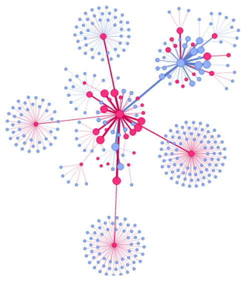
pyvis Documentation

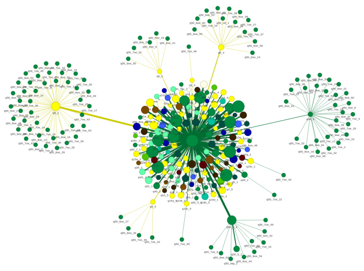
Release 0.1.3.1

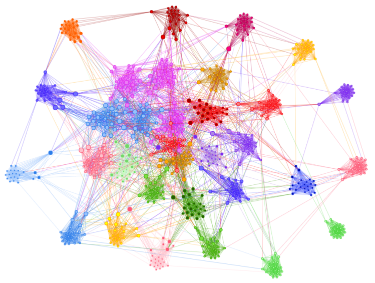
Giancarlo Perrone, Shashank Soni

Nov 11, 2022

Contents







CHAPTER 1

Contents:

1.1 Installation

1.1.1 Install with pip

```
$ pip install pyvis
```

Or you can download an archive of the project here. To install, unpack it and run the following in the top-level directory:

```
$ python setup.py install
```

1.2 Introduction

The goal of this project is to build a python based approach to constructing and visualizing network graphs in the same space. A pyvis network can be customized on a per node or per edge basis. Nodes can be given colors, sizes, labels, and other metadata. Each graph can be interacted with, allowing the dragging, hovering, and selection of nodes and edges. Each graph's layout algorithm can be tweaked as well to allow experimentation with rendering of larger graphs.

Pyvis is built around the amazing VisJS library.

1.3 Tutorial

The pyvis library is meant for quick generation of visual network graphs with minimal python code. It is designed as a wrapper around the popular Javascript visJS library found at this link.

1.3.1 Getting started

All networks must be instantiated as a `Network` class instance

```
>>> from pyvis.network import Network
>>> net = Network()
```

1.3.2 Add nodes to the network

```
>>> net.add_node(1, label="Node 1") # node id = 1 and label = Node 1
>>> net.add_node(2) # node id and label = 2
```

Here, the first parameter to the add_node method is the desired ID to give the Node. This can be a string or a numeric. The label argument is the string that will be visibly attached to the node in the final visualization. If no label argument is specified then the node id will be used as a label.

Note: The `ID` parameter must be unique

You can also add a list of nodes

```
>>> nodes = ["a", "b", "c", "d"]
>>> net.add_nodes(nodes) # node ids and labels = ["a", "b", "c", "d"]
>>> net.add_nodes("hello") # node ids and labels = ["h", "e", "l", "o"]
```

Note: `network.Network.add_nodes()` accepts any iterable as long as the contents are strings or numerics

1.3.3 Node properties

A call to `add_node()` supports various node properties that can be set individually. All of these properties can be found here, courtesy of VisJS. For the direct Python translation of these attributes, reference the `network.Network.add_node()` docs.

Note: Through no fault of pyvis, some of the attributes in the VisJS documentation do not work as expected, or at all. Pyvis can translate into the JavaScript elements for VisJS but after that it's up to VisJS!

1.3.4 Indexing a Node

Use the `get_node()` method to index a node by its ID:

```
>>> net.add_nodes(["a", "b", "c"])
>>> net.get_node("c")
>>> {'id': 'c', 'label': 'c', 'shape': 'dot'}
```

1.3.5 Adding list of nodes with properties

When using the `network.Network.add_nodes()` method optional keyword arguments can be passed in to add properties to these nodes as well. The valid properties in this case are

```
>>> ['size', 'value', 'title', 'x', 'y', 'label', 'color']
```

Each of these keyword args must be the same length as the nodes parameter to the method.

Example:

```
>>> g = Network()
>>> g.add_nodes([1,2,3], value=[10, 100, 400],
                         title=['I am node 1', 'node 2 here', 'and im node 3'],
                         x=[21.4, 54.2, 11.2],
                         y=[100.2, 23.54, 32.1],
                         label=['NODE 1', 'NODE 2', 'NODE 3'],
                         color=['#00ff1e', '#162347', '#dd4b39'])
```

Note: If you mouse over each node you will see that the `title` of a node attribute is responsible for rendering data on mouse hover. You can add `HTML` in your `title` string and it will be rendered as such.

Note: The `color` attribute can also be a plain HTML `color` like red or blue. You can also specify the full `rgba` specification if needed. The VisJS documentation has more details.

Detailed optional argument documentation for nodes are in the `network.Network.add_node()` method documentation.

1.3.6 Edges

Assuming the network's nodes exist, the edges can then be added according to node id's

```
>>> net.add_node(0, label='a')
>>> net.add_node(1, label='b')
>>> net.add_edge(0, 1)
```

Edges can contain a `weight` attribute as well

```
>>> net.add_edge(0, 1, weight=.87)
```

Edges can be customized and documentation on options can be found at `network.Network.add_edge()` method documentation, or by referencing the original VisJS edge module docs.

1.3.7 Networkx integration

An easy way to visualize and construct pyvis networks is to use Networkx and use pyvis's built-in networkx helper method to translate the graph. Note that the Networkx node properties with the same names as those consumed by pyvis (e.g., `title`) are translated directly to the correspondingly-named pyvis node attributes.

```
>>> from pyvis.network import Network
>>> import networkx as nx
>>> nx_graph = nx.cycle_graph(10)
>>> nx_graph.nodes[1]['title'] = 'Number 1'
>>> nx_graph.nodes[1]['group'] = 1
>>> nx_graph.nodes[3]['title'] = 'I belong to a different group!'
>>> nx_graph.nodes[3]['group'] = 10
```

(continues on next page)

(continued from previous page)

```
>>> nx_graph.add_node(20, size=20, title='couple', group=2)
>>> nx_graph.add_node(21, size=15, title='couple', group=2)
>>> nx_graph.add_edge(20, 21, weight=5)
>>> nx_graph.add_node(25, size=25, label='lonely', title='lonely node', group=3)
>>> nt = Network('500px', '500px')
# populates the nodes and edges data structures
>>> nt.from_nx(nx_graph)
>>> nt.show('nx.html')
```

1.3.8 Visualization

The displaying of a graph is achieved by a single method call on `network.Network.show()` after the underlying network is constructed. The interactive visualization is presented as a static HTML file.

```
>>> net.toggle_physics(True)
>>> net.show('mygraph.html')
```

Note: Triggering the `toggle_physics()` method allows for more fluid graph interactions

1.3.9 Example: Visualizing a Game of Thrones character network

The following code block is a minimal example of the capabilities of pyvis.

```
from pyvis.network import Network
import pandas as pd

got_net = Network(height="750px", width="100%", bgcolor="#222222", font_color="white")

# set the physics layout of the network
got_net.barnes_hut()
got_data = pd.read_csv("../../notebooks/NetworkOfThrones.csv")

sources = got_data['Source']
targets = got_data['Target']
weights = got_data['Weight']

edge_data = zip(sources, targets, weights)

for e in edge_data:
                src = e[0]
                dst = e[1]
                w = e[2]

                got_net.add_node(src, src, title=src)
                got_net.add_node(dst, dst, title=dst)
                got_net.add_edge(src, dst, value=w)

neighbor_map = got_net.get_adj_list()

# add neighbor data to node hover data
for node in got_net.nodes:
```

(continues on next page)

(continued from previous page)

```
                node["title"] += " Neighbors:<br>" + "<br>".join(neighbor_map[node["id
↪"]])
                node["value"] = len(neighbor_map[node["id"]])

got_net.show("gameofthrones.html")
```

If you want to try out the above code, the csv data source can be downloaded

Note: The `title` attribute of each node is responsible for rendering data on node hover.

1.3.10 Using the configuration UI to dynamically tweak `Network` settings

You also have the option of supplying your visualization with a UI used to dynamically alter some of the settings pertaining to your network. This could be useful for finding the most optimal parameters to your graph's physics and layout function.

```
>>> net.show_buttons(filter_=['physics'])
```

Note: You can copy/paste the output from the *generate options* button in the above UI into `network.Network.set_options()` to finalize your results from experimentation with the settings.

1.3.11 Filtering and Highlighting the nodes

You can highlight a node and its neighboring edges and nodes by clicking on the node or choosing the drop down above when select_menu option is set as True. The selected node and its neighbours will be highlighted while the rest of the network is greyed out.

```
>>> got_net = Network(height="750px", width="100%", bgcolor="#222222", font_color=
↪"white", select_menu=True)
```

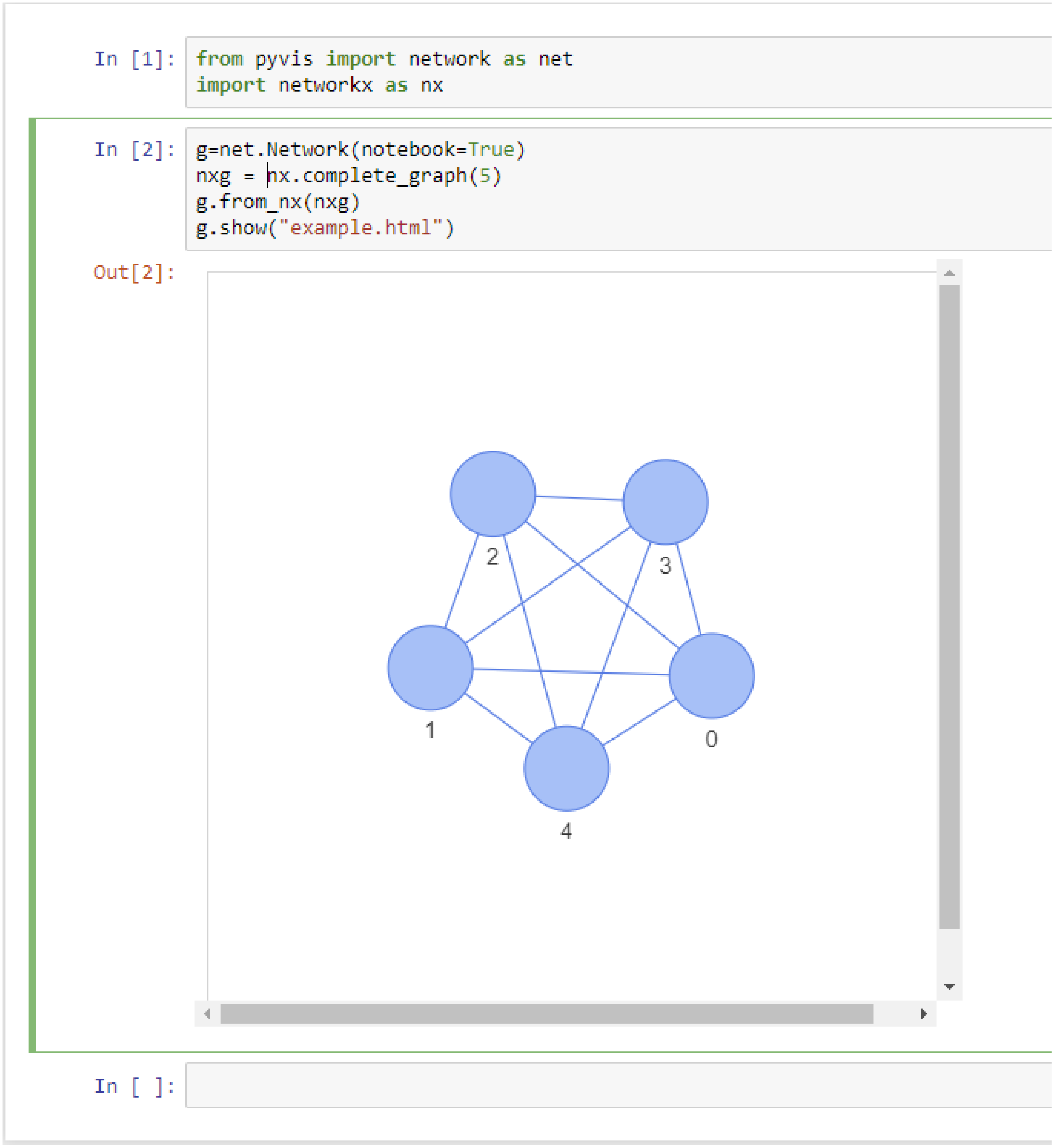
You can also filter on certain objects of the network like nodes and edges. You can activate this feature by passing filter_menu option as True. With this option turned on, you can build a query choosing edges or nodes, then choosing the attribute to filter on and finally a value or multiple values to filter. When you filter on nodes, the selected node will be highlighted and rest of the network will be hidden. If the selected nodes are connected the edges will also be highlighted. When you filter on edges, the nodes connecting them will be highlighted along with the edges.

```
>>> got_net = Network(height="750px", width="100%", bgcolor="#222222", font_color=
↪"white", filter_menu=True)
```

Note: You can use these two features independently and can also combine them to get a customized view of the network

1.3.12 Using pyvis within Jupyter notebook

Pyvis supports Jupyter notebook embedding through the use of the `network.Network()` constructor. The network instance must be "prepped" during instantiation by supplying the *notebook=True* kwarg. Example:



```
In [1]: from pyvis import network as net
        import networkx as nx

In [2]: g=net.Network(notebook=True)
        nxg = nx.complete_graph(5)
        g.from_nx(nxg)
        g.show("example.html")

Out[2]:

In [ ]:
```

Note: while using notebook in chrome browser, to render the graph, pass additional kwarg 'cdn_resources' as 'remote' or 'inline'

1.4 License

Pyvis is distributed with the BSD 3 Clause license.

1.5 Documentation

class pyvis.network.**Network**(*height='600px'*, *width='100%'*, *directed=False*, *notebook=False*, *neighborhood_highlight=False*, *select_menu=False*, *filter_menu=False*, *bgcolor='#ffffff'*, *font_color=False*, *layout=None*, *heading=''*, *cdn_resources='local'*)

The Network class is the focus of this library. All viz functionality should be implemented off of a Network instance.

To instantiate:

```
>>> nt = Network()
```

add_edge(*source*, *to*, ***options*)

Adding edges is done based off of the IDs of the nodes. Order does not matter unless dealing with a directed graph.

```
>>> nt.add_edge(0, 1) # adds an edge from node ID 0 to node ID
>>> nt.add_edge(0, 1, value = 4) # adds an edge with a width of 4
```

Parameters

- **arrowStrikethrough** (*bool*) – When false, the edge stops at the arrow. This can be useful if you have thick lines and you want the arrow to end in a point. Middle arrows are not affected by this.

- **from** (*str or num*) – Edges are between two nodes, one to and one from. This is where you define the from node. You have to supply the corresponding node ID. This naturally only applies to individual edges.
- **hidden** (*bool*) – When true, the edge is not drawn. It is part still part of the physics simulation however!
- **physics** (*bool*) – When true, the edge is part of the physics simulation. When false, it will not act as a spring.
- **title** (*str*) – The title is shown in a pop-up when the mouse moves over the edge.
- **to** (*str or num*) – Edges are between two nodes, one to and one from. This is where you define the to node. You have to supply the corresponding node ID. This naturally only applies to individual edges.
- **value** (*num*) – When a value is set, the edges' width will be scaled using the options in the scaling object defined above.
- **width** (*num*) – The width of the edge. If value is set, this is not used.

add_edges (*edges*)

This method serves to add multiple edges between existing nodes in the network instance. Adding of the edges is done based off of the IDs of the nodes. Order does not matter unless dealing with a directed graph.

Parameters **edges** – A list of tuples, each tuple consists of source of edge, edge destination and and optional width.

add_node (*n_id, label=None, shape='dot', color='#97c2fc', **options*)

This method adds a node to the network, given a mandatory node ID. Node labels default to node ids if no label is specified during the call.

```
>>> nt = Network("500px", "500px")
>>> nt.add_node(0, label="Node 0")
>>> nt.add_node(1, label="Node 1", color = "blue")
```

Parameters

- **n_id** (*str or int*) – The id of the node. The id is mandatory for nodes and they have to be unique. This should obviously be set per node, not globally.
- **label** (*str or int*) – The label is the piece of text shown in or under the node, depending on the shape.
- **borderWidth** (*num (optional)*) – The width of the border of the node.
- **borderWidthSelected** (*num (optional)*) – The width of the border of the node when it is selected. When undefined, the borderWidth * 2 is used.
- **brokenImage** (*str (optional)*) – When the shape is set to image or circularImage, this option can be an URL to a backup image in case the URL supplied in the image option cannot be resolved.
- **group** (*str (optional)*) – When not undefined, the node will belong to the defined group. Styling information of that group will apply to this node. Node specific styling overrides group styling.
- **hidden** (*bool (optional)*) – When true, the node will not be shown. It will still be part of the physics simulation though!

- **image** (*str (optional)*) – When the shape is set to image or circularImage, this option should be the URL to an image. If the image cannot be found, the brokenImage option can be used.
- **labelHighlightBold** (*bool (optional)*) – Determines whether or not the label becomes bold when the node is selected.
- **level** (*num (optional)*) – When using the hierarchical layout, the level determines where the node is going to be positioned.
- **mass** (*num (optional)*) – The barnesHut physics model (which is enabled by default) is based on an inverted gravity model. By increasing the mass of a node, you increase it's repulsion. Values lower than 1 are not recommended.
- **physics** (*bool (optional)*) – When false, the node is not part of the physics simulation. It will not move except for from manual dragging.
- **shape** (*str (optional)*) – The shape defines what the node looks like. There are two types of nodes. One type has the label inside of it and the other type has the label underneath it. The types with the label inside of it are: ellipse, circle, database, box, text. The ones with the label outside of it are: image, circularImage, diamond, dot, star, triangle, triangleDown, square and icon.
- **size** (*num (optional)*) – The size is used to determine the size of node shapes that do not have the label inside of them. These shapes are: image, circularImage, diamond, dot, star, triangle, triangleDown, square and icon.
- **title** (*str or html element (optional)*) – Title to be displayed when the user hovers over the node. The title can be an HTML element or a string containing plain text or HTML.
- **value** (*num (optional)*) – When a value is set, the nodes will be scaled using the options in the scaling object defined above.
- **x** (*num (optional)*) – This gives a node an initial x position. When using the hierarchical layout, either the x or y position is set by the layout engine depending on the type of view. The other value remains untouched. When using stabilization, the stabilized position may be different from the initial one. To lock the node to that position use the physics or fixed options.
- **y** (*num (optional)*) – This gives a node an initial y position. When using the hierarchical layout,either the x or y position is set by the layout engine depending on the type of view. The other value remains untouched. When using stabilization, the stabilized position may be different from the initial one. To lock the node to that position use the physics or fixed options.

add_nodes (*nodes*, ***kwargs*)

This method adds multiple nodes to the network from a list. Default behavior uses values of 'nodes' for node ID and node label properties. You can also specify other lists of properties to go along each node.

Example:

```
>>> g = net.Network()
>>> g.add_nodes([1, 2, 3], size=[2, 4, 6], title=["n1", "n2", "n3"])
>>> g.nodes
>>> [{'id': 1, 'label': 1, 'shape': 'dot', 'size': 2, 'title': 'n1'},
```

Output:

```
>>> {'id': 2, 'label': 2, 'shape': 'dot', 'size': 4, 'title': 'n2'},
>>> {'id': 3, 'label': 3, 'shape': 'dot', 'size': 6, 'title': 'n3'}]
```

Parameters **`nodes`** (*`list`*) – A list of nodes.

`barnes_hut`(*gravity=-80000*, *central_gravity=0.3*, *spring_length=250*, *spring_strength=0.001*, *damping=0.09*, *overlap=0*)

BarnesHut is a quadtree based gravity model. It is the fastest. default and recommended solver for non-hierarchical layouts.

Parameters

- **`gravity`** (*`int`*) – The more negative the gravity value is, the stronger the repulsion is.
- **`central_gravity`** (*`float`*) – The gravity attractor to pull the entire network to the center.
- **`spring_length`** (*`int`*) – The rest length of the edges
- **`spring_strength`** (*`float`*) – The strong the edges springs are
- **`damping`** (*`float`*) – A value ranging from 0 to 1 of how much of the velocity from the previous physics simulation iteration carries over to the next iteration.
- **`overlap`** (*`float`*) – When larger than 0, the size of the node is taken into account. The distance will be calculated from the radius of the encompassing circle of the node for both the gravity model. Value 1 is maximum overlap avoidance.

`force_atlas_2based`(*gravity=-50*, *central_gravity=0.01*, *spring_length=100*, *spring_strength=0.08*, *damping=0.4*, *overlap=0*)

The forceAtlas2Based solver makes use of some of the equations provided by them and makes use of the barnesHut implementation in vis. The main differences are the central gravity model, which is here distance independent, and the repulsion being linear instead of quadratic. Finally, all node masses have a multiplier based on the amount of connected edges plus one.

Parameters

- **`gravity`** (*`int`*) – The more negative the gravity value is, the stronger the repulsion is.
- **`central_gravity`** (*`float`*) – The gravity attractor to pull the entire network to the center.
- **`spring_length`** (*`int`*) – The rest length of the edges
- **`spring_strength`** (*`float`*) – The strong the edges springs are
- **`damping`** (*`float`*) – A value ranging from 0 to 1 of how much of the velocity from the previous physics simulation iteration carries over to the next iteration.
- **`overlap`** (*`float`*) – When larger than 0, the size of the node is taken into account. The distance will be calculated from the radius of the encompassing circle of the node for both the gravity model. Value 1 is maximum overlap avoidance.

`from_DOT`(*dot*)

This method takes the contents of .DOT file and converts it to a PyVis visualization.

Assuming the contents of test.dot contains: digraph sample3 { A -> {B ; C ; D} C -> {B ; A} }

Usage:

```
>>> nt.Network("500px", "500px")
>>> nt.from_DOT("test.dot")
>>> nt.show("dot.html")
```

Parameters **dot** (*dot file*) – The path of the dotfile being converted.

from_nx (*nx_graph*, *node_size_transf=<function Network.<lambda>>*, *edge_weight_transf=<function Network.<lambda>>*, *default_node_size=10*, *default_edge_weight=1*, *show_edge_weights=True*, *edge_scaling=False*)

This method takes an exisitng Networkx graph and translates it to a PyVis graph format that can be accepted by the VisJs API in the Jinja2 template. This operation is done in place.

Parameters

- **nx_graph** (*networkx.Graph instance*) – The Networkx graph object that is to be translated.
- **node_size_transf** (*func*) – function to transform the node size for plotting
- **edge_weight_transf** (*func*) – function to transform the edge weight for plotting
- **default_node_size** – default node size if not specified
- **default_edge_weight** – default edge weight if not specified

```
>>> nx_graph = nx.cycle_graph(10)
>>> nx_graph.nodes[1]['title'] = 'Number 1'
>>> nx_graph.nodes[1]['group'] = 1
>>> nx_graph.nodes[3]['title'] = 'I belong to a different group!'
>>> nx_graph.nodes[3]['group'] = 10
>>> nx_graph.add_node(20, size=20, title='couple', group=2)
>>> nx_graph.add_node(21, size=15, title='couple', group=2)
>>> nx_graph.add_edge(20, 21, weight=5)
>>> nx_graph.add_node(25, size=25, label='lonely', title='lonely node',␣
↪group=3)
>>> nt = Network("500px", "500px")
# populates the nodes and edges data structures
>>> nt.from_nx(nx_graph)
>>> nt.show("nx.html")
```

generate_html (*name='index.html'*, *local=True*, *notebook=False*)

This method gets the data structures supporting the nodes, edges, and options and updates the template to write the HTML holding the visualization. :type name_html: str

get_adj_list ()

This method returns the user an adjacency list representation of the network.

Returns dictionary mapping of Node ID to list of Node IDs it

is connected to.

get_edges ()

This method returns an iterable list of edge objects

Returns list

get_network_data ()

Extract relevant information about this network in order to inject into a Jinja2 template.

Returns:

nodes (list), edges (list), height (string), width (string), options (object)

Usage:

```
>>> nodes, edges, heading, height, width, options = net.get_network_data()
```

get_node (*n_id*)
Lookup node by ID and return it.

Parameters **n_id** – The ID given to the node.

Returns dict containing node properties

get_nodes ()
This method returns an iterable list of node ids

Returns list

hrepulsion (*node_distance=120*, *central_gravity=0.0*, *spring_length=100*, *spring_strength=0.01*, *damping=0.09*)
This model is based on the repulsion solver but the levels are taken into account and the forces are normalized.

Parameters

- **node_distance** (`int`) – This is the range of influence for the repulsion.
- **central_gravity** – The gravity attractor to pull the entire network to the center.
- **spring_length** – The rest length of the edges
- **spring_strength** – The strong the edges springs are
- **damping** – A value ranging from 0 to 1 of how much of the velocity from the previous physics simulation iteration carries over to the next iteration.

:type central_gravity float :type spring_length: int :type spring_strength: float :type damping: float

inherit_edge_colors (*status*)
Edges take on the color of the node they are coming from.

Parameters **status** (`bool`) – True if edges should adopt color coming from.

neighbors (*node*)
Given a node id, return the set of neighbors of this particular node.

Parameters **node** (`str or int`) – The node to get the neighbors from

Returns set

num_edges ()
Return number of edges

Returns `int`

num_nodes ()
Return number of nodes

Returns `int`

prep_notebook (*custom_template=False*, *custom_template_path=None*)
Loads the template data into the template attribute of the network. This should be done in a jupyter notebook environment before showing the network.

Example:

```
>>> net.prep_notebook()
>>> net.show("nb.html")
```

Parameters **path** (*string*) – the relative path pointing to a template html file

repulsion(*node_distance=100*, *central_gravity=0.2*, *spring_length=200*, *spring_strength=0.05*, *damping=0.09*)
Set the physics attribute of the entire network to repulsion. When called, it sets the solver attribute of physics to repulsion.

Parameters

- **node_distance** (*int*) – This is the range of influence for the repulsion.
- **central_gravity** – The gravity attractor to pull the entire network to the center.
- **spring_length** – The rest length of the edges
- **spring_strength** – The strong the edges springs are
- **damping** – A value ranging from 0 to 1 of how much of the velocity from the previous physics simulation iteration carries over to the next iteration.

:type central_gravity float :type spring_length: int :type spring_strength: float :type damping: float

save_graph(*name*)
Save the graph as html in the current directory with name.

Parameters **name** (*str*) – the name of the html file to save as

set_edge_smooth(*smooth_type*)
Sets the smooth.type attribute of the edges.

Parameters **smooth_type** (*string*) – Possible options: 'dynamic', 'continuous', 'discrete', 'diagonalCross', 'straightCross', 'horizontal', 'vertical', 'curvedCW', 'curvedCCW', 'cubicBezier'. When using dynamic, the edges will have an invisible support node guiding the shape. This node is part of the physics simulation. Default is set to continous.

set_options(*options*)
Overrides the default options object passed to the VisJS framework. Delegates to the `options.Options.set()` routine.

Parameters **options** (*str*) – The string representation of the Javascript-like object to be used to override default options.

set_template(*path_to_template: str*)
Path to full template assumes that it exists inside of a template directory. Use *set_template_dir* to set the relative template path to the template directory along with the directory location itself to change both values otherwise this function will infer the results. :path_to_template path: full os path string value of the template directory

set_template_dir(*template_directory*, *template_file='template.html'*)
Path to template directory along with the location of the template file. :template_directory path: template directory :template_file path: name of the template file that is going to be used to generate the html doc.

show(*name*, *local=True*)
Writes a static HTML file and saves it locally before opening.

Param name: the name of the html file to save as

show_buttons(*filter_=None*)
Displays or hides certain widgets to dynamically modify the network.

Usage: >>> g.show_buttons(filter_=['nodes', 'edges', 'physics'])

Or to show all options: >>> g.show_buttons()

Parameters

- **status** (*bool*) – When set to True, the widgets will be shown. Default is set to False.
- **filter** (*bool or list:*) – Only include widgets specified by *filter_*. Valid options: True (gives all widgets)

  List of *nodes*, *edges*, *layout*, *interaction*, *manipulation*, *physics*, *selection*, *renderer*.

toggle_drag_nodes(*status*)
Toggles the dragging of the nodes in the network.

Parameters **status** (*bool*) – When set to True, the nodes can be dragged around in the network. Default is set to True.

toggle_hide_edges_on_drag(*status*)
Displays or hides edges while dragging the network. This makes panning of the network easy.

Parameters **status** (*bool*) – True if edges should be hidden on drag

toggle_hide_nodes_on_drag(*status*)
Displays or hides nodes while dragging the network. This makes panning of the network easy.

Parameters **status** (*bool*) – When set to True, the nodes will hide on drag. Default is set to False.

toggle_physics(*status*)
Toggles physics simulation

Parameters **status** (*bool*) – When False, nodes are not part of the physics simulation. They will not move except for from manual dragging. Default is set to True.

toggle_stabilization(*status*)
Toggles the stablization of the network.

Parameters **status** (*bool*) – Default is set to True.

write_html(*name*, *local=True*, *notebook=False*)
This method gets the data structures supporting the nodes, edges, and options and updates the template to write the HTML holding the visualization. :type name_html: str

class pyvis.options.**Configure**(*enabled=False*, *filter_=None*)
Handles the HTML part of the canvas and generates an interactive option editor with filtering.

class pyvis.options.**EdgeOptions**
This is where the construction of the edges' options takes place. So far, the edge smoothness can be switched through this object as well as the edge color's inheritance.

class Color
The color object contains the color information of the edge in every situation. When the edge only needs a single color value like 'rgb(120,32,14)', '#ffffff' or 'red' can be supplied instead of an object.

class Smooth
When the edges are made to be smooth, the edges are drawn as a dynamic quadratic bezier curve. The drawing of these curves takes longer than that of the straight curves but it looks better. There is a difference between dynamic smooth curves and static smooth curves. The dynamic smooth curves have an invisible support node that takes part in the physics simulation. If there are a lot of edges, another kind of smooth than dynamic would be better for performance.

inherit_colors(*status*)
Whether or not to inherit colors from the source node. If this is set to *from* then the edge will take the color of the source node. If it is set to *to* then the color will be that of the destination node.

Note: If set to *True* then the *from* behavior is adopted and vice versa.

`toggle_smoothness`(*smooth_type*)
Change smooth option for edges. When using dynamic, the edges will have an invisible support node guiding the shape. This node is part of the physics simulation,

Parameters **`smooth_type`** (`str`) – Possible options are dynamic, continuous, discrete, diagonalCross, straightCross, horizontal, vertical, curvedCW, curvedCCW, cubicBezier

class `pyvis.options.`**`Interaction`**
Used for all user interaction with the network. Handles mouse and touch events as well as the navigation buttons and the popups.

class `pyvis.options.`**`Layout`**(*randomSeed=None*, *improvedLayout=True*)
Acts as the camera that looks on the canvas. Does the animation, zooming and focusing.

`set_edge_minimization`(*status*)
Method for reducing whitespace. Can be used alone or together with block shifting. Enabling block shifting will usually speed up the layout process. Each node will try to move along its free axis to reduce the total length of it's edges. This is mainly for the initial layout. If you enable physics, they layout will be determined by the physics. This will greatly speed up the stabilization time

`set_separation`(*distance*)
The distance between the different levels.

`set_tree_spacing`(*distance*)
Distance between different trees (independent networks). This is only for the initial layout. If you enable physics, the repulsion model will denote the distance between the trees.

class `pyvis.options.`**`Options`**(*layout=None*)
Represents the global options of the network. This object consists of indiviual sub-objects that map to VisJS's modules of:

- configure
- layout
- interaction
- physics
- edges

The JSON representation of this object is directly passed in to the VisJS framework. In the future this can be expanded to completely mimic the structure VisJS can expect.

`set`(*new_options*)
This method should accept a JSON string and replace its internal options structure with the given argument after parsing it. In practice, this method should be called after using the browser to experiment with different physics and layout options, using the generated JSON options structure that is spit out from the front end to serve as input to this method as a string.

Parameters **`new_options`** (`str`) – The JSON like string of the options that will override.

1.6 Indices and tables

- genindex
- modindex

- search
- Glossary

Python Module Index

Index