DyNetworkX Documentation

Release 0.1

Makan Arastuie

Jul 28, 2022

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 now available for `pip` install!

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 Reference

Date Jul 28, 2022

5.3.1 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.

5.3.2 Graph Types

Interval Graph

Overview

Methods

Adding and removing nodes and edges

IntervalGraph.__init__
IntervalGraph.add_node
IntervalGraph.add_nodes_from
IntervalGraph.remove_node
IntervalGraph.add_edge
IntervalGraph.add_edges_from
IntervalGraph.remove_edge

Reporting interval graph, nodes and edges

IntervalGraph.nodes
IntervalGraph.has_node
IntervalGraph.edges
IntervalGraph.has_edge
IntervalGraph.__contains__
IntervalGraph.__str__
IntervalGraph.interval

Counting nodes and edges

IntervalGraph.number_of_nodes
IntervalGraph.__len__

Making copies and subgraphs

IntervalGraph.to_subgraph
IntervalGraph.to_snapshots

Continued on next page

Table 4 – continued from previous page

IntervalGraph.to_snapshot_graph

Loading an interval graph

IntervalGraph.load_from_txt
IntervalGraph.save_to_txt
IntervalGraph.from_networkx_graph
IntervalGraph.from_snapshot_graph

Analyzing interval graphs

IntervalGraph.degree

Directed Interval Graph

Overview

Methods

Adding and removing nodes and edges

IntervalDiGraph.__init__
IntervalDiGraph.add_node
IntervalDiGraph.add_nodes_from
IntervalDiGraph.remove_node
IntervalDiGraph.add_edge
IntervalDiGraph.add_edges_from
IntervalDiGraph.remove_edge

Reporting interval graph, nodes and edges

IntervalDiGraph.nodes
IntervalDiGraph.has_node
IntervalDiGraph.edges
IntervalDiGraph.has_edge
IntervalDiGraph.__contains__
IntervalDiGraph.__str__
IntervalDiGraph.interval

Counting nodes and edges

IntervalDiGraph.number_of_nodes
IntervalDiGraph.__len__

Making copies and subgraphs

IntervalDiGraph.to_subgraph
IntervalDiGraph.to_snapshots
IntervalDiGraph.to_snapshot_graph

Loading an interval graph

IntervalDiGraph.load_from_txt
IntervalDiGraph.save_to_txt
IntervalDiGraph.from_networkx_graph
IntervalDiGraph.from_snapshot_graph

Analyzing interval graphs

IntervalDiGraph.degree
IntervalDiGraph.in_degree
IntervalDiGraph.out_degree

Impulse Graph

Overview

Methods

Adding and removing nodes and edges

ImpulseGraph.__init__
ImpulseGraph.add_node
ImpulseGraph.add_nodes_from
ImpulseGraph.remove_node
ImpulseGraph.add_edge
ImpulseGraph.add_edges_from
ImpulseGraph.remove_edge

Reporting impulse graph, nodes and edges

ImpulseGraph.nodes
ImpulseGraph.has_node
ImpulseGraph.edges
ImpulseGraph.has_edge
ImpulseGraph.__contains__
ImpulseGraph.__str__
ImpulseGraph.interval

Counting nodes and edges

ImpulseGraph.number_of_nodes
ImpulseGraph.__len__

Making copies and subgraphs

ImpulseGraph.to_subgraph
ImpulseGraph.to_snapshots
ImpulseGraph.to_snapshot_graph

Loading an impulse graph

ImpulseGraph.load_from_txt
ImpulseGraph.save_to_txt

Analyzing impulse graphs

ImpulseGraph.degree

Directed Impulse Graph

Overview

Methods

Adding and removing nodes and edges

ImpulseDiGraph.__init__
ImpulseDiGraph.add_node
ImpulseDiGraph.add_nodes_from
ImpulseDiGraph.remove_node
ImpulseDiGraph.add_edge
ImpulseDiGraph.add_edges_from
ImpulseDiGraph.remove_edge

Reporting impulse graph, nodes and edges

ImpulseDiGraph.nodes
ImpulseDiGraph.has_node
ImpulseDiGraph.edges
ImpulseDiGraph.has_edge
ImpulseDiGraph.__contains__
ImpulseDiGraph.__str__

Continued on next page

Table 20 – continued from previous page

ImpulseDiGraph.interval

Counting nodes and edges

ImpulseDiGraph.number_of_nodes
ImpulseDiGraph.__len__

Making copies and subgraphs

ImpulseDiGraph.to_subgraph
ImpulseDiGraph.to_snapshots
ImpulseDiGraph.to_snapshot_graph

Loading an impulse graph

ImpulseDiGraph.load_from_txt
ImpulseDiGraph.save_to_txt

Analyzing impulse graphs

ImpulseDiGraph.degree
ImpulseDiGraph.in_degree
ImpulseDiGraph.out_degree

Snapshot Graph

Overview

Methods

Adding and removing nodes and edges

SnapshotGraph.__init__
SnapshotGraph.add_nodes_from
SnapshotGraph.add_edges_from

Manipulating Snapshots

SnapshotGraph.insert
SnapshotGraph.add_snapshot

Reporting Snapshots

SnapshotGraph.__len__
SnapshotGraph.order
SnapshotGraph.has_node
SnapshotGraph.size
SnapshotGraph.is_directed
SnapshotGraph.is_multigraph
SnapshotGraph.number_of_nodes
SnapshotGraph.degree

Making copies and subgraphs

SnapshotGraph.subgraph
SnapshotGraph.to_directed
SnapshotGraph.to_undirected

Directed Snapshot Graph

Overview

Methods

Adding and removing nodes and edges

SnapshotDiGraph.__init__
SnapshotDiGraph.add_nodes_from
SnapshotDiGraph.add_edges_from

Manipulating Snapshots

SnapshotDiGraph.insert
SnapshotDiGraph.add_snapshot

Reporting Snapshots

SnapshotDiGraph.__len__
SnapshotDiGraph.order
SnapshotDiGraph.has_node
SnapshotDiGraph.size
SnapshotDiGraph.is_directed
SnapshotDiGraph.is_multigraph
SnapshotDiGraph.number_of_nodes
SnapshotDiGraph.degree

Making copies and subgraphs

`SnapshotDiGraph.subgraph`
`SnapshotDiGraph.to_directed`
`SnapshotDiGraph.to_undirected`

5.4 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.

5.5 License

5.6 Need Help?

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

CHAPTER 6

Indices and tables

- genindex
- modindex
- search