stream_interactions

DynDiGraph.**stream_interactions**()

Generate a temporal ordered stream of interactions. Only incoming interactions are returned.

Returns **nd_iter** – The iterator returns a 4-tuples of (node, node, op, timestamp).

Return type an iterator

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(0,1, t=0)
>>> G.add_interaction(1,2, t=0)
>>> G.add_interaction(2,3, t=0)
>>> G.add_interaction(3,4, t=1)
>>> G.add_interaction(4,5, t=1)
>>> G.add_interaction(5,6, t=1)
>>> list(G.stream_interactions())
[(0, 1, '+', 0), (1, 2, '+', 0), (2, 3, '+', 0), (3, 4, '+', 1), (4, 5, '+', 1),␣
↪(5, 6, '+', 1)]
```

dynetx.DynDiGraph.time_slice

DynDiGraph.**time_slice**(*t_from*, *t_to=None*)

Return an new graph containing nodes and interactions present in [t_from, t_to].

Parameters

- **t_from** (*snapshot id, mandatory*) –
- **t_to** (*snapshot id, optional (default=None)*) – If None t_to will be set equal to t_from

Returns **H** – the graph described by interactions in [t_from, t_to]

Return type a DynDiGraph object

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(0,1, t=0)
>>> G.add_interaction(1,2, t=0)
>>> G.add_interaction(2,3, t=0)
>>> G.add_interaction(0,4, t=1)
>>> G.add_interaction(4,5, t=1)
>>> G.add_interaction(5,6, t=1)
>>> G.add_interaction(7,1, t=2)
>>> G.add_interaction(1,2, t=2)
>>> G.add_interaction(2,3, t=2)
>>> H = G.time_slice(0)
>>> H.interactions()
[(0, 1), (1, 2), (1, 3)]
>>> H = G.time_slice(0, 1)
>>> H.interactions()
[(0, 1), (1, 2), (1, 3), (0, 4), (4, 5), (5, 6)]
```

dynetx.DynDiGraph.temporal_snapshots_ids

DynDiGraph.**temporal_snapshots_ids**()

Return the ordered list of snapshot ids present in the dynamic graph.

Returns **nd** – a list of snapshot ids

Return type list

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(0,1, t=0)
>>> G.add_interaction(1,2, t=0)
>>> G.add_interaction(2,3, t=0)
>>> G.add_interaction(0,4, t=1)
>>> G.add_interaction(4,5, t=1)
>>> G.add_interaction(5,6, t=1)
>>> G.add_interaction(7,1, t=2)
>>> G.add_interaction(1,2, t=2)
>>> G.add_interaction(2,3, t=2)
>>> G.temporal_snapshots_ids()
[0, 1, 2]
```

dynetx.DynDiGraph.interactions_per_snapshots

DynDiGraph.**interactions_per_snapshots**(*t=None*)

Return the number of interactions within snapshot t.

Parameters **t** (*snapshot id (default=None)*) – If None will be returned total number of interactions across all snapshots

Returns **nd** – A dictionary with snapshot ids as keys and interaction count as values or a number if a single snapshot id is specified.

Return type dictionary, or number

Examples

```
>>> G = dn.DynDiGraph()
>>> G.add_interaction(0,1, t=0)
>>> G.add_interaction(1,2, t=0)
>>> G.add_interaction(2,3, t=0)
>>> G.add_interaction(0,4, t=1)
>>> G.add_interaction(4,5, t=1)
>>> G.add_interaction(5,6, t=1)
>>> G.add_interaction(7,1, t=2)
>>> G.add_interaction(1,2, t=2)
>>> G.add_interaction(2,3, t=2)
>>> G.interactions_per_snapshots(t=0)
3
>>> G.interactions_per_snapshots()
{0: 3, 1: 3, 2: 3}
```

dynetx.DynDiGraph.inter_event_time_distribution

DynDiGraph.**inter_event_time_distribution**(*u=None*, *v=None*)

Return the distribution of inter event time. If u and v are None the dynamic graph intere event distribution is returned. If u is specified the inter event time distribution of interactions involving u is returned. If u and v are specified the inter event time distribution of (u, v) interactions is returned

Parameters

- **u** (*node id*) –
- **v** (*node id*) –

Returns **nd** – A dictionary from inter event time to number of occurrences

Return type dictionary

dynetx.DynDiGraph.inter_in_event_time_distribution

DynDiGraph.**inter_in_event_time_distribution**(*u=None*, *v=None*)

Return the distribution of inter event time for in interactions. If u and v are None the dynamic graph intere event distribution is returned. If u is specified the inter event time distribution of interactions involving u is returned. If u and v are specified the inter event time distribution of (u, v) interactions is returned

Parameters

- **u** (*node id*) –
- **v** (*node id*) –

Returns **nd** – A dictionary from inter event time to number of occurrences

Return type dictionary

dynetx.DynDiGraph.inter_out_event_time_distribution

DynDiGraph.**inter_out_event_time_distribution**(*u=None*, *v=None*)

Return the distribution of inter event time for out interactions. If u and v are None the dynamic graph intere

event distribution is returned. If u is specified the inter event time distribution of interactions involving u is returned. If u and v are specified the inter event time distribution of (u, v) interactions is returned

Parameters

- **u** (*node id*) –
- **v** (*node id*) –

Returns **nd** – A dictionary from inter event time to number of occurrences

Return type dictionary

5.2 Functions

Functional interface to graph methods and assorted utilities.

5.2.1 Graph

degree(G[, nbunch, t])	Return the degree of a node or nodes at time t.
degree_histogram(G[, t])	Return a list of the frequency of each degree value.
density(G[, t])	Return the density of a graph at timestamp t.
create_empty_copy(G[, with_data])	Return a copy of the graph G with all of the edges removed.
is_directed(G)	Return True if graph is directed.
add_star(G, nodes, t, **attr)	Add a star at time t.
add_path(G, nodes, t, **attr)	Add a path at time t.
add_cycle(G, nodes, t, **attr)	Add a cycle at time t.

dynetx.classes.function.degree

dynetx.classes.function.**degree**(*G*, *nbunch=None*, *t=None*)

Return the degree of a node or nodes at time t.

The node degree is the number of edges adjacent to that node.

Parameters

- **G** (*Graph opject*) – DyNetx graph object
- **nbunch** (*iterable container, optional (default=all nodes)*) – A container of nodes. The container will be iterated through once.
- **t** (*snapshot id (default=None)*) – If None will be returned the degree of nodes on the flattened graph.

Returns **nd** – A dictionary with nodes as keys and degree as values or a number if a single node is specified.

Return type dictionary, or number

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2,3], t=0)
>>> dn.degree(G, 0, t=0)
1
>>> dn.degree(G, [0,1], t=1)
{0: 0, 1: 0}
>>> list(dn.degree(G, [0,1], t=0).values())
[1, 2]
```

dynetx.classes.function.degree_histogram

dynetx.classes.function.**degree_histogram**(*G*, *t=None*)

Return a list of the frequency of each degree value.

Parameters **G** (*Graph opject*) – DynNetx graph object

t [snapshot id (default=None)] snapshot id

Returns **hist** – A list of frequencies of degrees. The degree values are the index in the list.

Return type list

Notes

Note: the bins are width one, hence len(list) can be large (Order(number_of_edges))

dynetx.classes.function.density

dynetx.classes.function.**density**(*G*, *t=None*)

Return the density of a graph at timestamp t. The density for undirected graphs is

$$d = \frac{2m}{n(n-1)},$$

and for directed graphs is

$$d = \frac{m}{n(n-1)},$$

where *n* is the number of nodes and *m* is the number of edges in *G*.

Parameters **G** (*Graph opject*) – DynNetx graph object

t [snapshot id (default=None)] If None the density will be computed on the flattened graph.

Notes

The density is 0 for a graph without edges and 1 for a complete graph.

Self loops are counted in the total number of edges so graphs with self loops can have density higher than 1.

dynetx.classes.function.create_empty_copy

dynetx.classes.function.**create_empty_copy**(*G*, *with_data=True*)
Return a copy of the graph G with all of the edges removed. :param G: A DyNetx graph :type G: graph :param with_data: Include data. :type with_data: bool (default=True)

Notes

Graph and edge data is not propagated to the new graph.

dynetx.classes.function.is_directed

dynetx.classes.function.**is_directed**(*G*)
Return True if graph is directed.

dynetx.classes.function.add_star

dynetx.classes.function.**add_star**(*G*, *nodes*, *t*, ***attr*)
Add a star at time t.

The first node in nodes is the middle of the star. It is connected to all other nodes.

Parameters

- **G** (*graph*) – A DyNetx graph
- **nodes** (*iterable container*) – A container of nodes.
- **t** (*snapshot id (default=None)*) – snapshot id

See also:

add_path(), *add_cycle()*

Examples

```
>>> G = dn.DynGraph()
>>> dn.add_star(G, [0,1,2,3], t=0)
```

dynetx.classes.function.add_path

dynetx.classes.function.**add_path**(*G*, *nodes*, *t*, ***attr*)
Add a path at time t.

Parameters

- **G** (*graph*) – A DyNetx graph
- **nodes** (*iterable container*) – A container of nodes.
- **t** (*snapshot id (default=None)*) – snapshot id

See also:

add_path(), *add_cycle()*

Examples

```
>>> G = dn.DynGraph()
>>> dn.add_path(G, [0,1,2,3], t=0)
```

dynetx.classes.function.add_cycle

dynetx.classes.function.**add_cycle**(*G*, *nodes*, *t*, ***attr*)

Add a cycle at time t.

Parameters

- **G** (*graph*) – A DyNetx graph
- **nodes** (*iterable container*) – A container of nodes.
- **t** (*snapshot id (default=None)*) – snapshot id

See also:

add_path(), *add_cycle()*

Examples

```
>>> G = dn.DynGraph()
>>> dn.add_cycle(G, [0,1,2,3], t=0)
```

5.2.2 Nodes

nodes(G[, t])	Return a list of the nodes in the graph at a given snapshot.
number_of_nodes(G[, t])	Return the number of nodes in the t snpashot of a dynamic graph.
all_neighbors(graph, node[, t])	Returns all of the neighbors of a node in the graph at time t.
non_neighbors(graph, node[, t])	Returns the non-neighbors of the node in the graph at time t.

dynetx.classes.function.nodes

dynetx.classes.function.**nodes**(*G*, *t=None*)

Return a list of the nodes in the graph at a given snapshot.

Parameters

- **G** (*Graph opject*) – DyNetx graph object
- **t** (*snapshot id (default=None)*) – If None the the method returns all the nodes of the flattened graph.

Returns **nlist** – A list of nodes. If data=True a list of two-tuples containing (node, node data dictionary).

Return type list

Examples

```
>>> G = dn.DynGraph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2], 0)
>>> dn.nodes(G, t=0)
[0, 1, 2]
>>> G.add_edge(1, 4, t=1)
>>> dn.nodes(G, t=0)
[0, 1, 2]
```

dynetx.classes.function.number_of_nodes

dynetx.classes.function.**number_of_nodes**(*G*, *t=None*)

Return the number of nodes in the t snpashot of a dynamic graph.

Parameters

- **G** (*Graph opject*) – DyNetx graph object
- **t** (*snapshot id (default=None)*) – If None return the number of nodes in the flattened graph.

Returns **nnodes** – The number of nodes in the graph.

Return type int

See also:

order()

Examples

```
>>> G = dn.DynGraph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2], t=0)
>>> dn.number_of_nodes(G, 0)
3
```

dynetx.classes.function.all_neighbors

dynetx.classes.function.**all_neighbors**(*graph*, *node*, *t=None*)

Returns all of the neighbors of a node in the graph at time t.

If the graph is directed returns predecessors as well as successors.

Parameters

- **graph** (*DyNetx graph*) – Graph to find neighbors.
- **node** (*node*) – The node whose neighbors will be returned.
- **t** (*snapshot id (default=None)*) – If None the neighbors are identified on the flattened graph.

Returns **neighbors** – Iterator of neighbors

Return type iterator

dynetx.classes.function.non_neighbors

dynetx.classes.function.**non_neighbors**(*graph*, *node*, *t=None*)

Returns the non-neighbors of the node in the graph at time t.

Parameters

- **graph** (*DyNetx graph*) – Graph to find neighbors.
- **node** (*node*) – The node whose neighbors will be returned.
- **t** (*snapshot id (default=None)*) – If None the non-neighbors are identified on the flattened graph.

Returns **non_neighbors** – Iterator of nodes in the graph that are not neighbors of the node.

Return type iterator

5.2.3 Interactions

interactions(G[, nbunch, t])	Return the list of edges present in a given snapshot.
number_of_interactions(G[, u, v, t])	Return the number of edges between two nodes at time t.
non_interactions(graph[, t])	Returns the non-existent edges in the graph at time t.

dynetx.classes.function.interactions

dynetx.classes.function.**interactions**(*G*, *nbunch=None*, *t=None*)

Return the list of edges present in a given snapshot.

Edges are returned as tuples in the order (node, neighbor).

Parameters

- **G** (*Graph opject*) – DyNetx graph object
- **nbunch** (*iterable container, optional (default= all nodes)*) – A container of nodes. The container will be iterated through once.
- **t** (*snapshot id (default=None)*) – If None the the method returns all the edges of the flattened graph.

Returns **edge_list** – Edges that are adjacent to any node in nbunch, or a list of all edges if nbunch is not specified.

Return type list of edge tuples

Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored. For directed graphs this returns the out-edges.

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2], t=0)
```

(continues on next page)

(continued from previous page)

```
>>> G.add_edge(2,3, t=1)
>>> dn.interactions(G, t=0)
[(0, 1), (1, 2)]
>>> dn.interactions(G)
[(0, 1), (1, 2), (2, 3)]
>>> dn.interactions(G, [0,3], t=0)
[(0, 1)]
```

dynetx.classes.function.number_of_interactions

dynetx.classes.function.**number_of_interactions**(*G*, *u=None*, *v=None*, *t=None*)

Return the number of edges between two nodes at time t.

Parameters

- **v** (*u,*) – If u and v are specified, return the number of edges between u and v. Otherwise return the total number of all edges.
- **t** (*snapshot id (default=None)*) – If None will be returned the number of edges on the flattened graph.

Returns **nedges** – The number of edges in the graph. If nodes u and v are specified return the number of edges between those nodes.

Return type int

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2,3], t=0)
>>> dn.number_of_interactions(G, t=0)
```

dynetx.classes.function.non_interactions

dynetx.classes.function.**non_interactions**(*graph*, *t=None*)

Returns the non-existent edges in the graph at time t.

Parameters

- **graph** (*NetworkX graph.*) – Graph to find non-existent edges.
- **t** (*snapshot id (default=None)*) – If None the non-existent edges are identified on the flattened graph.

Returns **non_edges** – Iterator of edges that are not in the graph.

Return type iterator

5.2.4 Freezing graph structure

freeze(G)	Modify graph to prevent further change by adding or removing nodes or edges.

Continued on next page

Table 12 – continued from previous page

is_frozen(G)	Return True if graph is frozen.

dynetx.classes.function.freeze

dynetx.classes.function.**freeze**(*G*)

Modify graph to prevent further change by adding or removing nodes or edges.

Node and edge data can still be modified.

Parameters **G** (*graph*) – A NetworkX graph

Notes

To "unfreeze" a graph you must make a copy by creating a new graph object.

See also:

is_frozen()

dynetx.classes.function.is_frozen

dynetx.classes.function.**is_frozen**(*G*)

Return True if graph is frozen.

Parameters **G** (*graph*) – A DyNetx graph

See also:

freeze()

5.2.5 Snapshots and Interaction Stream

stream_interactions(G)	Generate a temporal ordered stream of interactions.
time_slice(G, t_from[, t_to])	Return an iterator for (node, degree) at time t.
temporal_snapshots_ids(G)	Return the ordered list of snapshot ids present in the dynamic graph.
interactions_per_snapshots(G[, t])	Return the number of interactions within snapshot t.
inter_event_time_distribution(G[, u, v])	Return the distribution of inter event time.

dynetx.classes.function.stream_interactions

dynetx.classes.function.**stream_interactions**(*G*)

Generate a temporal ordered stream of interactions.

Parameters **G** (*graph*) – A DyNetx graph.

Returns **nd_iter** – The iterator returns a 4-tuples of (node, node, op, timestamp).

Return type an iterator

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2,3], t=0)
>>> G.add_path([3,4,5,6], t=1)
>>> list(dn.stream_interactions(G))
[(0, 1, '+', 0), (1, 2, '+', 0), (2, 3, '+', 0), (3, 4, '+', 1), (4, 5, '+', 1),␣
↪(5, 6, '+', 1)]
```

dynetx.classes.function.time_slice

dynetx.classes.function.**time_slice**(*G*, *t_from*, *t_to=None*)

Return an iterator for (node, degree) at time t.

The node degree is the number of edges adjacent to the node.

Parameters

- **G** (*graph*) – A DyNetx graph.
- **t_from** (*snapshot id, mandatory*) –
- **t_to** (*snapshot id, optional (default=None)*) – If None t_to will be set equal to t_from

Returns **H** – the graph described by interactions in [t_from, t_to]

Return type a DynGraph object

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2,3], t=0)
>>> G.add_path([0,4,5,6], t=1)
>>> G.add_path([7,1,2,3], t=2)
>>> H = dn.time_slice(G, 0)
>>> H.edges()
[(0, 1), (1, 2), (1, 3)]
>>> H = dn.time_slice(G, 0, 1)
>>> H.edges()
[(0, 1), (1, 2), (1, 3), (0, 4), (4, 5), (5, 6)]
```

dynetx.classes.function.temporal_snapshots_ids

dynetx.classes.function.**temporal_snapshots_ids**(*G*)

Return the ordered list of snapshot ids present in the dynamic graph.

Parameters **G** (*graph*) – A DyNetx graph.

Returns **nd** – a list of snapshot ids

Return type list

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2,3], t=0)
>>> G.add_path([0,4,5,6], t=1)
>>> G.add_path([7,1,2,3], t=2)
>>> dn.temporal_snapshots(G)
[0, 1, 2]
```

dynetx.classes.function.interactions_per_snapshots

dynetx.classes.function.**interactions_per_snapshots**(*G*, *t=None*)

Return the number of interactions within snapshot t.

Parameters

- **G** (*graph*) – A DyNetx graph.
- **t** (*snapshot id (default=None)*) – If None will be returned total number of interactions across all snapshots

Returns **nd** – A dictionary with snapshot ids as keys and interaction count as values or a number if a single snapshot id is specified.

Return type dictionary, or number

Examples

```
>>> G = dn.DynGraph()
>>> G.add_path([0,1,2,3], t=0)
>>> G.add_path([0,4,5,6], t=1)
>>> G.add_path([7,1,2,3], t=2)
>>> dn.number_of_interactions(G, t=0)
3
>>> dn.interactions_per_snapshots(G)
{0: 3, 1: 3, 2: 3}
```

dynetx.classes.function.inter_event_time_distribution

dynetx.classes.function.**inter_event_time_distribution**(*G*, *u=None*, *v=None*)

Return the distribution of inter event time. If u and v are None the dynamic graph intere event distribution is returned. If u is specified the inter event time distribution of interactions involving u is returned. If u and v are specified the inter event time distribution of (u, v) interactions is returned

Parameters

- **G** (*graph*) – A DyNetx graph.
- **u** (*node id*) –
- **v** (*node id*) –

Returns **nd** – A dictionary from inter event time to number of occurrences

Return type dictionary

5.3 Reading and writing graphs

5.3.1 Edge List

Read and write DyNetx graphs as edge lists.

The multi-line adjacency list format is useful for graphs with nodes that can be meaningfully represented as strings.

With the edgelist format simple edge data can be stored but node or graph data is not. There is no way of representing isolated nodes unless the node has a self-loop edge.

Format

You can read or write three formats of edge lists with these functions.

Node pairs with **timestamp** (u, v, t):

```
>>> 1 2 0
```

Sequence of **Interaction** events (u, v, +/-, t):

```
>>> 1 2 + 0
>>> 1 2 - 3
```

Interaction Graph

write_interactions(G, path[, delimiter, ...])	Write a DyNetx graph in interaction list format.
read_interactions(path[, comments, ...])	Read a DyNetx graph from interaction list format.

dynetx.readwrite.edgelist.write_interactions

dynetx.readwrite.edgelist.**write_interactions**(*G*, *path*, *delimiter=' '*, *encoding='utf-8'*)

Write a DyNetx graph in interaction list format.

Parameters

- **G** (*graph*) – A DyNetx graph.
- **path** (*basestring*) – The desired output filename
- **delimiter** (*character*) – Column delimiter

dynetx.readwrite.edgelist.read_interactions

dynetx.readwrite.edgelist.**read_interactions**(*path*, *comments='#'*, *directed=False*, *delimiter=None*, *nodetype=None*, *timestamptype=None*, *encoding='utf-8'*, *keys=False*)

Read a DyNetx graph from interaction list format.

Parameters

- **path** (*basestring*) – The desired output filename
- **delimiter** (*character*) – Column delimiter

Snapshot Graphs

write_snapshots(G, path[, delimiter, encoding])	Write a DyNetx graph in snapshot graph list format.
read_snapshots(path[, comments, directed, ...])	Read a DyNetx graph from snapshot graph list format.

dynetx.readwrite.edgelist.write_snapshots

dynetx.readwrite.edgelist.**write_snapshots**(*G*, *path*, *delimiter=' '*, *encoding='utf-8'*)

Write a DyNetx graph in snapshot graph list format.

Parameters

- **G** (*graph*) – A DyNetx graph.
- **path** (*basestring*) – The desired output filename
- **delimiter** (*character*) – Column delimiter

dynetx.readwrite.edgelist.read_snapshots

dynetx.readwrite.edgelist.**read_snapshots**(*path*, *comments='#'*, *directed=False*, *delimiter=None*, *nodetype=None*, *timestamptype=None*, *encoding='utf-8'*, *keys=False*)

Read a DyNetx graph from snapshot graph list format.

Parameters

- **path** (*basestring*) – The desired output filename
- **delimiter** (*character*) – Column delimiter

5.3.2 JSON

JSON data

Generate and parse JSON serializable data for DyNetx graphs.

node_link_data(G[, attrs])	Return data in node-link format that is suitable for JSON serialization and use in Javascript documents.
node_link_graph(data[, directed, attrs])	Return graph from node-link data format.

dynetx.readwrite.json_graph.node_link_data

dynetx.readwrite.json_graph.**node_link_data**(*G*, *attrs={'id': 'id', 'source': 'source', 'target': 'target'}*)

Return data in node-link format that is suitable for JSON serialization and use in Javascript documents.

Parameters

- **G** (*DyNetx graph*) –
- **attrs** (*dict*) –

  A dictionary that contains three keys 'id', 'source' and 'target'. The corresponding values provide the attribute names for storing

DyNetx-internal graph data. The values should be unique. Default value: `dict(id='id', source='source', target='target')`.

Returns **data** – A dictionary with node-link formatted data.

Return type dict

Examples

```
>>> from dynetx.readwrite import json_graph
>>> G = dn.DynGraph([(1,2)])
>>> data = json_graph.node_link_data(G)
```

To serialize with json

```
>>> import json
>>> s = json.dumps(data)
```

Notes

Graph, node, and link attributes are stored in this format. Note that attribute keys will be converted to strings in order to comply with JSON.

See also:

node_link_graph()

dynetx.readwrite.json_graph.node_link_graph

dynetx.readwrite.json_graph.**node_link_graph**(*data*, *directed=False*, *attrs={'id': 'id', 'source': 'source', 'target': 'target'}*)

Return graph from node-link data format.

Parameters

- **data** (*dict*) – node-link formatted graph data
- **directed** (*bool*) – If True, and direction not specified in data, return a directed graph.
- **attrs** (*dict*) – A dictionary that contains three keys 'id', 'source', 'target'. The corresponding values provide the attribute names for storing Dynetx-internal graph data. Default value: `dict(id='id', source='source', target='target')`.

Returns **G** – A DyNetx graph object

Return type DyNetx graph

Examples

```
>>> from dynetx.readwrite import json_graph
>>> G = dn.DynGraph([(1,2)])
>>> data = json_graph.node_link_data(G)
>>> H = json_graph.node_link_graph(data)
```

See also:

node_link_data()

Python Module Index

d

Index

Symbols

A

C

D

F

H

I

N

O

P

R

S

T

W