algorithmx Documentation

Release 1.1.2

alexsocha

Dec 08, 2019

INSTALLATION AND USAGE

Version: 1.1.2

A library for network visualization and algorithm simulation.

CHAPTER

ONE

CONTENTS

1.1 Installation

Python 3.6 or higher is required.

AlgorithmX can be installed using pip:

```
pip install algorithmx
```

1.1.1 Jupyter Widget

In classic Jupyter notebooks, the widget will typically be enabled by default. However, if you installed using pip with notebook version <5.3, you will have to manually enable it by running:

```
jupyter nbextension enable --sys-prefix --py algorithmx
```

with the appropriate flag. To enable in JupyterLab, run:

```
jupyter labextension install @jupyter-widgets/jupyterlab-manager
jupyter labextension install algorithmx-jupyter
```

1.2 HTTP Server

To use the library normally (i.e. not through Jupyter), you will need to set up a local server for displaying the interactive network. The library comes with all the tools needed to do this:

```
import algorithmx

# Create a new HTTP server
server = algorithmx.http_server(port=5050)
# Create a CanvasSelection interface
canvas = server.canvas()

def start():
    # Use the library normally, for example:
    canvas.nodes([1, 2]).add()
    canvas.edge((1, 2)).add()

    canvas.pause(1)
```

(continues on next page)

(continued from previous page)

```
    canvas.node(1).highlight().size('1.5x').pause(0.5)
    canvas.edge((1, 2)).animate('traverse').color('blue')

# Call the function above when the client broadcasts a 'start' message
# (which will happen when the user clicks the start or restart button)
canvas.listen('start', start)

# Start the server, blocking all further execution on the current thread
# You can press 'CTRL-C' to exit the script
server.start()
```

After running this code, open a browser and go to the address `http://localhost:5050/` to see the network. The library provides a simple HTML interface with buttons for starting, stopping and restarting the simulation. If you wish to customize this further, you can tell the server to open a different HTML file, and configure the port:

```
server = algorithmx.http_server(file='my/custom/interface.html', port=8090)
```

Use the provided HTML file as a guide for creating your own.

`algorithmx.`**`http_server`**(*file: str = None*, *port: int = 5050*) → algorithmx.server.Server.Server

Creates a new HTTP server for displaying the network, using WebSockets to transmit data. The server will only start once its *`start()`* method is called. After the server has started, the network can be viewed by opening a browser and navigating to the address `http://localhost:5050/` (change the port as necessary).

File (Optional) The path to the HTML file which the server should display, relative to the current runtime directory. If unspecified, the default HTML file will be used. When creating a custom HTML interface, use the default file as a guide.

Port (Optional) The port on which the server should start, defaulting to to 5050. Note that the next port (by default 5051) will also be used to transmit data through WebSockets.

class `algorithmx.server.`**`Server`**(*file: str*, *port: int*)

A local HTTP server using WebSockets to transmit data.

`start`()

Starts the server on the current threat, blocking all further execution until the server shuts down.

`shutdown`()

Shuts down the server. This must be called on a different thread to the one used to start the server.

`canvas`(*name: str = 'output'*) → algorithmx.graphics.CanvasSelection.CanvasSelection

Creates a new *`CanvasSelection`* which will dispatch and receive events through a WebSocket connected to the server.

Parameters **name** – (Optional) The name of the canvas. By default, each server will only render one canvas, and so this argument has no affect. However, if you wish to design a custom interface with more than one canvas per page, you can use this to differentiate between them.

1.3 Jupyter Widget

After *installing and enabling* the Jupyter widget, you can use the library within a notebook in the following way:

```
import algorithmx

# Create a Jupyter canvas interface
```

(continues on next page)

(continued from previous page)

```
canvas = algorithmx.jupyter_canvas()

# Set the size of the canvas
canvas.size((300, 200))

# Use the library normally, for example:
canvas.nodes([1, 2]).add()
canvas.edge((1, 2)).add()

# Display the canvas
display(canvas)
```

Note that you need to hold down the `ctrl/cmd` key to zoom in. If you are creating an algorithm simulation, you can also enable start/stop/restart buttons:

```
canvas = algorithmx.jupyter_canvas(buttons=True)
```

`algorithmx.`**`jupyter_canvas`**(*buttons: bool = False*) → algorithmx.jupyter.JupyterCanvas.JupyterCanvas

Creates a new *`CanvasSelection`* which will dispatch and receive events through a Jupyter widget, and which can be displayed using the IPython `display` function.

By default, the canvas size is (400, 250), and requires the `ctrl/cmd` to be held down while zooming.

1.4 Overview

The AlgorithmX graphics library provides a selection-based interface for creating interactive network visualizations. At the root of each visualization is a *`CanvasSelection`*, which can be created either through a HTTP Server (*`canvas()`*), or a Jupyter widget (*`jupyter_canvas()`*).

The purpose of the library is to provide a way to manipulate the graphics representing a network, by sending events directly to the client. As such, it does not keep track of any state (except for callbacks). In order to store and analyze the network, you can combine this with another library, such as NetworkX.

1.4.1 Using Selections

Every selection corresponds to one or more graphical objects in the network. If a selection is created with objects that do not exist in the network yet, these can be added by calling *`add()`*. Selections will provide a range of methods for setting custom attributes, configuring animations, and interacting with event queues.

Below is an example showing how selections can be created, added, modified and removed:

```
# Add a big red node
canvas.node('A').add().color('red').size(30)

# Add a label to the node
canvas.node('A').label(1).add().text('My Label')

# Pause for half a second
canvas.pause(0.5)

# Modify the color of the node
canvas.node('A').color('blue')
```

(continues on next page)

(continued from previous page)

```
# Temporarily make the node 1.25 times as big
canvas.node('A').highlight().size('1.25x')

# Add a few more modes
canvas.nodes([1, 2, 3]).add()

# Add an edge
canvas.edge((2, 3)).add()

# Remove the first node
canvas.node('A').remove()
```

Attributes can also be configured using the *set()* method:

```
# Configure the attributes of a label
canvas.node(1).label(2).set(
    text='Hello',
    color='red'
    size=45,
    font='Courier'
)
```

1.4.2 Functional Arguments

All selection methods can take functions as arguments, allowing attributes to be configured differently depending on each element's data and index within the selection.

```
# Conditionally set color using id
canvas.nodes(['A', 'B']).color(lambda n: 'red' if n == 'A' else 'blue')

# Conditionally set color using index
colors = ['red', 'blue']
canvas.nodes(['A', 'B']).color(lambda n, i: colors[i])

# Conditionally set color using data binding
canvas.nodes(['A', 'B']).data(colors).color(lambda c: c)
```

graphics.types.**ElementFn = typing.Union[typing.Callable[[typing.Any], ~T], typing.Callable**

A function taking a selected element's data as input. This is typically provided as an argument in a selection method, allowing attributes to be configured differently for each element.

Parameters

- **ElementFn.data** – The data associated with the element. If the *data()* method was used previously in the method chain, it will determine the type of data used. If the selection has no associated data, it will fall back on its parent's data (as is the case for *LabelSelection*). Otherwise, the information used to construct the selection will serve as its data (such as node ID values and edge tuples).
- **ElementFn.index** – (Optional) The index of the element in the selection, beginning at 0, determined by its position in the list initially used to construct the selection.

graphics.types.**ElementArg = typing.Union[typing.Callable[[typing.Any], ~T], typing.Callable**

Allows an argument to be provided either directly, or as a function of each element's data (see *ElementFn* and *data()*).

1.4.3 Expressions

Most numerical attributes can also be specified as linear expressions, often allowing for easier and more powerful configuration. Expressions use variables corresponding to other attributes; for example, a label could be positioned relative to it's parent node without needing to know the node's size, and would be re-positioned accordingly when the node's size changes.

```
# Position a label in the top-left corner of a node
canvas.node('A').label().align('top-left').pos(('-x+5', 'y-5'))

# Pin a node to the canvas using a relative position
canvas.node('A').fixed(True).pos(('-0.5cx', '-0.5cy'))

# Change the size of a node relative to it's current size
canvas.node('C').shape('rect').size(('1.25x', '1.25y'))
```

graphics.types.**NumExpr = typing.Union[int, float, str, typing.Dict]**

A number, or an expression evaluating to a number. Expressions must be in the form `mx+c`, described by either an `{ m, x, c }` dictionary, or an expression string such as "-2x+8". Both `m` and `c` are constants, while `x` is a variable corresponding to some other attribute. Below is a list of valid variables and the context in which they can be used:

- "cx": Half the width of the canvas.
- "cy": Half the height of the canvas.
- **nodes**
  - "x": Half the width of the node.
  - "y": Half the height of the node.
  - **labels**
    - "r": Distance from the center of the node to its boundary given the angle attribute of the label.

1.5 Selections

1.5.1 Selection

class graphics.**Selection**(*context*)

add() → self

Adds all elements in the current selection to the canvas. This should be called immediately after a selection of new elements is created. If the selection contains multiple elements, they will not necessarily be added in order.

Returns A new instance of the current selection with animations disabled, allowing initial attributes to be configured.

remove() → self

Removes all elements in the current selection from the canvas.

set(*attrs*, ***kwargs*) → self

Sets one or more custom attributes on all elements in the current selection. The attributes are provided using a dictionary, where each (key, value) pair corresponds to the method and argument setting the same attribute. Keyword arguments can also be used in the same way. For example:

```
node.color('red').size((20, 30)).svgattr('stroke', 'blue')
# is equivalent to
node.set(color = 'red',
         size = (20, 30),
         svgattr = {
            'stroke': 'blue'
         })
```

Parameters

- **attrs** (*ElementArg*[Dict[str, Any]]) – (Optional) A dictionary of custom attributes.
- **kwargs** (*Dict[str, Any]*) – Custom attributes as keywork arguments.

visible (*visible*) → self

Sets whether or not the elements in the current selection should be visible. This can be animated in the same way as additions and removals. However, in contrast to removing, disabling visibility will not clear attributes or affect layout.

Parameters **visible** – Whether or not the elements should be visible.

eventQ (*queue*) → self

Sets the queue onto which all events triggered by the selection should be added. Each queue handles events independently, and all queues execute in parallel. Since queues can be delayed (see `pause()`), this effectively enables multiple animations to run simultaneously.

The `None` queue is special; all events added to it will execute immediately. The default queue is named "default".

Parameters **queue** (*Union[Any, None]*) – The ID of the queue, which will be converted to a string, or `None` for the immediate queue. Defaults to "default".

Returns A new instance of the current selection using the specified event queue.

duration (*seconds*) → self

Configures the duration of all animations triggered by the selection. A duration of `0` will ensure that changes occur immediately. The default duration is `0.5`.

Parameters **seconds** (*ElementArg*[Union[int, float]]) – The animation duration, in seconds.

Returns A new instance of the current selection using the specified animation duration.

ease (*ease*) → self

Configures the ease function used in all animations triggered by the selection. This will affect the way attributes transition from one value to another. More information is available here: https://github.com/d3/d3-ease.

Parameters **ease** (*ElementArg*[str]) – The name of the ease function, based on the functions found in D3. The full list is below:

"linear", "poly", "poly-in", "poly-out", "poly-in-out", "quad", "quad-in", "quad-out", "quad-in-out", "cubic", "cubic-in", "cubic-out", "cubic-in-out", "sin", "sin-in", "sin-out", "sin-in-out", "exp", "exp-in", "exp-out", "exp-in-out", "circle", "circle-in", "circle-out", "circle-in-out", "elastic", "elastic-in", "elastic-out", "elastic-in-out", "back", "back-in", "back-out", "back-in-out", "bounce", "bounce-in", "bounce-out", "bounce-in-out".

Returns A new instance of the current selection using the specified animation ease.

highlight (*seconds*) → self
Returns a new selection through which all attribute changes are temporary. This is typically used to draw attention to a certain element without permanently changing its attributes.

Parameters **seconds** (Optional[*ElementArg*[Union[int, float]]]) – The amount of time attributes should remain 'highlighted', in seconds, before changing back to their original values. Defaults to `0.5`.

Returns A new instance of the current selection, where all attribute changes are temporary.

data (*data*) → self
Binds the selection to a list of data values. This will decide the arguments provided whenever an attribute is configured using a function (see *ElementArg*).

Parameters **data** – An iterable container of values to use as the data of this selection, which should have the same length as the number of elements in the selection. Alternatively, a function (*ElementFn*) transforming the selection's previous data. Use `null` to unbind the selection from its data, in which case the selection will fall back on its parent's data.

Type data: Union[Iterable[Any], ElementFn[Any]]

Raises **Exception** – If the length of the data does not equal the number of elements in the selection.

Returns A new instance of the current selection bound to the given data.

pause (*seconds*) → self
Adds a pause to the event queue, delaying the next event by the given number of seconds.

Parameters **seconds** (*Union[int, float]*) – The duration of the pause, in seconds.

stop (*queue*) → self
Stops the execution of all scheduled events on the given event queue. Note that this will still be added as an event onto the current queue.

Parameters **queue** (*Any*) – The ID of the queue to stop, which will be converted to a string.

stopall () → self
Stops the execution of all scheduled events on all event queues. Note this will still be added as an event onto the current queue.

start (*queue*) → self
Starts/resumes the execution of all scheduled events on the given event queue. Note this will still be added as an event onto the current queue.

Parameters **queue** (*Any*) – The name of the queue to start, or an iterable container of names. Defaults to "default".

startall () → self
Starts/resumes the execution of all scheduled events on all event queues. Note that this will still be added as an event onto the current queue.

cancel (*queue*) → self
Cancels all scheduled events on the given event queue. Note this will still be added as an event onto the current queue.

Parameters **queue** (*Any*) – The name of the queue to cancel, or an iterable container of names. Defaults to "default".

cancelall () → self
Cancels all scheduled events on all event queues. Note that this will still be added as an event onto the current queue.

broadcast (*message*) → self
Adds a message to the event queue, which will trigger a corresponding listener (see `listen()`). This can be used to detect when a queue reaches a certain point in execution, or to enable communication between a server.

Parameters **message** (*str*) – The message.

listen (*message*, *on_receive*) → self
Registers a function to listen for a specific broadcast message (see `broadcast()`). The function will be called when the corresponding broadcast event is processed by the event queue. If the same message is broadcast multiple times, the function will be called each time. This will also override any previous function listening for the same message.

Parameters

- **message** (*str*) – The message to listen for.
- **on_receive** (*Callable*) – The function to call when the message is received.

callback (*on_callback*) → self
Adds a callback to the event queue. This is roughly equivalent to broadcasting a unique message and setting up a corresponding listener. The callback function is guaranteed to only execute once.

Parameters **on_callback** (*Callable*) – The function to call when the callback event is processed by the event queue.

1.5.2 CanvasSelection

class graphics.**CanvasSelection** (*context*)

node (*id*) → graphics.NodeSelection.NodeSelection
Selects a single node by its ID.

Parameters **id** (*Any*) – The ID of the node, which will be converted to a string.

Returns A new selection corresponding to the given node.

nodes (*ids*) → graphics.NodeSelection.NodeSelection
Selects multiple nodes using a list of ID values.

Parameters **ids** (*Iterable[Any]*) – An iterable container of node IDs, which will be converted to strings.

Returns A new selection corresponding to the given nodes.

edge (*edge*) → None
Selects a single edge by its source, target, and optional ID. The additional ID value will distinguish edges connected to the same nodes. Once the edge has been added, source and target nodes can be provided in any order.

Parameters **edge** (*Tuple[Any, Any, Any]*) – A (source, target) or (source, target, ID) tuple. All values will be converted to strings.

Returns A new selection corresponding to the given edge.

edges (*edges*) → None
Selects multiple edges using a list of source, target, and optional ID tuples.

Parameters **edges** (*Iterable[Union[Tuple[Any, Any], Tuple[Any, Any, Any]]]*) – An iterable container of (source, target) or (source, target, ID) tuples. All values will be converted to strings.

Returns A new selection corresponding to the given edges.

label (*id*) → graphics.LabelSelection.LabelSelection
Selects a single label, attached to the canvas, by its ID.

Parameters **id** (*Any*) – The ID of the label, which will be converted to a string. Defaults to "title".

Returns A new selection corresponding to the given label.

labels (*ids*) → graphics.LabelSelection.LabelSelection
Selects multiple labels, attached to the canvas, using an array of ID values.

Parameters **ids** (*Iterable[Any]*) – An iterable container of labels IDs, which will be converted to strings.

Returns A new selection corresponding to the given labels.

size (*size*) → self
/** Sets the width and height of the canvas. This will determine the coordinate system, and will update the `width` and `height` attributes of the main SVG element, unless otherwise specified with *svgattr()*. Note that size is not animated by default.

Parameters **size** (*ElementArg*[Tuple[*NumExpr*, *NumExpr*]]) – A (width, height) tuple describing the size of the canvas.

edgelengths (*length_info*) → self
Sets method used to calculate edge lengths. Edges can either specify individual length values (see *length()*), or have their lengths dynamically calculated, in which case an 'average length' value can be provided. More information is available here: https://github.com/tgdwyer/WebCola/wiki/link-lengths.

The default setting is: (type="jaccard", average length=70).

Parameters **length_info** (*ElementArg*[Union[str, Tuple[str, *NumExpr*]]]) – Either a single string describing the edge length type, or a (type, average length) tuple. The valid edge length types are:

- "individual": Uses each edge's length attribute individually.
- "jaccard", "symmetric": Dynamic calculation using an 'average length' value.

pan (*location*) → self
Sets the location of the canvas camera. The canvas uses a Cartesian coordinate system with (0, 0) at the center.

Parameters **location** (*ElementArg*[Tuple[*NumExpr*, *NumExpr*]]) – An (x, y) tuple describing the new pan location.

zoom (*zoom*) → self
Sets the zoom level of the canvas camera. A zoom level of 2.0 will make objects appear twice as large, 0.5 will make them half as large, etc.

Parameters **zoom** (*ElementArg*[*NumExpr*]) – The new zoom level.

panlimit (*box*) → self
Restricts the movement of the canvas camera to the given bounding box, centered at (0, 0). The canvas will only be draggable when the camera is within the bounding box (i.e. the coordinates currently in view are a subset of the bounding box).

The default pan limit is: (-Infinity, Infinity).

Parameters **box** (*ElementArg*[Tuple[*NumExpr*, *NumExpr*]]) – A (width/2, height/2) tuple describing the bounding box.

zoomlimit (*limit*) → self
Restricts the zoom level of the canvas camera to the given range. The lower bound describes how far away the camera can zoom, while the upper bound describes the maximum enlarging zoom.

The default zoom limit is: (0.1, 10).

Parameters **limit** (*ElementArg*[Tuple[*NumExpr*, *NumExpr*]]) – A (min, max) tuple describing the zoom limit.

zoomkey (*required*) → self
Sets whether or not zooming requires the `ctrl`/`cmd` key to be held down. Disabled by default.

Parameters **required** (*ElementArg*[bool]) – True if the `ctrl`/`cmd` key is required, false otherwise.

svgattr (*key*, *value*)
Sets a custom SVG attribute on the canvas.

Parameters

- **key** (*str*) – The name of the SVG attribute.
- **value** (*ElementArg*[Union[str, int, float, None]]) – The value of the SVG attribute.

1.5.3 NodeSelection

class graphics.**NodeSelection** (*context*)

remove () → self
Removes all nodes in the current selection from the canvas. Additionally, removes any edges connected to the nodes.

label (*id*) → graphics.LabelSelection.LabelSelection
Selects a single label, attached to the node, by its ID.

By default, each node is initialized with a "value" label, located at the center of the node and displaying its ID. Any additional labels will be automatically positioned along the boundary of the node.

Parameters **id** (*Any*) – The ID of the label, which will be converted to a string. Defaults to "value".

Returns A new selection corresponding to the given label.

labels (*ids*) → graphics.LabelSelection.LabelSelection
Selects multiple labels, attached to the node, using a list of ID values.

Parameters **ids** (*Iterable[Any]*) – An iterable container of label IDs, which will be converted to strings.

Returns A new selection corresponding to the given labels.

shape (*shape*) → self
Sets the shape of the node. Note that shape cannot be animated or highlighted.

Parameters **shape** (*ElementArg*[str]) – One of the following strings:

- "circle": Standard circular node with a single radius dimension.
- "rect": Rectangular node with separate width and height dimensions, and corner rounding.
- "ellipse": Elliptical node with width and height dimensions.

color (*color*) → self
Sets the color of the node. The default color is "dark-gray".

Parameters **color** (*ElementArg*[str]) – A CSS color string.

size (*size*) → self
Sets the size of the node. If the node is a circle, a single radius value is sufficient. Otherwise, a tuple containing both the horizontal and vertical radius should be provided.

Note that size can be set relative to the node's current size using string expressions, e.g. "1.5x" for circles or ("1.5x", "1.5y") for rectangles and other shapes.

The default size is (12, 12).

Parameters **size** (*ElementArg*[Union[*NumExpr*, Tuple[*NumExpr*, *NumExpr*]]]) – The radius of the node, or a (width/2, height/2) tuple.

pos (*pos*) → self
Sets the position of the node. The canvas uses a Cartesian coordinate system with (0, 0) at the center.

Parameters **pos** (*ElementArg*[Tuple[*NumExpr*, *NumExpr*]]) – An (x, y) tuple describing the new position of the node.

fixed (*fixed*) → self
When set to true, this prevents the node from being automatically moved during the layout process. This does not affect manual dragging.

Parameters **fixed** (*ElementArg*[bool]) – True if the position of the node should be fixed, false otherwise.

draggable (*draggable*) → self
Sets whether or not the node can be manually dragged around.

Parameters **draggable** (*ElementArg*[bool]) – True if the node should be draggable, false otherwise.

click (*on_click*) → self
Registers a function to listen for node click events. This will override any previous function listening for click events on the same node.

Parameters **on_click** (*ElementFn*) – A function taking the node's data (see `data()`) and, optionally, index.

hoverin (*on_hoverin*) → self
Registers a function to listen for node mouse-over events, triggered when the mouse enters the node. This will override any previous function listening for hover-in events on the same node.

Parameters **on_hoverin** (*ElementFn*) – A function taking the node's data (see `data()`) and, optionally, index.

hoverout (*on_hoverout*) → self
Registers a function to listen for node mouse-over events, triggered when the mouse leaves the node. This will override any previous function listening for hover-out events on the same node.

Parameters **on_hoverout** (*ElementFn*) – A function taking the node's data (see `data()`) and, optionally, index.

svgattr (*key*, *value*)
Sets a custom SVG attribute on the node's shape.

Parameters

- **key** (*str*) – The name of the SVG attribute.

- **value** (*ElementArg*[Union[str, int, float, None]]) – The value of the SVG attribute.

1.5.4 EdgeSelection

class graphics.**EdgeSelection**(*context*)

traverse(*source*) → self

Sets the selection's animation type such that color (`color()`) is animated with a traversal, and configures the node at which the traversal should begin.

If no source is given, the first node in each edge tuple used to construct the selection will be used. If the source is not connected, the edge's actual source will be used.

Parameters **source** (Optional[*ElementArg*[Any]) – The ID of the node at which the traversal animation should begin, which will be converted to a string.

label(*id*) → graphics.LabelSelection.LabelSelection

Selects a single label, attached to the edge, by its ID.

Parameters **id** (*Any*) – The ID of the label, which will be converted to a string. Defaults to "weight".

Returns A new selection corresponding to the given label.

labels(*ids*) → graphics.LabelSelection.LabelSelection

Selects multiple labels, attached to the edge, using a list of ID values.

Parameters **ids** (*Iterable[Any]*) – An iterable container of label IDs, which will be converted to strings.

Returns A new selection corresponding to the given labels.

directed(*directed*) → self

Sets whether or not the edge should include an arrow pointing towards its target node.

Parameters **directed** (*ElementArg*[bool]) – True if the edge should be directed, false otherwise.

length(*length*) → self

Sets the length of the edge. This will only take effect when `edgelengths()` is set to "individual".

Parameters **length** (*ElementArg*[*NumExpr*]) – The length of the edge.

thickness(*thickness*) → self

Sets the thickness of the edge.

Parameters **thickness** (*ElementArg*[*NumExpr*]) – The thickness of the edge.

color(*color*) → self

Sets color of the edge. Note that this can be animated with a traversal animation (see `traverse()`). The default color is "light-gray".

Parameters **color** (*ElementArg*[str]) – A CSS color string.

flip(*flip*) → self

Sets whether or not the edge should be 'flipped' after exceeding a certain angle, such that it is never rendered upside-down. This only applies to edges connecting two nodes.

Parameters **flip** (*ElementArg*[bool]) – True if the edge should flip automatically, false otherwise.

curve (*curve*) → self

Sets the curve function used to interpolate the edge's path. The default setting is "cardinal". More information is available here: https://github.com/d3/d3-shape#curves.

Parameters **curve** (*ElementArg*[str]) – The name of the curve function, based on the functions found in D3. The full list is below:

"basis", "bundle", "cardinal", "catmull-rom", "linear", "monotone-x", "monotone-y", "natural", "step", "step-before", "step-after"

path (*path*) → self

Sets a custom path for the edge. The path is a list of (x, y) tuples, relative to the edge's origin, which will automatically connect to the boundaries of the source and target nodes.

If the edge connects two nodes, (0, 0) will be the midpoint between the two nodes. If edge is a looping edge connecting one node, (0, 0) will be a point along the node's boundary, in the direction of the edge.

Parameters **path** (*ElementArg*[Iterable[Tuple[*NumExpr*, *NumExpr*]]]) – An iterable container of (x, y) tuples.

svgattr (*key*, *value*)

Sets a custom SVG attribute on the edge's path.

Parameters

- **key** (*str*) – The name of the SVG attribute.
- **value** (*ElementArg*[Union[str, int, float, None]]) – The value of the SVG attribute.

1.5.5 LabelSelection

class graphics.**LabelSelection** (*context*)

text (*text*) → self

Sets the text displayed by the label. The newline character ("\n") can be used to break the text into multiple lines. Note that text cannot be animated or highlighted.

Parameters **text** (*ElementArg*[str]) – The text displayed by the label.

align (*align*) → self

Sets alignment of the label's text. This will affect the direction in which text is appended, as well as its positioning relative to the label's base position. For example, an alignment of "top-left" will ensure that the top left corner of the label is located at its base position.

A special "radial" alignment can be used to dynamically calculate the label's alignment based on its *angle()* and *rotate()* attributes, such that text is optimally positioned around an element.

Parameters **align** (*ElementArg*[str]) – A string describing the alignment, typically in the form "vertical-horizontal". The full list is below:

"top-left", "top-middle", "top-right", "middle-left", "middle", "middle-right", "bottom-left", "bottom-middle", "bottom-right", "radial".

pos (*pos*) → self

Sets the position of the the label relative to its parent element. This will always involve a Cartesian coordinate system. If the parent is a node, (0, 0) will be its center. If the parent is an edge connecting two nodes, (0, 0) will be the midpoint between the two nodes. If the parent is a looping edge connecting one node, (0, 0) will be a point along the node's boundary, in the direction of the edge.

Parameters **pos** (*ElementArg*[Tuple[*NumExpr*, *NumExpr*]]) – An (x, y) tuple describing the position of the label.

radius (*radius*) → self
Allows the label to be positioned using polar coordinates, together with the `angle()` attribute. This will specify the distance from the label's base position (see `pos()`).

Parameters **radius** (*ElementArg*[*NumExpr*]) – The polar radius, defined as the distance from the label's base position.

angle (*angle*) → self
Allows the label to be positioned using polar coordinates, together with the `radius()` attribute. This will specify the angle, in degrees, along a standard unit circle centered at the label's base position (see `pos()`).

Additionally, this will affect the rotation of the label, if enabled (see `rotate()`).

Parameters **angle** (*ElementArg*[*NumExpr*]) – The polar angle, in degrees, increasing counter-clockwise from the x-axis.

rotate (*rotate*) → self
Sets whether or not the label should rotate, using its `angle()` attribute. The exact rotation will also depend on the label's alignment. For example, an alignment of "top-center" together with an angle of 90 degrees will result in the text being upside-down.

Parameters **rotate** (*ElementArg*[bool]) – Whether or not the label should rotate.

color (*color*) → self
Sets the color of the label's text. The default color is "gray".

Parameters **color** (*ElementArg*[str]) – A CSS color string.

font (*font*) → self
Sets the font of the label's text.

Parameters **font** (*ElementArg*[str]) – A CSS font-family string.

size (*size*) → self
Sets the size of the label's text.

Parameters **size** (*ElementArg*[*NumExpr*]) – The size of the label's text, in pixels.

svgattr (*key*, *value*)
Sets a custom SVG attribute on the label's text.

Parameters

- **key** (*str*) – The name of the SVG attribute.
- **value** (*ElementArg*[Union[str, int, float, None]]) – The value of the SVG attribute.

1.6 Utilities

1.6.1 NetworkX

NetworkX graphs can be directly added to a canvas in the following way:

```
import networkx as nx
from algorithmx.networkx import add_graph

G = nx.MultiDiGraph()
G.add_nodes_from([1, 2, 3])
G.add_weighted_edges_from([(1, 2, 3.0), (2, 3, 7.5)])
```

(continues on next page)

(continued from previous page)

```
# canvas = ...

add_graph(canvas, G)
```

`algorithmx.networkx.add_graph`(*canvas: algorithmx.graphics.CanvasSelection.CanvasSelection*, *graph*, *weight: Optional[str] = 'weight'*) → algorithmx.graphics.CanvasSelection.CanvasSelection

Adds all nodes and edges from a NetworkX graph to the given canvas. Edges will automatically set the `directed()` attribute and/or add a weight `label()` depending on the provided graph.

Parameters

- **canvas** (`CanvasSelection`) – The CanvasSelection onto which the graph should be added.
- **graph** (`Any type of NetworkX graph`) – The NetworkX graph
- **weight** (`Union[str, None]`) – The name of the attribute which describes edge weight in the the NetworkX graph. Edges without the attribute will not display a weight, and a value of `None` will prevent any weight from being displayed. Defaults to "weight".

Returns The provided CanvasSelection with animations disabled, allowing initial attributes to be configured.

Return type `CanvasSelection`

1.7 Notebooks

Below are some examples, created using Jupyter notebooks:

1.7.1 Basic Examples

Let's import the library and create a simple network. You can hold down `ctrl`/`cmd` to zoom in.

```
import algorithmx

canvas = algorithmx.jupyter_canvas()
canvas.size((300, 200))

canvas.nodes([1, 2]).add()
canvas.edge((1, 2)).add()

canvas
```

That's nice, but now we would like to animate it. Let's also add some buttons so that we can easily start/stop/restart our animation.

```
canvas = algorithmx.jupyter_canvas(buttons=True)
canvas.size((300, 200))

canvas.nodes([1, 2]).add()
canvas.edge((1, 2)).add()

canvas.pause(0.5)
```

(continues on next page)

(continued from previous page)

```
canvas.node(1).highlight().size('1.25x').pause(0.5)
canvas.edge((1, 2)).traverse().color('blue')

canvas
```

Finally, lets apply all of this to a larger graph.

```
canvas = algorithmx.jupyter_canvas(buttons=True)

canvas.nodes(range(1, 8)).add()
canvas.edges([(i, i+1) for i in range(1, 7)]
           + [(1, 3), (2, 4), (2, 7)]).add()

for i in range(1, 8):
    canvas.pause(0.5)
    canvas.node(i).color('green').highlight().size('1.25x')

    if i < 8:
        canvas.pause(0.5)
        canvas.edge((i, i+1)).traverse().color('green')

canvas
```

1.7.2 NetworkX Examples

Let's begin by creating a directed graph with random edge weights.

```
import algorithmx
import networkx as nx
from random import randint

canvas = algorithmx.jupyter_canvas()

# Create a directed graph
G = nx.circular_ladder_graph(5).to_directed()
# Randomize edge weights
nx.set_edge_attributes(G, {e: {'weight': randint(1, 9)} for e in G.edges})

# Add nodes
canvas.nodes(G.nodes).add()

# Add directed edges with weight labels
canvas.edges(G.edges).add().directed(True) \
    .label().text(lambda e: G.edges[e]['weight'])

canvas
```

Next, we can use NetworkX run a breadth-first search, and AlgorithmX to animate it.

```
canvas = algorithmx.jupyter_canvas(buttons=True)
canvas.size((500, 400))

# Generate a 'caveman' graph with 3 cliques of size 4
G = nx.connected_caveman_graph(3, 4)
```

(continues on next page)

(continued from previous page)

```
# Add nodes and edges
canvas.nodes(G.nodes).add()
canvas.edges(G.edges).add()
canvas.pause(1)

# Traverse the graph using breadth-first search
bfs = nx.edge_bfs(G, 0)

# Animate traversal
source = None
for e in bfs:
    if e[0] != source:
        # Make the new source large
        canvas.node(e[0]).size('1.25x').color('purple')
        # Make the previous source small again
        if source is not None:
            canvas.node(source).size('0.8x')
        # Update source node
        source = e[0]
        canvas.pause(0.5)

    # Traverse edges
    canvas.edge(e).traverse().color('pink')
    canvas.pause(0.5)

# Make the remaining source small again
canvas.node(source).size('0.8x')

canvas
```

For our final visualization, let's find the shortest path on a random graph using Dijkstra's algorithm.

```
import random
random.seed(436)

canvas = algorithmx.jupyter_canvas(buttons=True)
canvas.size((500, 400))

# Generate random graph with random edge weights
G = nx.newman_watts_strogatz_graph(16, 2, 0.4, seed=537)
nx.set_edge_attributes(G, {e: randint(1, 20) for e in G.edges}, 'weight')

# Add nodes and edges with weight labels
canvas.nodes(G.nodes).add()
canvas.edges(G.edges).add().label().text(lambda e: G.edges[e]['weight'])
canvas.pause(1)

# Select source and target
source = 0
target = 8
canvas.node(source).color('green').highlight().size('1.25x')
canvas.node(target).color('red').highlight().size('1.25x')
canvas.pause(1.5)

# Run Dijkstra's shortest path algorithm
path = nx.dijkstra_path(G, source, target)
```

(continues on next page)

(continued from previous page)

```
# Animate the algorithm
path_length = 0
for i in range(len(path) - 1):
    u, v = path[i], path[i + 1]

    # Update path length
    path_length += G[u][v]['weight']

    # Traverse edge
    canvas.edge((u, v)).traverse().color('blue')
    canvas.pause(0.4)

    # Make the next node blue, unless it's the target
    if v != target:
        canvas.node(v).color('blue')

    # Add a label to indicate current path length
    canvas.node(v).label('path').add().color('blue').text(path_length)
    canvas.pause(0.4)

canvas
```

1.7.3 NetworkX Tutorial

In this tutorial we will take a look at ways of combining the analysis tools provided by NetworkX with the visualization capailities of AlgorithmX.

Simple Graph

Let's start by creating a simple NetworkX graph. We will use add_path to quickly add both nodes and edges.

```
import networkx as nx

G = nx.Graph()

nx.add_path(G, [1, 2, 3])
nx.add_path(G, [4, 2, 5])

print('Nodes:', G.nodes)
print('Edges:', G.edges)
```

```
Nodes: [1, 2, 3, 4, 5]
Edges: [(1, 2), (2, 3), (2, 4), (2, 5)]
```

Now that we have all the data we need, we can create an AlgorithmX canvas to display our nodes and edges.

```
import algorithmx

canvas = algorithmx.jupyter_canvas()

canvas.nodes(G.nodes).add()
canvas.edges(G.edges).add()
```

(continues on next page)

(continued from previous page)

```
canvas
```

So we have our simple graph, but we think it could look a little more interesting. Let's define a custom style for our nodes, and also give each one a different color. We can take advantage of the fact that nearly any argument in AlgorithmX can be passed as a lambda function, making our code much more concise.

```
canvas = algorithmx.jupyter_canvas()

node_style = {
    'shape': 'rect',
    'size': (20, 12)
}
node_colors = {1: 'red', 2: 'green', 3: 'blue', 4: 'orange', 5: 'purple'}

canvas.nodes(G.nodes).add() \
    .set(node_style) \
    .color(lambda n: node_colors[n])

canvas.edges(G.edges).add()

canvas
```

Making the graph directed is easy - all we have to do is call G.to_directed(), and then tell AlgorithmX that the edges should be rendered with an arrow.

Weighted and Directed Graphs To create a directed graph, all we need to do is use a NetworkX DiGraph, and tell AlgrithmX that edges should be rendered with an arrow.

```
G = nx.DiGraph()

G.add_nodes_from([1, 2, 3])
G.add_edges_from([(1, 2), (2, 3), (3, 1)])

canvas = algorithmx.jupyter_canvas()

canvas.nodes(G.nodes).add()
canvas.edges(G.edges).add().directed(True)

canvas
```

To create wighted graph, we will first ensure that our NetworkX edges have a 'weight' attribute. Then, we will add a label to each edge displaying the attribute.

```
G = nx.Graph()

G.add_nodes_from([1, 2, 3])
G.add_weighted_edges_from([(1, 2, 0.4), (2, 3, 0.2), (3, 1, 0.3)])

canvas = algorithmx.jupyter_canvas()

canvas.nodes(G.nodes).add()
canvas.edges(G.edges).add() \
    .label().add() \
        .text(lambda e: G.edges[e]['weight'])

canvas
```

Finally, AlgorithmX provides a uility to simplify this process.

```
from algorithmx.networkx import add_graph

G = nx.DiGraph()

G.add_nodes_from([1, 2, 3])
G.add_weighted_edges_from([(1, 2, 0.4), (2, 3, 0.2), (3, 1, 0.3)])

canvas = algorithmx.jupyter_canvas()

add_graph(canvas, G)
```

Random Graph

NetworkX provides a range of functions for generating graphs. For generating a random graph, we will use the basic gnp_random_graph function. By providing a seed, we can ensure that we get the same graph every time (otherwise there is no guarantee of it being connected!).

```
G = nx.gnp_random_graph(10, 0.3, 138)

canvas = algorithmx.jupyter_canvas()
canvas.nodes(G.nodes).add()
canvas.edges(G.edges).add()

canvas
```

To make the graph directed, we will simply use G.to_directed. To make the graph weighted, we will need to configure a weight attribute for each edge. Since our graph is random, we'll make our edge weights random as well. For this we will use the set_edge_attributes function.

```
from random import randint

G = G.to_directed()
nx.set_edge_attributes(G, {e: {'weight': randint(1, 10)} for e in G.edges})
```

We can now display the graph using the utility from before.

```
canvas = algorithmx.jupyter_canvas()
add_graph(canvas, G)
```

Detailed Graph

Now we are going to create a graph that displays a range of interesting properties. Let's begin by generating a random weighted graph, as before.

```
G = nx.gnp_random_graph(10, 0.3, 201)
nx.set_edge_attributes(G, {e: {'weight': randint(1, 10)} for e in G.edges})
```

Next, we will use NetworkX to calculate the graph's coloring and edge centrality.

```
coloring = nx.greedy_color(G)
centrality = nx.edge_betweenness_centrality(G, weight='weight', normalized=True)
```

We can now begin displaying the graph. First, we will add the nodes and assign them a color based on their calculated priority. We happen to know that any graph requires at most 4 different colors, and so we prepare these beforehand.

```
canvas = algorithmx.jupyter_canvas()

color_priority = {0: 'red', 1: 'orange', 2: 'yellow', 3: 'green'}

canvas.nodes(G.nodes).add() \
    .color(lambda n: color_priority[coloring[n]])

print(coloring)
```

```
{4: 0, 2: 1, 3: 2, 0: 1, 1: 2, 6: 0, 8: 1, 7: 2, 9: 2, 5: 0}
```

Afterwards, we will add the edges. Each one will have two labels; one to display it's weight, and another to display it's calculated centrality.

```
init_edges = canvas.edges(G.edges).add()

formatted_centrality = {k: '{0:.2f}'.format(v) for k, v in centrality.items()}

init_edges.label().add() \
    .text(lambda e: G.edges[e]['weight']) \

init_edges.label('centrality').add() \
    .color('blue') \
    .text(lambda e: formatted_centrality[e])

print(formatted_centrality)
```

```
{(0, 1): '0.00', (0, 3): '0.11', (0, 4): '0.09', (1, 4): '0.33', (1, 8): '0.18', (2,
↪3): '0.11', (2, 4): '0.22', (2, 5): '0.20', (2, 6): '0.02', (2, 7): '0.09', (3, 4):
↪'0.22', (3, 6): '0.24', (4, 8): '0.00', (4, 9): '0.18', (6, 7): '0.11', (8, 9): '0.
↪02'}
```

Finally, we can see the whole graph.

```
canvas
```

1.8 Developer Install

To install a developer version of algorithmx, you will first need to clone the repository:

```
git clone https://github.com/algorithmx/algorithmx-python
cd algorithmx-python
```

Next, install it with a develop install using pip:

```
pip install -e .
```

If you are planning on working on the JS/frontend code, you should also do a link installation of the extension:

```
jupyter nbextension install [--sys-prefix / --user / --system] --symlink --py␣
↪algorithmx

jupyter nbextension enable [--sys-prefix / --user / --system] --py algorithmx
```

with the appropriate flag. Or, if you are using Jupyterlab:

```
jupyter labextension install .
```

PYTHON MODULE INDEX

a

INDEX

S

T

V

Z