Drawing

NetworkX provides basic functionality for visualizing graphs, but its main goal is to enable graph analysis rather than perform grap visualization. In the future, graph visualization functionality may be removed from NetworkX or only available as an add-on package.

Proper graph visualization is hard, and we highly recommend that people visualize their graphs with tools dedicated to that task. Notable examples of dedicated and fully-featured graph visualization tools are Cytoscape, Gephi, Graphviz and, for LaTeX typesetting, PGF/TikZ. To use these and other such tools, you should export your NetworkX graph into a format that can be read b those tools. For example, Cytoscape can read the GraphML format, and so, `networkx.write_graphml(G, path)` might be an appropriate choice.

More information on the features provided here are available at

- matplotlib: http://matplotlib.org/
- pygraphviz: http://pygraphviz.github.io/

Matplotlib

Draw networks with matplotlib.

Examples

```
>>> G = nx.complete_graph(5)
>>> nx.draw(G)
```

See Also

Skip to main content

- matplotlib
- `matplotlib.pyplot.scatter()`
- `matplotlib.patches.FancyArrowPatch`

`draw`(G[, pos, ax])	Draw the graph G with Matplotlib.
`draw_networkx`(G[, pos, arrows, with_labels])	Draw the graph G using Matplotlib.
`draw_networkx_nodes`(G, pos[, nodelist, ...])	Draw the nodes of the graph G.
`draw_networkx_edges`(G, pos[, edgelist, ...])	Draw the edges of the graph G.
`draw_networkx_labels`(G, pos[, labels, ...])	Draw node labels on the graph G.
`draw_networkx_edge_labels`(G, pos[, ...])	Draw edge labels.
`draw_circular`(G, **kwargs)	Draw the graph `G` with a circular layout.
`draw_kamada_kawai`(G, **kwargs)	Draw the graph `G` with a Kamada-Kawai force-directed layout.
`draw_planar`(G, **kwargs)	Draw a planar networkx graph `G` with planar layout.
`draw_random`(G, **kwargs)	Draw the graph `G` with a random layout.
`draw_spectral`(G, **kwargs)	Draw the graph `G` with a spectral 2D layout.
`draw_spring`(G, **kwargs)	Draw the graph `G` with a spring layout.
`draw_shell`(G[, nlist])	Draw networkx graph `G` with shell layout.

Graphviz AGraph (dot)

Interface to pygraphviz AGraph class.

Examples

```
>>> G = nx.complete_graph(5)
>>> A = nx.nx_agraph.to_agraph(G)
>>> H = nx.nx_agraph.from_agraph(A)
```

See Also

Skip to main content

- Graphviz: https://www.graphviz.org
- DOT Language: http://www.graphviz.org/doc/info/lang.html

`from_agraph`(A[, create_using])	Returns a NetworkX Graph or DiGraph from a PyGraphviz graph.
`to_agraph`(N)	Returns a pygraphviz graph from a NetworkX graph N.
`write_dot`(G, path)	Write NetworkX graph G to Graphviz dot format on path.
`read_dot`(path)	Returns a NetworkX graph from a dot file on path.
`graphviz_layout`(G[, prog, root, args])	Create node positions for G using Graphviz.
`pygraphviz_layout`(G[, prog, root, args])	Create node positions for G using Graphviz.

Graphviz with pydot

Import and export NetworkX graphs in Graphviz dot format using pydot.

Either this module or nx_agraph can be used to interface with graphviz.

Examples

```
>>> G = nx.complete_graph(5)
>>> PG = nx.nx_pydot.to_pydot(G)
>>> H = nx.nx_pydot.from_pydot(PG)
```

See Also

- pydot: erocarrera/pydot
- Graphviz: https://www.graphviz.org
- DOT Language: http://www.graphviz.org/doc/info/lang.html

`from_pydot`(P)	Returns a NetworkX graph from a Pydot graph.
`to_pydot`(N)	Returns a pydot graph from a NetworkX graph N.
`write_dot`(G, path)	Write NetworkX graph G to Graphviz dot format on path.

Skip to main content

`graphviz_layout`(G[, prog, root])	Create node positions using Pydot and Graphviz.
`pydot_layout`(G[, prog, root])	Create node positions using `pydot` and Graphviz.

Graph Layout

Node positioning algorithms for graph drawing.

For `random_layout()` the possible resulting shape is a square of side [0, scale] (default: [0, 1]) Changing `center` shifts the layout b
that amount.

For the other layout routines, the extent is [center - scale, center + scale] (default: [-1, 1]).

Warning: Most layout routines have only been tested in 2-dimensions.

`bipartite_layout`(G, nodes[, align, scale, ...])	Position nodes in two straight lines.
`circular_layout`(G[, scale, center, dim])	Position nodes on a circle.
`kamada_kawai_layout`(G[, dist, pos, weight, ...])	Position nodes using Kamada-Kawai path-length cost-function.
`planar_layout`(G[, scale, center, dim])	Position nodes without edge intersections.
`random_layout`(G[, center, dim, seed])	Position nodes uniformly at random in the unit square.
`rescale_layout`(pos[, scale])	Returns scaled position array to (-scale, scale) in all axes.
`rescale_layout_dict`(pos[, scale])	Return a dictionary of scaled positions keyed by node
`shell_layout`(G[, nlist, rotate, scale, ...])	Position nodes in concentric circles.
`spring_layout`(G[, k, pos, fixed, ...])	Position nodes using Fruchterman-Reingold force-directed algorithm.
`spectral_layout`(G[, weight, scale, center, dim])	Position nodes using the eigenvectors of the graph Laplacian.
`spiral_layout`(G[, scale, center, dim, ...])	Position nodes in a spiral layout.
`multipartite_layout`(G[, subset_key, align, ...])	Position nodes in layers of straight lines.

LaTeX Code

Export NetworkX graphs in LaTeX format using the TikZ library within TeX/LaTeX. Usually, you will want the drawing to appear in a figure environment so you use `to_latex(G, caption="A caption")`. If you want the raw drawing commands without a figure environment use `to_latex_raw()`. And if you want to write to a file instead of just returning the latex code as a string, use `write_latex(G, "filename.tex", caption="A caption")`.

Skip to main content

To construct a figure with subfigures for each graph to be shown, provide `to_latex` or `write_latex` a list of graphs, a list of subcaptions, and a number of rows of subfigures inside the figure.

To be able to refer to the figures or subfigures in latex using `\\ref`, the keyword `latex_label` is available for figures and `sub_labels` for a list of labels, one for each subfigure.

We intend to eventually provide an interface to the TikZ Graph features which include e.g. layout algorithms.

Let us know via github what you'd like to see available, or better yet give us some code to do it, or even better make a github pull request to add the feature.

The TikZ approach

Drawing options can be stored on the graph as node/edge attributes, or can be provided as dicts keyed by node/edge to a string of the options for that node/edge. Similarly a label can be shown for each node/edge by specifying the labels as graph node/edge attributes or by providing a dict keyed by node/edge to the text to be written for that node/edge.

Options for the tikzpicture environment (e.g. "[scale=2]") can be provided via a keyword argument. Similarly default node and edg options can be provided through keywords arguments. The default node options are applied to the single TikZ "path" that draws a nodes (and no edges). The default edge options are applied to a TikZ "scope" which contains a path for each edge.

Examples

```
>>> G = nx.path_graph(3)
>>> nx.write_latex(G, "just_my_figure.tex", as_document=True)
>>> nx.write_latex(G, "my_figure.tex", caption="A path graph", latex_label="fig1")
>>> latex_code = nx.to_latex(G)  # a string rather than a file
```

You can change many features of the nodes and edges.

```
>>> G = nx.path_graph(4, create_using=nx.DiGraph)
>>> pos = {n: (n, n) for n in G}  # nodes set on a line
```

```
>>> G.nodes[0]["style"] = "blue"
>>> G.nodes[2]["style"] = "line width=3,draw"
>>> G.nodes[3]["label"] = "Stop"
```

Skip to main content

```
>>> G.edges[(1, 2)]["style"] = "line width=3"
>>> G.edges[(1, 2)]["label"] = "2nd Step"
>>> G.edges[(2, 3)]["style"] = "green"
>>> G.edges[(2, 3)]["label"] = "3rd Step"
>>> G.edges[(2, 3)]["label_opts"] = "near end"
```


Then compile the LaTeX using something like `pdflatex latex_graph.tex` and view the pdf file created: `latex_graph.pdf`.

If you want **subfigures** each containing one graph, you can input a list of graphs.

```
>>> H1 = nx.path_graph(4)
>>> H2 = nx.complete_graph(4)
>>> H3 = nx.path_graph(8)
>>> H4 = nx.complete_graph(8)
>>> graphs = [H1, H2, H3, H4]
>>> caps = ["Path 4", "Complete graph 4", "Path 8", "Complete graph 8"]
>>> lbls = ["fig2a", "fig2b", "fig2c", "fig2d"]
>>> nx.write_latex(graphs, "subfigs.tex", n_rows=2, sub_captions=caps, sub_labels=lbls)
>>> latex_code = nx.to_latex(graphs, n_rows=2, sub_captions=caps, sub_labels=lbls)
```

```
>>> node_color = {0: "red", 1: "orange", 2: "blue", 3: "gray!90"}
>>> edge_width = {e: "line width=1.5" for e in H3.edges}
>>> pos = nx.circular_layout(H3)
>>> latex_code = nx.to_latex(H3, pos, node_options=node_color, edge_options=edge_width)
>>> print(latex_code)
\documentclass{report}
\usepackage{tikz}
\usepackage{subcaption}

\begin{document}
\begin{figure}
  \begin{tikzpicture}
      \draw
        (1.0, 0.0) node[red] (0){0}
        (0.707, 0.707) node[orange] (1){1}
        (-0.0, 1.0) node[blue] (2){2}
        (-0.707, 0.707) node[gray!90] (3){3}
        (-1.0, -0.0) node (4){4}
        (-0.707, -0.707) node (5){5}
        (0.0, -1.0) node (6){6}
        (0.707, -0.707) node (7){7};
      \begin{scope}[-]
        \draw[line width=1.5] (0) to (1);
        \draw[line width=1.5] (1) to (2);
```

```
        \draw[line width=1.5] (4) to (5);
        \draw[line width=1.5] (5) to (6);
        \draw[line width=1.5] (6) to (7);
      \end{scope}
    \end{tikzpicture}
\end{figure}
\end{document}
```

graphviz_layout

pydot_layout

Notes

If you want to change the preamble/postamble of the figure/document/subfigure environment, use the keyword arguments: `figure_wrapper`, `document_wrapper`, `subfigure_wrapper`. The default values are stored in private variables e.g. `nx.nx_layout._DOCUMENT_WRAPPER`

References

TikZ: https://tikz.dev/

TikZ options details: https://tikz.dev/tikz-actions

`to_latex_raw`(G[, pos, tikz_options, ...])	Return a string of the LaTeX/TikZ code to draw `G`
`to_latex`(Gbunch[, pos, tikz_options, ...])	Return latex code to draw the graph(s) in `Gbunch`