DyNetworkX Documentation

Release 0.1

Makan Arastuie

Mar 13, 2020

Contents

DyNetworkX is a Python package for the study of dynamic network analysis (DNA). It is a fork of NetworkX package. Thus, implementation, documentation and the development of DyNetworkX is heavily influenced by NetworkX.

DyNetworkX provides

- tools for the study of the structure of dynamic networks.
- all dynamic graph types can be converted to one or more of NetworkX graph types, allowing access to a verity of network algorithms.

CHAPTER 1

Audience

The audience for DyNetworkX includes mathematicians, physicists, biologists, computer scientists, and social scientists. Overall, everyone interested in analyzing dynamic networks.

CHAPTER 2

Python

Python is a powerful programming language that allows simple and flexible representations of networks as well as clear and concise expressions of network algorithms. Python has a vibrant and growing ecosystem of packages that NetworkX uses to provide more features such as numerical linear algebra and drawing. In order to make the most out of NetworkX you will want to know how to write basic programs in Python. Among the many guides to Python, we recommend the Python documentation.

CHAPTER 3

Free software

Released under the 3-Clause BSD license. More information can be found under Licence.

CHAPTER 4

History

DyNetworkX is developed by IDEAS Lab @ The University of Toledo.

CHAPTER 5

Documentation

5.1 Install

Just like NetworkX, DyNetworkX requires Python 3.4, 3.5, or 3.6.

Below we assume you have the default Python environment already configured on your computer and you intend to install `networkx` inside of it. If you want to create and work with Python virtual environments, please follow instructions on venv and virtual environments.

First, make sure you have the latest version of `pip` (the Python package manager) installed. If you do not, refer to the Pip documentation and install `pip` first.

5.1.1 Note

DyNetworkX is not yet available for `pip` install. If you are interested in this DyNetworkX, contact us from *Need Help?* and we will keep you updated.

5.2 Tutorial

This guide can help you start working with IntervalGraph module of DyNetworkX.

Disclaimer: this tutorial, similar to DyNetworkX itself, is heavily influenced by NetworkX's tutorial. This is done on purpose, in order to point out the similarities between the two packages.

5.2.1 Creating an interval graph

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

```
>>> import dynetworkx as dnx
>>> IG = dnx.IntervalGraph()
```

By definition, an *IntervalGraph* is a collection of nodes (vertices) along with identified pairs of nodes (called interval edges, edges, links, etc) each of which is coupled with a given interval. In DyNetworkX, just like NetworkX, nodes can be any hashable object e.g., a text string, an image, an XML object, another Graph, a customized node object, etc.

Note: Python's `None` object should not be used as a node as it determines whether optional function arguments have been assigned in many functions.

5.2.2 Nodes

Using DyNetworkX's *IntervalGraph.load_from_txt()* method, the graph `IG` can be grown by importing an existing network. However, we first look at simple ways to manipulate an interval graph. The simplest form is adding a single node,

```
>>> IG.add_node(1)
```

add a list of nodes,

```
>>> IG.add_nodes_from([2, 3])
```

or add any iterable container of nodes. You can also add nodes along with node attributes if your container yields 2-tuples (node, node_attribute_dict). Node attributes are discussed further below.

```
>>> H = dnx.IntervalGraph()
>>> IG.add_node(H)
```

Note that interval graph `IG` now contains interval graph `H` as a node. This flexibility is very powerful as it allows graphs of graphs, graphs of files, graphs of functions and much more. It is worth thinking about how to structure your application so that the nodes are useful entities. Of course you can always use a unique identifier in `IG` and have a separate dictionary keyed by identifier to the node information if you prefer.

Note: You should not change the node object if the hash depends on its contents.

5.2.3 Edges

Edges are what make an interval graph possible. Every edge is defined by 2 nodes, the inclusive beginning of the interval when the edge first appears and its non-inclusive end. Beginning of an interval must be strictly smaller than its end and both can be of any orderable types.

Note: In this tutotial as well as IntervalGraph documentation, the two terms `edge` and `interval edge` are used interchangeably.

`IG` can also be grown by adding one edge at a time,

```
>>> IG.add_edge(1, 2, 1, 4) # n1, n2, beginning, end of the edge interval
>>> ie = (2, 3, 2, 5)
>>> IG.add_edge(*ie) # unpack interval edge tuple*
```

by adding a list of edges,

```
>>> IG.add_edges_from([(1, 2, 2, 6), (1, 3, 6, 9)])
```

or by adding any *ebunch* of edges. An *ebunch* is any iterable container of interval edge-tuples. An interval edge-tuple is a 4-tuple of nodes and intervals.

Note: In above example it is worth noting that the two added interval edges, `(1, 2, 1, 4)` and `(1, 2, 2, 6)` are two different interval edges, since they exists on different intervals.

If a new interval edge is to be added with nodes that are not currently in the interval graph, nodes will be added automatically.

There are no complaints when adding existing nodes or edges. As we add new nodes/edges, DyNetworkX quietly ignores any that are already present.

```
>>> IG.add_edge(1, 2, 1, 4)
>>> IG.add_node(1)
```

At this stage the interval graph `IG` consist of 4 nodes and 4 edges,

```
>>> IG.number_of_nodes()
4
>>> len(IG.edges())
4
```

We can examine nodes and edges with two interval graph methods which facilitate reporting: *`IntervalGraph.nodes()`* and *`IntervalGraph.edges()`*. These are lists of the nodes and interval edges. They offer a continually updated read-only view into the graph structure.

```
>>> IG.nodes()
[1, 2, 3, <dynetworkx.classes.intervalgraph.IntervalGraph object at 0x100000000>]
```

`IG.edges()` is an extremely flexible and useful method to query the interval graph for various interval edges. It returns a list of Interval objects which are in the form `Interval(begin, end, (node_1, node_2)`.

Using this method you have access to 4 constraints in order to restrict your query. *u*, *v*, *begin* and *end*. Defining any of them narrows down your query.

```
>>> IG.edges() # returns a list of all edges
[Interval(6, 9, (1, 3)), Interval(2, 5, (2, 3)), Interval(2, 6, (1, 2)), Interval(1,␣
↪4, (1, 2))]
>>> IG.edges(begin=5) # all edges which have an overlapping interval with interval [5,
↪ end of the interval graph]
[Interval(6, 9, (1, 3)), Interval(2, 6, (1, 2))]
>>> IG.edges(end=3) # all edges which have an overlapping interval with interval␣
↪[beginning of the interval graph, 3)
[Interval(2, 5, (2, 3)), Interval(2, 6, (1, 2)), Interval(1, 4, (1, 2))]
>>> IG.edges(u=1, v=2) # all edge between nodes 1 and 2
[Interval(2, 6, (1, 2)), Interval(1, 4, (1, 2))]
>>> IG.edges(1, 2, 5, 6) # all edges between nodes 1 and 2 which have an overlapping␣
↪interval with [5, 6)
[Interval(2, 6, (1, 2))]
```

One can also take advantage of this method to obtain more information such as *degree*. Since in an interval graph these parameters change depending on the interval in question, you need to adjust your query.

Accessing *degree* of a node:

```
>>> len(IG.edges(u=1)) # total number of edges associated with node 1 over the entire␣
↪interval
3
>>> len(IG.edges(u=1, begin=2, end=4)) # Adding interval restriction
2
```

Keep in mind that `end` is non-inclusive. Thus, depening on what time increment you use to define your interval, if you set `end = begin + smallest_increment` it will return all the edges which are present at time `begin`.

```
>>> len(IG.edges(u=1, begin=5, end=6))
1
```

If you are using a truly continuous time interval, you can add your machine epsilon to `begin` to achieve the same result. As an example:

```
>>> import numpy as np
>>> eps = np.finfo(np.float64).eps
>>> begin = 5
>>> IG.edges(u=1, begin=begin, end=begin + eps)
[Interval(2, 6, (1, 2))]
```

As it is shown, `IG.edges()` is a powerful method to query the network for edges. You can also take advantage of *`IntervalGraph.has_node()`* and *`IntervalGraph.has_edge()`* as it is shown below,

```
>>> IG.has_node(3)
True
>>> 1 in IG # this is equivalent to IG.has_node(1)
True
>>> IG.has_node(5)
False
>>> IG.has_edge(2, 3)
True
>>> IG.has_edge(1, H)
False
```

Or constraint the begin and/or end of your search:

```
>>> IG.has_node(3, end=2) # end is non-inclusive
False
```

```
>>> IG.has_edge(2, 3, 3, 7) # matching an interval edge with nodes 2 and 3, and␣
↪overlapping interval [3, 7)
True
>>> IG.has_edge(2, 3, 3, 7, overlapping=False) # setting overlapping=False, searches␣
↪for an exact interval match
False
```

One can remove nodes and edges from the graph in a similar fashion to adding. by using *`IntervalGraph.remove_node()`* and *`IntervalGraph.remove_edge()`*, e.g.

```
>>> IG.remove_node(H)
[1, 2, 3]
>>> IG.remove_edge(1, 3, 6, 9, overlapping=False)
>>> IG.edges()
[Interval(2, 5, (2, 3)), Interval(2, 6, (1, 2)), Interval(1, 4, (1, 2))]
```

5.2.4 What to use as nodes and edges

Just like NetworkX, DyNetworkX does not have a specific type for nodes an edges. This allows you to represent nodes and edges with any hashable object to add more depth and meanning to your interval graph. The most common choices are numbers or strings, but a node can be any hashable object (except `None`), and an edge can be associated with any object `x` using `IG.add_edge(n1, n2, begin, end, object=x)`.

As an example, `n1` and `n2` could be real people's profile url or a custom python object and `x` can be another python object which describes the detail of their contact. This way, you are not bound to only associating weights with the edges.

Based on the NetworkX's experience, this is quite useful, but its abuse can lead to unexpected surprises unless one is familiar with Python.

5.2.5 Adding attributes to graphs, nodes, and edges

Attributes such as weights, labels, colors, or whatever Python object you like, can be attached to graphs, nodes, or edges.

Each graph, node, and edge can hold key/value attribute pairs in an associated attribute dictionary (the keys must be hashable). By default these are empty, but attributes can be added or changed using `add_edge`, `add_node`.

Graph attributes

Assign graph attributes when creating a new graph,

```
>>> IG = dnx.IntervalGraph(state='Ohio')
>>> IG.graph
{'state': 'Ohio'}
```

Or you can modify attributes later,

```
>>> IG.graph['state'] = 'Michigan'
>>> IG.graph
{'state': 'Michigan'}
```

There is also an spacial attribute for interval graphs called `name`. You can either set it just like any other attribute or you can take advantage of the `IG.name` property:

```
>>> IG.name = "USA"
>>> IG.name
USA
```

Node attributes

Add node attributes using `add_node()` or `add_nodes_from()`,

```
>>> IG.add_node(1, time='5pm', day="Friday") # Adds node 1 and sets its two attributes
>>> IG.add_nodes_from([2, 3], time='2pm') # Adds nodes 2 and 3 and sets both of their
↪'time' attributes to '2pm'
>>> IG.add_node(1, time='10pm') # Updates node 1's 'time' attribute to '10pm'
```

Note that you can update a node's attribute by adding the node and setting a new value for its attribute.

Edge attributes

Similarly, add/change edge attributes using `add_edge()` or `add_edges_from()`,

```
>>> G.add_edge(1, 2, 4, 6, contact_type='call') # Adds the edge and sets its 'contact_
↪type' attribute.
>>> G.add_edges_from([(3, 4, 1, 5), (1, 2, 4, 6)], weight=5.8)
>>> G.add_edge(1, 2, 4, 6, weight=6.6) # Updates the weight attribute of the edge.
```

Note that updating an edge's attribute is similar to updating nodes' attributes.

5.2.6 Subgraphs and snapshots

You can create one, or a series of snapshots of, NetworkX *Graph* or *MultiGraph* from an interval graph if you wish to analyze a portion, or your entire interval graph, using well-known static network algorithms that are available in NetworkX.

Subgraphs

To extract a portion of an interval graph, given an interval, you can utilize *`IntervalGraph.to_subgraph()`*,

```
>>> IG = dnx.IntervalGraph()
>>> IG.add_edges_from([(1, 2, 3, 10), (2, 4, 1, 11), (6, 4, 12, 19), (2, 4, 8, 15)])
>>> H = IG.to_subgraph(4, 12)
>>> type(H)
<class 'networkx.classes.graph.Graph'>
>>> list(H.edges(data=True))
[(1, 2, {}), (2, 4, {})]
```

Note that you can also use *`IntervalGraph.interval()`* to get the interval for the entire interval graph, and use that to convert an interval graph to a NetworkX Graph.

You can also keep the information about each edge's interval as attributes on the NetworkX's Graph:

```
>>> H = G.to_subgraph(4, 12, edge_interval_data=True)
>>> type(H)
<class 'networkx.classes.graph.Graph'>
>>> list(H.edges(data=True))
[(1, 2, {'end': 10, 'begin': 3}), (2, 4, {'end': 15, 'begin': 8})]
```

Notice that if there are multiple edges available between two nodes, the interval information is going to reflect only one of the edges. Another option is to retrieve a *MultiGraph* to lose less information in the conversion process:

```
>>> M = G.to_subgraph(4, 12, multigraph=True, edge_interval_data=True)
>>> type(M)
<class 'networkx.classes.multigraph.MultiGraph'>
>>> list(M.edges(data=True))
[(1, 2, {'end': 10, 'begin': 3}), (2, 4, {'end': 11, 'begin': 1}), (2, 4, {'end': 15,
↪'begin': 8})]
```

Snapshots

A more traditional method of analyzing continuous dynamic networks has been dividing the network into a series of fixed-interval snapshots. Although some information will be lost in the conversion due to the classic limitations of representing a continuous network in a discrete format, you will gain access to numerous well-defined algorithms which do not exist for continuous networks.

To do so, you can simply use `IntervalGraph.to_snapshots()` and set the number of snapshots you wish to divided the network into:

```
>>> S, l = G.to_snapshots(2, edge_interval_data=True, return_length=True)
>>> S # a list of NetworkX Graphs
[<networkx.classes.graph.Graph object at 0x100000>, <networkx.classes.graph.Graph␣
↪object at 0x150d00>]
>>> l # length of the interval of a single snapshot
9.0
>>> for g in S:
>>> ... g.edges(data=True))
[(1, 2, {'begin': 3, 'end': 10}), (2, 4, {'begin': 8, 'end': 15})]
[(2, 4, {'begin': 8, 'end': 15}), (4, 6, {'begin': 12, 'end': 19})]
```

Combining this method with `SnapshotGraph` can be a powerful tool to gain access to all the methods available through DyNetworkX's `SnapshotGraph`.

Similar to *to_subgraph* method, you can also divide the interval graph into a series of NetworkX's *MultiGraph*, if that is what you need.

5.2.7 Importing from text file

Using *load_from_txt* you can also read in an IntervalGraph or ImpulseGraph from a text file in a specific edge-list format. For more detail checkout the documentation on `IntervalGraph.load_from_txt()`.

5.2.8 Saving to text file

Using *save_to_txt* you can also write an IntervalGraph or ImpulseGraph to a text file in a specific edge-list format. For more detail checkout the documentation on `IntervalGraph.save_to_txt()`.

5.3 Download

5.3.1 Software

DyNetworkX package is still under development as of now, and there is no stable version available through PyPI. However, if you wish to try the dev version for yourself, you can fork the github repository .

5.4 Reference

Date Mar 13, 2020

5.4.1 Introduction

The structure of DyNetworkX closely (and intentionally) resembles the structure of NetworkX, since it is a fork of NetworkX.

DyNetworkX Basics

After starting Python, import the dynetworkx module with (the recommended way)

```
>>> import dynetworkx as dnx
```

To save repetition, in the documentation we assume that DyNetworkX has been imported this way.

If importing networkx fails, it means that Python cannot find the installed module. Check your installation and your `PYTHONPATH`.

The following basic graph types are provided as Python classes:

`IntervalGraph` This class implements an undirected interval graph. Each edge must have a beginning and ending as an interval. It ignores multiple edges (edges with the same nodes and interval) between two nodes. It does allow self-loop edges between a node and itself.

`SnapshotGraph` This class implements an easy way to gain access to a list of NetworkX networks and provides various methods to interact, manipulate and analyze the networks.

5.4.2 Graph Types

Interval Graph

Overview

`class` `dynetworkx.`**`IntervalGraph`**`(`***attr*`)`

Base class for undirected interval graphs.

The IntervalGraph class allows any hashable object as a node and can associate key/value attribute pairs with each undirected edge.

Each edge must have two integers, begin and end for its interval.

Self-loops are allowed but multiple edges (two or more edges with the same nodes, begin and end interval) are not.

Two nodes can have more than one edge with different overlapping or non-overlapping intervals.

Parameters **`attr`** (*`keyword arguments, optional (default= no attributes)`*) – Attributes to add to graph as key=value pairs.

Examples

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

```
>>> G = dnx.IntervalGraph()
```

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).

Add the nodes from any container (a list, dict, set)

```
>>> G.add_nodes_from([2, 3])
>>> G.add_nodes_from(range(100, 110))
```

Edges:

G can also be grown by adding edges. This can be considered the primary way to grow G, since nodes with no edge will not appear in G in most cases. See `G.to_snapshot()`.

Add one edge, which starts at 0 and ends at 10. Keep in mind that the interval is [0, 10). Thus, it does not include the end.

```
>>> G.add_edge(1, 2, 0, 10)
```

a list of edges,

```
>>> G.add_edges_from([(1, 2, 0, 10), (1, 3, 3, 11)])
```

If some edges connect nodes not yet in the graph, the nodes are added automatically. There are no errors when adding nodes or edges that already exist.

Attributes:

Each interval graph, node, and edge can hold key/value attribute pairs in an associated attribute dictionary (the keys must be hashable). By default these are empty, but can be added or changed using add_edge, add_node.

Keep in mind that the edge interval is not an attribute of the edge.

```
>>> G = dnx.IntervalGraph(day="Friday")
>>> G.graph
{'day': 'Friday'}
```

Add node attributes using add_node(), add_nodes_from()

```
>>> G.add_node(1, time='5pm')
>>> G.add_nodes_from([3], time='2pm')
```

Add edge attributes using add_edge(), add_edges_from().

```
>>> G.add_edge(1, 2, 0, 10, weight=4.7 )
>>> G.add_edges_from([(3, 4, 3, 11), (4, 5, 0, 33)], color='red')
```

Shortcuts:

Here are a couple examples of available shortcuts:

```
>>> 1 in G   # check if node in interval graph during any interval
True
>>> len(G)   # number of nodes in the entire interval graph
5
```

Subclasses (Advanced): Edges in interval graphs are represented by Interval Objects and are kept in an IntervalTree. Both are based on intervaltree available in pypi (https://pypi.org/project/intervaltree). IntervalTree allows for fast interval based search through edges, which makes interval graph analysis possible.

The Graph class uses a dict-of-dict-of-dict data structure. The outer dict (node_dict) holds adjacency information keyed by nodes. The next dict (adjlist_dict) represents the adjacency information and holds edge data keyed by interval objects. The inner dict (edge_attr_dict) represents the edge data and holds edge attribute values keyed by attribute names.

Methods

Adding and removing nodes and edges

`IntervalGraph.__init__`(**attr)	Initialize an interval graph with edges, name, or graph attributes.
`IntervalGraph.add_node`(node_for_adding, **attr)	Add a single node *node_for_adding* and update node attributes.
`IntervalGraph.add_nodes_from`(. . .)	Add multiple nodes.
`IntervalGraph.remove_node`(n[, begin, end])	Remove the presence of a node n within the given interval.
`IntervalGraph.add_edge`(u, v, begin, end, **attr)	Add an edge between u and v, during interval [begin, end).
`IntervalGraph.add_edges_from`(ebunch_to_add, . . .)	Add all the edges in ebunch_to_add.
`IntervalGraph.remove_edge`(u, v[, begin, . . .])	Remove the edge between u and v in the interval graph, during the given interval.

dynetworkx.IntervalGraph.__init__

`IntervalGraph.`**`__init__`**`(`***attr*`)`

Initialize an interval graph with edges, name, or graph attributes.

Parameters **`attr`** (*`keyword arguments, optional (default= no attributes)`*) – Attributes to add to graph as key=value pairs.

Examples

```
>>> G = dnx.IntervalGraph()
>>> G = dnx.IntervalGraph(name='my graph')
>>> G.graph
{'name': 'my graph'}
```

dynetworkx.IntervalGraph.add_node

`IntervalGraph.`**`add_node`**`(`*node_for_adding*, ***attr*`)`

Add a single node *node_for_adding* and update node attributes.

Parameters

- **`node_for_adding`** (*`node`*) – A node can be any hashable Python object except None.
- **`attr`** (*`keyword arguments, optional`*) – Set or change node attributes using key=value.

See also:

`add_nodes_from()`

Examples

```
>>> G = dnx.IntervalGraph()
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> G.number_of_nodes()
2
```

Use keywords set/change node attributes:

```
>>> G.add_node(1, size=10)
>>> G.add_node(3, weight=0.4, UTM=('13S', 382871, 3972649))
```

Notes

A hashable object is one that can be used as a key in a Python dictionary. This includes strings, numbers, tuples of strings and numbers, etc.

On many platforms hashable items also include mutables such as NetworkX Graphs, though one should be careful that the hash doesn't change on mutables.

dynetworkx.IntervalGraph.add_nodes_from

`IntervalGraph.`**`add_nodes_from`**(*nodes_for_adding*, ***attr*)

Add multiple nodes.

Parameters

- **`nodes_for_adding`** (*`iterable container`*) – A container of nodes (list, dict, set, etc.). OR A container of (node, attribute dict) tuples. Node attributes are updated using the attribute dict.
- **`attr`** (*`keyword arguments, optional (default= no attributes)`*) – Update attributes for all nodes in nodes. Node attributes specified in nodes as a tuple take precedence over attributes specified via keyword arguments.

See also:

`add_node()`

Examples

```
>>> G = dnx.IntervalGraph()
>>> G.add_nodes_from('Hello')
>>> G.has_node('e')
True
```

Use keywords to update specific node attributes for every node.

```
>>> G.add_nodes_from([1, 2], size=10)
>>> G.add_nodes_from([3, 4], weight=0.4)
```

Use (node, attrdict) tuples to update attributes for specific nodes.

```
>>> G.add_nodes_from([(1, dict(size=11)), (2, {'color':'blue'})])
```

dynetworkx.IntervalGraph.remove_node

`IntervalGraph.`**`remove_node`**(*n*, *begin=None*, *end=None*)

Remove the presence of a node n within the given interval.

Removes the presence node n and all adjacent edges within the given interval.

If interval is specified, all the edges of n will be removed within that interval.

Quiet if n is not in the interval graph.

Parameters

- **n** (*node*) – A node in the graph
- **begin** (*int or float, optional (default= beginning of the entire interval graph)*) – Inclusive beginning time of the node appearing in the interval graph.
- **end** (*int or float, optional (default= end of the entire interval graph + 1)*) – Non-inclusive ending time of the node appearing in the interval graph. Must be bigger than or equal to begin. Note that the default value is shifted up by 1 to make it an inclusive end.

Examples

```
>>> G.add_edges_from([(1, 2, 3, 10), (2, 4, 1, 11), (6, 4, 12, 19), (2, 4, 8, 
→15)])
>>> G.add_nodes_from([(1, {'time': '1pm'}), (2, {'time': '2pm'}), (4, {'time':
→'4pm'})])
>>> G.nodes(begin=4, end=6)
[1, 2, 4, 6]
>>> G.remove_node(2, begin=4, end=6)
>>> G.nodes(begin=4, end=6)
[4, 6]
>>> G.nodes(data=True)
[(1, {'time': '1pm'}), (2, {'time': '2pm'}), (4, {'time': '4pm'}), (6, {})]
>>> G.remove_node(2)
>>> G.nodes(data=True)
[(1, {'time': '1pm'}), (4, {'time': '4pm'}), (6, {})]
```

dynetworkx.IntervalGraph.add_edge

`IntervalGraph.`**`add_edge`**(*u*, *v*, *begin*, *end*, ***attr*)

Add an edge between u and v, during interval [begin, end).

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

Edge attributes can be specified with keywords or by directly accessing the edge's attribute dictionary. See examples below.

Parameters

- **v** (*u,*) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.
- **begin** (*orderable type*) – Inclusive beginning time of the edge appearing in the interval graph.
- **end** (*orderable type*) – Non-inclusive ending time of the edge appearing in the interval graph. Must be bigger than begin.
- **attr** (*keyword arguments, optional*) – Edge data (or labels or objects) can be assigned using keyword arguments.

See also:

add_edges_from() add a collection of edges

Notes

Adding an edge that already exists updates the edge data.

Both begin and end must be the same type across all edges in the interval graph. Also, to create snapshots, both must be integers.

Many NetworkX algorithms designed for weighted graphs use an edge attribute (by default *weight*) to hold a numerical value.

Examples

The following all add the edge e=(1, 2, 3, 10) to graph G:

```
>>> G = dnx.IntervalGraph()
>>> e = (1, 2, 3, 10)
>>> G.add_edge(1, 2, 3, 10)           # explicit two-node form with interval
>>> G.add_edge(*e)             # single edge as tuple of two nodes and interval
>>> G.add_edges_from([(1, 2, 3, 10)])  # add edges from iterable container
```

Associate data to edges using keywords:

```
>>> G.add_edge(1, 2, 3, 10 weight=3)
>>> G.add_edge(1, 3, 4, 9, weight=7, capacity=15, length=342.7)
```

dynetworkx.IntervalGraph.add_edges_from

`IntervalGraph.`**`add_edges_from`**(*ebunch_to_add*, ***attr*)

Add all the edges in ebunch_to_add.

Parameters

- **ebunch_to_add** (*container of edges*) – Each edge given in the container will be added to the interval graph. The edges must be given as as 4-tuples (u, v, being, end). Both begin and end must be orderable and the same type across all edges.
- **attr** (*keyword arguments, optional*) – Edge data (or labels or objects) can be assigned using keyword arguments.

See also:

`add_edge()` add a single edge

Notes

Adding the same edge (with the same interval) twice has no effect but any edge data will be updated when each duplicate edge is added.

Examples

```
>>> G = dnx.IntervalGraph()
>>> G.add_edges_from([(1, 2, 3, 10), (2, 4, 1, 11)]) # using a list of edge tuples
```

Associate data to edges

```
>>> G.add_edges_from([(1, 2, 3, 10), (2, 4, 1, 11)], weight=3)
>>> G.add_edges_from([(3, 4, 2, 19), (1, 4, 1, 3)], label='WN2898')
```

dynetworkx.IntervalGraph.remove_edge

`IntervalGraph.`**`remove_edge`**(*u*, *v*, *begin=None*, *end=None*, *overlapping=True*)

Remove the edge between u and v in the interval graph, during the given interval.

Quiet if the specified edge is not present.

Parameters

- **v** (*u,*) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.
- **begin** (*int or float, optional (default= beginning of the entire interval graph)*) – Inclusive beginning time of the edge appearing in the interval graph.
- **end** (*int or float, optional (default= end of the entire interval graph + 1)*) – Non-inclusive ending time of the edge appearing in the interval graph. Must be bigger than or equal to begin. Note that the default value is shifted up by 1 to make it an inclusive end.
- **overlapping** (*bool, optional (default= True)*) – if True, remove the edge between u and v with overlapping interval with *begin* and *end*. if False, remove the edge between u and v with the exact interval. Note: if False, both *begin* and *end* must be defined, otherwise an exception is raised.

Raises `NetworkXError` – If *begin* and *end* are not defined and *overlapping= False*

Examples

```
>>> G = dnx.IntervalGraph()
>>> G.add_edges_from([(1, 2, 3, 10), (2, 4, 1, 11), (6, 4, 5, 9), (1, 2, 8, 15)])
>>> G.remove_edge(1, 2)
>>> G.has_edge(1, 2)
False
```

With specific overlapping interval

```
>>> G = dnx.IntervalGraph()
>>> G.add_edges_from([(1, 2, 3, 10), (2, 4, 1, 11), (6, 4, 5, 9), (1, 2, 8, 15)])
>>> G.remove_edge(1, 2, begin=2, end=4)
>>> G.has_edge(1, 2, begin=2, end=4)
False
>>> G.has_edge(1, 2)
True
```

Exact interval match

```
>>> G.remove_edge(2, 4, begin=1, end=11, overlapping=False)
>>> G.has_edge(2, 4, begin=1, end=11)
False
```

Reporting interval graph, nodes and edges

`IntervalGraph.nodes`([begin, end, data, default])	A NodeDataView of the IntervalGraph nodes.
`IntervalGraph.has_node`(n[, begin, end])	Return True if the interval graph contains the node n, during the given interval.
`IntervalGraph.edges`([u, v, begin, end, . . .])	Returns a list of Interval objects of the IntervalGraph edges.
`IntervalGraph.has_edge`(u, v[, begin, end, . . .])	Return True if there exists an edge between u and v in the interval graph, during the given interval.
`IntervalGraph.__contains__`(n)	Return True if n is a node, False otherwise.
`IntervalGraph.__str__`()	Return the interval graph name.
`IntervalGraph.interval`()	Return a 2-tuple as (begin, end) interval of the entire

dynetworkx.IntervalGraph.nodes

`IntervalGraph.`**`nodes`**(*begin=None*, *end=None*, *data=False*, *default=None*)

A NodeDataView of the IntervalGraph nodes.

A nodes is considered to be present during an interval, if it has an edge with overlapping interval.

Parameters

- **`begin`** (*`int or float, optional (default= beginning of the entire interval graph)`*) – Inclusive beginning time of the node appearing in the interval graph.
- **`end`** (*`int or float, optional (default= end of the entire interval graph + 1)`*) – Non-inclusive ending time of the node appearing in the interval graph. Must be bigger than or equal to begin. Note that the default value is shifted up by 1 to make it an inclusive end.
- **`data`** (*`string or bool, optional (default=False)`*) – The node attribute returned in 2-tuple (n, dict[data]). If False, return just the nodes n.
- **`default`** (*`value, optional (default=None)`*) – Value used for nodes that don't have the requested attribute. Only relevant if data is not True or False.

Returns

A NodeDataView iterates over *(n, data)* and has no set operations.

When called, if data is False, an iterator over nodes. Otherwise an iterator of 2-tuples (node, attribute value) where data is True.

Return type NodeDataView

Examples

There are two simple ways of getting a list of all nodes in the graph:

```
>>> G = dnx.IntervalGraph()
>>> G.add_edges_from([(1, 2, 3, 10), (2, 4, 1, 11), (6, 4, 12, 19), (2, 4, 8,␣
↪15)])
[1, 2, 4, 6]
```

To get the node data along with the nodes:

```
>>> G.add_nodes_from([(1, {'time': '1pm'}), (2, {'time': '2pm'}), (4, {'time':
↪'4pm'}), (6, {'day': 'Friday'})])
[(1, {'time': '1pm'}), (2, {'time': '2pm'}), (4, {'time': '4pm'}), (6, {'day':
↪'Friday'})]
```

```
>>> G.nodes(data="time")
[(1, '1pm'), (2, '2pm'), (4, '4pm'), (6, None)]
>>> G.nodes(data="time", default="5pm")
[(1, '1pm'), (2, '2pm'), (4, '4pm'), (6, '5pm')]
```

To get nodes which appear in a specific interval. nodes without an edge are not considered present.

```
>>> G.nodes(begin=11, data=True)
[(2, {'time': '2pm'}), (4, {'time': '4pm'}), (6, {'day': 'Friday'})]
>>> G.nodes(begin=4, end=12) # non-inclusive end
[1, 2, 4]
```

dynetworkx.IntervalGraph.has_node

`IntervalGraph.`**`has_node`**(*n*, *begin=None*, *end=None*)

Return True if the interval graph contains the node n, during the given interval.

Identical to *n in G* when 'begin' and 'end' are not defined.

Parameters

- **n** (*node*) –
- **begin** (*int or float, optional (default= beginning of the entire interval graph)*) – Inclusive beginning time of the node appearing in the interval graph.
- **end** (*int or float, optional (default= end of the entire interval graph + 1)*) – Non-inclusive ending time of the node appearing in the interval graph. Must be bigger than or equal begin. Note that the default value is shifted up by 1 to make it an inclusive end.

Examples

```
>>> G = dnx.IntervalGraph()
>>> G.add_ndoe(1)
>>> G.has_node(1)
True
```

It is more readable and simpler to use

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

With interval query:

```
>>> G.add_edge(3, 4, 2, 5)
>>> G.has_node(3)
True
>>> G.has_node(3, begin=2)
True
>>> G.has_node(3, end=2) # end is non-inclusive
False
```

dynetworkx.IntervalGraph.edges

`IntervalGraph.`**`edges`**(*u=None*, *v=None*, *begin=None*, *end=None*, *data=False*, *default=None*)

Returns a list of Interval objects of the IntervalGraph edges.

All edges which are present within the given interval.

All parameters are optional. *u* and *v* can be thought of as constraints. If no node is defined, all edges within the interval are returned. If one node is defined, all edges which have that node as one end, will be returned, and finally if both nodes are defined then all edges between the two nodes are returned.

Parameters

- **v** (*u,*) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects. If the node does not exist in the graph, a key error is raised.
- **begin** (*int or float, optional (default= beginning of the entire interval graph)*) – Inclusive beginning time of the edge appearing in the interval graph.
- **end** (*int or float, optional (default= end of the entire interval graph + 1)*) – Non-inclusive ending time of the edge appearing in the interval graph. Must be bigger than or equal to begin. Note that the default value is shifted up by 1 to make it an inclusive end.
- **data** (*string or bool, optional (default=False)*) – If True, return 2-tuple (Interval object, dict of attributes). If False, return just the Interval objects. If string (name of the attribute), return 2-tuple (Interval object, attribute value).
- **default** (*value, optional (default=None)*) – Default Value to be used for edges that don't have the requested attribute. Only relevant if *data* is a string (name of an attribute).

Returns

An interval object has the following format: (begin, end, (u, v))

When called, if *data* is False, a list of interval objects. If *data* is True, a list of 2-tuples: (Interval, dict of attribute(s) with values), If *data* is a string, a list of 2-tuples (Interval, attribute value).

Return type List of Interval objects

Examples

To get a list of all edges:

```
>>> G = dnx.IntervalGraph()
>>> G.add_edges_from([(1, 2, 3, 10), (2, 4, 1, 11), (6, 4, 12, 19), (2, 4, 8,␣
↪15)])
>>> G.edges()
[Interval(8, 15, (2, 4)), Interval(3, 10, (1, 2)), Interval(1, 11, (2, 4)),␣
↪Interval(12, 19, (6, 4))]
```

To get edges which appear in a specific interval:

```
>>> G.edges(begin=10)
[Interval(12, 19, (6, 4)), Interval(1, 11, (2, 4)), Interval(8, 15, (2, 4))]
>>> G.edges(end=5)
[Interval(3, 10, (1, 2)), Interval(1, 11, (2, 4))]
>>> G.edges(begin=2, end=4)
[Interval(3, 10, (1, 2)), Interval(1, 11, (2, 4))]
```

To get edges with either of the two nodes being defined:

```
>>> G.edges(u=2)
[Interval(3, 10, (1, 2)), Interval(1, 11, (2, 4)), Interval(8, 15, (2, 4))]
>>> G.edges(u=2, begin=11)
[Interval(1, 11, (2, 4)), Interval(8, 15, (2, 4))]
>>> G.edges(u=2, v=4, end=8)
[Interval(1, 11, (2, 4))]
>>> G.edges(u=1, v=6)
[]
```

To get a list of edges with data:

```
>>> G = dnx.IntervalGraph()
>>> G.add_edge(1, 3, 1, 4, weight=8, height=18)
>>> G.add_edge(1, 2, 3, 10, weight=10)
>>> G.add_edge(2, 6, 2, 10)
>>> G.edges(data="weight")
[(Interval(2, 8, (2, 3)), None), (Interval(3, 10, (1, 2)), 10), (Interval(1, 4,␣
↪(1, 3)), 8)]
>>> G.edges(data="weight", default=5)
[(Interval(2, 8, (2, 3)), 5), (Interval(3, 10, (1, 2)), 10), (Interval(1, 4, (1,␣
↪3)), 8)]
>>> G.edges(data=True)
[(Interval(2, 8, (2, 3)), {}), (Interval(3, 10, (1, 2)), {'weight': 10}),␣
↪(Interval(1, 4, (1, 3)), {'height': 18, 'weight': 8})]
>>> G.edges(u=1, begin=5, end=9, data="weight")
[(Interval(3, 10, (1, 2)), 10)]
```

dynetworkx.IntervalGraph.has_edge

`IntervalGraph.`**`has_edge`**`(`*u*, *v*, *begin=None*, *end=None*, *overlapping=True*`)`

Return True if there exists an edge between u and v in the interval graph, during the given interval.

Parameters

- **`v`** (*`u,`*) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.
- **`begin`** (*`int or float, optional (default= beginning of the entire interval graph)`*) – Inclusive beginning time of the node appearing in the interval graph.
- **`end`** (*`int or float, optional (default= end of the entire interval graph + 1)`*) – Non-inclusive ending time of the node appearing in the interval graph. Must be bigger than or equal begin. Note that the default value is shifted up by 1 to make it an inclusive end.
- **`overlapping`** (*`bool, optional (default= True)`*) – if True, it returns True if there exists an edge between u and v with overlapping interval with *begin* and *end*. if False, it returns true only if there exists an edge between u and v with the exact interval. Note: if False, both *begin* and *end* must be defined, otherwise an exception is raised.

Raises `NetworkXError` – If *begin* and *end* are not defined and *overlapping= False*

Examples

```
>>> G = dnx.IntervalGraph()
>>> G.add_edges_from([(1, 2, 3, 10), (2, 4, 1, 11)])
>>> G.has_edge(1, 2)
True
```

With specific overlapping interval:

```
>>> G.has_edge(1, 2, begin=2)
True
>>> G.has_edge(2, 4, begin=12)
False
```

Exact interval match:

```
>>> G.has_edge(2, 4, begin=1, end=11)
True
>>> G.has_edge(2, 4, begin=2, end=11)
False
```

dynetworkx.IntervalGraph.__contains__

`IntervalGraph.`**`__contains__`**`(`*n*`)`

Return True if n is a node, False otherwise. Use: ‘n in G’.

Examples

```
>>> G = dnx.IntervalGraph()
>>> G.add_node(2)
>>> 2 in G
True
```

dynetworkx.IntervalGraph.__str__

`IntervalGraph.`**`__str__`**`()`

Return the interval graph name.

Returns name – The name of the interval graph.

Return type string

Examples

```
>>> G = dnx.IntervalGraph(name='foo')
>>> str(G)
'foo'
```

dynetworkx.IntervalGraph.interval

`IntervalGraph.`**`interval`**`()`

Return a 2-tuple as (begin, end) interval of the entire interval graph.

Note that end is non-inclusive.

Examples

```
>>> G = dnx.IntervalGraph()
>>> G.add_edges_from([(1, 2, 0, 10), (3, 7, 9, 16)])
>>> G.interval()
(0, 16)
```

Counting nodes and edges

IntervalGraph.number_of_nodes([begin, end])	Return the number of nodes in the interval graph between the given interval.
IntervalGraph.__len__()	Return the number of nodes.

dynetworkx.IntervalGraph.number_of_nodes

`IntervalGraph.`**`number_of_nodes`**`(`*begin=None*, *end=None*`)`

Return the number of nodes in the interval graph between the given interval.

Parameters

- **begin** (*int or float, optional (default= beginning of the entire interval graph)*) – Inclusive beginning time of the node appearing in the interval graph.
- **end** (*int or float, optional (default= end of the entire interval graph + 1)*) – Non-inclusive ending time of the node appearing in the interval graph. Must be bigger than or equal begin. Note that the default value is shifted up by 1 to make it an inclusive end.

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

Return type int

See also:

`__len__()`

Examples

```
>>> G = dnx.IntervalGraph()
>>> G.add_edges_from([(1, 2, 0, 5), (3, 4, 8, 11)])
>>> len(G)
4
>>> G.number_of_nodes()
4
>>> G.number_of_nodes(begin=6)
2
>>> G.number_of_nodes(begin=5, end=8) # end in non-inclusive
2
>>> G.number_of_nodes(end=8)
4
```

dynetworkx.IntervalGraph.__len__

`IntervalGraph.`**`__len__`**`()`

Return the number of nodes. Use: ‘len(G)’.

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

Return type int

Examples

```
>>> G = dnx.IntervalGraph()
>>> G.add_nodes_from([2, 4, 5])
>>> len(G)
3
```

Making copies and subgraphs

IntervalGraph.to_subgraph(begin, end[, . . .])	Return a networkx Graph or MultiGraph which includes all the nodes and edges which have overlapping intervals with the given interval.
IntervalGraph.to_snapshots(number_of_snapshots)	Return a list of networkx Graph or MultiGraph objects as snapshots of the interval graph in consecutive order.

dynetworkx.IntervalGraph.to_subgraph

`IntervalGraph.`**`to_subgraph`**(*begin*, *end*, *multigraph=False*, *edge_data=False*, *edge_interval_data=False*, *node_data=False*)

Return a networkx Graph or MultiGraph which includes all the nodes and edges which have overlapping intervals with the given interval.

Parameters

- **begin** (*int or float*) – Inclusive beginning time of the edge appearing in the interval graph.
- **end** (*int or float*) – Non-inclusive ending time of the edge appearing in the interval graph. Must be bigger than or equal to begin.
- **multigraph** (*bool, optional (default= False)*) – If True, a networkx MultiGraph will be returned. If False, networkx Graph.
- **edge_data** (*bool, optional (default= False)*) – If True, edges will keep their attributes.
- **edge_interval_data** (*bool, optional (default= False)*) – If True, each edge's attribute will also include its begin and end interval data. If *edge_data= True* and there already exist edge attributes with names begin and end, they will be overwritten.
- **node_data** (*bool, optional (default= False)*) – if True, each node's attributes will be included.

See also:

to_snapshots() divide the interval graph to snapshots

Notes

If multigraph= False, and edge_data=True or edge_interval_data=True, in case there are multiple edges, only one will show with one of the edge's attributes.

Note: nodes with no edges will not appear in any subgraph.

Examples

```
>>> G = dnx.IntervalGraph()
>>> G.add_edges_from([(1, 2, 3, 10), (2, 4, 1, 11), (6, 4, 12, 19), (2, 4, 8,␣
↪15)])
>>> H = G.to_subgraph(4, 12)
>>> type(H)
<class 'networkx.classes.graph.Graph'>
>>> list(H.edges(data=True))
[(1, 2, {}), (2, 4, {})]
```

```
>>> H = G.to_subgraph(4, 12, edge_interval_data=True)
>>> type(H)
<class 'networkx.classes.graph.Graph'>
>>> list(H.edges(data=True))
[(1, 2, {'end': 10, 'begin': 3}), (2, 4, {'end': 15, 'begin': 8})]
```

```
>>> M = G.to_subgraph(4, 12, multigraph=True, edge_interval_data=True)
>>> type(M)
<class 'networkx.classes.multigraph.MultiGraph'>
>>> list(M.edges(data=True))
[(1, 2, {'end': 10, 'begin': 3}), (2, 4, {'end': 11, 'begin': 1}), (2, 4, {'end':␣
↪15, 'begin': 8})]
```

dynetworkx.IntervalGraph.to_snapshots

`IntervalGraph.`**`to_snapshots`**(*number_of_snapshots*, *multigraph=False*, *edge_data=False*, *edge_interval_data=False*, *node_data=False*, *return_length=False*)

Return a list of networkx Graph or MultiGraph objects as snapshots of the interval graph in consecutive order.

Parameters

- **`number_of_snapshots`** (*`integer`*) – Number of snapshots to divide the interval graph into. Must be bigger than 1.
- **`multigraph`** (*`bool, optional (default= False)`*) – If True, a networkx MultiGraph will be returned. If False, networkx Graph.
- **`edge_data`** (*`bool, optional (default= False)`*) – If True, edges will keep their attributes.
- **`edge_interval_data`** (*`bool, optional (default= False)`*) – If True, each edge's attribute will also include its begin and end interval data. If *edge_data= True* and there already exist edge attributes with names begin and end, they will be overwritten.
- **`node_data`** (*`bool, optional (default= False)`*) – if True, each node's attributes will be included.
- **`return_length`** (*`bool, optional (default= False)`*) – If true, the length of snapshots will be returned as the second argument.

See also:

`to_subgraph()` subgraph based on an interval

Notes

In order to create snapshots, begin and end interval objects of the interval graph must be numbers.

If multigraph= False, and edge_data=True or edge_interval_data=True, in case there are multiple edges, only one will show with one of the edge's attributes.

Examples

Snapshots of NetworkX Graph

```
>>> G = dnx.IntervalGraph()
>>> G.add_edges_from([(1, 2, 3, 10), (2, 4, 1, 11), (6, 4, 12, 19), (2, 4, 8,␣
↪15)])
>>> S, l = G.to_snapshots(2, edge_interval_data=True, return_length=True)
>>> S
[<networkx.classes.graph.Graph object at 0x100000>, <networkx.classes.graph.Graph␣
↪object at 0x150d00>]
>>> l
9.0
>>> for g in S:
>>> ... g.edges(data=True))
[(1, 2, {'begin': 3, 'end': 10}), (2, 4, {'begin': 8, 'end': 15})]
[(2, 4, {'begin': 8, 'end': 15}), (4, 6, {'begin': 12, 'end': 19})]
```

Snapshots of NetworkX MultiGraph

```
>>> S, l = G.to_snapshots(3, multigraph=True, edge_interval_data=True, return_
↪length=True)
>>> S
[<networkx.classes.multigraph.MultiGraph object at 0x1060d40b8>, <networkx.
↪classes.multigraph.MultiGraph object at 0x151020c9e8>, <networkx.classes.
↪multigraph.MultiGraph object at 0x151021d390>]
>>> l
6.0
>>> for g in S:
>>> ... g.edges(data=True))
[(1, 2, {'end': 10, 'begin': 3}), (2, 4, {'end': 11, 'begin': 1})]
[(1, 2, {'end': 10, 'begin': 3}), (2, 4, {'end': 11, 'begin': 1}), (2, 4, {'end':␣
↪15, 'begin': 8}), (4, 6, {'end': 19, 'begin': 12})]
[(2, 4, {'end': 15, 'begin': 8}), (4, 6, {'end': 19, 'begin': 12})]
```

Loading an interval graph

`IntervalGraph.load_from_txt`(path[, …])	Read interval graph in from path.
`IntervalGraph.save_to_txt`(path[, delimiter])	Write interval graph to path.

dynetworkx.IntervalGraph.load_from_txt

static `IntervalGraph.`**`load_from_txt`**(*path*, *delimiter=' '*, *nodetype=None*, *intervaltype=<class 'float'>*, *comments='#'*)

Read interval graph in from path. Every line in the file must be an edge in the following format: "node node begin end". Both interval times must be integers or floats. Nodes can be any hashable objects.

Parameters

- **`path`** (*`string or file`*) – Filename to read.
- **`nodetype`** (*`Python type, optional`*) – Convert nodes to this type.
- **`intervaltype`** (*`Python type, optional (default= float)`*) –
- **`interval begin and end to this type.`** (*`Convert`*) –
- **`must be an orderable type, ideally int or float. Other orderable types have not been fully tested.`** (*`This`*) –

- **comments** (*string, optional*) – Marker for comment lines
- **delimiter** (*string, optional*) – Separator for node labels. The default is whitespace.

Returns **G** – The graph corresponding to the lines in edge list.

Return type *IntervalGraph*

Examples

```
>>> G=dnx.IntervalGraph.load_from_txt("my_dygraph.txt")
```

The optional nodetype is a function to convert node strings to nodetype.

For example

```
>>> G=dnx.IntervalGraph.load_from_txt("my_dygraph.txt", nodetype=int)
```

will attempt to convert all nodes to integer type.

Since nodes must be hashable, the function nodetype must return hashable types (e.g. int, float, str, frozenset - or tuples of those, etc.)

dynetworkx.IntervalGraph.save_to_txt

`IntervalGraph.`**`save_to_txt`**(*path*, *delimiter=' '*)

Write interval graph to path. Every line in the file must be an edge in the following format: "node node begin end". Begin, end must be integers or floats. Nodes can be any hashable objects.

Parameters

- **path** (*string or file*) – Filename to read.
- **delimiter** (*string, optional*) – Separator for node labels. The default is whitespace. Cannot be =.

Examples

```
>>> G.save_to_txt("my_dygraph.txt")
```

Analyzing interval graphs

IntervalGraph.degree([node, begin, end, delta])	Return the degree of a specified node between time begin and end.

dynetworkx.IntervalGraph.degree

`IntervalGraph.`**`degree`**(*node=None*, *begin=None*, *end=None*, *delta=False*)

Return the degree of a specified node between time begin and end.

Parameters

- **node** (*Nodes can be, for example, strings or numbers, optional.*) – Nodes must be hashable (and not None) Python objects.
- **begin** (*int or float, optional (default= beginning of the entire interval graph)*) – Inclusive beginning time of the edge appearing in the interval graph.
- **end** (*int or float, optional (default= end of the entire interval graph)*) – Non-inclusive ending time of the edge appearing in the interval graph.

Returns

- *Integer value of degree of specified node.*
- *If no node is specified, returns float mean degree value of graph.*
- *If delta is True, return list of tuples.* – First indicating the time a degree change occurred, Second indicating the degree after the change occured

Examples

```
>>> G = IntervalGraph()
>>> G.add_edge(1, 2, 3, 5)
>>> G.add_edge(2, 3, 8, 11)
>>> G.degree(2)
2
>>> G.degree(2,2)
2
>>> G.degree(2,end=8)
1
>>> G.degree()
1.33333
>>> G.degree(2,delta=True)
[(3, 1), (5, 0), (8, 1)]
```

Impulse Graph

Overview

class `dynetworkx.`**`ImpulseGraph`**(*name=''*, ***attr*)

Base class for undirected interval graphs.

The ImpulseGraph class allows any hashable object as a node and can associate key/value attribute pairs with each undirected edge.

Each edge must have one integer, timestamp.

Self-loops are allowed. Multiple edges between two nodes are allowed.

Parameters **attr** (*keyword arguments, optional (default= no attributes)*) – Attributes to add to graph as key=value pairs.

Examples

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

```
>>> G = dnx.ImpulseGraph()
```

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).

Add the nodes from any container (a list, dict, set)

```
>>> G.add_nodes_from([2, 3])
>>> G.add_nodes_from(range(100, 110))
```

Edges:

G can also be grown by adding edges. This can be considered the primary way to grow G, since nodes with no edge will not appear in G in most cases. See `G.to_snapshot()`.

Add one edge, with timestamp of 10.

```
>>> G.add_edge(1, 2, 10)
```

a list of edges,

```
>>> G.add_edges_from([(1, 2, 10), (1, 3, 11)])
```

If some edges connect nodes not yet in the graph, the nodes are added automatically. There are no errors when adding nodes or edges that already exist.

Attributes:

Each impulse graph, node, and edge can hold key/value attribute pairs in an associated attribute dictionary (the keys must be hashable). By default these are empty, but can be added or changed using add_edge, add_node.

Keep in mind that the edge timestamp is not an attribute of the edge.

```
>>> G = dnx.IntervalGraph(day="Friday")
>>> G.graph
{'day': 'Friday'}
```

Add node attributes using add_node(), add_nodes_from()

```
>>> G.add_node(1, time='5pm')
>>> G.add_nodes_from([3], time='2pm')
```

Add edge attributes using add_edge(), add_edges_from().

```
>>> G.add_edge(1, 2, 10, weight=4.7 )
>>> G.add_edges_from([(3, 4, 11), (4, 5, 33)], color='red')
```

Shortcuts:

Here are a couple examples of available shortcuts:

```
>>> 1 in G  # check if node in impulse graph during any timestamp
True
>>> len(G)  # number of nodes in the entire impulse graph
5
```

Subclasses (Advanced): Edges in impulse graphs are represented by tuples kept in a SortedDict (http://www.grantjenks.com/docs/sortedcontainers/) keyed by timestamp.

The Graph class uses a dict-of-dict-of-dict data structure. The outer dict (node_dict) holds adjacency information keyed by nodes. The next dict (adjlist_dict) represents the adjacency information and holds edge data keyed by interval objects. The inner dict (edge_attr_dict) represents the edge data and holds edge attribute values keyed by attribute names.

Methods

Adding and removing nodes and edges

ImpulseGraph.__init__([name])	Initialize an interval graph with edges, name, or graph attributes.
ImpulseGraph.add_node(node_for_adding, **attr)	Add a single node *node_for_adding* and update node attributes.
ImpulseGraph.add_nodes_from(. . .)	Add multiple nodes.
ImpulseGraph.remove_node(n[, begin, end, . . .])	Remove the presence of a node n within the given interval.
ImpulseGraph.add_edge(u, v, t, **attr)	Add an edge between u and v, at t.
ImpulseGraph.add_edges_from(ebunch_to_add, . . .)	Add all the edges in ebunch_to_add.
ImpulseGraph.remove_edge(u, v[, begin, end, . . .])	Remove the edge between u and v in the impulse graph, during the given interval.

dynetworkx.ImpulseGraph.__init__

ImpulseGraph.**__init__**(*name=''*, ***attr*)

Initialize an interval graph with edges, name, or graph attributes.

Parameters **attr** (*keyword arguments, optional (default= no attributes)*) – Attributes to add to graph as key=value pairs.

Examples

```
>>> G = dnx.ImpulseGraph()
>>> G = dnx.ImpulseGraph(name='my graph')
>>> G.graph
{'name': 'my graph'}
```

dynetworkx.ImpulseGraph.add_node

ImpulseGraph.**add_node**(*node_for_adding*, ***attr*)

Add a single node *node_for_adding* and update node attributes.

Parameters

- **`node_for_adding`** (*node*) – A node can be any hashable Python object except None.
- **`attr`** (*keyword arguments, optional*) – Set or change node attributes using key=value.

See also:

add_nodes_from()

Examples

```
>>> G = dnx.ImpulseGraph()
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> G.number_of_nodes()
2
```

Use keywords set/change node attributes:

```
>>> G.add_node(1, size=10)
>>> G.add_node(3, weight=0.4, UTM=('13S', 382871, 3972649))
```

Notes

A hashable object is one that can be used as a key in a Python dictionary. This includes strings, numbers, tuples of strings and numbers, etc.

On many platforms hashable items also include mutables such as NetworkX Graphs, though one should be careful that the hash doesn't change on mutables.

dynetworkx.ImpulseGraph.add_nodes_from

`ImpulseGraph.`**`add_nodes_from`**(*nodes_for_adding*, ***attr*)

Add multiple nodes.

Parameters

- **`nodes_for_adding`** (*iterable container*) – A container of nodes (list, dict, set, etc.). OR A container of (node, attribute dict) tuples. Node attributes are updated using the attribute dict.
- **`attr`** (*keyword arguments, optional (default= no attributes)*) – Update attributes for all nodes in nodes. Node attributes specified in nodes as a tuple take precedence over attributes specified via keyword arguments.

See also:

add_node()

Examples

```
>>> G = dnx.ImpulseGraph()
>>> G.add_nodes_from('Hello')
>>> G.has_node('e')
True
```

Use keywords to update specific node attributes for every node.

```
>>> G.add_nodes_from([1, 2], size=10)
>>> G.add_nodes_from([3, 4], weight=0.4)
```

Use (node, attrdict) tuples to update attributes for specific nodes.

```
>>> G.add_nodes_from([(1, dict(size=11)), (2, {'color':'blue'})])
```

dynetworkx.ImpulseGraph.remove_node

`ImpulseGraph.`**`remove_node`**(*n*, *begin=None*, *end=None*, *inclusive=(True*, *True)*)

Remove the presence of a node n within the given interval.

Removes the presence node n and all adjacent edges within the given interval.

If interval is specified, all the edges of n will be removed within that interval.

Quiet if n is not in the impulse graph.

Parameters

- **n** (*node*) – A node in the graph
- **begin** (*int or float, optional (default= beginning of the entire impulse graph)*) –
- **end** (*int or float, optional (default= end of the entire impulse graph)*) – Must be bigger than or equal to begin.
- **inclusive** (*2-tuple boolean that determines inclusivity of begin and end*) –

Examples

```
>>> G.add_edges_from([(1, 2, 10), (2, 4, 11), (6, 4, 19), (2, 4, 15)])
>>> G.add_nodes_from([(1, {'time': '1pm'}), (2, {'time': '2pm'}), (4, {'time':
↪'4pm'})])
>>> G.nodes(begin=10, end=19)
[1, 2, 4, 6]
>>> G.remove_node(6, begin=10, end=20)
>>> G.nodes()
[1, 2, 4]
>>> G.nodes(data=True)
[(1, {'time': '1pm'}), (2, {'time': '2pm'}), (4, {'time': '4pm'})]
>>> G.remove_node(2)
>>> G.nodes(data=True)
[(1, {'time': '1pm'}), (4, {'time': '4pm'})]
```

dynetworkx.ImpulseGraph.add_edge

`ImpulseGraph.`**`add_edge`**(*u*, *v*, *t*, ***attr*)

Add an edge between u and v, at t.

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

Edge attributes can be specified with keywords or by directly accessing the edge's attribute dictionary. See examples below.

Parameters

- **v** (*u,*) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.
- **t** (*timestamp*) – Timestamps can be, for example, strings or numbers. Timestamps must be hashable (and not None) Python objects.
- **attr** (*keyword arguments, optional*) – Edge data (or labels or objects) can be assigned using keyword arguments.

See also:

`add_edges_from()` add a collection of edges

Notes

Adding an edge that already exists updates the edge data.

Timestamps must be the same type across all edges in the impulse graph. Also, to create snapshots, timestamps must be integers.

Many NetworkX algorithms designed for weighted graphs use an edge attribute (by default *weight*) to hold a numerical value.

Examples

The following all add the edge e=(1, 2, 3, 10) to graph G:

```
>>> G = dnx.ImpulseGraph()
>>> e = (1, 2, 10)
>>> G.add_edge(1, 2, 10)           # explicit two-node form with timestamp
>>> G.add_edge(*e)              # single edge as tuple of two nodes and timestamp
>>> G.add_edges_from([(1, 2, 10)])  # add edges from iterable container
```

Associate data to edges using keywords:

```
>>> G.add_edge(1, 2, 10 weight=3)
>>> G.add_edge(1, 3, 9, weight=7, capacity=15, length=342.7)
```

dynetworkx.ImpulseGraph.add_edges_from

`ImpulseGraph.`**`add_edges_from`**(*ebunch_to_add*, ***attr*)

Add all the edges in ebunch_to_add.

Parameters

- **ebunch_to_add** (*container of edges*) – Each edge given in the container will be added to the interval graph. The edges must be given as as 3-tuples (u, v, t). Timestamp must be orderable and the same type across all edges.
- **attr** (*keyword arguments, optional*) – Edge data (or labels or objects) can be assigned using keyword arguments.

See also:

`add_edge()` add a single edge

Notes

Adding the same edge (with the same timestamp) twice has no effect but any edge data will be updated when each duplicate edge is added.

Examples

```
>>> G = dnx.ImpulseGraph()
>>> G.add_edges_from([(1, 2, 10), (2, 4, 11)]) # using a list of edge tuples
```

Associate data to edges

```
>>> G.add_edges_from([(1, 2, 10), (2, 4, 11)], weight=3)
>>> G.add_edges_from([(3, 4, 19), (1, 4, 3)], label='WN2898')
```

dynetworkx.ImpulseGraph.remove_edge

`ImpulseGraph.`**`remove_edge`**(*u*, *v*, *begin=None*, *end=None*, *inclusive=(True*, *True)*)

Remove the edge between u and v in the impulse graph, during the given interval.

Quiet if the specified edge is not present.

Parameters

- **v** (*u,*) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.
- **begin** (*int or float, optional (default= beginning of the entire interval graph)*) –
- **end** (*int or float, optional (default= end of the entire interval graph + 1)*) – Must be bigger than or equal to begin.
- **inclusive** (*2-tuple boolean that determines inclusivity of begin and end*) –

Examples

```
>>> G = dnx.ImpulseGraph()
>>> G.add_edges_from([(1, 2, 10), (2, 4, 11), (6, 4, 9), (1, 2, 15)])
>>> G.remove_edge(1, 2)
>>> G.has_edge(1, 2)
False
```

```
>>> G = dnx.ImpulseGraph()
>>> G.add_edges_from([(1, 2, 10), (2, 4, 11), (6, 4, 9), (1, 2, 15)])
>>> G.remove_edge(1, 2, begin=2, end=11)
>>> G.has_edge(1, 2, begin=2, end=11)
False
>>> G.has_edge(1, 2)
True
```

Reporting impulse graph, nodes and edges

ImpulseGraph.nodes([begin, end, inclusive, . . .])	A NodeDataView of the ImpulseGraph nodes.
ImpulseGraph.has_node(n[, begin, end, inclusive])	Return True if the impulse graph contains the node n, during the given interval.
ImpulseGraph.edges([u, v, begin, end, . . .])	Returns a list of Interval objects of the IntervalGraph edges.
ImpulseGraph.has_edge(u, v[, begin, end, . . .])	Return True if there exists an edge between u and v in the impulse graph, during the given interval.
ImpulseGraph.__contains__(n)	Return True if n is a node, False otherwise.
ImpulseGraph.__str__()	Return the interval graph name.
ImpulseGraph.interval()	Return a 2-tuple as (begin, end) interval of the entire

dynetworkx.ImpulseGraph.nodes

`ImpulseGraph.`**`nodes`** (*begin=None*, *end=None*, *inclusive=(True*, *True)*, *data=False*, *default=None*)

A NodeDataView of the ImpulseGraph nodes.

A nodes is considered to be present during an interval, if it has an edge with overlapping interval.

Parameters

- **begin** (*int or float, optional (default= beginning of the entire impulse graph)*) –
- **end** (*int or float, optional (default= end of the entire impulse graph)*) – Must be bigger than or equal to begin.
- **inclusive** (*2-tuple boolean that determines inclusivity of begin and end*) –
- **data** (*string or bool, optional (default=False)*) – The node attribute returned in 2-tuple (n, dict[data]). If False, return just the nodes n.
- **default** (*value, optional (default=None)*) – Value used for nodes that don’t have the requested attribute. Only relevant if data is not True or False.

Returns

A NodeDataView iterates over *(n, data)* and has no set operations.

When called, if data is False, an iterator over nodes. Otherwise an iterator of 2-tuples (node, attribute value) where data is True.

Return type NodeDataView

Examples

There are two simple ways of getting a list of all nodes in the graph:

```
>>> G = dnx.ImpulseGraph()
>>> G.add_edges_from([(1, 2, 10), (2, 4, 11), (6, 4, 19), (2, 4, 15)])
>>> G.nodes()
[1, 2, 4, 6]
```

To get the node data along with the nodes:

```
>>> G.add_nodes_from([(1, {'time': '1pm'}), (2, {'time': '2pm'}), (4, {'time':
↪'4pm'}), (6, {'day': 'Friday'})])
>>> G.nodes(data=True)
[(1, {'time': '1pm'}), (2, {'time': '2pm'}), (4, {'time': '4pm'}), (6, {'day':
↪'Friday'})]
```

```
>>> G.nodes(data="time")
[(1, '1pm'), (2, '2pm'), (4, '4pm'), (6, None)]
>>> G.nodes(data="time", default="5pm")
[(1, '1pm'), (2, '2pm'), (4, '4pm'), (6, '5pm')]
```

To get nodes which appear in a specific interval. Nodes without an edge are not considered present.

```
>>> G.nodes(begin=11, data=True)
[(2, {'time': '2pm'}), (4, {'time': '4pm'}), (6, {'day': 'Friday'})]
>>> G.nodes(begin=4, end=12)
[1, 2, 4]
```

dynetworkx.ImpulseGraph.has_node

`ImpulseGraph.`**`has_node`**(*n*, *begin=None*, *end=None*, *inclusive=(True*, *True)*)

Return True if the impulse graph contains the node n, during the given interval.

Identical to *n in G* when 'begin' and 'end' are not defined.

Parameters

- **n** (*node*) –
- **begin** (*int or float, optional (default= beginning of the entire impulse graph)*) –
- **end** (*int or float, optional (default= end of the entire impulse graph)*) – Must be bigger than or equal begin.
- **inclusive** (*2-tuple boolean that determines inclusivity of begin and end*) –

Examples

```
>>> G = dnx.ImpulseGraph()
>>> G.add_node(1)
>>> G.has_node(1)
True
```

It is more readable and simpler to use

```
>>> 1 in G
True
```

With interval query:

```
>>> G.add_edge(3, 4, 5)
>>> G.has_node(3)
True
>>> G.has_node(3, begin=2)
True
>>> G.has_node(3, end=2)
False
```

dynetworkx.ImpulseGraph.edges

`ImpulseGraph.`**`edges`**(*u=None*, *v=None*, *begin=None*, *end=None*, *inclusive=(True*, *True)*, *data=False*, *default=None*)

Returns a list of Interval objects of the IntervalGraph edges.

All edges which are present within the given interval.

All parameters are optional. *u* and *v* can be thought of as constraints. If no node is defined, all edges within the interval are returned. If one node is defined, all edges which have that node as one end, will be returned, and finally if both nodes are defined then all edges between the two nodes are returned.

Parameters

- **`v`** (*`u,`*) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects. If the node does not exist in the graph, a key error is raised.
- **`begin`** (*`int or float, optional (default= beginning of the entire impulse graph)`*) –
- **`end`** (*`int or float, optional (default= end of the entire impulse graph)`*) – Must be bigger than or equal to begin.
- **`inclusive`** (*`2-tuple boolean that determines inclusivity of begin and end`*) –
- **`data`** (*`string or bool, optional (default=False)`*) – If True, return 2-tuple (Edge Tuple, dict of attributes). If False, return just the Edge Tuples. If string (name of the attribute), return 2-tuple (Edge Tuple, attribute value).
- **`default`** (*`value, optional (default=None)`*) – Default Value to be used for edges that don't have the requested attribute. Only relevant if *data* is a string (name of an attribute).

Returns

An edge tuple has the following format: (u, v, edge_id, timestamp)

When called, if *data* is False, a list of edge tuples. If *data* is True, a list of 2-tuples: (Edge Tuple, dict of attribute(s) with values), If *data* is a string, a list of 2-tuples (Edge Tuple, attribute value).

Return type List of Edge Tuples

Examples

To get a list of all edges:

```
>>> G = dnx.IntervalGraph()
>>> G.add_edges_from([(1, 2, 10), (2, 4, 11), (6, 4, 19), (2, 4, 15)])
>>> G.edges()
[(1, 2, 10), (2, 4, 11), (2, 4, 15), (6, 4, 19)]
```

To get edges which appear in a specific interval:

```
>>> G.edges(begin=10)
[(1, 2, 10), (2, 4, 11), (2, 4, 15), (6, 4, 19)]
>>> G.edges(end=11)
[(1, 2, 10), (2, 4, 11)]
>>> G.edges(begin=11, end=15)
[(2, 4, 11), (2, 4, 15)]
```

To get edges with either of the two nodes being defined:

```
>>> G.edges(u=2)
[(2, 4, 11), (2, 4, 15)]
>>> G.edges(u=2, begin=11)
[(2, 4, 11), (2, 4, 15)]
>>> G.edges(u=2, v=4, end=11)
[(2, 4, 11)]
>>> G.edges(u=1, v=6)
[]
```

To get a list of edges with data:

```
>>> G = dnx.ImpulseGraph()
>>> G.add_edge(1, 3, 4, weight=8, height=18)
>>> G.add_edge(1, 2, 10, weight=10)
>>> G.add_edge(2, 6, 10)
>>> G.edges(data="weight")
[((1, 3, 4), 8), ((1, 2, 10), 10), ((2, 6, 10), None)]
>>> G.edges(data="weight", default=5)
[((1, 3, 4), 8), ((1, 2, 10), 10), ((2, 6, 10), 5)]
>>> G.edges(data=True)
[((1, 3, 4), {'weight': 8, 'height': 18}), ((1, 2, 10), {'weight': 10}), ((2, 6,␣
↪10), {})]
>>> G.edges(u=1, begin=2, end=9, data="weight")
[((1, 3, 4), 8)]
```

dynetworkx.ImpulseGraph.has_edge

`ImpulseGraph.`**`has_edge`**(*u*, *v*, *begin=None*, *end=None*, *inclusive=(True*, *True)*)

Return True if there exists an edge between u and v in the impulse graph, during the given interval.

Parameters

- **v** (*u,*) – Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.
- **begin** (*int or float, optional (default= beginning of the entire impulse graph)*) –
- **end** (*int or float, optional (default= end of the entire impulse graph)*) – Must be bigger than or equal begin.

- **inclusive** (*2-tuple boolean that determines inclusivity of begin and end*) –

Examples

```
>>> G = dnx.ImpulseGraph()
>>> G.add_edges_from([(1, 2, 10), (2, 4, 11)])
>>> G.has_edge(1, 2)
True
>>> G.has_edge(1, 2, begin=2)
True
>>> G.has_edge(2, 4, begin=12)
False
```

dynetworkx.ImpulseGraph.__contains__

`ImpulseGraph.`**`__contains__`**`(n)`

Return True if n is a node, False otherwise. Use: ‘n in G’.

Examples

```
>>> G = dnx.IntervalGraph()
>>> G.add_node(2)
>>> 2 in G
True
```

dynetworkx.ImpulseGraph.__str__

`ImpulseGraph.`**`__str__`**`()`

Return the interval graph name.

Returns **name** – The name of the interval graph.

Return type string

Examples

```
>>> G = dnx.IntervalGraph(name='foo')
>>> str(G)
'foo'
```

dynetworkx.ImpulseGraph.interval

`ImpulseGraph.`**`interval`**`()`

Return a 2-tuple as (begin, end) interval of the entire impulse graph.

Examples

```
>>> G = dnx.ImpulseGraph()
>>> G.add_edges_from([(1, 2, 10), (3, 7, 16)])
>>> G.interval()
(10, 16)
```

Counting nodes and edges

ImpulseGraph.number_of_nodes([begin, end, …])	Return the number of nodes in the impulse graph between the given interval.
ImpulseGraph.__len__()	Return the number of nodes.

dynetworkx.ImpulseGraph.number_of_nodes

`ImpulseGraph.`**`number_of_nodes`**(*begin=None*, *end=None*, *inclusive=(True*, *True)*)

Return the number of nodes in the impulse graph between the given interval.

Parameters

- **begin** (*int or float, optional (default= beginning of the entire impulse graph)*) –
- **end** (*int or float, optional (default= end of the entire impulse graph)*) – Must be bigger than or equal begin.
- **inclusive** (*2-tuple boolean that determines inclusivity of begin and end*) –

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

Return type int

See also:

__len__()

Examples

```
>>> G = dnx.ImpulseGraph()
>>> G.add_edges_from([(1, 2, 5), (3, 4, 11)])
>>> len(G)
4
>>> G.number_of_nodes()
4
>>> G.number_of_nodes(begin=6)
2
>>> G.number_of_nodes(begin=5, end=8)
2
>>> G.number_of_nodes(end=11)
4
```

dynetworkx.ImpulseGraph.__len__

`ImpulseGraph.`**`__len__`**`()`

Return the number of nodes. Use: 'len(G)'.

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

Return type int

Examples

```
>>> G = dnx.IntervalGraph()
>>> G.add_nodes_from([2, 4, 5])
>>> len(G)
3
```

Making copies and subgraphs

`ImpulseGraph.to_subgraph`(begin, end[, …])	Return a networkx Graph or MultiGraph which includes all the nodes and edges which have timestamps within the given interval.
`ImpulseGraph.to_snapshots`(number_of_snapshots)	Return a list of networkx Graph or MultiGraph objects as snapshots of the impulse graph in consecutive order.

dynetworkx.ImpulseGraph.to_subgraph

`ImpulseGraph.`**`to_subgraph`**(*begin*, *end*, *inclusive=(True, True)*, *multigraph=False*, *edge_data=False*, *edge_timestamp_data=False*, *node_data=False*)

Return a networkx Graph or MultiGraph which includes all the nodes and edges which have timestamps within the given interval.

Parameters

- **begin** (*int or float*) –
- **end** (*int or float*) – Must be bigger than or equal to begin.
- **inclusive** (*2-tuple boolean that determines inclusivity of begin and end*) –
- **multigraph** (*bool, optional (default= False)*) – If True, a networkx MultiGraph will be returned. If False, networkx Graph.
- **edge_data** (*bool, optional (default= False)*) – If True, edges will keep their attributes.
- **edge_timestamp_data** (*bool, optional (default= False)*) – If True, each edge's attribute will also include its timestamp data. If *edge_data= True* and there already exist edge attributes named timestamp it will be overwritten.
- **node_data** (*bool, optional (default= False)*) – if True, each node's attributes will be included.

See also:

`to_snapshots()` divide the impulse graph to snapshots

Notes

If multigraph= False, and edge_data=True or edge_interval_data=True, in case there are multiple edges, only one will show with one of the edge's attributes.

Note: nodes with no edges will not appear in any subgraph.

Examples

```
>>> G = dnx.ImpulseGraph()
>>> G.add_edges_from([(1, 2, 10), (2, 4, 11), (6, 4, 19), (2, 4, 15)])
>>> H = G.to_subgraph(4, 12)
>>> type(H)
<class 'networkx.classes.graph.Graph'>
>>> list(H.edges(data=True))
[(1, 2, {}), (2, 4, {})]
```

```
>>> H = G.to_subgraph(10, 12, edge_timestamp_data=True)
>>> type(H)
<class 'networkx.classes.graph.Graph'>
>>> list(H.edges(data=True))
[(1, 2, {'timestamp': 10}), (2, 4, {'timestamp': 11})]
```

```
>>> M = G.to_subgraph(4, 12, multigraph=True, edge_timestamp_data=True)
>>> type(M)
<class 'networkx.classes.multigraph.MultiGraph'>
>>> list(M.edges(data=True))
[(1, 2, {'timestamp': 10}), (2, 4, {'timestamp': 11})]
```

dynetworkx.ImpulseGraph.to_snapshots

`ImpulseGraph.`**`to_snapshots`**(*number_of_snapshots*, *multigraph=False*, *edge_data=False*, *edge_timestamp_data=False*, *node_data=False*, *return_length=False*)

Return a list of networkx Graph or MultiGraph objects as snapshots of the impulse graph in consecutive order.

Parameters

- **number_of_snapshots** (*integer*) – Number of snapshots to divide the interval graph into. Must be bigger than 1.
- **multigraph** (*bool, optional (default= False)*) – If True, a networkx MultiGraph will be returned. If False, networkx Graph.
- **edge_data** (*bool, optional (default= False)*) – If True, edges will keep their attributes.
- **edge_timestamp_data** (*bool, optional (default= False)*) – If True, each edge's attribute will also include its timestamp data. If *edge_data= True* and there already exist edge attributes named timestamp it will be overwritten.
- **node_data** (*bool, optional (default= False)*) – if True, each node's attributes will be included.
- **return_length** (*bool, optional (default= False)*) – If true, the length of snapshots will be returned as the second argument.

See also:

`to_subgraph()` subgraph based on an interval

Notes

In order to create snapshots, timestamp of edges of the impulse graph must be numbers.

If multigraph= False, and edge_data=True or edge_timestamp_data=True, in case there are multiple edges, only one will show with one of the edge's attributes.

Examples

Snapshots of NetworkX Graph

```
>>> G = dnx.ImpulseGraph()
>>> G.add_edges_from([(1, 2, 10), (2, 4, 11), (6, 4, 19), (2, 4, 15)])
>>> S, l = G.to_snapshots(2, edge_timestamp_data=True, return_length=True)
>>> S
[<networkx.classes.graph.Graph object at 0x100000>, <networkx.classes.graph.Graph␣
↪object at 0x150d00>]
>>> l
4.5
>>> for g in S:
>>> ... g.edges(data=True))
[(1, 2, {'timestamp': 10}), (2, 4, {'timestamp': 11})]
[(2, 4, {'timestamp': 15}), (4, 6, {'timestamp': 19})]
```

Snapshots of NetworkX MultiGraph

```
>>> S, l = G.to_snapshots(3, multigraph=True, edge_timestamp_data=True, return_
↪length=True)
>>> S
[<networkx.classes.multigraph.MultiGraph object at 0x1060d40b8>, <networkx.
↪classes.multigraph.MultiGraph object at 0x151020c9e8>, <networkx.classes.
↪multigraph.MultiGraph object at 0x151021d390>]
>>> l
3.0
>>> for g in S:
>>> ... g.edges(data=True))
[(1, 2, {'timestamp': 10}), (2, 4, {'timestamp': 11})]
[(2, 4, {'timestamp': 15})]
[(6, 4, {'timestamp': 19})]
```

Loading an impulse graph

ImpulseGraph.load_from_txt(path[, ...])	Read impulse graph in from path.
ImpulseGraph.save_to_txt(path[, delimiter])	Write impulse graph to path.

dynetworkx.ImpulseGraph.load_from_txt

static ImpulseGraph.**load_from_txt**(*path*, *delimiter=' '*, *nodetype=None*, *timestamptype=<class 'float'>*, *comments='#'*)

Read impulse graph in from path. Every line in the file must be an edge in the following format: "node node timestamp". Timestamps must be integers or floats. Nodes can be any hashable objects.

Parameters

- **`path`** (*`string or file`*) – Filename to read.
- **`nodetype`** (*`Python type, optional`*) – Convert nodes to this type.
- **`timestamptype`** (*`Python type, optional (default= float)`*) –
- **`timestamp to this type.`** (*`Convert`*) –
- **`must be an orderable type, ideally int or float. Other orderable types have not been fully tested.`** (*`This`*) –
- **`comments`** (*`string, optional`*) – Marker for comment lines
- **`delimiter`** (*`string, optional`*) – Separator for node labels. The default is whitespace. Cannot be =.

Returns **G** – The graph corresponding to the lines in edge list.

Return type *ImpulseGraph*

Examples

```
>>> G=dnx.ImpulseGraph.load_from_txt("my_dygraph.txt")
```

The optional nodetype is a function to convert node strings to nodetype.

For example

```
>>> G=dnx.ImpulseGraph.load_from_txt("my_dygraph.txt", nodetype=int)
```

will attempt to convert all nodes to integer type.

Since nodes must be hashable, the function nodetype must return hashable types (e.g. int, float, str, frozenset - or tuples of those, etc.)

dynetworkx.ImpulseGraph.save_to_txt

`ImpulseGraph.`**`save_to_txt`**(*path*, *delimiter=' '*)

Write impulse graph to path. Every line in the file must be an edge in the following format: "node node timestamp". Timestamps must be integers or floats. Nodes can be any hashable objects.

Parameters

- **`path`** (*`string or file`*) – Filename to read.
- **`delimiter`** (*`string, optional`*) – Separator for node labels. The default is whitespace. Cannot be =.

Examples

```
>>> G.save_to_txt("my_dygraph.txt")
```

Analyzing impulse graphs

ImpulseGraph.degree([node, begin, end, delta])	Return the degree of a specified node between time begin and end.

dynetworkx.ImpulseGraph.degree

`ImpulseGraph.`**`degree`**(*node=None*, *begin=None*, *end=None*, *delta=False*)

Return the degree of a specified node between time begin and end.

Parameters

- **node** (*Nodes can be, for example, strings or numbers.*) – Nodes must be hashable (and not None) Python objects.
- **begin** (*int or float, optional (default= beginning of the entire impulse graph)*) – Inclusive beginning time of the edge appearing in the impulse graph.
- **end** (*int or float, optional (default= end of the entire impulse graph)*) – Non-inclusive ending time of the edge appearing in the impulse graph.

Returns

Return type Integer value of degree of specified node.

Examples

```
>>> G = ImpulseGraph()
>>> G.add_edge(1, 2, 3)
>>> G.add_edge(2, 3, 8)
>>> G.degree(2)
2
>>> G.degree(2,2)
2
>>> G.degree(2,end=8)
1
>>> G.mean_degree()
1.33333
>>> G.degree(2,delta=True)
[(8, 1), (3, 1)]
```

Snapshot Graph

Overview

class `dynetworkx.`**`SnapshotGraph`**(***attr*)

Methods

Adding and removing nodes and edges

SnapshotGraph.__init__(**attr)	Initialize self.
SnapshotGraph.add_nodes_from(nbunch[, sbunch])	Adds nodes to snapshots in sbunch.
SnapshotGraph.add_edges_from(ebunch[, sbunch])	Adds edges to snapshots in sbunch.

dynetworkx.SnapshotGraph.__init__

SnapshotGraph.**__init__**(***attr*)

Initialize self. See help(type(self)) for accurate signature.

dynetworkx.SnapshotGraph.add_nodes_from

SnapshotGraph.**add_nodes_from**(*nbunch*, *sbunch=None*, ***attrs*)

Adds nodes to snapshots in sbunch.

Parameters

- **nbunch** (*container of nodes*) – Each node in the nbunch list will be added to all graphs indexed in sbunch.
- **sbunch** (*container of snapshot indexes, optional (default= None)*) – Each snapshot index in this list will be included in the returned list of node degrees. It is highly recommended that this list is sequential, however it can be out of order.

Returns

Return type None

Examples

```
>>> G = dnx.SnapshotGraph()
>>> G.add_snapshot([(1, 2), (1, 3)])
>>> G.add_snapshot([(1, 4), (1, 3)])
```

```
>>> G.add_nodes_from([5, 6, 7], [0])
>>> G.add_nodes_from([8, 9, 10, 11], [1])
>>> nx.adjacency_matrix(G.get()[0]).todense()
[[0 1 1 0 0 0]
 [1 0 0 0 0 0]
 [1 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]]
>>> nx.adjacency_matrix(G.get()[1]).todense()
[[0 1 1 0 0 0 0]
 [1 0 0 0 0 0 0]
 [1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]]
```

dynetworkx.SnapshotGraph.add_edges_from

`SnapshotGraph.`**`add_edges_from`**(*ebunch*, *sbunch=None*, ***attrs*)

Adds edges to snapshots in sbunch.

Parameters

- **`ebunch`** (*`container of edges`*) – Each edge in the ebunch list will be added to all graphs indexed in sbunch.
- **`sbunch`** (*`container of snapshot indexes, optional (default= None)`*) – Each snapshot index in this list will be included in the returned list of node degrees. It is highly recommended that this list is sequential, however it can be out of order.

Returns

Return type None

Examples

```
>>> G = dnx.SnapshotGraph()
>>> G.add_snapshot([(1, 2), (1, 3)])
>>> G.add_snapshot([(1, 4), (1, 3)])
```

```
>>> G.add_edges_from([(5, 6), (7, 6)], [0])
>>> G.add_edges_from([(8, 9), (10, 11)], [0, 1])
>>> nx.adjacency_matrix(G.get()[0]).todense()
[[0 1 1 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0]
 [0 0 0 1 0 1 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0 1 0]]
>>> nx.adjacency_matrix(G.get()[1]).todense()
[[0 1 1 0 0 0 0]
 [1 0 0 0 0 0 0]
 [1 0 0 0 0 0 0]
 [0 0 0 0 1 0 0]
 [0 0 0 1 0 0 0]
 [0 0 0 0 0 0 1]
 [0 0 0 0 0 1 0]]
```

Manipulating Snapshots

`SnapshotGraph.insert`(graph[, snap_len, . . .])	Insert a graph into the snapshot graph, with options for inserting at a given index, with some snapshot length.
`SnapshotGraph.add_snapshot`([ebunch, graph, . . .])	Add a snapshot with a bunch of edge values.

dynetworkx.SnapshotGraph.insert

`SnapshotGraph.`**`insert`**(*graph*, *snap_len=None*, *num_in_seq=None*)

Insert a graph into the snapshot graph, with options for inserting at a given index, with some snapshot length.

Parameters

- **graph** (*networkx graph object*) – A networkx graph to be inserted into snapshot graph.
- **snap_len** (*integer, optional (default= None)*) – Length of the snapshot.
- **num_in_seq** (*integer, optional (default= None)*) – Time slot to begin insertion at.

Returns

Return type None

Examples

```
>>> nxG1 = nx.Graph()
>>> nxG1.add_edges_from([(1, 2), (1, 3)])
>>> G = dnx.SnapshotGraph()
>>> G.insert(nxG1, 0)
```

dynetworkx.SnapshotGraph.add_snapshot

`SnapshotGraph.`**`add_snapshot`**(*ebunch=None*, *graph=None*, *num_in_seq=None*)

Add a snapshot with a bunch of edge values.

Parameters

- **ebunch** (*container of edges, optional (default= None)*) – Each edge in the ebunch list will be included to all added graphs.
- **graph** (*networkx graph object, optional (default= None)*) – networkx graph to be inserted into snapshot graph.
- **num_in_seq** (*integer, optional (default= None)*) – Time slot to begin insertion at.

Returns

Return type None

Examples

```
>>> G = dnx.SnapshotGraph()
>>> G.add_snapshot([(1, 4), (1, 3)])
```

Reporting Snapshots

SnapshotGraph.__len__()	Return the number of snapshots.
SnapshotGraph.order([sbunch])	Returns order of each graph requested in 'sbunch'.
SnapshotGraph.has_node(n[, sbunch])	Gets boolean list of if a snapshot in 'sbunch' contains node 'n'.
SnapshotGraph.size([sbunch, weight])	Returns the size of each graph index as specified in sbunch as a list.
SnapshotGraph.is_directed([sbunch])	Returns a list of boolean values for if the graph at the index is a directed graph.
SnapshotGraph.is_multigraph([sbunch])	Returns a list of boolean values for if the graph at the index is a multigraph.
SnapshotGraph.number_of_nodes([sbunch])	Gets number of nodes in each snapshot requested in 'sbunch'.
SnapshotGraph.degree([sbunch, nbunch, weight])	Return a list of tuples containing the degrees of each node in each snapshot

dynetworkx.SnapshotGraph.__len__

SnapshotGraph.**__len__**()

Return the number of snapshots. Use: 'len(G)'.

Returns num_snapshots – The number of snapshots in the graph.

Return type int

Examples

```
>>> nxG1 = nx.Graph()
>>> nxG2 = nx.Graph()
>>>
>>> nxG1.add_edges_from([(1, 2), (1, 3)])
>>> nxG2.add_edges_from([(1, 4), (1, 3)])
>>>
>>> G = dnx.SnapshotGraph()
>>> G.add_snapshot(graph=nxG1)
>>> G.add_snapshot(graph=nxG2)
>>> len(G)
2
```

dynetworkx.SnapshotGraph.order

SnapshotGraph.**order**(*sbunch=None*)

Returns order of each graph requested in 'sbunch'.

Parameters sbunch (*container of snapshot indexes, optional (default= None)*) – Each snapshot index in this list will be included in the returned list of node orders. It is highly recommended that this list is sequential, however it can be out of order.

Returns snapshot_orders – A list of the orders of each snapshot.

Return type list

Examples

```
>>> G = dnx.SnapshotGraph()
>>> G.add_snapshot([(1, 2), (1, 3)])
>>> G.add_snapshot([(1, 4), (1, 3)])
>>> G.order(sbunch=[1])
[3]
>>> G.order(sbunch=[0, 1])
[3, 3]
```

dynetworkx.SnapshotGraph.has_node

`SnapshotGraph.`**`has_node`**(*n*, *sbunch=None*)

Gets boolean list of if a snapshot in 'sbunch' contains node 'n'.

Parameters

- **n** (*node*) – Node to be checked for in requested snapshots.
- **sbunch** (*container of snapshot indexes, optional (default= None)*) – Each snapshot index in this list will be included in the returned list of if the snapshot graph includes the node. It is highly recommended that this list is sequential, however it can be out of order.

Returns

Return type List of boolean values if index in sbunch contains n.

Examples

```
>>> G = dnx.SnapshotGraph()
>>> G.add_snapshot([(1, 2), (1, 3)])
>>> G.add_snapshot([(1, 4), (1, 3)])
>>> G.has_node(1, sbunch=[1])
[True]
>>> G.has_node(1)
[True, True]
```

dynetworkx.SnapshotGraph.size

`SnapshotGraph.`**`size`**(*sbunch=None*, *weight=None*)

Returns the size of each graph index as specified in sbunch as a list.

Parameters

- **sbunch** (*container of snapshot indexes, optional (default= None)*) – Each snapshot index in this list will be included in the returned list of sizes. It is highly recommended that this list is sequential, however it can be out of order.
- **weight** (*string, optional (default=None)*) – The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1.

Returns **size_list** – List of sizes of each graph indexed in sbunch.

Return type list

Examples

```
>>> G = dnx.SnapshotGraph()
>>> G.add_snapshot([(1, 2), (1, 3)])
>>> G.add_snapshot([(1, 4), (1, 3)])
>>> G.size(sbunch=[0, 1])
[2, 2]
>>> G.size()
[2, 2]
```

dynetworkx.SnapshotGraph.is_directed

`SnapshotGraph.`**`is_directed`**`(`*sbunch=None*`)`

Returns a list of boolean values for if the graph at the index is a directed graph.

Parameters **`sbunch`** (*`container of snapshot indexes, optional (default= None)`*) – Each snapshot index in this list will be included in the returned list of booleans. It is highly recommended that this list is sequential, however it can be out of order.

Returns **is_direct_list** – List of boolean values if index in sbunch is a directed graph.

Return type list

Examples

```
>>> G = dnx.SnapshotGraph()
>>> G.add_snapshot([(1, 2), (1, 3)])
>>> G.add_snapshot([(1, 4), (1, 3)])
>>> G.is_directed(sbunch=[0, 1])
[False, False]
>>> G.is_directed()
[False, False]
```

dynetworkx.SnapshotGraph.is_multigraph

`SnapshotGraph.`**`is_multigraph`**`(`*sbunch=None*`)`

Returns a list of boolean values for if the graph at the index is a multigraph.

Parameters **`sbunch`** (*`container of snapshot indexes, optional (default= None)`*) – Each snapshot index in this list will be included in the returned list of booleans. It is highly recommended that this list is sequential, however it can be out of order.

Returns **mutli_list** – List of boolean values if index in sbunch is a multigraph.

Return type list

Examples

```
>>> G = dnx.SnapshotGraph()
>>> G.add_snapshot([(1, 2), (1, 3)])
>>> G.add_snapshot([(1, 4), (1, 3)])
>>> G.is_multigraph(sbunch=[0, 1])
```

(continues on next page)

(continued from previous page)

```
[False, False]
>>> G.is_multigraph()
[False, False]
```

dynetworkx.SnapshotGraph.number_of_nodes

`SnapshotGraph.`**`number_of_nodes`**`(`*sbunch=None*`)`

Gets number of nodes in each snapshot requested in 'sbunch'.

Parameters **`sbunch`** (*`container of snapshot indexes, optional (default= None)`*) – Each snapshot index in this list will be included in the returned list of number of nodes in the snapshot. It is highly recommended that this list is sequential, however it can be out of order.

Returns **num_nodes** – A list of of the number of nodes in each requested snapshot.

Return type list

Examples

```
>>> G = dnx.SnapshotGraph()
>>> G.add_snapshot([(1, 2), (1, 3)])
>>> G.add_snapshot([(1, 4), (1, 3)])
>>> G.number_of_nodes(sbunch=[1])
[3]
>>> G.number_of_nodes(sbunch=[0, 1])
[3, 3]
```

dynetworkx.SnapshotGraph.degree

`SnapshotGraph.`**`degree`**`(`*sbunch=None*, *nbunch=None*, *weight=None*`)`

Return a list of tuples containing the degrees of each node in each snapshot

Parameters

- **`sbunch`** (*`container of snapshot indexes, optional (default= None)`*) – Each snapshot index in this list will be included in the returned list of node degrees. It is highly recommended that this list is sequential, however it can be out of order.
- **`nbunch`** (*`container of nodes, optional (default= None)`*) – Each node in the nbunch list will be included in the returned list of node degrees.
- **`weight`** (*`string, optional (default= None)`*) – The edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1. The degree is the sum of the edge weights adjacent to the node.

Returns **degree_list** – List of DegreeView objects containing the degree of each node, indexed by requested snapshot.

Return type list

Examples

```
>>> G = dnx.SnapshotGraph()
>>> G.add_snapshot([(1, 2), (1, 3)])
>>> G.add_snapshot([(1, 4), (1, 3)])
>>> G.degree(sbunch=[1])
[DegreeView({1: 2, 4: 1, 3: 1})]
>>> G.degree(nbunch=[1, 2])
[DegreeView({1: 2, 2: 1}), DegreeView({1: 2})]
```

Making copies and subgraphs

`SnapshotGraph.subgraph`(nbunch[, sbunch])	Return a snapshot graph containing only the nodes in bunch, and snapshot indexes in sbunch.
`SnapshotGraph.to_directed`([sbunch])	Returns a list of networkx directed graph objects.
`SnapshotGraph.to_undirected`([sbunch])	Returns a list of networkx graph objects.

dynetworkx.SnapshotGraph.subgraph

`SnapshotGraph.`**`subgraph`**(*nbunch*, *sbunch=None*)

Return a snapshot graph containing only the nodes in bunch, and snapshot indexes in sbunch.

Parameters

- **`nbunch`** (*`container of nodes`*) – Each node in the nbunch list will be included in all subgraphs indexed in sbunch.
- **`sbunch`** (*`container of edges, optional (default= None)`*) – Each snapshot index in this list will be included in the returned list of subgraphs. It is highly recommended that this list is sequential, however it can be out of order.

Returns **snap_graph** – Contains only the nodes in bunch, and snapshot indexes in sbunch.

Return type SnapshotGraph object

Examples

```
>>> G = dnx.SnapshotGraph()
>>> G.add_snapshot([(1, 2), (2, 3), (4, 6), (2, 4)])
>>> G.add_snapshot([(1, 2), (2, 3), (4, 6), (2, 4)])
>>> H = G.subgraph([4, 6])
>>> type(H)
<class 'snapshotgraph.SnapshotGraph'>
>>> list(H.get([0])[0].edges(data=True))
[(4, 6, {})]
```

dynetworkx.SnapshotGraph.to_directed

`SnapshotGraph.`**`to_directed`**(*sbunch=None*)

Returns a list of networkx directed graph objects.

Parameters **sbunch** (*container of snapshot indexes, optional (default= None)*) – Each snapshot index in this list will be included in the returned list of directed graphs. It is highly recommended that this list is sequential, however it can be out of order.

Returns **direct_list** – List of networkx directed graph objects.

Return type list

Examples

```
>>> G = dnx.SnapshotGraph()
>>> G.add_snapshot([(1, 2), (1, 3)])
>>> G.add_snapshot([(1, 4), (1, 3)])
>>> G.to_directed(sbunch=[0, 1])
[<networkx.classes.digraph.DiGraph object at 0x7f1a6de49dd8>, <networkx.classes.
↪digraph.DiGraph object at 0x7f1a6de49e10>]
```

dynetworkx.SnapshotGraph.to_undirected

`SnapshotGraph.`**`to_undirected`**`(`*sbunch=None*`)`

Returns a list of networkx graph objects.

Parameters **sbunch** (*container of snapshot indexes, optional (default= None)*) – Each snapshot index in this list will be included in the returned list of undirected graphs. It is highly recommended that this list is sequential, however it can be out of order.

Returns **undirect_list** – List of networkx graph objects.

Return type list

Examples

```
>>> G = dnx.SnapshotGraph()
>>> G.add_snapshot([(1, 2), (1, 3)])
>>> G.add_snapshot([(1, 4), (1, 3)])
>>> G.to_directed(sbunch=[0, 1])
[<networkx.classes.graph.Graph object at 0x7ff532219e10>, <networkx.classes.graph.
↪Graph object at 0x7ff532219e48>]
```

5.5 Developer Guide

DyNetworkX is still under development and the repository is kept private. If you are interested in getting access to the project as a developer, go to *Need Help?* for contact information.

5.6 License

BSD 3-Clause License

Copyright (c) 2018, IDEAS Lab @ The University of Toledo.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

5.7 Need Help?

If you have any trouble with DyNetworkX, please email Makan.Arastuie@rockets.utoledo.edu

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index