Netgraph: Publication-quality Network Visualisations in Python

Paul J. N. Brodersen [1]

1 Department of Pharmacology, University of Oxford, United Kingdom

DOI: 10.21105/joss.05372

Software

- Review
- Repository
- Archive

Editor: Rachel Kurchin

Reviewers:

- @jonjoncardoso
- @ortega2247
- @idoby

Submitted: 16 March 2023
Published: 19 July 2023

Statement of need

The empirical study and scholarly analysis of networks have increased manifold in recent decades, fuelled by the new prominence of network structures in our lives (the web, social networks, artificial neural networks, ecological networks, etc.) and the data available on them. While there are several comprehensive Python libraries for network analysis such as NetworkX (Hagberg et al., 2008), igraph (Csardi & Nepusz, 2006), and graph-tool (Peixoto, 2014), their inbuilt visualisation capabilities lag behind specialised software solutions such as Graphviz (Ellson et al., 2002), Cytoscape (Shannon et al., 2003), and Gephi (Bastian et al., 2009). However, although Python bindings for these applications exist in the form of PyGraphviz, py4cytoscape, and GephiStreamer, respectively, their outputs are not manipulable Python objects, which restricts customisation, limits their extensibility, and prevents a seamless integration within a wider Python application.

Summary

Netgraph is a Python library that aims to complement the existing network analysis libraries with publication-quality visualisations within the Python ecosystem. To facilitate a seamless integration, Netgraph supports a variety of input formats, including NetworkX, igraph, and graph-tool Graph objects. At the time of writing, Netgraph provides the following node layout algorithms:

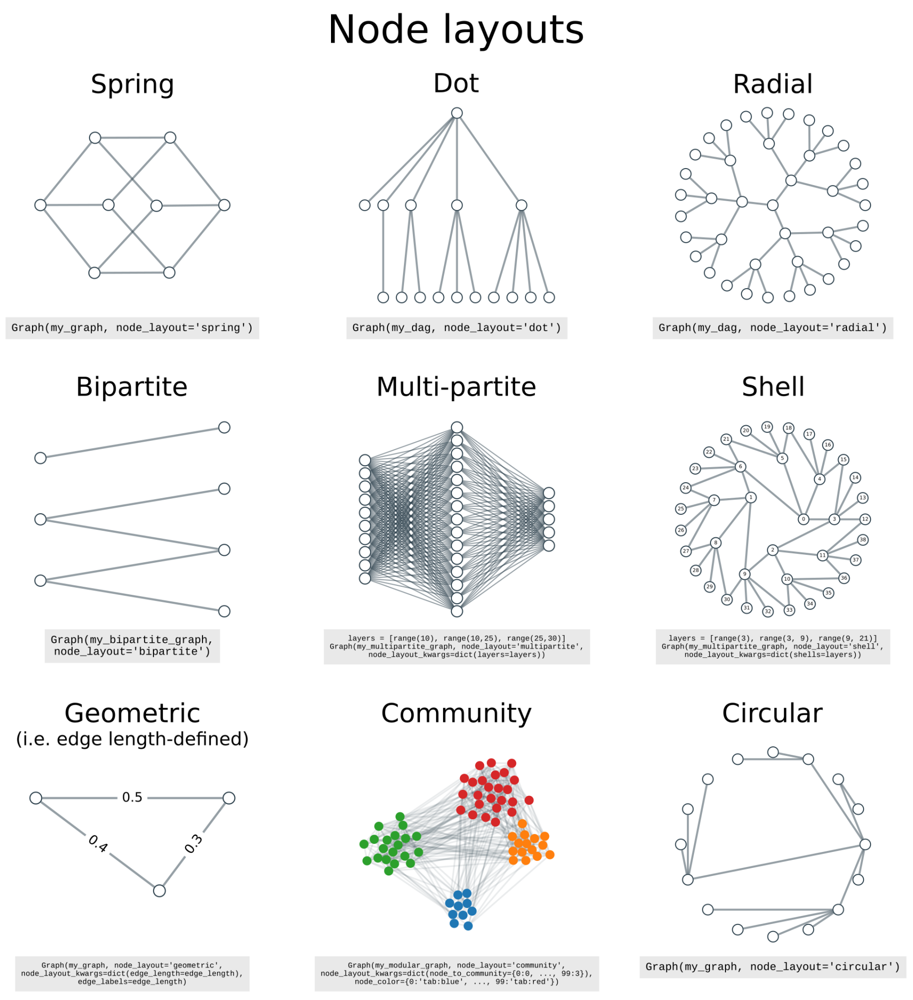
- the Fruchterman-Reingold algorithm, a.k.a. the “spring” layout,
- the Sugiyama algorithm, a.k.a. the “dot” layout for directed, acyclic graphs,
- a radial tree layout for directed, acyclic graphs,
- a circular node layout (with optional edge crossing reduction),
- a bipartite node layout for bipartite graphs (with optional edge crossing reduction),
- a layered node layout for multipartite graphs (with optional edge crossing reduction),
- a shell layout for multipartite graphs (with optional edge crossing reduction),
- a community node layout for modular graphs, and
- a “geometric” node layout for graphs with defined edge lengths but unknown node positions.

Additionally, links or edges between the nodes can be straight, curved (avoiding collisions with other nodes and edges), or bundled. However, new layout routines are added regularly to Netgraph; for an up-to-date list, consult the online documentation here.

Uniquely among Python alternatives, Netgraph handles networks with multiple components gracefully (which otherwise break most node layout routines), and it post-processes the output of the node layout and edge routing algorithms with several heuristics to increase the interpretability of the visualisation (reduction of overlaps between nodes, edges, and labels; edge crossing minimisation and edge unbundling where applicable). The highly customisable plots are created using Matplotlib (Hunter, 2007), a popular Python plotting library, and the resulting Matplotlib objects are exposed in an easily queryable format such that they can be further manipulated and/or animated using standard Matplotlib syntax. The visualisations can also be altered interactively: nodes and edges can be added on-the-fly through hotkeys, positioned using the mouse, and (re-)labelled through standard text-entry. For a comprehensive tutorial on Netgraph's interactive features, consult the online documentation here.

Netgraph is licensed under the General Public License version 3 (GPLv3). The repository is hosted on GitHub, and distributed via PyPI and conda-forge. It includes an extensive automated test suite that can be executed using pytest. The comprehensive documentation – including a complete API reference as well as numerous examples and tutorials – is hosted on ReadTheDocs.

Figures



```
Graph(my_graph, node_layout='spring')
```

```
Graph(my_dag, node_layout='dot')
```

```
Graph(my_dag, node_layout='radial')
```

```
Graph(my_bipartite_graph,
node_layout='bipartite')
```

```
layers = [range(10), range(10,25), range(25,30)]
Graph(my_multipartite_graph, node_layout='multipartite',
node_layout_kwargs=dict(layers=layers))
```

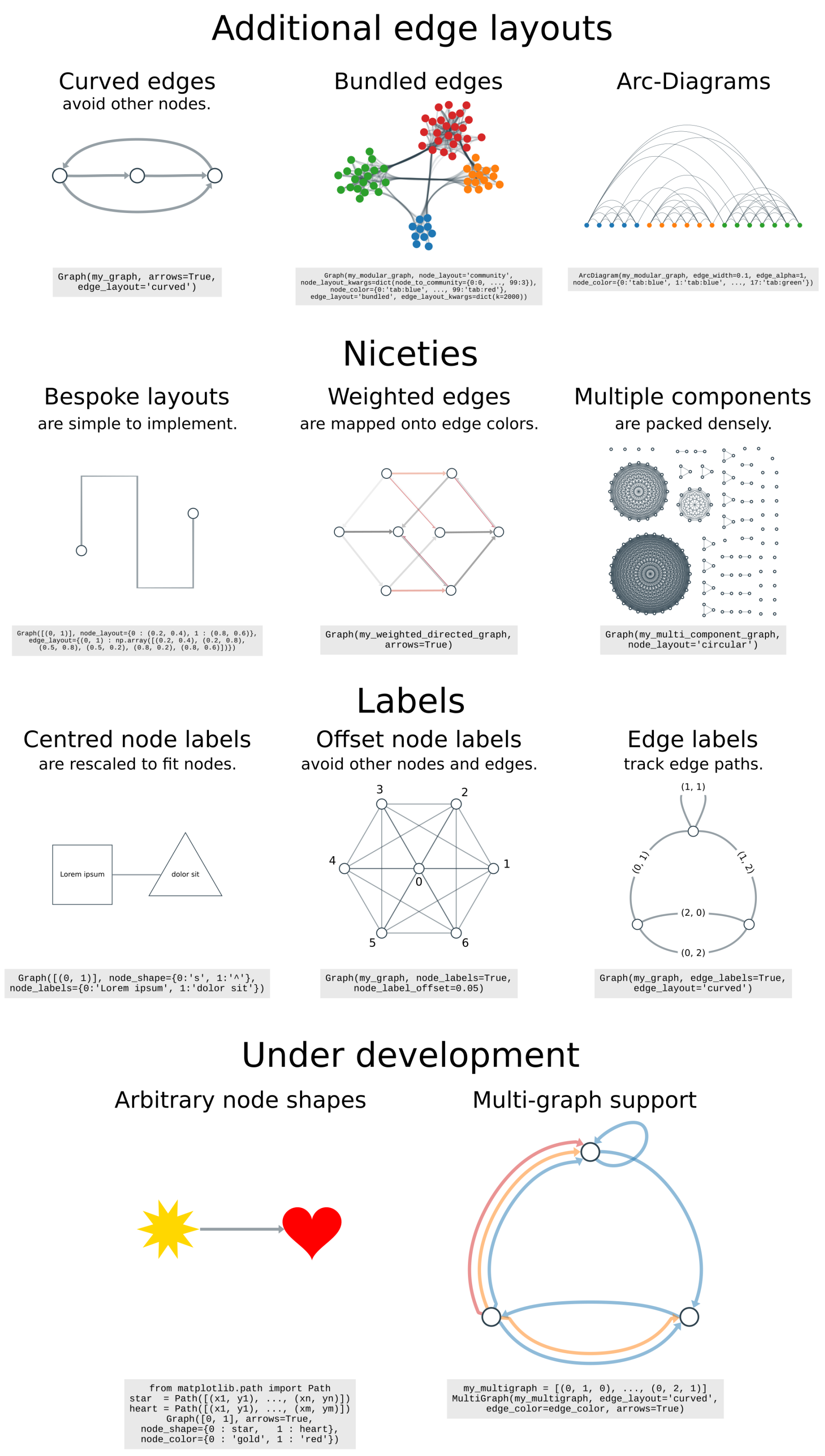
```
layers = [range(3), range(3, 9), range(9, 21)]
Graph(my_multipartite_graph, node_layout='shell',
node_layout_kwargs=dict(shells=layers))
```

```
Graph(my_graph, node_layout='geometric',
node_layout_kwargs=dict(edge_length=edge_length),
edge_labels=edge_length)
```

```
Graph(my_modular_graph, node_layout='community',
node_layout_kwargs=dict(node_to_community={0:0, ..., 99:3}),
node_color={0:'tab:blue', ..., 99:'tab:red'})
```

```
Graph(my_graph, node_layout='circular')
```

Figure 1: Netgraph's node layouts



```
Graph(my_graph, arrows=True,
edge_layout='curved')
```

```
Graph(my_modular_graph, node_layout='community',
node_layout_kwargs=dict(node_to_community={0:0, ..., 99:3}),
node_color={0:'tab:blue', ..., 99:'tab:red'},
edge_layout='bundled', edge_layout_kwargs=dict(k=2000))
```

```
ArcDiagram(my_modular_graph, edge_width=0.1, edge_alpha=1,
node_color={0:'tab:blue', 1:'tab:blue', ..., 17:'tab:green'})
```

```
Graph([(0, 1)], node_layout={0 : (0.2, 0.4), 1 : (0.8, 0.6)},
edge_layout={(0, 1) : np.array([(0.2, 0.4), (0.2, 0.8),
(0.5, 0.8), (0.5, 0.2), (0.8, 0.2), (0.8, 0.6)])})
```

```
Graph(my_weighted_directed_graph,
arrows=True)
```

```
Graph(my_multi_component_graph,
node_layout='circular')
```

```
Graph([(0, 1)], node_shape={0:'s', 1:'^'},
node_labels={0:'Lorem ipsum', 1:'dolor sit'})
```

```
Graph(my_graph, node_labels=True,
node_label_offset=0.05)
```

```
Graph(my_graph, edge_labels=True,
edge_layout='curved')
```

```
from matplotlib.path import Path
star  = Path([(x1, y1), ..., (xn, yn)])
heart = Path([(x1, y1), ..., (xm, ym)])
Graph([0, 1], arrows=True,
node_shape={0 : star,   1 : heart},
node_color={0 : 'gold', 1 : 'red'})
```

```
my_multigraph = [(0, 1, 0), ..., (0, 2, 1)]
MultiGraph(my_multigraph, edge_layout='curved',
edge_color=edge_color, arrows=True)
```

Figure 2: Netgraph's edge layouts and other key features

A Basic Example

The following script shows a minimum working example. The graph structure is defined by an edge list, and the visualisation is created using (mostly) default parameters.

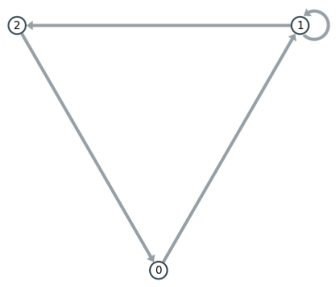


Figure 3: Basic example output

```python
import matplotlib.pyplot as plt

from netgraph import Graph

triangle = [
    (0, 1),
    (1, 2),
    (2, 0),
    (1, 1), # self-loop
]
Graph(
    triangle,
    node_labels=True,
    arrows=True,
)
plt.show()
```

Interoperability & Customisability

Netgraph can be easily integrated into existing network analysis workflows as it accepts a variety of graph structures. The example below uses a NetworkX `Graph` object, but igraph and graph-tool objects are also valid inputs, as are plain edge lists and full-rank adjacency matrices. The output visualisations are created using Matplotlib and can hence form subplots in larger Matplotlib figures.

Each visualisation can be customised in various ways. Most parameters can be set using a scalar or string. In this case, the value is applied to all nodes or edges (depending on the parameter). To style each node or each edge differently, supply a dictionary instead. Furthermore, Netgraph's `NodeArtist` and `EdgeArtist` classes, i.e. the Python objects that instruct a renderer to paint nodes and edges onto the canvas, are derived from Matplotlib's `PathPatch` class; similarly, node and edge labels are Matplotlib `Text` instances. Hence all node artists, edge artists, and labels can be manipulated using standard matplotlib syntax after the initial draw.

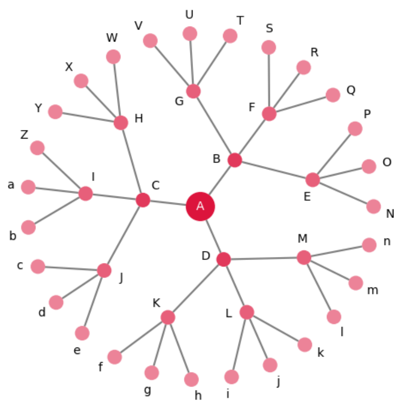


Figure 4: Advanced example output

```python
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx

from netgraph import Graph

# initialize the figure
fig, ax = plt.subplots(figsize=(6,6))

# initialize the graph structure
balanced_tree = nx.balanced_tree(3, 3)

# initialize the visualisation
g = Graph(
    balanced_tree,
    node_layout='radial',
    edge_layout='straight',
    node_color='crimson',
    node_size={node : 4 if node == 0 else 2 for node in balanced_tree},
    node_edge_width=0,
    edge_color='black',
    edge_width=0.5,
    node_labels=dict(
        zip(balanced_tree, 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz')
    ),
    node_label_offset=0.05,
    node_label_fontdict=dict(fontsize=10),
    ax=ax,
)

# center the label of the root node on the corresponding node artist and make it white
root = 0 # NetworkX graph generator convention
center = g.node_positions[root]
g.node_label_artists[root].set_position(center)
g.node_label_artists[root].set_color('white')
```

```
# decrease the node artist alpha parameter from the root to the leaves or the graph
for node in balanced_tree:
    distance = np.linalg.norm(center - g.node_positions[node])
    g.node_artists[node].set_alpha(1 - distance)

# redraw figure to display changes
fig.canvas.draw()
```

Key Design Decisions

The creation of Netgraph was motivated by the desire to make high-quality, easily customisable, and reproducible network visualisations, whilst maintaining an extensible code base. To that end, a key design decision was to have a single reference frame for all node artist and edge artist attributes that determine their extent (e.g., in the case of a circular node artist, its position and its radius).

Good data visualisations are both accurate and legible. The legibility of a visualisation is influenced predominantly by the size of the plot elements, and occlusions between them. However, there is often tension between these two requirements, as larger plot elements are more visible but also more likely to cause overlaps with other plot elements. Most data visualisation tools focus on accuracy and visibility. To that end, they operate in two reference frames: a data-derived reference frame and a display-derived reference frame. For example, in a standard line-plot, the data-derived reference frame determines the x and y values of the line. The thickness of the line, however, scales with the size of the display, and its width (measured in pixels) remains constant across different figure sizes and aspect ratios. Having two reference frames ensures that the line (1) is an accurate representation of the data, and (2) is visible and discernible independent of figure dimensions. The trade-off of this setup is that (1) the precise extents of plot elements can only be computed after the figure is initialised, and (2) occlusions are not managed and hence common, for example, if multiple lines are plotted in the same figure. Nevertheless, most network visualisation tools follow this standard. For example, NetworkX specifies node positions and edge paths in data coordinates, but uses display units for node sizes and edge widths.

However, network visualisations differ from other data visualisations in two aspects: (1) the precise positions of nodes and the precise paths of edges often carry no inherent meaning, and (2) most figures contain a multitude of node and edge artists instead of just a few lines typically present in a line-plot. As a consequence, a common goal of most algorithms for node layout, edge routing, and label placement is to minimize occlusions between different plot elements, as they reduce the ease with which a visualisation is interpreted. To that end, precise knowledge of the extent of all plot elements is paramount, motivating the use of a single reference frame. In Netgraph, this reference frame derives from the data. Specifically, node positions and edge paths are specified in data units, and node sizes and edge widths are specified in 1/100s of data units (as this makes the node sizes and edge widths more comparable to typical values in NetworkX, igraph, and graph-tool). This decouples layout computations from rendering the figure, simplifies computing the extent of the different plot elements, facilitates the reduction of overlaps, and makes it possible to create pixel-perfect reproductions independent of display parameters.

Acknowledgements

We thank GitHub users adleris, Allan L. R. Hansen, chenghuzi, Hamed Mohammadpour, and Pablo for contributing various bug fixes, as well as ortega2247 and idoby for their thoughtful reviews of earlier versions of this manuscript and their inspired suggestions, such as adding a section on "Key Design Decisions".

References

Bastian, M., Heymann, S., & Jacomy, M. (2009). *Gephi: An open source software for exploring and manipulating networks*. https://doi.org/10.1609/icwsm.v3i1.13937

Csardi, G., & Nepusz, T. (2006). The igraph software package for complex network research. *InterJournal*, *Complex Systems*, 1695. http://igraph.org

Ellson, J., Gansner, E., Koutsofios, L., North, S. C., & Woodhull, G. (2002). Graphviz— open source graph drawing tools. In P. Mutzel, M. Jünger, & S. Leipert (Eds.), *Graph drawing* (pp. 483–484). Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-45848-4_57

Hagberg, A. A., Schult, D. A., & Swart, P. J. (2008). Exploring Network Structure, Dynamics, and Function using NetworkX. In G. Varoquaux, T. Vaught, & J. Millman (Eds.), *Proceedings of the 7th Python in science conference* (pp. 11–15).

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, *9*(3), 90–95. https://doi.org/10.1109/MCSE.2007.55

Peixoto, T. P. (2014). The graph-tool Python library. *Figshare*. https://doi.org/10.6084/m9.figshare.1164194

Shannon, P., Markiel, A., Ozier, O., Baliga, N. S., Wang, J. T., Ramage, D., Amin, N., Schwikowski, B., & Ideker, T. (2003). Cytoscape: A software environment for integrated models of biomolecular interaction networks. *Genome Research*, *13*(11), 2498–2504. https://doi.org/10.1101/gr.1239303