N2G

Release 0.3.0

Denis Mulyalin

Feb 11, 2023

CONTENTS

CHAPTER

ONE

OVERVIEW

N2G is a library to produce XML text files structured in a format supported for opening and editing by these applications:

- yWorsk yEd Graph Editor and yEd web application
- Diagrams DrawIO desktop and DrawIO web application

N2G contains dedicated modules for each format with very similar API that can help create, load, modify and save diagrams.

However, due to discrepancy in functionality and peculiarities of applications itself, N2G modules API is not 100% identical and differ to reflect particular application capabilities.

CHAPTER

TWO

INSTALLATION

Install from PYPI using pip:

```
pip install N2G
```

Or copy repository from GitHub and run:

```
python -m pip install .
```

N2G core functionality uses Python built-in libraries, but additional features require 3rd party dependencies that can be installed using `full` extras:

```
pip install N2G[full]
```

CHAPTER

THREE

DIAGRAM PLUGINS

Diagram plugins take structured data or API calls as input and produce results that can be used with diagramming application supported by plugin.

3.1 yEd Diagram Plugin

N2G yEd Module supports producing graphml XML structured text files that can be opened by yWorsk yEd Graph Editor or yEd web application.

3.1.1 Quick start

Nodes and links can be added one by one using `add_node` and `add_link` methods

```
from N2G import yed_diagram

diagram = yed_diagram()
diagram.add_node('R1', top_label='Core', bottom_label='ASR1004')
diagram.add_node('R2', top_label='Edge', bottom_label='MX240')
diagram.add_link('R1', 'R2', label='DF', src_label='Gi0/1', trgt_label='ge-0/1/2')
diagram.layout(algo="kk")
diagram.dump_file(filename="Sample_graph.graphml", folder="./Output/")
```

After opening and editing diagram, it might look like this:

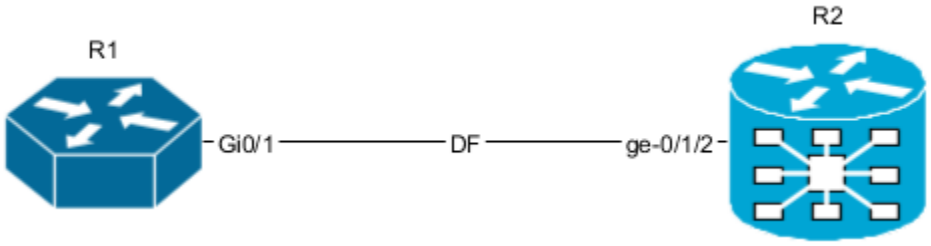
3.1.2 Adding SVG nodes

By default N2G uses shape nodes, but svg image can be sourced from directory on your system and used as node image instead. However, svg images as nodes can support only one label attribute, that label will be displayed above svg picture.

```
from N2G import yed_diagram

diagram = yed_diagram()
diagram.add_node('R1', pic="router.svg", pic_path="./Pics/")
diagram.add_node('R2', pic="router_edge.svg", pic_path="./Pics/")
diagram.add_link('R1', 'R2', label='DF', src_label='Gi0/1', trgt_label='ge-0/1/2')
diagram.layout(algo="kk")
diagram.dump_file(filename="Sample_graph.graphml", folder="./Output/")
```

After opening and editing diagram, it might look like this:



3.1.3 Nodes and links data attributes

Description and URL attributes can be added to node and link. Description attribute can be used by yEd to search for elements as well as diagrams exported in svg format can display data attributes as a tooltips.

```
from N2G import yed_diagram

diagram = yed_diagram()
diagram.add_node('R1', top_label='Core', bottom_label='ASR1004', description="loopback0:␣
↪192.168.1.1", url="google.com")
diagram.add_node('R2', top_label='Edge', bottom_label='MX240', description="loopback0:␣
↪192.168.1.2")
diagram.add_link('R1', 'R2', label='DF', src_label='Gi0/1', trgt_label='ge-0/1/2',␣
↪description="link media-type: 10G-LR", url="github.com")
diagram.layout(algo="kk")
diagram.dump_file(filename="Sample_graph.graphml", folder="./Output/")
```

After opening and editing diagram, it might look like this:

Node R1 and link should be clickable on above image as they contain URL information, tooltip should be displayed if svg will be open on its own.

3.1.4 Loading graph from dictionary

Diagram elements can be loaded from dictionary structure. That dictionary may contain nodes, links and edges keys, these keys should contain list of dictionaries where each dictionary item will contain elements attributes such as id, labels, description etc.

```
from N2G import yed_diagram

diagram = yed_diagram()
sample_graph={
'nodes': [
    {'id': 'a', 'pic': 'router.svg', 'label': 'R1' },
    {'id': 'R2', 'bottom_label':'CE12800', 'top_label':'1.1.1.1'},
    {'id': 'c', 'label': 'R3', 'bottom_label':'FI', 'top_label':'fns751', 'description':
↪'role: access'},
    {'id': 'd', 'pic':'firewall.svg', 'label': 'FW1', 'description': 'location: US'},
    {'id': 'R4', 'pic': 'router'}
],
```

(continues on next page)

(continued from previous page)

```
'links': [
    {'source': 'a', 'src_label': 'Gig0/0\nUP', 'label': 'DF', 'target': 'R2', 'trgt_label
↪': 'Gig0/1', 'description': 'role: uplink'},
    {'source': 'R2', 'src_label': 'Gig0/0', 'label': 'Copper', 'target': 'c', 'trgt_label
↪': 'Gig0/2'},
    {'source': 'c', 'src_label': 'Gig0/0', 'label': 'ZR', 'target': 'a', 'trgt_label':
↪'Gig0/3'},
    {'source': 'd', 'src_label': 'Gig0/10', 'label': 'LR', 'target': 'c', 'trgt_label':
↪'Gig0/8'},
    {'source': 'd', 'src_label': 'Gig0/11', 'target': 'R4', 'trgt_label': 'Gig0/18'}
]}
diagram.from_dict(sample_graph)
diagram.layout(algo="kk")
diagram.dump_file(filename="Sample_graph.graphml", folder="./Output/")
```

After opening and editing diagram, it might look like this:

3.1.5 Loading graph from list

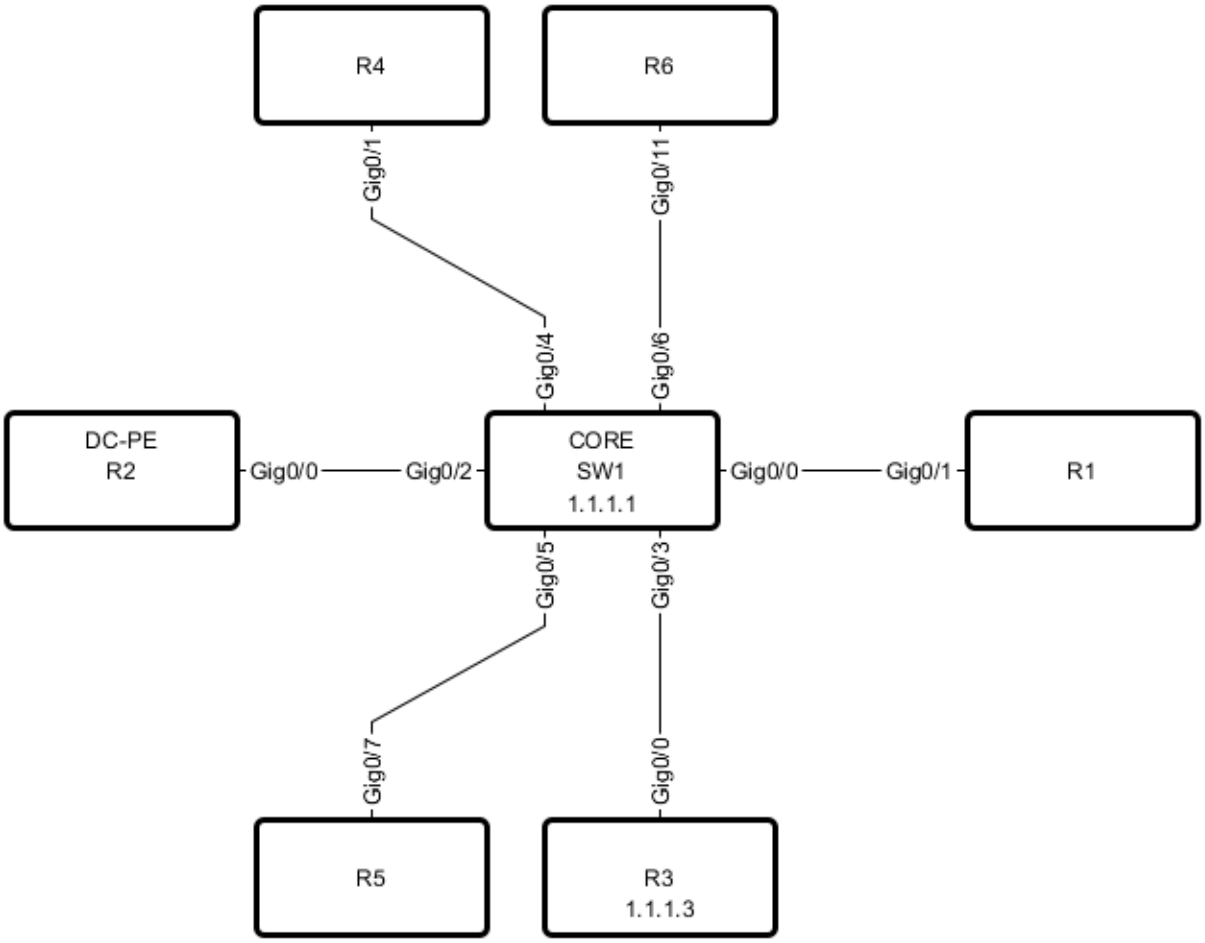
From list method allows to load graph from list of dictionaries, generally containing link details like source, target, labels. Additionally source and target can be defined using dictionaries as well, containing nodes details.

Note: Non-existing node will be automatically added on first encounter, by default later occurrences of same node will not lead to node attributes change, that behavior can be changed setting `node_duplicates` yed_diagram attribute equal to *update* value.

```
from N2G import yed_diagram

diagram = yed_diagram()
sample_list_graph = [
    {'source': {'id': 'SW1', 'top_label': 'CORE', 'bottom_label': '1,1,1,1'}, 'src_label
↪': 'Gig0/0', 'target': 'R1', 'trgt_label': 'Gig0/1'},
    {'source': {'id': 'R2', 'top_label': 'DC-PE'}, 'src_label': 'Gig0/0', 'target': 'SW1
↪', 'trgt_label': 'Gig0/2'},
    {'source': {'id':'R3', 'bottom_label': '1.1.1.3'}, 'src_label': 'Gig0/0', 'target':
↪'SW1', 'trgt_label': 'Gig0/3'},
    {'source': 'SW1', 'src_label': 'Gig0/4', 'target': 'R4', 'trgt_label': 'Gig0/1'},
    {'source': 'SW1', 'src_label': 'Gig0/5', 'target': 'R5', 'trgt_label': 'Gig0/7'},
    {'source': 'SW1', 'src_label': 'Gig0/6', 'target': 'R6', 'trgt_label': 'Gig0/11'}
]
diagram.from_list(sample_list_graph)
diagram.dump_file(filename="Sample_graph.graphml", folder="./Output/")
```

After opening and editing diagram, it might look like this:



3.1.6 Loading graph from csv

Similar to `from_dict` or `from_list` methods, from_csv method can take csv data with elements details and add them to diagram. Two types of csv table should be provided - one for nodes, another for links.

```
from N2G import yed_diagram

diagram = yed_diagram()
csv_links_data = """"source","src_label","label","target","trgt_label","description"
"a","Gig0/0\nUP","DF","R1","Gig0/1","vlans_trunked: 1,2,3\nstate: up"
"R1","Gig0/0","Copper","c","Gig0/2",
"R1","Gig0/0","Copper","e","Gig0/2",
d,Gig0/21,FW,e,Gig0/23,
"""
csv_nodes_data="""id","pic","label","bottom_label","top_label","description"
a,router,"R12",,,
"R1",,,"SGD1378","servers",
"c",,"R3","SGE3412","servers","1.1.1.1"
"d","firewall.svg","FW1",,,"2.2.2.2"
"e","router","R11",,,
```

(continues on next page)

(continued from previous page)

```
"""
diagram.from_csv(csv_nodes_data)
diagram.from_csv(csv_links_data)
diagram.dump_file(filename="Sample_graph.graphml", folder="./Output/")
```

After opening and editing diagram, it might look like this:

3.1.7 Loading existing diagrams

N2G yEd module uses custom `nmetadata` and `emetadata` attributes to store original node and link id. For nodes, `nmetadata` contains node id in a format `{'id': 'node_id_value'}`, for links `emetadata` contains source and target node ids as well as link id, e.g. `{"sid": "SW1", "tid": "R6", "id": "8e96ade0d90d33c3308721dc2a53b391"}`, where link id calculated using rules described in *API reference* section.

`nmetadata` and `emetadata` custom attributes used to properly load previously produced diagrams for modification, as a result:

> **Warning:** currently, N2G yEd module can properly load only diagrams that were created by this module in the first place or diagrams that had manually added `nmetadata` and `emetadata` attributes.

N2G yEd module provides `from_file` and `from_text` methods to load existing diagram content, to load diagram from file one can use this as example:

```
from N2G import yed_diagram

diagram = yed_diagram()
diagram.from_file("./source/diagram_old.graphml")
```

After diagram loaded it can be modified or updated using `add_x`, `from_x`, `delete_x` or `compare` methods.

3.1.8 Diagram layout

To arrange diagram nodes in certain way one can use `layout` method that relies on igraph library to calculate node coordinates in accordance with certain algorithm. List of supported layout algorithms and their details can be found here together with brief description in *API Reference* section.

Sample code to layout diagram:

```
from N2G import yed_diagram

diagram = yed_diagram()
diagram.from_file("./source/diagram_old.graphml")
diagram.layout(algo="drl", width=500, height=500)
diagram.dump_file(filename="Sample_graph.graphml", folder="./Output/")
```

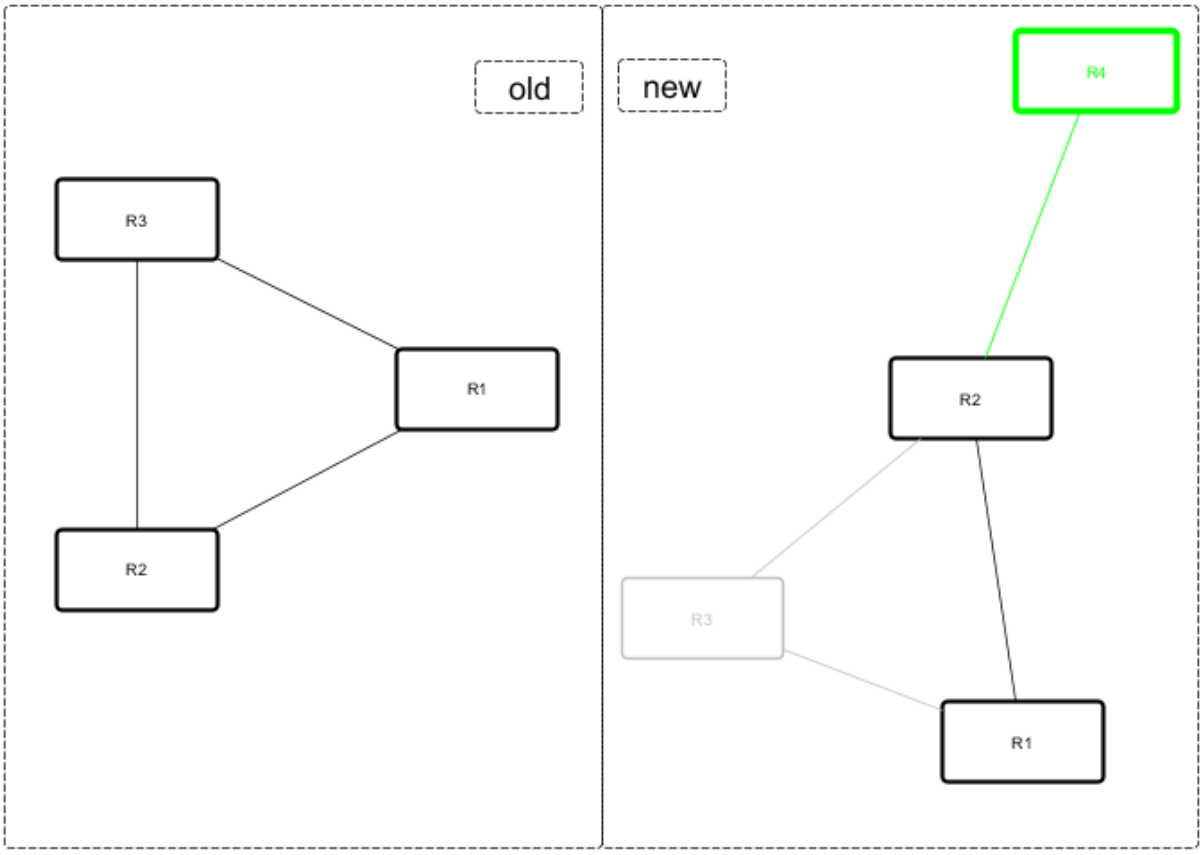
3.1.9 Comparing diagrams

Comparing diagrams can be useful to spot changes in your system. N2G `compare` method allow to calculate differences between old and new graphs and produce resulting diagram highlighting changes.

```
from N2G import yed_diagram

diagram = yed_diagram()
old_graph = {
'nodes': [
    {'id': 'R1'}, {'id': 'R2'}, {'id': 'R3'},
    ],
'edges': [
    {'source': 'R1', 'target': 'R2'},
    {'source': 'R2', 'target': 'R3'},
    {'source': 'R3', 'target': 'R1'}
]}
new_graph = {
'nodes': [
    {'id': 'R1'}, {'id': 'R2'}, {'id': 'R4'},
    ],
'edges': [
    {'source': 'R1', 'target': 'R2'},
    {'source': 'R2', 'target': 'R4'}
]}
diagram.from_dict(old_graph)
diagram.compare(new_graph)
diagram.layout(algo="kk", width=500, height=500)
diagram.dump_file(filename="Sample_graph.graphml", folder="./Output/")
```

Original and after diagrams:



R3 and its links are missing - highlighted in gray, but R4 and its link is new - highlighted in green.

3.1.10 API reference

API reference for N2G yEd module.

class N2G.plugins.diagrams.N2G_yEd.**yed_diagram**(*node_duplicates='skip'*, *link_duplicates='skip'*)

N2G yEd module allows to produce diagrams in yEd .graphml format.

Parameters

- `node_duplicates` (str) can be of value skip, log, update
- `link_duplicates` (str) can be of value skip, log, update

add_link(*source*, *target*, *label=''*, *src_label=''*, *trgt_label=''*, *description=''*, *attributes=None*, *url=''*, *link_id=None*)

Method to add link between nodes.

Parameters

- `source` (str) mandatory, id of source node
- `target` (str) mandatory, id of target node
- `label` (str) label at the center of the edge, by default equal to id attribute
- `src_label` (str) label to display at the source end of the edge
- `trgt_label` (str) label to display at target end of the edge

- `description` (str) string to save as link `description` attribute
- `url` (str) string to save as link `url` attribute
- `attributes` (dict) dictionary of yEd graphml tag names and attributes
- `link_id` (str or int) optional link id value, must be unique across all links

Attributes dictionary keys will be used as xml tag names and values dictionary will be used as xml tag attributes, example:

```
{
    "LineStyle": {"color": "#00FF00", "width": "1.0"},
    "EdgeLabel": {"textColor": "#00FF00"},
}
```

Note: If source or target nodes does not exists, they will be automatically created

add_node(*id*, ***kwargs*)

Convenience method to add node, by calling one of node add methods following these rules:

- If `pic` attribute in kwargs, `add_svg_node` is called
- If `group` kwargs attribute equal to *True*, `_add_group_node` called
- `add_shape_node` called otherwise

Parameters

- `id` (str) mandatory, unique node identifier, usually equal to node name

add_shape_node(*id*, *label=''*, *top_label=''*, *bottom_label=''*, *attributes=None*, *description=''*, *shape_type='roundrectangle'*, *url=''*, *width=120*, *height=60*, *x_pos=200*, *y_pos=150*, ***kwargs*)

Method to add node of type "shape".

Parameters

- `id` (str) mandatory, unique node identifier, usually equal to node name
- `label` (str) label at the center of the node, by default equal to id attribute
- `top_label` (str) label displayed at the top of the node
- `bottom_label` (str) label displayed at the bottom of the node
- `description` (str) string to save as node `description` attribute
- `shape_type` (str) shape type, default - "roundrectangle"
- `url` (str) url string to save a node `url` attribute
- `width` (int) node width in pixels
- `height` (int) node height in pixels
- `x_pos` (int) node position on x axis
- `y_pos` (int) node position on y axis
- `attributes` (dict) dictionary of yEd graphml tag names and attributes

Attributes dictionary keys will be used as xml tag names and values dictionary will be used as xml tag attributes, example:

```
{
    'Shape'     : {'type': 'roundrectangle'},
    'DropShadow': { 'color': '#B3A691', 'offsetX': '5', 'offsetY': '5'}
}
```

add_svg_node(*pic*, *id*, *pic_path='./Pics/'*, *label=''*, *attributes=None*, *description=''*, *url=''*, *width=50*, *height=50*, *x_pos=200*, *y_pos=150*, ***kwargs*)

Method to add SVG picture as node by loading SVG file content into graphml

Parameters

- `id` (str) mandatory, unique node identifier, usually equal to node name
- `pic` (str) mandatory, name of svg file
- `pic_path` (str) OS path to SVG file folder, default is `./Pics/`
- `label` (str) label displayed above SVG node, if not provided, label set equal to id
- `description` (str) string to save as node `description` attribute
- `url` (str) url string to save as node `url` attribute
- `width` (int) node width in pixels
- `height` (int) node height in pixels
- `x_pos` (int) node position on x axis
- `y_pos` (int) node position on y axis
- `attributes` (dict) dictionary of yEd graphml tag names and attributes

Attributes dictionary keys will be used as xml tag names and values dictionary will be used as xml tag attributes, example:

```
{
    'DropShadow': { 'color': '#B3A691', 'offsetX': '5', 'offsetY': '5'}
}
```

compare(*data*, *missing_nodes=None*, *new_nodes=None*, *missing_links=None*, *new_links=None*)

Method to combine two graphs - existing and new - and produce resulting graph following these rules:

- nodes and links present in new graph but not in existing graph considered as new and will be updated with `new_nodes` and `new_links` attributes by default highlighting them in green
- nodes and links missing from new graph but present in existing graph considered as missing and will be updated with `missing_nodes` and `missing_links` attributes by default highlighting them in gray
- nodes and links present in both graphs will remain unchanged

Parameters

- `data` (dict) dictionary containing new graph data, dictionary format should be the same as for `from_dict` method.
- `missing_nodes` (dict) dictionary with attributes to apply to missing nodes
- `new_nodes` (dict) dictionary with attributes to apply to new nodes
- `missing_links` (dict) dictionary with attributes to apply to missing links
- `new_links` (dict) dictionary with attributes to apply to new links

Sample usage:

```
from N2G import yed_diagram
diagram = yed_diagram()
new_graph = {
    'nodes': [
        {'id': 'a', 'pic': 'router_round', 'label': 'R1' }
    ],
    'edges': [
        {'source': 'f', 'src_label': 'Gig0/21', 'label': 'DF', 'target': 'b'}
    ]
}
diagram.from_file("./old_graph.graphml")
diagram.compare(new_graph)
diagram.dump_file(filename="compared_graph.graphml")
```

delete_link(*id=None*, *ids=None*, *label=''*, *src_label=''*, *trgt_label=''*, *source=''*, *target=''*)
Method to delete link by its id. Bulk delete operation supported by providing list of link ids to delete.

If link `id` or `ids` not provided, id calculated based on - `label, src_label, trgt_label, source, target` - attributes using this algorithm:

1. Edge tuple produced: `tuple(sorted([label, src_label, trgt_label, source, target]))`
2. MD5 hash derived from tuple: `hashlib.md5(",".join(edge_tup).encode()).hexdigest()`

Parameters

- `id` (str) id of single link to delete
- `ids` (list) list of link ids to delete
- `label` (str) link label to calculate id of single link to delete
- `src_label` (str) link source label to calculate id of single link to delete
- `trgt_label` (str) link target label to calculate id of single link to delete
- `source` (str) link source to calculate id of single link to delete
- `target` (str) link target to calculate id of single link to delete

delete_node(*id=None*, *ids=None*)
Method to delete node by its id. Bulk delete operation supported by providing list of node ids to delete.

Parameters

- `id` (str) id of single node to delete
- `ids` (list) list of node ids to delete

dump_file(*filename=None*, *folder='./Output/'*)
Method to save current diagram in .graphml file.

Parameters

- `filename` (str) name of the file to save diagram into
- `folder` (str) OS path to folder where to save diagram file

If no `filename` provided, timestamped format will be used to produce filename, e.g.: `Sun Jun 28 20-30-57 2020_output.graphml`

dump_xml()
Method to return current diagram XML text

from_csv(*text_data*)

Method to build graph from CSV tables

Parameters

- `text_data` (str) CSV text with links or nodes details

This method supports loading CSV text data that contains nodes or links information. If `id` in headers, `from_dict` method will be called for CSV processing, `from_list` method will be used otherwise.

CSV data with nodes details should have headers matching add node methods arguments and rules.

CSV data with links details should have headers matching `add_link` method arguments and rules.

Sample CSV table with link details:

```
"source","src_label","label","target","trgt_label","description"
"a","Gig0/0","DF","b","Gig0/1","vlans_trunked: 1,2,3"
"b","Gig0/0","Copper","c","Gig0/2",
"b","Gig0/0","Copper","e","Gig0/2",
d,Gig0/21,FW,e,Gig0/23,
```

Sample CSV table with node details:

```
"id","pic","label","bottom_label","top_label","description"
a,router_1,"R1,2",,,
"b",,,"some","top_some",
"c",,"somelabel","botlabel","toplabel","some node description"
"d","firewall.svg","somelabel1",,,"some node description"
"e","router_2","R1",,,
```

from_dict(*data*)

Method to build graph from dictionary.

Parameters

- `data` (dict) dictionary with nodes and link/edges details.

Example `data` dictionary:

```
sample_graph = {
    'nodes': [
        {
            'id': 'a',
            'pic': 'router',
            'label': 'R1'
        },
        {
            'id': 'b',
            'label': 'somelabel',
            'bottom_label':'botlabel',
            'top_label':'toplabel',
            'description': 'some node description'
        },
        {
            'id': 'e',
            'label': 'E'
        }
```

(continues on next page)

(continued from previous page)

```
    ],
    'edges': [
        {
            'source': 'a',
            'src_label': 'Gig0/0',
            'label': 'DF',
            'target': 'b',
            'trgt_label': 'Gig0/1',
            'description': 'vlans_trunked: 1,2,3'
        }
    ],
    'links': [
        {
            'source': 'a',
            'target': 'e'
        }
    ]
}
```

Dictionary Content Rules

- dictionary may contain `nodes` key with a list of nodes dictionaries
- each node dictionary must contain unique `id` attribute, other attributes are optional
- dictionary may contain `edges` or `links` key with a list of edges dictionaries
- each link dictionary must contain `source` and `target` attributes, other attributes are optional

from_file(*filename*, *file_load='xml'*)

Method to load data from file for processing. File format can be yEd graphml (XML) or CSV

Parameters

- `filename` (str) OS path to file to load
- `file_load` (str) indicated the load of the file, supports `xml`, `csv`

from_list(*data*)

Method to build graph from list.

Parameters

- `data` (list) list of link dictionaries,

Example `data` list:

```
sample_graph = [
    {
        'source': 'a',
        'src_label': 'Gig0/0\nUP',
        'label': 'DF',
        'target': 'b',
        'trgt_label': 'Gig0/1',
        'description': 'vlans_trunked: 1,2,3\nstate: up'
    },
    {
        'source': 'a',
```

(continues on next page)

(continued from previous page)

```
        'target': {
                'id': 'e',
                'label': 'somelabel',
                'bottom_label':'botlabel',
                'top_label':'toplabel',
                'description': 'some node description'
            }
        }
    }
]
```

List Content Rules

- each list item must have `target` and `source` attributes defined
- `target`/`source` attributes can be either a string or a dictionary
- dictionary `target`/`source` node must contain `id` attribute and other supported node attributes

---

Note: By default yed_diagram object `node_duplicates` action set to 'skip' meaning that node will be added on first occurrence and ignored after that. Set `node_duplicates` to 'update' if node with given id need to be updated by later occurrences in the list.

---

from_xml(*text_data*)
Method to load yEd graphml XML formatted text for processing

Parameters

- `text_data` (str) text data to load

layout(*algo='kk'*, *width=1360*, *height=864*, ***kwargs*)
Method to calculate graph layout using Python igraph library

Parameters

- `algo` (str) name of layout algorithm to use, default is 'kk'. Reference *Layout algorithms* table below for valid algo names
- `width` (int) width in pixels to fit layout in
- `height` (int) height in pixels to fit layout in
- `kwargs` any additional kwargs to pass to igraph `Graph.layout` method

Layout algorithms

algo name	description
circle, circular	Deterministic layout that places the vertices on a circle
drl	The Distributed Recursive Layout algorithm for large graphs
fr	Fruchterman-Reingold force-directed algorithm
fr3d, fr_3d	Fruchterman-Reingold force-directed algorithm in three dimensions
grid_fr	Fruchterman-Reingold force-directed algorithm with grid heuristics for large graphs
kk	Kamada-Kawai force-directed algorithm
kk3d, kk_3d	Kamada-Kawai force-directed algorithm in three dimensions
large, lgl, large_graph	The Large Graph Layout algorithm for large graphs
random	Places the vertices completely randomly
random_3d	Places the vertices completely randomly in 3D
rt, tree	Reingold-Tilford tree layout, useful for (almost) tree-like graphs
rt_circular, tree	Reingold-Tilford tree layout with a polar coordinate post-transformation, useful for (almost) tree-like graphs
sphere, spherical, circular_3d	Deterministic layout that places the vertices evenly on the surface of a sphere

`set_attributes`(*element*, *attributes=None*)

Method to set attributes for XML element

Parameters

- `element` (object) xml etree element object to set attributes for
- `attributes` (dict) dictionary of yEd graphml tag names and attributes

Attributes dictionary keys will be used as xml tag names and values dictionary will be used as xml tag attributes, example:

```
{
    "LineStyle": {"color": "#00FF00", "width": "1.0"},
    "EdgeLabel": {"textColor": "#00FF00"},
}
```

`update_link`(*edge_id=''*, *label=''*, *src_label=''*, *trgt_label=''*, *source=''*, *target=''*, *new_label=None*, *new_src_label=None*, *new_trgt_label=None*, *description=''*, *attributes=None*)

Method to update edge/link details.

Parameters

- `edge_id` (str) md5 hash edge id, if not provided, will be generated based on edge attributes
- `label` (str) existing edge label
- `src_label` (str) existing edge src_label
- `trgt_label` (str) existing edge tgt_label
- `source` (str) existing edge source node ID
- `target` (str) existing edge target node id
- `new_label` (str) new edge label
- `new_src_label` (str) new edge src_label
- `new_trgt_label` (str) new edge tgt_label

- `description` (str) new edge description
- `attributes` (str) dictionary of attributes to apply to edge element

Either of these must be provided to find edge element to update:

- `edge_id` MD5 hash or
- `label, src_label, trgt_label, source, target` attributes to calculate `edge_id`

`edge_id` calculated based on - `label, src_label, trgt_label, source, target` - attributes following this algorithm:

1. Edge tuple produced: `tuple(sorted([label, src_label, trgt_label, source, target]))`
2. MD5 hash derived from tuple: `hashlib.md5(",".join(edge_tup).encode()).hexdigest()`

This method will replace existing and add new labels to the link.

Existing description attribute will be replaced with new value.

Attributes will replace existing values.

update_node(*id*, *label=None*, *top_label=None*, *bottom_label=None*, *attributes=None*, *description=None*, *width=''*, *height=''*)

Method to update node details

Parameters

- `id` (str) mandatory, unique node identifier, usually equal to node name
- `label` (str) label at the center of the shape node or above SVG node
- `top_label` (str) label displayed at the top of the node
- `bottom_label` (str) label displayed at the bottom of the node
- `description` (str) string to save as node `description` attribute
- `width` (int) node width in pixels
- `height` (int) node height in pixels
- `attributes` (dict) dictionary of yEd graphml tag names and attributes

Attributes dictionary keys will be used as xml tag names and values dictionary will be used as xml tag attributes, example:

```
{
    'Shape'     : {'type': 'roundrectangle'},
    'DropShadow': { 'color': '#B3A691', 'offsetX': '5', 'offsetY': '5'}
}
```

This method will replace existing and add new labels to the node.

Existing description attribute will be replaced with new value.

Height and width will override existing values.

Attributes will replace existing values.

3.2 Drawlo Diagram Plugin

N2G Drawio Module supports producing XML structured text files that can be opened by Diagrams DrawIO desktop or DrawIO web applications

3.2.1 Quick start

Nodes and links can be added one by one using `add_node` and `add_link` methods

```
from N2G import drawio_diagram

diagram = drawio_diagram()
diagram.add_diagram("Page-1")
diagram.add_node(id="R1")
diagram.add_node(id="R2")
diagram.add_link("R1", "R2", label="DF", src_label="Gi1/1", trgt_label="GE23")
diagram.layout(algo="kk")
diagram.dump_file(filename="Sample_graph.drawio", folder="./Output/")
```

After opening and editing diagram, it might look like this:

Working with drawio module should be started with adding new diagram, after that nodes and links can be added. It is possible to switch between diagrams to edit using `go_to_diagram` method.

Note: link `src_label` and `trgt_label` attributes supported starting with 0.2.0 version

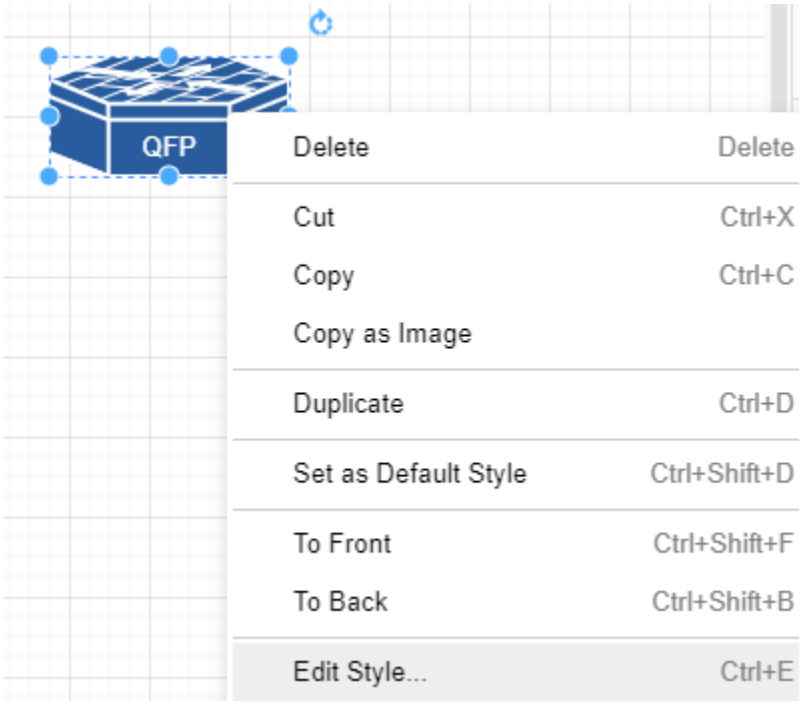
---

3.2.2 Adding styles

Styles used to change the way how things look and can be applied to nodes or links. Styles attributes in DrawIO encoded using strings similar to this one:

```
shape=mxgraph.cisco.misc.asr_1000_series;html=1;pointerEvents=1;dashed=0;fillColor=
↪#036897;strokeColor=#ffffff;strokeWidth=2;verticalLabelPosition=bottom;
↪verticalAlign=top;align=center;outlineConnect=0;
```

above strings can be found in node and link settings:



and can be used to reference by node and links style `attribute`, additionally, style string can be saved in text file and `style` attribute can reference that file OS path location.

```
from N2G import drawio_diagram

new_link_style="endArrow=classic;fillColor=#f8cecc;strokeColor=#FF3399;dashed=1;
↪edgeStyle=entityRelationEdgeStyle;startArrow=diamondThin;startFill=1;endFill=0;
↪strokeWidth=5;"
building_style="shape=mxgraph.cisco.buildings.generic_building;html=1;pointerEvents=1;
↪dashed=0;fillColor=#036897;strokeColor=#ffffff;strokeWidth=2;
↪verticalLabelPosition=bottom;verticalAlign=top;align=center;outlineConnect=0;"

diagram = drawio_diagram()
diagram.add_diagram("Page-1")
diagram.add_node(id="HQ", style=building_style, width=90, height=136)
diagram.add_node(id="R1", style="./styles/router.txt")
diagram.add_link("R1", "HQ", label="DF", style=new_link_style)
```

where *./styles/router.txt* content is:

```
shape=mxgraph.cisco.routers.atm_router;html=1;pointerEvents=1;dashed=0;fillColor=#036897;
↪strokeColor=#ffffff;strokeWidth=2;verticalLabelPosition=bottom;verticalAlign=top;
↪align=center;outlineConnect=0;
```

After opening and editing diagram, it might look like this:

Note: DrawIO does not encode node width and height attributes in style string, as a result width and height should be provided separately or will be set to default values: 120 and 60 pixels

3.2.3 Nodes and links data attributes

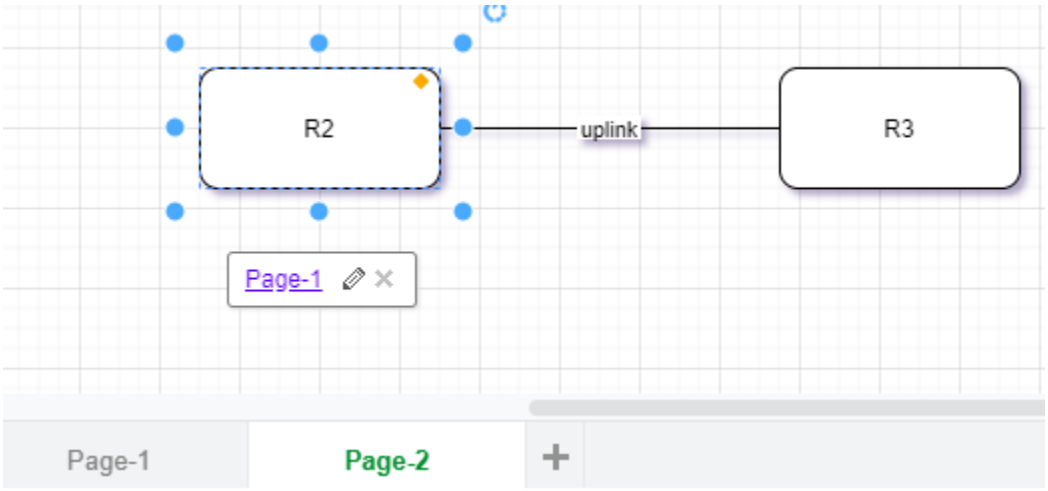
Data and URL attributes can be added to links and nodes to encode additional information. Data attribute should be a dictionary of key value pairs to add, where values can be of type string only.

URL attribute can point to WEB link or can reference another diagram/tab name.

```
from N2G import drawio_diagram

diagram = drawio_diagram()
diagram.add_diagram("Page-1")
diagram.add_node(id="R1", data={"a": "b", "c": "d"}, url="http://google.com")
diagram.add_diagram("Page-2")
diagram.add_node(id="R2", url="Page-1")
diagram.add_node(id="R3")
diagram.add_link("R2", "R3", label="uplink", data={"speed": "1G", "media": "10G-LR"},␣
↪url="http://cmdb.local")
diagram.dump_file(filename="Sample_graph.drawio", folder="./Output/")
```

After opening and editing diagram, it might look like this:



3.2.4 Loading graph from dictionary

Diagram elements can be loaded from dictionary structure. That dictionary may contain nodes, links and edges keys, these keys should contain list of dictionaries where each dictionary item will contain element attributes such as id, labels, data, url etc.

```
from N2G import drawio_diagram

diagram = drawio_diagram()
sample_graph={
'nodes': [
    {'id': 'a', 'style': './styles/router.txt', 'label': 'R1', 'width': 78, 'height': 53}
↪,
    {'id': 'R2', 'label':'CE12800'},
    {'id': 'c', 'label': 'R3', 'data': {'role': 'access', 'make': 'VendorX'}}
],
```

(continues on next page)

(continued from previous page)

```
'links': [
    {'source': 'a', 'label': 'DF', 'target': 'R2', 'data': {'role': 'uplink'}},
    {'source': 'R2', 'label': 'Copper', 'target': 'c'},
    {'source': 'c', 'label': 'ZR', 'target': 'a'}
]}
diagram.from_dict(sample_graph, width=300, height=200, diagram_name="Page-2")
diagram.layout(algo="kk")
diagram.dump_file(filename="Sample_graph.drawio", folder="./Output/")
```

After opening and editing diagram, it might look like this:

3.2.5 Loading graph from list

From list method allows to load graph from list of dictionaries, generally containing link details like source, target, label. Additionally source and target can be defined using dictionaries as well, containing nodes details.

Note: Non-existing node will be automatically added on first encounter, by default later occurrences of same node will not lead to node attributes change, that behavior can be changed setting `node_duplicates` drawio_diagram attribute equal to *update* value.

```
from N2G import drawio_diagram

diagram = drawio_diagram()
sample_list_graph = [
    {'source': {'id': 'SW1'}, 'target': 'R1', 'label': 'Gig0/1--Gi2'},
    {'source': 'R2', 'target': 'SW1', "data": {"speed": "1G", "media": "10G-LR"}},
    {'source': {'id':'a', 'label': 'R3'}, 'target': 'SW1'},
    {'source': 'SW1', 'target': 'R4'}
]
diagram.from_list(sample_list_graph, width=300, height=200, diagram_name="Page-2")
diagram.layout(algo="kk")
diagram.dump_file(filename="Sample_graph.drawio", folder="./Output/")
```

After opening and editing diagram, it might look like this:

3.2.6 Loading graph from csv

Similar to `from_dict` or `from_list`, from_csv method can take csv data with elements details and add them to diagram. Two types of csv table should be provided - one for nodes, another for links.

```
from N2G import drawio_diagram

diagram = drawio_diagram()
csv_links_data = """source","label","target"
"a","DF","b"
"b","Copper","c"
"b","Copper","e"
d,FW,e
"""
```

(continues on next page)

(continued from previous page)

```
csv_nodes_data="""id","label","style","width","height"
a,"R12","./styles/router.txt",78,53
"b","R2",,,
"c","R3",,,
"d","SW22",,,
"e","R1",,,
"""
diagram.from_csv(csv_nodes_data)
diagram.from_csv(csv_links_data)
diagram.layout(algo="kk")
diagram.dump_file(filename="Sample_graph.drawio", folder="./Output/")
```

After opening and editing diagram, it might look like this:

3.2.7 Loading existing diagrams

N2G DrawIO module provides `from_file` and `from_text` methods to load existing diagram content, to load diagram from file one can use this as example:

```
from N2G import drawio_diagram

diagram = drawio_diagram()
diagram.from_file("./source/old_office_diagram.drawio")
```

After diagram loaded it can be modified or updated using `add_x`, `from_x`, `delete_x` or `compare` methods.

3.2.8 Diagram layout

To arrange diagram nodes in certain way one can use `layout` method that relies on igraph library to calculate node coordinates in accordance with certain algorithm. List of supported layout algorithms and their details can be found here together with brief description in *API Reference* section.

Sample code to layout diagram:

```
from N2G import drawio_diagram

diagram = drawio_diagram()
diagram.from_file("./source/old_office_diagram.graphml")
diagram.layout(algo="drl")
diagram.dump_file(filename="updated_office_diagram.graphml", folder="./Output/")
```

3.2.9 Comparing diagrams

Comparing diagrams can help to spot changes in your system. N2G `compare` method allow to calculate differences between old and new graphs and produce resulting diagram highlighting changes.

```
from N2G import drawio_diagram

diagram = drawio_diagram()
old_graph = {
```

(continues on next page)

(continued from previous page)

```
'nodes': [
    {'id': 'R1'}, {'id': 'R2'}, {'id': 'R3'},
    ],
'edges': [
    {'source': 'R1', 'target': 'R2'},
    {'source': 'R2', 'target': 'R3'},
    {'source': 'R3', 'target': 'R1'}
]}
new_graph = {
'nodes': [
    {'id': 'R1'}, {'id': 'R2'}, {'id': 'R4'},
    ],
'edges': [
    {'source': 'R1', 'target': 'R2'},
    {'source': 'R2', 'target': 'R4'}
]}
diagram.add_diagram("Page-1", width=500, height=500)
diagram.from_dict(old_graph)
diagram.compare(new_graph)
diagram.layout(algo="kk")
diagram.dump_file(filename="Sample_graph.drawio", folder="./Output/")
```

Original and after diagrams combined:

R3 and its links are missing - highlighted in gray, but R4 and its link is new - highlighted in green.

3.2.10 API reference

API reference for N2G DrawIO module.

class N2G.plugins.diagrams.N2G_DrawIO.**drawio_diagram**(*node_duplicates='skip'*, *link_duplicates='skip'*)

N2G DrawIO module allows to produce diagrams compatible with DrawIO XML format.

Parameters

- `node_duplicates` (str) can be of value skip, log, update
- `link_duplicates` (str) can be of value skip, log, update

add_diagram(*id*, *name=''*, *width=1360*, *height=864*)

Method to add new diagram tab and switch to it.

> **Warning:** This method must be called to create at list one diagram tab to work with prior to nodes and links can be added to the drawing calling `add_link` or `add_node` methods.

Parameters

- `id` (str) id of the diagram, should be unique across other diagrams
- `name` (str) tab name
- `width` (int) width of diagram in pixels
- `height` (int) height of diagram in pixels

add_link(*source*, *target*, *style=''*, *label=''*, *data=None*, *url=''*, *src_label=''*, *trgt_label=''*, *src_label_style=''*, *trgt_label_style=''*, *link_id=None*, ***kwargs*)

Method to add link between nodes to the diagram.

Parameters

- `source` (str) mandatory, source node id
- `source` (str) mandatory, target node id
- `label` (str) link label to display at the center of the link
- `data` (dict) dictionary of key value pairs to add as link data
- `url` (str) url string to save as link `url` attribute
- `style` (str) string or OS path to text file with style to apply to the link
- `src_label` (str) link label to display next to source node
- `trgt_label` (str) link label to display next to target node
- `src_label_style` (str) source label style string
- `trgt_label_style` (str) target label style string
- `link_id` (str or int) optional link id value, must be unique across all links

Sample DrawIO style string for the link:

```
endArrow=classic;fillColor=#f8cecc;strokeColor=#FF3399;dashed=1;
edgeStyle=entityRelationEdgeStyle;startArrow=diamondThin;startFill=1;
endFill=0;strokeWidth=5;
```

Note: If source or target nodes does not exists, they will be automatically created

All labels are optional and substituted with empty values to calculate link id.

add_node(*id*, *label=''*, *data=None*, *url=''*, *style=''*, *width=120*, *height=60*, *x_pos=200*, *y_pos=150*, ***kwargs*)

Method to add node to the diagram.

Parameters

- `id` (str) mandatory, unique node identifier, usually equal to node name
- `label` (str) node label, if not provided, set equal to id
- `data` (dict) dictionary of key value pairs to add as node data
- `url` (str) url string to save as node `url` attribute
- `width` (int) node width in pixels
- `height` (int) node height in pixels
- `x_pos` (int) node position on x axis
- `y_pos` (int) node position on y axis
- `style` (str) string containing DrawIO style parameters to apply to the node

Sample DrawIO style string for the node:

```
shape=mxgraph.cisco.misc.asr_1000_series;html=1;pointerEvents=1;
dashed=0;fillColor=#036897;strokeColor=#ffffff;strokeWidth=2;
verticalLabelPosition=bottom;verticalAlign=top;align=center;
outlineConnect=0;
```

compare(*data*, *diagram_name=None*, *missing_colour='#C0C0C0'*, *new_colour='#00FF00'*)

Method to combine two graphs - existing and new - and produce resulting graph following these rules:

- nodes and links present in new graph but not in existing graph considered as new and will be updated with `new_colour` style attribute by default highlighting them in green
- nodes and links missing from new graph but present in existing graph considered as missing and will be updated with `missing_colour` style attribute by default highlighting them in gray
- nodes and links present in both graphs will remain unchanged

Parameters

- `data` (dict) dictionary containing new graph data, dictionary format should be the same as for `from_dict` method.
- `missing_colour` (str) colour to apply to missing elements
- `new_colour` (str) colour to apply to new elements

Sample usage:

```
from N2G import drawio_diagram
existing_graph = {
    "nodes": [
        {"id": "node-1"},
        {"id": "node-2"},
        {"id": "node-3"}
],
    "links": [
        {"source": "node-1", "target": "node-2", "label": "bla1"},
        {"source": "node-2", "target": "node-3", "label": "bla2"},
    ]
}
new_graph = {
    "nodes": [
        {"id": "node-99"},
        {"id": "node-100", "style": "./Pics/router_1.txt", "width": 78, "height
↪": 53},
    ],
    "links": [
        {"source": "node-2", "target": "node-3", "label": "bla2"},
        {"source": "node-99", "target": "node-3", "label": "bla99"},
        {"source": "node-100", "target": "node-99", "label": "bla10099"},
    ]
}
drawing = drawio_diagram()
drawing.from_dict(data=existing_graph)
drawing.compare(new_graph)
drawing.layout(algo="kk")
drawing.dump_file(filename="compared_graph.drawio")
```

delete_link(*id=None*, *ids=None*, *label=''*, *source=''*, *target=''*, *src_label=''*, *trgt_label=''*)
Method to delete link by its id. Bulk delete operation supported by providing list of link ids to delete.

If link `id` or `ids` not provided, id calculated based on - `label, source, target` - attributes using this algorithm:

1. Edge tuple produced: `tuple(sorted([label, source, target]))`
2. MD5 hash derived from tuple: `hashlib.md5(",".join(edge_tup).encode()).hexdigest()`

Parameters

- `id` (str) id of single link to delete
- `ids` (list) list of link ids to delete
- `label` (str) link label to calculate id of single link to delete
- `source` (str) link source to calculate id of single link to delete
- `target` (str) link target to calculate id of single link to delete
- `src_label` (str) link source label to calculate id of single link to delete
- `trgt_label` (str) link target label to calculate id of single link to delete

delete_node(*id=None*, *ids=None*)
Method to delete node by its id. Bulk delete operation supported by providing list of node ids to delete.

Parameters

- `id` (str) id of single node to delete
- `ids` (list) list of node ids to delete

dump_file(*filename=None*, *folder='./Output/'*)
Method to save current diagram in .drawio file.

Parameters

- `filename` (str) name of the file to save diagram into
- `folder` (str) OS path to folder where to save diagram file

If no `filename` provided, timestamped format will be used to produce filename, e.g.: `Sun Jun 28 20-30-57 2020_output.drawio`

dump_xml()
Method to return current diagram XML text

from_csv(*text_data*)
Method to build graph from CSV tables

Parameters

- `text_data` (str) CSV text with links or nodes details

This method supports loading CSV text data that contains nodes or links information. If `id` in headers, `from_dict` method will be called for CSV processing, `from_list` method will be used otherwise.

CSV data with nodes details should have headers matching `add_node` method arguments and rules.

CSV data with links details should have headers matching `add_link` method arguments and rules.

Sample CSV table with links details:

```
"source","label","target","src_label","trgt_label"
"a","DF","b","Gi1/1","Gi2/2"
"b","Copper","c","Te2/1",
"b","Copper","e","","GE3"
"d","FW","e",,
```

Sample CSV table with nodes details:

```
"id","label","style","width","height"
a,"R1,2","./Pics/cisco_router.txt",78,53
"b","some",,,
"c","somelabel",,,
"d","somelabel1",,,
"e","R1",,,
```

from_dict(*data*, *diagram_name='Page-1'*, *width=1360*, *height=864*)

Method to build graph from dictionary.

Parameters

- `diagram_name` (str) name of the diagram tab where to add links and nodes. Diagram tab will be created if it does not exists
- `width` (int) diagram width in pixels
- `height` (int) diagram height in pixels
- `data` (dict) dictionary with nodes and link/edges details, example:

```
sample_graph = {
    'nodes': [
        {
            'id': 'a',
            'label': 'R1'
        },
        {
            'id': 'b',
            'label': 'somelabel',
            'data': {'description': 'some node description'}
        },
        {
            'id': 'e',
            'label': 'E'
        }
    ],
    'edges': [
        {
            'source': 'a',
            'label': 'DF',
            'src_label': 'Gi1/1',
            'trgt_label': 'Gi2/2',
            'target': 'b',
            'url': 'google.com'
        }
    ],
```

(continues on next page)

(continued from previous page)

```
    'links': [
        {
            'source': 'a',
            'target': 'e'
        }
    ]
}
```

Dictionary Content Rules

- dictionary may contain `nodes` key with a list of nodes dictionaries
- each node dictionary must contain unique `id` attribute, other attributes are optional
- dictionary may contain `edges` or `links` key with a list of edges dictionaries
- each link dictionary must contain `source` and `target` attributes, other attributes are optional

from_file(*filename*, *file_load='xml'*)

Method to load nodes and links from Drawio diagram file for further processing

Args

- filename - OS path to .drawio file to load

from_list(*data*, *diagram_name='Page-1'*, *width=1360*, *height=864*)

Method to build graph from list.

Parameters

- `diagram_name` (str) name of the diagram tab where to add links and nodes. Diagram tab will be created if it does not exists
- `width` (int) diagram width in pixels
- `height` (int) diagram height in pixels
- `data` (list) list of link dictionaries, example:

```
sample_graph = [
    {
        'source': 'a',
        'label': 'DF',
        'src_label': 'Gi1/1',
        'trgt_label': 'Gi2/2',
        'target': 'b',
        'data': {'vlans': 'vlans_trunked: 1,2,3\nstate: up'}
    },
    {
        'source': 'a',
        'target': {
                'id': 'e',
                'label': 'somelabel',
                'data': {'description': 'some node description'}
            }
        }
    }
]
```

List Content Rules

- each list item must have `target` and `source` attributes defined
- `target`/`source` attributes can be either a string or a dictionary
- dictionary `target`/`source` node must contain `id` attribute and other supported node attributes

---

Note: By default drawio_diagram object `node_duplicates` action set to 'skip' meaning that node will be added on first occurrence and ignored after that. Set `node_duplicates` to 'update' if node with given id need to be updated by later occurrences in the list.

`from_xml`(*text_data*)
Method to load graph from .drawio XML text produced by DrawIO

Args

- text_data - text data to load

`go_to_diagram`(*diagram_name=None*, *diagram_index=None*)
DrawIO supports adding multiple diagram tabs within single document. This method allows to switch between diagrams in different tabs. That way each tab can be updated separately.

Parameters

- `diagram_name` (str) name of diagram tab to switch to
- `diagram_index` (int) index of diagram tab to switch to, will change to last tab if index is out of range. Index can be positive or negative number and follows Python list index behavior. For instance, index equal to "-1" we go to last tab, "0" will go to first tab

`layout`(*algo='kk'*, *ig_kwargs=None*, ***kwargs*)
Method to calculate graph layout using Python igraph library

Parameters

- `algo` (str) name of layout algorithm to use, default is 'kk'. Reference *Layout algorithms* table below for valid algo names
- `ig_kwargs` (dict) arguments to use to instantiate igraph's Graph instance as per documentation
- `kwargs` any additional kwargs to pass to igraph `Graph.layout` method as per documentation

Layout algorithms

algo name	description
circle, circular	Deterministic layout that places the vertices on a circle
drl	The Distributed Recursive Layout algorithm for large graphs
fr	Fruchterman-Reingold force-directed algorithm
fr3d, fr_3d	Fruchterman-Reingold force-directed algorithm in three dimensions
grid_fr	Fruchterman-Reingold force-directed algorithm with grid heuristics for large graphs
kk	Kamada-Kawai force-directed algorithm
kk3d, kk_3d	Kamada-Kawai force-directed algorithm in three dimensions
large, lgl, large_graph	The Large Graph Layout algorithm for large graphs
random	Places the vertices completely randomly
random_3d	Places the vertices completely randomly in 3D
rt, tree	Reingold-Tilford tree layout, useful for (almost) tree-like graphs
rt_circular, tree	Reingold-Tilford tree layout with a polar coordinate post-transformation, useful for (almost) tree-like graphs
sphere, spherical, circular_3d	Deterministic layout that places the vertices evenly on the surface of a sphere

update_link(*edge_id=''*, *label=''*, *source=''*, *target=''*, *data=None*, *url=''*, *style=''*, *src_label=''*, *trgt_label=''*, *new_label=None*, *new_src_label=None*, *new_trgt_label=None*, *src_label_style=''*, *trgt_label_style=''*)

Method to update edge/link details.

Parameters

- `edge_id` (str) - md5 hash edge id, if not provided, will be generated based on link attributes
- `label` (str) - existing edge label
- `src_label` (str) - existing edge source label
- `trgt_label` (str) - existing edge target label
- `source` (str) - existing edge source node id
- `target` (str) - existing edge target node id
- `new_label` (str) - new edge label
- `data` (dict) - edge new data attributes
- `url` (str) - edge new url attribute
- `style` (str) - OS path to file or sting containing style to apply to edge
- `new_src_label` (str) - new edge source label`
- `new_trgt_label` (str) - new edge target label
- `src_label_style` (str) - string with style to apply to source label
- `trgt_label_style` (str) - strung with style to apply to target label

Either of these must be provided to find link element to update:

- `edge_id` MD5 hash or
- `label, source, target, src_label, trgt_label` existing link attributes to calculate `edge_id`

`edge_id` calculated based on - `label, source, target, src_label, trgt_label` - attributes following this algorithm:

1. Edge tuple produced: `tuple(sorted([label, source, target, src_label, trgt_label]))`
2. MD5 hash derived from tuple: `hashlib.md5(",".join(edge_tup).encode()).hexdigest()`

If no `label, src_label, trgt_label` provided, they substituted with empty values in assumption that values for existing link are empty as well.

This method will replace existing or add new labels to the link.

Existing data attribute will be amended with new values using dictionary like update method.

New style will replace existing style.

update_node(*id*, *label=None*, *data=None*, *url=None*, *style=''*, *width=''*, *height=''*, ***kwargs*)

Method to update node details. Uses node `id` to search for node to update

Parameters

- `id` (str) mandatory, unique node identifier
- `label` (str) label at the center of the node
- `data` (dict) dictionary of data items to add to the node
- `width` (int) node width in pixels
- `height` (int) node height in pixels
- `url` (str) url string to save as node *url* attribute
- `style` (str) string containing DrawIO style parameters to apply to the node

3.3 V3D Diagram Plugin

Module to produce JSON structure compatible with 3D Force-Directed Graph library JSON input syntax

Why? Because network 3D visualisation is awesome. However, author is not aware of complete application that is capable of displaying produced results utilizing 3D Force-Directed Graph library. There is an attempt to make such an application described in *Built-in Diagram Viewer* section, but it is very (very) far from being perfect. Hence, if you are aware of better option to visualize data compatible with JSON input syntax please let the author know about it.

3.3.1 Quick start

Nodes and links can be added one by one using `add_node` and `add_link` methods

```
from N2G import v3d_diagramm as create_v3d_diagram

v3d_drawing = create_v3d_diagram()
v3d_drawing.add_node(id="node-1")
v3d_drawing.add_node(id="node-2")
v3d_drawing.add_link("node-1", "node-2", label="link 1")
v3d_drawing.dump_file()
```

After opening and editing produced JSON text file, it might look like this:

```
{
    "nodes": [
        {
            "id": "node-1",
            "label": "node-1",
            "color": "green",
            "nodeResolution": 8,
            "data": {}
        },
        {
            "id": "node-2",
            "label": "node-2",
            "color": "green",
            "nodeResolution": 8,
            "data": {}
        }
    ],
    "links": [
        {
            "id": "b35ebf8a6eeb7084dd9f3e14ec85eb9c",
            "label": "bla1",
            "source": "node-1",
            "target": "node-2",
            "src_label": "",
            "trgt_label": "",
            "data": {}
        }
    ]
}
```

3.3.2 Nodes and links default attributes

Node dictionaries have these attributes added by default:

- `id` - node unique identifier
- `label` - node label to display
- `color` - node color, default is green
- `nodeResolution` - how smooth node sphere is, default value is 8
- `data` - data dictionary

Link dictionaries have these attributes added by default:

- `source` - source node id
- `target` - target node id
- `id` - link unique identifier, calculated automatically if not supplied
- `label` - link label
- `data` - data dictionary
- `src_label` - link label to use next to source node

- `trgt_label` - link label to use next to target node

3.3.3 Loading graph from dictionary

Graph can be loaded from dictionary data using `from_dict` method, sample code:

```
from N2G import v3d_diagramm as create_v3d_diagram

sample_data = {
    'links': [{'data': {}, 'label': 'bla1', 'source': 'node-1', 'src_label': '', 'target
↪': 'node-2', 'trgt_label': ''},
              {'data': {}, 'label': 'bla2', 'source': 'node-1', 'src_label': '', 'target
↪': 'node-3', 'trgt_label': ''},
              {'data': {}, 'label': 'bla3', 'source': 'node-3', 'src_label': '', 'target
↪': 'node-5', 'trgt_label': ''},
              {'data': {}, 'label': 'bla4', 'source': 'node-3', 'src_label': '', 'target
↪': 'node-4', 'trgt_label': ''},
              {'data': {}, 'label': 'bla77', 'source': 'node-33', 'src_label': '',
↪'target': 'node-44', 'trgt_label': ''},
              {'data': {'cd': 123, 'ef': 456}, 'label': 'bla6', 'source': 'node-6', 'src_
↪label': '', 'target': 'node-1', 'trgt_label': ''}],
    'nodes': [{'color': 'green', 'data': {}, 'id': 'node-1', 'label': 'node-1',
↪'nodeResolution': 16},
              {'color': 'green', 'data': {}, 'id': 'node-2', 'label': 'node-2',
↪'nodeResolution': 8},
              {'color': 'blue', 'data': {'val': 4}, 'id': 'node-3', 'label': 'node-3',
↪'nodeResolution': 8},
              {'color': 'green', 'data': {}, 'id': 'node-4', 'label': 'node-4',
↪'nodeResolution': 8},
              {'color': 'green', 'data': {}, 'id': 'node-5', 'label': 'node-5',
↪'nodeResolution': 8},
              {'color': 'green', 'data': {'a': 'b', 'c': 'd'}, 'id': 'node-6', 'label':
↪'node-6', 'nodeResolution': 8},
              {'color': 'green', 'data': {}, 'id': 'node-33', 'label': 'node-33',
↪'nodeResolution': 8},
              {'color': 'green', 'data': {}, 'id': 'node-44', 'label': 'node-44',
↪'nodeResolution': 8},
              {'color': 'green', 'data': {}, 'id': 'node-25', 'label': 'node-25',
↪'nodeResolution': 8}]
}

v3d_drawing = create_v3d_diagram()
v3d_drawing.from_dict(sample_data)
v3d_drawing.dump_file()
```

3.3.4 Loading graph from list

Graph can be loaded from list data using `from_list` method, sample code:

```
from N2G import v3d_diagramm as create_v3d_diagram

sample_data_list = [
    {'data': {}, 'label': 'bla1', 'source': {'id': 'node-1', 'nodeResolution': 16}, 'src_
↪label': '', 'target': {'id': 'node-2'}, 'trgt_label': ''},
    {'data': {}, 'label': 'bla2', 'source': 'node-1', 'src_label': '', 'target': 'node-3
↪', 'trgt_label': ''},
    {'data': {}, 'label': 'bla3', 'source': {'id': 'node-3'}, 'src_label': '', 'target':
↪'node-5', 'trgt_label': ''},
    {'data': {}, 'label': 'bla4', 'source': {'id': 'node-3', 'data': {'val': 4}}, 'src_
↪label': '', 'target': 'node-4', 'trgt_label': ''},
    {'data': {}, 'label': 'bla77', 'source': 'node-33', 'src_label': '', 'target': 'node-
↪44', 'trgt_label': ''},
    {'data': {'cd': 123, 'ef': 456}, 'label': 'bla6', 'source': {'id': 'node-6', 'data':
↪{'a': 'b', 'c': 'd'}}, 'src_label': '', 'target': 'node-1', 'trgt_label': ''}
]

v3d_drawing = create_v3d_diagram()
v3d_drawing.from_list(sample_data_list)
v3d_drawing.dump_file()
```

3.3.5 Loading existing diagrams

Existing JSON input syntax data can be loaded into V3D plugin for processing and manipulation using sample code:

```
from N2G import v3d_diagramm as create_v3d_diagram

data = '''{
    "nodes": [
        {
        "id": "id1",
        "name": "name1",
        "val": 1
        },
        {
        "id": "id2",
        "name": "name2",
        "val": 10
        }
    ],
    "links": [
        {
            "source": "id1",
            "target": "id2"
        }
    ]
}'''
```

(continues on next page)

(continued from previous page)

```
v3d_drawing = create_v3d_diagram()
v3d_drawing.from_v3d_json(data)
```

3.3.6 Diagram layout

To arrange diagram nodes in certain way one can use `layout` method that relies on igraph library to calculate node coordinates in accordance with certain algorithm. List of supported layout algorithms and their details can be found here together with brief description in *API Reference* section.

Sample code to layout diagram:

```
from N2G import v3d_diagramm as create_v3d_diagram

sample_data_list = [
    {'data': {}, 'label': 'bla1', 'source': {'id': 'node-1', 'nodeResolution': 16}, 'src_
↪label': '', 'target': {'id': 'node-2'}, 'trgt_label': ''},
    {'data': {}, 'label': 'bla2', 'source': 'node-1', 'src_label': '', 'target': 'node-3
↪', 'trgt_label': ''},
    {'data': {}, 'label': 'bla3', 'source': {'id': 'node-3'}, 'src_label': '', 'target':
↪'node-5', 'trgt_label': ''},
    {'data': {}, 'label': 'bla4', 'source': {'id': 'node-3', 'data': {'val': 4}}, 'src_
↪label': '', 'target': 'node-4', 'trgt_label': ''},
    {'data': {}, 'label': 'bla77', 'source': 'node-33', 'src_label': '', 'target': 'node-
↪44', 'trgt_label': ''},
    {'data': {'cd': 123, 'ef': 456}, 'label': 'bla6', 'source': {'id': 'node-6', 'data':
↪{'a': 'b', 'c': 'd'}}, 'src_label': '', 'target': 'node-1', 'trgt_label': ''}
]

v3d_drawing = create_v3d_diagram()
v3d_drawing.from_list(sample_data_list)
v3d_drawing.layout(algo='kk3d', dx=200, dy=200, dz=200)
```

Where `dx, dy and dz` help to set diagram 3d size.

3.3.7 Built-in Diagram Viewer

V3D plugin comes with simple 3d diagram viewer for the purpose of demonstration and to explore produced diagram.

Built in WEB server uses Flask in debug mode, hence not suitable for production use.

To install Flask WEB framework - `pip install Flask`

Sample code to run built-in WEB server:

```
from N2G import v3d_diagramm as create_v3d_diagram

sample_data_list = [
    {'data': {}, 'label': 'bla1', 'source': {'id': 'node-1', 'nodeResolution': 16}, 'src_
↪label': '', 'target': {'id': 'node-2'}, 'trgt_label': ''},
    {'data': {}, 'label': 'bla2', 'source': 'node-1', 'src_label': '', 'target': 'node-3
↪', 'trgt_label': ''},
    {'data': {}, 'label': 'bla3', 'source': {'id': 'node-3'}, 'src_label': '', 'target':
↪'node-5', 'trgt_label': ''},
```

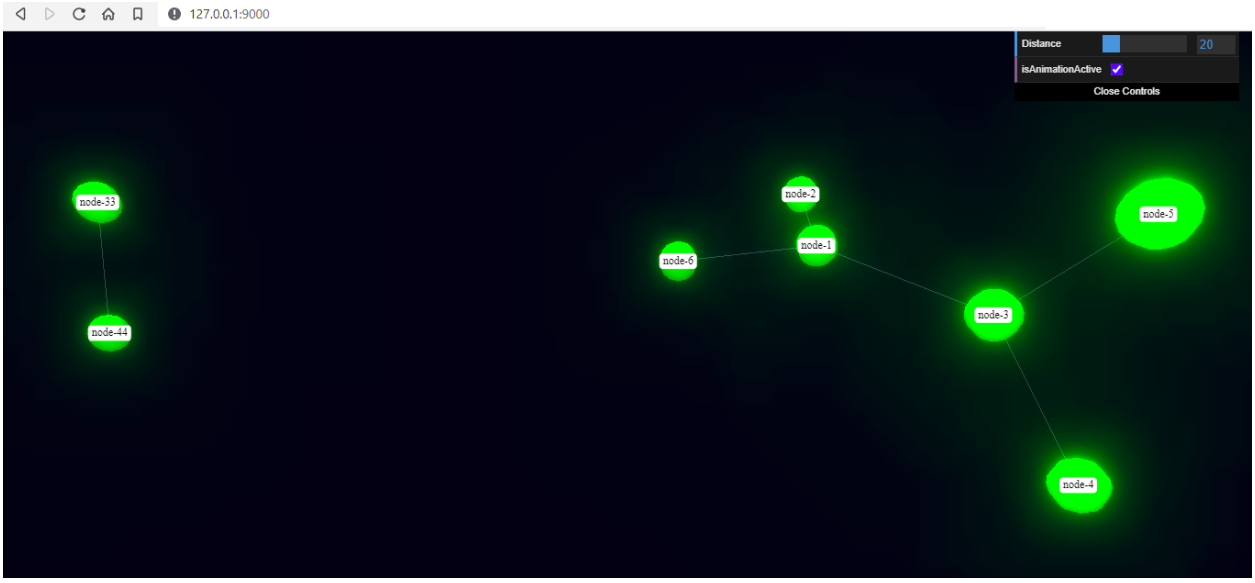
(continues on next page)

(continued from previous page)

```
    {'data': {}, 'label': 'bla4', 'source': {'id': 'node-3', 'data': {'val': 4}}, 'src_
↪label': '', 'target': 'node-4', 'trgt_label': ''},
    {'data': {}, 'label': 'bla77', 'source': 'node-33', 'src_label': '', 'target': 'node-
↪44', 'trgt_label': ''},
    {'data': {'cd': 123, 'ef': 456}, 'label': 'bla6', 'source': {'id': 'node-6', 'data':
↪{'a': 'b', 'c': 'd'}}, 'src_label': '', 'target': 'node-1', 'trgt_label': ''}
]

v3d_drawing = create_v3d_diagram()
v3d_drawing.from_list(sample_data_list)
v3d_drawing.run(ip="0.0.0.0", "port"=9000)
```

If all good, browsing to `http://127.0.0.1:9000` URL should load similar to below 3D diagram:



3.3.8 API reference

API reference for N2G V3D module.

class N2G.plugins.diagrams.N2G_V3D.**v3d_diagramm**(*node_duplicates='skip'*, *link_duplicates='skip'*)

Class to produce JSON data structure compatible with 3D Force-Directed Graph library JSON input syntax

Parameters

- **node_duplicates** – (str) what to do with node duplicates - `skip` (default), `update` or `log`
- **link_duplicates** – (str) what to do with link duplicates - `skip` (default), `update` or `log`

add_link(*source*, *target*, *label=''*, *src_label=''*, *trgt_label=''*, *data=None*, *id=None*, ***kwargs*)

Method to add link between nodes.

Parameters

- **source** – (str) mandatory, source node id
- **target** – (str) mandatory, target node id
- **label** – (str) link label to display at the center of the link

- **data** – (dict) dictionary of key value pairs to add as link data
- **src_label** – (str) link label to use next to source node
- **trgt_label** – (str) link label to use next to target node
- **id** – (str) explicit link identifier to use
- **kwargs** – (dict) any additional kwargs to add to link dictionary

---

Note: If source or target nodes does not exists, they will be automatically created

---

All labels are optional and substituted with empty values to calculate link id.

By default V3D uses below code to produce MD5 hash digest for link id:

```
link_tup = tuple(sorted([label, source, target, src_label, trgt_label]))
link_id = hashlib.md5(",".join(edge_tup).encode()).hexdigest()
```

add_node(*id*, *label=''*, *data=None*, *color='green'*, *nodeResolution=8*, ***kwargs*)
Method to add node to the diagram.

Parameters

- **id** – (str) mandatory, unique node identifier, usually equal to node name
- **label** – (str) node label, if not provided, set equal to id
- **data** – (dict) dictionary of key value pairs to add as node data
- **fx** – (int) node position on x axis
- **fy** – (int) node position on y axis
- **fz** – (int) node position on z axis
- **color** – (str) node color e.g. `blue`, default is `green`
- **nodeResolution** – (int) geometric resolution of the node, expressed in how many slice segments to divide the circumference. Higher values yield smoother spheres.
- **kwargs** – (dict) any additional kwargs to add to node dictionary as per node styling attributes such as `nodeRelSize`, `nodeOpacity`, `nodeVal` etc.

delete_link(*source=None*, *target=None*, *label=''*, *src_label=''*, *trgt_label=''*, *id=None*)
Method to delete link. Uses link `id` to search for link to delete, if no `id` provided uses `source, target, label, src_label, trgt_label` to calculate edge id.

Parameters

- **source** – (str) source node id
- **target** – (str) target node id
- **label** – (str) existing link label
- **src_label** – (str) link source label
- **trgt_label** – (str) link target label
- **id** – (str) link identifier to find the link to delete

delete_node(*id*)
Method to delete node. Uses node `id` to search for node to delete.

Parameters **id** – (str) mandatory, unique node identifier

dump_dict()
Method to populate `self.drawing` dictionary with current links and nodes items, return `self.drawing` dictionary content after that.

dump_file(*filename=None*, *folder='./Output/'*, *json_kwargs=None*)
Method to save current diagram to text file in a JSON format.

Parameters

- **filename** – (str) name of the file to save diagram into
- **folder** – (str) OS path to folder where to save diagram file, default is `./Output/`
- **json_kwargs** – (dict) kwargs to use with `json.dumps` method

If no `filename` provided, timestamped format used to produce filename, e.g.: `Sun Jun 28 20-30-57 2020_output.txt`

dump_json(***kwargs*)
Method to transform graph data in a JSON formatted string.

Parameters **kwargs** – (dict) kwargs to use with `json.dumps` method

from_dict(*data*)
Method to build graph from dictionary.

Parameters **data** – (dict) dictionary with nodes and link/edges details

Sample data dictionary:

```
sample_graph = {
    'nodes': [
        {
            'id': 'a',
            'label': 'R1'
        },
        {
            'id': 'b',
            'label': 'somelabel',
            'data': {'description': 'some node description'}
        },
        {
            'id': 'e',
            'label': 'E'
        }
    ],
    'edges': [
        {
            'source': 'a',
            'label': 'DF',
            'src_label': 'Gi1/1',
            'trgt_label': 'Gi2/2',
            'target': 'b',
            'url': 'google.com'
        }
    ],
    'links': [
        {
            'source': 'a',
```

(continues on next page)

(continued from previous page)

```
            'target': 'e'
        }
    ]
}
```

Dictionary Content Rules

- dictionary may contain `nodes` key with a list of nodes dictionaries
- each node dictionary must contain unique `id` attribute, other attributes are optional
- dictionary may contain `edges` or `links` key with a list of edges dictionaries
- each link dictionary must contain `source` and `target` attributes, other attributes are optional

`from_list`(*data*)

Method to build graph from list.

Parameters `data` – (list) list of link dictionaries

Sample list data:

```
sample_graph = [
    {
        'source': 'a',
        'label': 'DF',
        'src_label': 'Gi1/1',
        'trgt_label': 'Gi2/2',
        'target': 'b',
        'data': {'vlans': 'vlans_trunked: 1,2,3\nstate: up'}
    },
    {
        'source': 'a',
        'target': {
                'id': 'e',
                'label': 'somelabel',
                'data': {'description': 'some node description'}
            }
        }
    }
]
```

List Content Rules

- each list item must have `target` and `source` attributes defined
- `target`/`source` attributes can be either a string or a dictionary
- dictionary `target`/`source` node must contain `id` attribute and other supported node attributes

---

Note: By default drawio_diagram object `node_duplicates` action set to 'skip' meaning that node will be added on first occurrence and ignored after that. Set `node_duplicates` to 'update' if node with given id need to be updated by later occurrences in the list.

`from_v3d_json`(*data*)

Method to load JSON input syntax data into diagram plugin, presumably to perform various manipulations.

Parameters **data** – (str) string of JSON input syntax format

layout(*algo='kk3d'*, *dx=100*, *dy=100*, *dz=100*, ***kwargs*)
Method to calculate graph layout using Python igraph library

Parameters

- **algo** – (str) name of igraph layout algorithm to use, default is 'kk3d'. Reference *Layout algorithms* table below for valid algo names
- **kwargs** – (dict) any additional kwargs to pass to igraph `Graph.layout` method

Layout algorithms

algo name	description
circle, circular	Deterministic layout that places the vertices on a circle
drl	The Distributed Recursive Layout algorithm for large graphs
fr	Fruchterman-Reingold force-directed algorithm
fr3d, fr_3d	Fruchterman-Reingold force-directed algorithm in three dimensions
grid_fr	Fruchterman-Reingold force-directed algorithm with grid heuristics for large graphs
kk	Kamada-Kawai force-directed algorithm
kk3d, kk_3d	Kamada-Kawai force-directed algorithm in three dimensions
large, lgl, large_graph	The Large Graph Layout algorithm for large graphs
random	Places the vertices completely randomly
random_3d	Places the vertices completely randomly in 3D
rt, tree	Reingold-Tilford tree layout, useful for (almost) tree-like graphs
rt_circular, tree	Reingold-Tilford tree layout with a polar coordinate post-transformation, useful for (almost) tree-like graphs
sphere, spherical, circular_3d	Deterministic layout that places the vertices evenly on the surface of a sphere

Note: if 2d layout algorithm called, z axis coordinate set to 0

run(*ip: str = '0.0.0.0'*, *port: int = 9000*, *debug: bool = True*, *dry_run: bool = False*) → None
Method to run FLASK web server using built-in browser app.

Parameters

- **ip** – IP address to bound WEB server to
- **port** – port number to run WEB server on
- **debug** – If True run Flask server in debug mode
- **dry_run** – (bool) if True, do not start, return status info instead

update_link(*source=None*, *target=None*, *label=''*, *src_label=''*, *trgt_label=''*, *new_label=''*, *new_src_label=''*, *new_trgt_label=''*, *data=None*, *url=''*, *id=None*, ***kwargs*)
Method to update link details. Uses link `id` to search for link to update, if no `id` provided uses `source, target, label, src_label, trgt_label` to calculate edge id.

Parameters

- **source** – (str) source node id
- **target** – (str) target node id

- **label** – (str) existing link label
- **src_label** – (str) existing link source label
- **trgt_label** – (str) existing link target label
- **new_label** – (str) new link label to replace existing label
- **new_src_label** – (str) new link source label to replace existing source label
- **new_trgt_label** – (str) new link target label to replace existing target label
- **data** – (dict) dictionary of key value pairs to update link data
- **url** – (str) url string to save as link `url` attribute
- **id** – (str) link identifier to find the link to update
- **kwargs** – (dict) any additional kwargs to update link dictionary

update_node(*id*, *data=None*, ***kwargs*)

Method to update node details. Uses node `id` to search for node to update

Parameters

- **id** – (str) mandatory, unique node identifier
- **data** – (dict) data argument/key dictionary content to update existing values
- **kwargs** – (dict) any additional arguments to update node dictionary

CHAPTER

FOUR

DATA PLUGINS

Data plugins take data of certain format as input and produce structured data that can serve as input for one of diagram plugins.

Data plugin device platform names correspond to Netmiko SSH device_type values.

4.1 CLI IP Data Plugin

This plugin populates diagram with IP related information, such as subnets and IP addresses.

IP data plugin mainly useful in networking domain, it can take show commands output from network devices, parse it with TTP templates in a structure that processed further to load into one of diagram plugin objects using `from_dict` method

4.1.1 Features Supported

Support matrix

Platform Name	IP/Subnets	ARP	interface config	links grouping	FHRP Protocols
cisco_ios	YES	YES	YES	YES	YES
cisco_xr	YES	—	YES	YES	YES
cisco_nxos	YES	YES	YES	YES	YES
huawei	YES	YES	YES	YES	YES
fortinet	YES	YES	YES	YES	—
arista_eos	YES	YES	YES	YES	—

4.1.2 Required Commands output

cisco_ios:

- `show running-config` or `show running-config | section interface` - mandatory output, used to parse interfaces IP addresses
- `show ip arp` and/or `show ip arp vrf xyz` - required by ARP visualization feature

cisco_xr:

- `show running-config` or `show running-config interface` - mandatory output, used to parse interfaces IP addresses
- `show arp` and/or `show arp vrf xyz/all` - required by ARP visualization feature

cisco_nxos:

- `show running-config` or `show running-config | section interface` - mandatory output, used to parse interfaces IP addresses
- `show ip arp` - required by ARP visualization feature

huawei:

- `display current-configuration interface` - mandatory output, used to parse interfaces IP addresses
- `display arp all` - required by ARP visualization feature

fortinet:

- `get system config` - mandatory output, used to parse interfaces IP addresses
- `get system arp` - required by ARP visualization feature

arista_eos:

- `show running-config` or `show running-config | section interface` - mandatory output, used to parse interfaces IP addresses
- `show ip arp` and/or `show ip arp vrf all` - required by ARP visualization feature

4.1.3 Sample Usage

Code to populate yEd diagram object with IP and subnet nodes using data dictionary:

```
data = {"huawei": ['''
<hua_sw1>dis current-configuration interface
#
interface Vlanif140
 ip binding vpn-instance VRF_MGMT
 ip address 10.1.1.2 255.255.255.0
 vrrp vrid 200 virtual-ip 10.1.1.1
#
interface Eth-Trunk5.123
 vlan-type dot1q 123
 description hua_sw2 BGP  peering
 ip binding vpn-instance VRF_MGMT
 ip address 10.0.0.1 255.255.255.252
 ipv6 address FD00:1::1/126
#
interface Eth-Trunk5.200
 vlan-type dot1q 200
 description hua_sw3 OSPF  peering
 ip address 192.168.2.2 255.255.255.252

<hua_sw1>dis arp all
10.1.1.2        a008-6fc1-1101        I         Vlanif140       VRF_MGMT
10.1.1.1        a008-6fc1-1102   0    D         Vlanif140       VRF_MGMT
10.1.1.3        a008-6fc1-1103   10   D/200     Vlanif140       VRF_MGMT
10.1.1.9        a008-6fc1-1104   10   D/200     Vlanif140       VRF_MGMT
10.0.0.2        a008-6fc1-1105   10   D/200     Eth-Trunk5.123  VRF_MGMT
    ''',
    '''
```

(continues on next page)

(continued from previous page)

```
<hua_sw2>dis current-configuration interface
#
interface Vlanif140
 ip binding vpn-instance VRF_MGMT
 ip address 10.1.1.3 255.255.255.0
 vrrp vrid 200 virtual-ip 10.1.1.1
#
interface Eth-Trunk5.123
 vlan-type dot1q 123
 description hua_sw1 BGP  peering
 ip binding vpn-instance VRF_MGMT
 ip address 10.0.0.2 255.255.255.252
 ipv6 address FD00:1::2/126
    ''',
    '''
<hua_sw3>dis current-configuration interface
#
interface Vlanif200
 ip binding vpn-instance VRF_CUST1
 ip address 192.168.1.1 255.255.255.0
#
interface Eth-Trunk5.200
 vlan-type dot1q 200
 description hua_sw1 OSPF  peering
 ip address 192.168.2.1 255.255.255.252

<hua_sw3>dis arp
192.168.1.1         a008-6fc1-1111       I      Vlanif200
192.168.1.10        a008-6fc1-1110   30  D/300  Vlanif200
    '''],
"cisco_nxos": ['''
switch_1# show run | sec interface
interface Vlan133
  description OOB
  vrf member MGMT_OOB
  ip address 10.133.137.2/24
  hsrp 133
    preempt
    ip 10.133.137.1
!
interface Vlan134
  description OOB-2
  vrf member MGMT_OOB
  ip address 10.134.137.2/24
  vrrpv3 1334 address-family ipv4
    address 10.134.137.1 primary
!
interface Vlan222
  description PTP OSPF Routing pat to  siwtch2
  ip address 10.222.137.1/30
!
interface Vlan223
```

(continues on next page)

(continued from previous page)

```
  description PTP OSPF Routing pat to siwtch3
  ip address 10.223.137.1/30

switch_1# show ip arp vrf all
10.133.137.2    -  d094.7890.1111  Vlan133
10.133.137.1    -  d094.7890.1111  Vlan133
10.133.137.30   -  d094.7890.1234  Vlan133
10.133.137.91   -  d094.7890.4321  Vlan133
10.134.137.1    -  d094.7890.1111  Vlan134
10.134.137.2    -  d094.7890.1111  Vlan134
10.134.137.3   90  d094.7890.2222  Vlan134
10.134.137.31  91  d094.7890.beef  Vlan134
10.134.137.81  81  d094.7890.feeb  Vlan134
10.222.137.2   21  d094.7890.2222  Vlan222
'''
}

drawing = create_yed_diagram()
drawer = cli_ip_data(drawing, add_arp=True, add_fhrp=True)
drawer.work(data)
drawer.drawing.dump_file(filename="ip_graph_dc_1.graphml", folder="./Output/")
```

4.1.4 API Reference

class N2G.plugins.data.cli_ip_data.**cli_ip_data**(*drawing*, *ttp_vars=None*, *group_links=False*, *add_arp=False*, *label_interface=False*, *label_vrf=False*, *collapse_ptp=True*, *add_fhrp=False*, *bottom_label_length=0*, *lbl_next_to_subnet=False*, *platforms=None*)

Class to instantiate IP Data Plugin.

Parameters

- **drawing** – (obj) - N2G drawing object instantiated using drawing module e.g. yed_diagram or drawio_diagram
- **ttp_vars** – (dict) - dictionary to use as TTP parser object template variables
- **platforms** – (list) - list of platform names to process e.g. `cisco_ios`, `cisco_xr` etc, default is `_all_`
- **group_links** – (bool) - if True, will group links between same nodes, default is False
- **add_arp** – (bool) - if True, will add IP nodes from ARP parsing results, default is False
- **label_interface** – (bool) - if True, will add interface name to the link's source and target labels, default is False
- **label_vrf** – (bool) - if True, will add VRF name to the link's source and target labels, default is False
- **collapse_ptp** – (bool) - if True (default) combines links for `/31` and `/30` IPv4 and `/127` IPv6 subnets into a single ink
- **add_fhrp** – (bool) - if True adds HSRP and VRRP IP addresses to the diagram, default is False

- **bottom_label_length** – (int) - bottom label length of interface description to use for subnet nodes, if False or 0, bottom label will not be set for subnet nodes
- **lbl_next_to_subnet** – (bool) - if True, put link `port:vrf:ip` label next to subnet node, default is False

work(*data*)

Method to parse text data and add nodes and links to drawing object.

Parameters **data** – (dict or str) dictionary or OS path string to directories with data files

If data is dictionary, keys must correspond to **Platform** column in *Features Supported* section table, values are lists of text items to process.

Data dictionary sample:

```
data = {
    "cisco_ios" : ["h1", "h2"],
    "cisco_xr": ["h3", "h4"],
    "cisco_nxos": ["h5", "h6"],
    ...etc...
}
```

Where `hX` devices show commands output.

If data is an OS path directory string, child directories' names must correspond to **Platform** column in *Features Supported* section table. Each child directory should contain text files with show commands output for each device, names of files are arbitrary, but output should contain device prompt to extract device hostname.

Directories structure sample:

```
├──folder_with_data
    ├──cisco_ios
        switch1.txt
        switch2.txt
    └──cisco_nxos
        nxos_switch_1.txt
        nxos_switch_2.txt
```

To point N2G to above location `data` attribute string can be `/var/data/n2g/folder_with_data/`

4.2 CLI ISIS LSDB Data Plugin

This module designed to process ISIS Link State Database (LSDB) of network devices CLI output and make diagram out of it.

Show commands output from devices parsed using TTP Templates into a dictionary structure.

After parsing, results processed further to form a dictionary of nodes and links keyed by unique nodes and links identifiers, dictionary values are nodes dictionaries and for links it is a list of dictionaries of links between pair of nodes. For nodes ISIS RID used as a unique ID, for links it is sorted tuple of `source`, `target` and `label` keys' values. This structure helps to eliminate duplicates.

Next step is post processing, such as packing links between nodes or IP lookups.

Last step is to populate N2G drawing with new nodes and links using `from_dict` method.

4.2.1 Features Supported

Support matrix

Platform Name	ISIS LSDB
cisco_ios	—
cisco_xr	YES
cisco_nxos	—
huawei	—

4.2.2 Required Commands output

cisco_xr:

- `show isis database verbose` - mandatory output, used to parse ISIS LSDB content

4.2.3 Sample usage

Code to populate yEd diagram object with ISIS LSDB sourced nodes and links:

```
from N2G import yed_diagram as create_yed_diagram
from N2G import cli_isis_data

isis_lsdb_data = {"cisco_xr": ['''
RP/0/RP0/CPU0:ROUTER-X1#show isis database verbose

IS-IS 1234 (Level-2) Link State Database
LSPID                 LSP Seq Num  LSP Checksum  LSP Holdtime/Rcvd  ATT/P/OL
ROUTER-X1.00-00     * 0x00000832   0x74bc        64943/*            0/0/0
Auth:           Algorithm HMAC-MD5, Length: 17
Area Address:   49.1234
NLPID:          0xcc
Router ID:      10.211.1.1
Hostname:       ROUTER-X1
Metric: 0          IP-Extended 10.211.1.1/32
    Prefix Attribute Flags: X:1 R:0 N:0 E:0 A:0
Metric: 16777214   IS-Extended ROUTER-X2.00
    Local Interface ID: 9, Remote Interface ID: 50
    Interface IP Address: 10.123.0.17
    Neighbor IP Address: 10.123.0.18
    Affinity: 0x00000000
    Physical BW: 10000000 kbits/sec
    Reservable Global pool BW: 0 kbits/sec
    Global Pool BW Unreserved:
    [0]: 0        kbits/sec          [1]: 0        kbits/sec
    [2]: 0        kbits/sec          [3]: 0        kbits/sec
    [4]: 0        kbits/sec          [5]: 0        kbits/sec
    [6]: 0        kbits/sec          [7]: 0        kbits/sec
    Admin. Weight: 1000
    Ext Admin Group: Length: 32
    0x00000000   0x00000000
```

(continues on next page)

(continued from previous page)

```
    0x00000000   0x00000000
    0x00000000   0x00000000
    0x00000000   0x00000000
    Physical BW: 10000000 kbits/sec
Metric: 802        IS-Extended ROUTER-X5.00
    Local Interface ID: 7, Remote Interface ID: 53
    Interface IP Address: 10.123.0.25
    Neighbor IP Address: 10.123.0.26
    Affinity: 0x00000000
    Physical BW: 10000000 kbits/sec
    Reservable Global pool BW: 0 kbits/sec
    Global Pool BW Unreserved:
    [0]: 0        kbits/sec          [1]: 0        kbits/sec
    [2]: 0        kbits/sec          [3]: 0        kbits/sec
    [4]: 0        kbits/sec          [5]: 0        kbits/sec
    [6]: 0        kbits/sec          [7]: 0        kbits/sec
    Admin. Weight: 802
    Ext Admin Group: Length: 32
    0x00000000   0x00000000
    0x00000000   0x00000000
    0x00000000   0x00000000
    0x00000000   0x00000000
    Physical BW: 10000000 kbits/sec
ROUTER-X2.00-00        0x00000826   0x4390        65258/65535        0/0/0
Auth:           Algorithm HMAC-MD5, Length: 17
Area Address:   49.1234
NLPID:          0xcc
Router ID:      10.211.1.2
Hostname:       ROUTER-X2
Metric: 0          IP-Extended 10.211.1.2/32
    Prefix Attribute Flags: X:1 R:0 N:0 E:0 A:0
Metric: 301        IS-Extended ROUTER-X6.00
    Local Interface ID: 48, Remote Interface ID: 53
    Interface IP Address: 10.123.0.33
    Neighbor IP Address: 10.123.0.34
    Affinity: 0x00000000
    Physical BW: 10000000 kbits/sec
    Reservable Global pool BW: 0 kbits/sec
    Global Pool BW Unreserved:
    [0]: 0        kbits/sec          [1]: 0        kbits/sec
    [2]: 0        kbits/sec          [3]: 0        kbits/sec
    [4]: 0        kbits/sec          [5]: 0        kbits/sec
    [6]: 0        kbits/sec          [7]: 0        kbits/sec
    Admin. Weight: 301
    Ext Admin Group: Length: 32
    0x00000000   0x00000000
    0x00000000   0x00000000
    0x00000000   0x00000000
    0x00000000   0x00000000
    Physical BW: 10000000 kbits/sec
Metric: 16777214   IS-Extended ROUTER-X1.00
    Local Interface ID: 50, Remote Interface ID: 9
```

(continues on next page)

(continued from previous page)

```
    Interface IP Address: 10.123.0.18
    Neighbor IP Address: 10.123.0.17
    Affinity: 0x00000000
    Physical BW: 10000000 kbits/sec
    Reservable Global pool BW: 0 kbits/sec
    Global Pool BW Unreserved:
    [0]: 0        kbits/sec          [1]: 0        kbits/sec
    [2]: 0        kbits/sec          [3]: 0        kbits/sec
    [4]: 0        kbits/sec          [5]: 0        kbits/sec
    [6]: 0        kbits/sec          [7]: 0        kbits/sec
    Admin. Weight: 1000
    Ext Admin Group: Length: 32
    0x00000000   0x00000000
    0x00000000   0x00000000
    0x00000000   0x00000000
    0x00000000   0x00000000
    Physical BW: 10000000 kbits/sec

Total Level-2 LSP count: 2     Local Level-2 LSP count: 1
RP/0/RP0/CPU0:ROUTER-X1#
    '''']
}

drawing = create_yed_diagram()
drawer = cli_isis_data(drawing)
drawer.work(isis_lsdb_data)
drawing.dump_file()
```

4.2.4 API Reference

class N2G.plugins.data.cli_isis_data.**cli_isis_data**(*drawing*, *ttp_vars: Optional[dict] = None*, *ip_lookup_data: Optional[dict] = None*, *add_connected: bool = False*, *ptp_filter: Optional[list] = None*, *add_data: bool = True*, *platforms: Optional[list] = None*)

Main class to instantiate ISIS LSDB Data Plugin object.

Parameters

- **drawing** – (obj) N2G Diagram object
- **ttp_vars** – (dict) Dictionary to use as vars attribute while instantiating TTP parser object
- **platforms** – (list) - list of platform names to process e.g. `cisco_ios`, `cisco_xr` etc, default is `_all_`
- **ip_lookup_data** – (dict or str) IP Lookup dictionary or OS path to CSV file
- **add_connected** – (bool) if True, will add connected subnets as nodes, default is False
- **ptp_filter** – (list) list of glob patterns to filter point-to-point links based on link IP
- **add_data** – (bool) if True (default) adds data information to nodes and links

`ip_lookup_data` dictionary must be keyed by ISSI RID IP address, with values being dictionary which must contain `hostname` key with optional additional keys to use for N2G diagram module node, e.g. `label`,

top_label, bottom_label, interface``etc. If ``ip_lookup_data is an OS path to CSV file, that file's first column header must be ip , file must contain hostname column, other columns values set to N2G diagram module node attributes, e.g. label, top_label, bottom_label, interface etc.

If lookup data contains interface key, it will be added to link label.

Sample ip_lookup_data dictionary:

```
{
    "1.1.1.1": {
        "hostname": "router-1",
        "bottom_label": "1 St address, City X",
        "interface": "Gi1"
    }
}
```

Sample ip_lookup_data CSV file:

```
ip,hostname,bottom_label,interface
1.1.1.1,router-1,"1 St address, City X",Gi1
```

work(*data*)

Method to parse text data and add nodes and links to N2G drawing.

Parameters **data** – (dict or str) dictionary keyed by platform name or OS path string to directories with text files

If data is dictionary, keys must correspond to "Platform" column in *Features Supported* section table, values are lists of text items to process.

Data dictionary sample:

```
data = {
    "cisco_ios" : ["h1", "h2"],
    "cisco_xr": ["h3", "h4"],
    "cisco_nxos": ["h5", "h6"],
    ...etc...
}
```

Where hX device's show commands output.

If data is an OS path directory string, child directories' names must correspond to **Platform** column in *Features Supported* section table. Each child directory should contain text files with show commands output for each device, names of files are arbitrary, but output should contain device prompt to extract device hostname.

Directories structure sample:

```
├──folder_with_data
    ├──cisco_ios
        │       switch1.txt
        │       switch2.txt
    └──cisco_nxos
            nxos_switch_1.txt
            nxos_switch_2.txt
```

To point N2G to above location data attribute string can be /var/data/n2g/folder_with_data/

4.3 CLI L2 Data Plugin

CLI L2 Data Plugin can produce diagrams based on OSI model layer 2 information, hence the name "layer 2". This plugin builds network diagrams with relationships and nodes using CDP and LLDP protocols neighbors information. In addition, adding L1/L2 related data to diagram elements.

CLI L2 Data Plugin uses TTP templates to parse show commands output and transform them in Python dictionary structure. That structure processed further to build a dictionary compatible with N2G's diagram plugins `from_dict` method. That method used to populate diagrams with devices and links information.

In addition to parsing relationships for CDP and LLDP protocols, L2 Data Plugin can help to manipulate diagrams by combining links based on certain criteria, adding additional information to elements meta data and adding unknown (to CDP and LLDP) but connected nodes to diagram.

4.3.1 Features Supported

Features Support Matrix

Platform Name	CDP peers	LLDP peers	interface config	interface state	LAG links	links grouping	node facts	Add all connected	Combine peers
cisco_ios	YES	YES	YES	YES	YES	YES	YES	YES	YES
cisco_xr	YES	YES	YES	YES	YES	YES	—	YES	YES
cisco_nxos	YES	YES	YES	YES	YES	YES	YES	YES	YES
huawei	—	YES	YES	—	YES	YES	YES	—	YES
juniper	—	YES	YES	YES	YES	YES	—	YES	YES

Features Description

- `CDP peers` - adds links and nodes for CDP neighbors
- `LLDP peers` - adds links and nodes for LLDP neighbors
- `interface config` - adds interfaces configuration to links data
- `interface state` - add links state information to links data
- `LAG links` - combines links based on LAG membership
- `links grouping` - groups links between nodes
- `node facts` - adds information to nodes for vlans configuration
- `Add all connected` - add nodes for connected interfaces that has no peers via CDP or LLDP
- `Combine peers` - groups CDP/LLDP peers behind same port by adding "L2" node

4.3.2 Required Commands output

cisco_xr:

- `show cdp neighbor details` and/or `show lldp neighbor details` - mandatory
- `show lldp` - optional, to extract local hostname
- `show running-configuration interface` - optional, used for LAG and interfaces config
- `show interfaces` - optional, used for interfaces state and to add all connected nodes

cisco_ios, cisco_nxos:

- `show cdp neighbor details` and/or `show lldp neighbor details` - mandatory
- `show running-configuration` - optional, used for LAG and interfaces config
- `show interface` - optional, used for interfaces state and to add all connected nodes

huawei:

- `display lldp neighbor details` - mandatory
- `display current-configuration interface` - optional, used for LAG and interfaces config
- `display interface` - optional, used for interfaces state and to add all connected nodes

juniper:

- `show lldp local-information` - to extract local hostname
- `show lldp neighbors` - to extract LLDP neighbors
- `show configuration interfaces | display set` - to extract interfaces configuration and LAG details
- `show interfaces detail` - to extract interfaces state to add all connected devices

4.3.3 Sample Usage

Code to populate yEd diagram object with CDP and LLDP sourced nodes and links:

```
from N2G import cli_l2_data, yed_diagram

data = {"cisco_ios": ['''
switch-1#show cdp neighbors detail
-------------------------
Device ID: switch-2
Entry address(es):
  IP address: 10.2.2.2
Platform: cisco WS-C6509,  Capabilities: Router Switch IGMP
Interface: GigabitEthernet4/6,  Port ID (outgoing port): GigabitEthernet1/5

-------------------------
Device ID: switch-3
Entry address(es):
  IP address: 10.3.3.3
Platform: cisco WS-C3560-48TS,  Capabilities: Switch IGMP
Interface: GigabitEthernet1/1,  Port ID (outgoing port): GigabitEthernet0/1

-------------------------
```

(continues on next page)

(continued from previous page)

```
Device ID: switch-4
Entry address(es):
  IP address: 10.4.4.4
Platform: cisco WS-C3560-48TS,  Capabilities: Switch IGMP
Interface: GigabitEthernet1/2,  Port ID (outgoing port): GigabitEthernet0/10

switch-1#show run
interface GigabitEthernet4/6
 description switch-2: access
 switchport
 switchport access vlan 2150
 switchport mode access
 spanning-tree portfast edge
!
interface GigabitEthernet1/1
 description switch-3:Gi0/1
 switchport
 switchport trunk allowed vlan 1771,1887
 switchport mode trunk
 mtu 9216
!
interface GigabitEthernet1/2
 description SW4 Routing Peering
 vrf forwarding VRF1
 ip address 10.0.0.1 255.255.255.0
    ''',
    '''
switch-2#show cdp neighbors detail
------------------------
Device ID: switch-1
Entry address(es):
  IP address: 10.1.1.1
Platform: cisco WS-C6509,  Capabilities: Router Switch IGMP
Interface: GigabitEthernet1/5,  Port ID (outgoing port): GigabitEthernet4/6

switch-2#show run
interface GigabitEthernet1/5
 description switch-1: access
 switchport
 switchport access vlan 2150
 switchport mode access
 spanning-tree portfast edge
    ''']
    }

config = {
    "add_interfaces_data": True,
    "group_links": False,
    "add_lag": False,
    "add_all_connected": False,
    "combine_peers": False,
    "platforms": ["_all_"]
```

(continues on next page)

(continued from previous page)

```
}

drawing_l2 = yed_diagram()
drawer = cli_l2_data(drawing_l2, **config)
drawer.work(data)
drawer.drawing.dump_file()
```

4.3.4 API Reference

class N2G.plugins.data.cli_l2_data.**cli_l2_data**(*drawing*, *ttp_vars=None*, *add_interfaces_data=True*, *group_links=False*, *add_lag=False*, *add_all_connected=False*, *combine_peers=False*, *skip_lag=True*, *platforms=None*)

Class to instantiate L2 (layer two) data plugin to process CDP and LLDP neighbors together with devices' running configuration and state and produce diagram out of it.

Parameters

- **drawing** – (obj) N2G drawing object instantiated using drawing module e.g. yed_diagram or drawio_diagram
- **ttp_vars** – (dict) dictionary to use as TTP parser object template variables
- **platforms** – (list) - list of platform names to process e.g. `cisco_ios`, `cisco_xr` etc, default is `_all_`
- **add_interfaces_data** – (bool) default `True`, add interfaces configuration and state data to links
- **group_links** – (bool) default `False`, group links between nodes
- **add_lag** – (bool) default `False`, add LAG/MLAG links to diagram
- **add_all_connected** – (bool) default `False`, add all nodes connected to devices based on interfaces state
- **combine_peers`** – (bool) default `False`, combine CDP/LLDP peers behind same interface by adding L2 node
- **skip_lag** – (bool) default `True`, skip CDP peers for LAG, some platforms send CDP/LLDP PDU from LAG ports

work(*data*)

Method to parse text data and add nodes and links to N2G drawing.

Parameters **data** – (dict or str) dictionary or OS path string to directories with text files

If data is dictionary, keys must correspond to "Platform" column in *Features Supported* section table, values are lists of text items to process.

Data dictionary sample:

```
data = {
    "cisco_ios" : ["h1", "h2"],
    "cisco_ios": ["h3", "h4"],
    "cisco_nxos": ["h5", "h6"],
    ...etc...
}
```

Where `hX` devices show commands output.

If data is an OS path directory string, child directories' names must correspond to **Platform** column in *Features Supported* section table. Each child directory should contain text files with show commands output for each device, names of files are arbitrary, but output should contain device prompt to extract device hostname.

Directories structure sample:

```
├──folder_with_data
    ├──cisco_ios
        switch1.txt
        switch2.txt
    └──cisco_nxos
        nxos_switch_1.txt
        nxos_switch_2.txt
```

To point N2G to above location `data` attribute string can be `/var/data/n2g/folder_with_data/`

4.4 CLI OSPFv2 LSDB Data Plugin

CLI OSPFv2 LSDB Data Plugin can process network devices CLI output of OSPFv2 LSDB content to populate N2G drawing with OSPF topology nodes and links.

CLI output from devices parsed using TTP Templates into a dictionary structure.

After parsing, results processed further to form a dictionary of nodes and links keyed by unique nodes and links identifiers wit values being nodes dictionaries and for links it is a list of dictionaries of links between same pair of nodes. For nodes OSPF RID used as a unique ID, for links it is sorted tuple of `source`, `target` and `label` keys' values. This structure helps to eliminate duplicates.

Next step is post processing, such as packing links between nodes. By default cli_ospf_data tries to match and pack nodes based on the IP addresses and their subnets, it checks if IP addresses are part of same subnet using prefix lengths - 31, 30, 29, … 22 - if IP addresses happens to be part of same subnet, link packed in one link.

Last step is to populate N2G drawing with new nodes and links using `from_dict` method.

4.4.1 Features Supported

Support matrix

Platform Name	Router LSA	OSPF Peers	External LSA	Summary LSA	interface config	interface state
cisco_ios	YES	—	—	—	—	—
cisco_xr	YES	—	—	—	—	—
cisco_nxos	—	—	—	—	—	—
huawei	YES	—	—	—	—	—

4.4.2 Required Commands output

cisco_ios:

- `show ip ospf database router` - mandatory, used to source nodes and links for topology
- `show ip ospf database summary`
- `show ip ospf database external`

cisco_xr:

- `show ospf database router` - mandatory, used to source nodes and links for topology
- `show ospf database summary`
- `show ospf database external`

huawei:

- `display ospf lsdb router` - mandatory, used to source nodes and links for topology

4.4.3 Sample usage

Code to populate yEd diagram object with OSPF LSDB sourced nodes and links:

```
from N2G import cli_l2_data, yed_diagram

data = {"cisco_xr": ['''
RP/0/RP0/CPU0:router-1#show ospf database router

            OSPF Router with ID (10.0.1.1) (Process ID 1)

                Router Link States (Area 0.0.0.0)

  LS age: 406
  Options: (No TOS-capability, DC)
  LS Type: Router Links
  Link State ID: 10.0.1.1
  Advertising Router: 10.0.1.1
  LS Seq Number: 8000010c
  Checksum: 0x24dd
  Length: 132
   Number of Links: 9

    Link connected to: another Router (point-to-point)
     (Link ID) Neighboring Router ID: 10.0.1.4
     (Link Data) Router Interface address: 0.0.0.12
      Number of TOS metrics: 0
       TOS 0 Metrics: 1100

    Link connected to: another Router (point-to-point)
     (Link ID) Neighboring Router ID: 10.0.1.2
     (Link Data) Router Interface address: 0.0.0.10
      Number of TOS metrics: 0
       TOS 0 Metrics: 1100
```

(continues on next page)

(continued from previous page)

```
  Routing Bit Set on this LSA
  LS age: 1604
  Options: (No TOS-capability, DC)
  LS Type: Router Links
  Link State ID: 10.0.1.2
  Advertising Router: 10.0.1.2
  LS Seq Number: 8000010b
  Checksum: 0xdc96
  Length: 132
   Number of Links: 9

    Link connected to: another Router (point-to-point)
     (Link ID) Neighboring Router ID: 10.0.1.3
     (Link Data) Router Interface address: 0.0.0.52
      Number of TOS metrics: 0
       TOS 0 Metrics: 1100

    Link connected to: another Router (point-to-point)
     (Link ID) Neighboring Router ID: 10.0.1.4
     (Link Data) Router Interface address: 0.0.0.53
      Number of TOS metrics: 0
       TOS 0 Metrics: 1100
    ''']
}

drawing = yed_diagram()
drawer = cli_ospf_data(drawing)
drawer.work(data)
drawer.drawing.dump_file()
```

4.4.4 API Reference

class N2G.plugins.data.cli_ospf_data.**cli_ospf_data**(*drawing*, *ttp_vars: Optional[dict] = None*, *ip_lookup_data: Optional[dict] = None*, *add_connected: bool = False*, *ptp_filter: Optional[list] = None*, *add_data: bool = True*)

Main class to instantiate OSPFv2 LSDB CLI Data Plugin object.

Parameters

- **drawing** – (obj) N2G Diagram object
- **ttp_vars** – (dict) Dictionary to use as vars attribute while instantiating TTP parser object
- **ip_lookup_data** – (dict or str) IP Lookup dictionary or OS path to CSV file
- **add_connected** – (bool) if True, will add connected subnets as nodes, default is False
- **ptp_filter** – (list) list of glob patterns to filter point-to-point links based on link IP
- **add_data** – (bool) if True (default) adds data information to nodes and links

`ip_lookup_data` dictionary must be keyed by OSPF RID IP address, with values being dictionary which must contain `hostname` key with optional additional keys to use for N2G diagram module node, e.g. `label`, `top_label`, `bottom_label`, `interface``etc. If ``ip_lookup_data` is an OS path to CSV file, that file's

first column header must be `ip` , file must contain `hostname` column, other columns values set to N2G diagram module node attributes, e.g. `label`, `top_label`, `bottom_label`, `interface` etc.

If lookup data contains `interface` key, it will be added to link label.

Sample ip_lookup_data dictionary:

```
{
    "1.1.1.1": {
        "hostname": "router-1",
        "bottom_label": "1 St address, City X",
        "interface": "Gi1"
    }
}
```

Sample ip_lookup_data CSV file:

```
ip,hostname,bottom_label,interface
1.1.1.1,router-1,"1 St address, City X",Gi1
```

`ptp_filter` default list of patterns are:

- `0*` - Cisco MPLS TE forwarding adjacencies links
- `112*` - huawei DCN OSPF links

work(*data*)

Method to parse OSPF LSDB data and add nodes and links to N2G drawing.

Parameters **data** – (dict or str) dictionary keyed by platform name or OS path string to directories with text files

If data is dictionary, keys must correspond to "Platform" column in *Supported platforms* table, values are lists of text items to process.

Data dictionary sample:

```
data = {
    "cisco_ios" : ["h1", "h2"],
    "cisco_ios": ["h3", "h4"],
    "cisco_nxos": ["h5", "h6"],
    ...etc...
}
```

Where `hX` device's show commands output.

If data is an OS path directory string, child directories' names must correspond to **Platform** column in *Features Supported* section table. Each child directory should contain text files with show commands output for each device, names of files are arbitrary, but output should contain device prompt to extract device hostname.

Directories structure sample:

```
├──folder_with_data
    ├──cisco_ios
        │       switch1.txt
        │       switch2.txt
    └──cisco_nxos
```

(continues on next page)

(continued from previous page)

```
            nxos_switch_1.txt
            nxos_switch_2.txt
```

To point N2G to above location `data` attribute string can be `/var/data/n2g/folder_with_data/`

4.5 JSON Data Plugin

JSON data plugin loads structured data from JSON string and inputs it into diagram class - if JSON string produces list, uses `from_list` method, if JSON string produces dictionary uses `from_dict` method.

4.5.1 Sample Usage

Code to demonstrate how to use `json_data` plugin:

```
from N2G import v3d_diagramm
from N2G import json_data

sample_json_data = '''{
    "links": [{"data": {}, "label": "bla1", "source": "node-1", "src_label": "", "target
↪": "node-2", "trgt_label": ""},
            {"data": {}, "label": "bla2", "source": "node-1", "src_label": "", "target":
↪"node-3", "trgt_label": ""},
            {"data": {}, "label": "bla3", "source": "node-3", "src_label": "", "target":
↪"node-5", "trgt_label": ""},
            {"data": {}, "label": "bla4", "source": "node-3", "src_label": "", "target":
↪"node-4", "trgt_label": ""},
            {"data": {}, "label": "bla77", "source": "node-33", "src_label": "", "target
↪": "node-44", "trgt_label": ""},
            {"data": {"cd": 123, "ef": 456}, "label": "bla6", "source": "node-6", "src_
↪label": "", "target": "node-1", "trgt_label": ""}],
    "nodes": [{"color": "green", "data": {}, "id": "node-1", "label": "node-1",
↪"nodeResolution": 16},
            {"color": "green", "data": {}, "id": "node-2", "label": "node-2",
↪"nodeResolution": 8},
            {"color": "blue", "data": {"val": 4}, "id": "node-3", "label": "node-3",
↪"nodeResolution": 8},
            {"color": "green", "data": {}, "id": "node-4", "label": "node-4",
↪"nodeResolution": 8},
            {"color": "green", "data": {}, "id": "node-5", "label": "node-5",
↪"nodeResolution": 8},
            {"color": "green", "data": {"a": "b", "c": "d"}, "id": "node-6", "label":
↪"node-6", "nodeResolution": 8},
            {"color": "green", "data": {}, "id": "node-33", "label": "node-33",
↪"nodeResolution": 8},
            {"color": "green", "data": {}, "id": "node-44", "label": "node-44",
↪"nodeResolution": 8},
            {"color": "green", "data": {}, "id": "node-25", "label": "node-25",
↪"nodeResolution": 8}]
}'''
```

(continues on next page)

(continued from previous page)

```
v3d_drawing = create_v3d_diagram()
json_data(v3d_drawing, sample_json_data)
v3d_drawing.dump_file()
```

4.5.2 API Reference

N2G.plugins.data.json_data.**json_data**(*drawing*, *data*)

Function to load graph data from JSON text.

Parameters

- **drawing** – (obj) class object of one of N2G diagram plugins
- **data** – (str) JSON string to load

If JSON string produces list, uses `frm_list` method, if dictionary uses `from_dict` method

4.6 XLSX Data Plugin

This plugin loads data from xlsx tables and transforms it in a dictionary supported by N2G diagram plugins. Using `from_dict` method, this plugin loads data into diagram plugin.

4.6.1 Guidelines and Limitations

- openpyxl >= 3.0.0 library need to be installed: `pip install openpyxl`
- Nodes and links tabs' first row must contain headers
- nodes tab should have at least `id` header, other headers should comply with `from_dict` method attributes or simply ignored
- links tab should have at least `source` and `target` headers, other headers should comply with `from_dict` method attributes or simply ignored

4.6.2 Sample Usage

Code to invoke `xlsx_data`:

```
from N2G import drawio_diagram
from N2G import xlsx_data

drawio_drawing = drawio_diagram()

xlsx_data(
    drawio_drawing,
    "./Data/nodes_and_links_data.xlsx",
    node_tabs="nodes",
    link_tabs="links"
)
```

(continues on next page)

(continued from previous page)

```
drawio_drawing.layout(algo="kk")
drawio_drawing.dump_file(filename="diagram.drawio", folder="./Output/")
```

Where `nodes_and_links_data.xlsx` content for nodes tab:

id	pic	label	bottom_label	top_label	description
r1		r1	core	1.1.1.1	Core Router
r2		r2	core	2.2.2.2	Core Router
r3		r3	edge	3.3.3.3	Edge Router

for links tab:

source	src_label	label	target	trgt_label	description
r1	Gi1/1	DF-10Km	r2	Gi3/4	DF link between R1 and R2
r3	10GE2/1/1	DF-32Km	r2	Ten1/1	DF link between R3 and R2

Support available to translate headers to comply with N2G diagram modules `from_dict` or `from_list` methods through the use of `node_headers_map` and `link_headers_map`. For instance consider this table:

```
# nodes tab:
hostname   lo0_ip    bgp_asn
r1         1.1.1.1   65123
r2         1.1.1.2   65123
r3         1.1.1.3   65123

# links tab:
device:a    interface:a    label    device:b  interface:b
r1          Gi1/1          DF-10Km  r2        Gi3/4
r3          10GE2/1/1      DF-32Km  r2        Ten1/1
```

If `node_headers_map` is:

```
node_headers_map = {
    "id": ["device", "hostname"],
    "tob_label": ["lo0_ip"],
    "bottom_label": ["bgp_asn"]
}
```

And `link_headers_map` is:

```
link_headers_map = {
    "source": ["device:a", "hostname:a"],
    "target": ["device:b", "hostname:b"],
    "src_label": ["interface:a", "ip:a"],
    "trgt_label": ["interface:b", "ip:b"]
}
```

Above table will be transformed to:

```
# nodes tab:
id         tob_label bottom_label
r1         1.1.1.1   65123
r2         1.1.1.2   65123
```

(continues on next page)

(continued from previous page)

```
r3          1.1.1.3   65123

# links tab:
source      src_label      label    target    trgt_label
r1          Gi1/1          DF-10Km  r2        Gi3/4
r3          10GE2/1/1      DF-32Km  r2        Ten1/1
```

4.6.3 API Reference

N2G.plugins.data.xlsx_data.**xlsx_data**(*drawing*, *data*, *node_tabs=None*, *link_tabs=None*, *node_headers_map=None*, *link_headers_map=None*)

Function to load data from XLSX file and add it to diagram using `from_dict` method.

Parameters

- **drawing** – N2G drawing module object
- **data** – (str) OS path to xlsx file to load
- **node_tabs** – (list) list of tabs with nodes data, default `["nodes"]`
- **link_tabs** – (list) list of tabs with links data, default `["links"]`
- **node_headers_map** – (dict) dictionary to use to translate node tabs headers
- **link_headers_map** – (dict) dictionary to use to translate link tabs headers

Returns `True` on success and `False` on failure to load data

CHAPTER

FIVE

VIEWER PLUGINS

Viewer plugins are standalone WEB UI applications that can be used to view diagrams content.

5.1 yEd SVG Viewer

yED SVG Viewer allows to start simple Flask WEB UI application to visualize network data using D3.js library.

yED SVG Viewer supports diagram files in SVG format produced by yED Graph Editor application using `File -> Export -> Save as type: SVG format` feature.

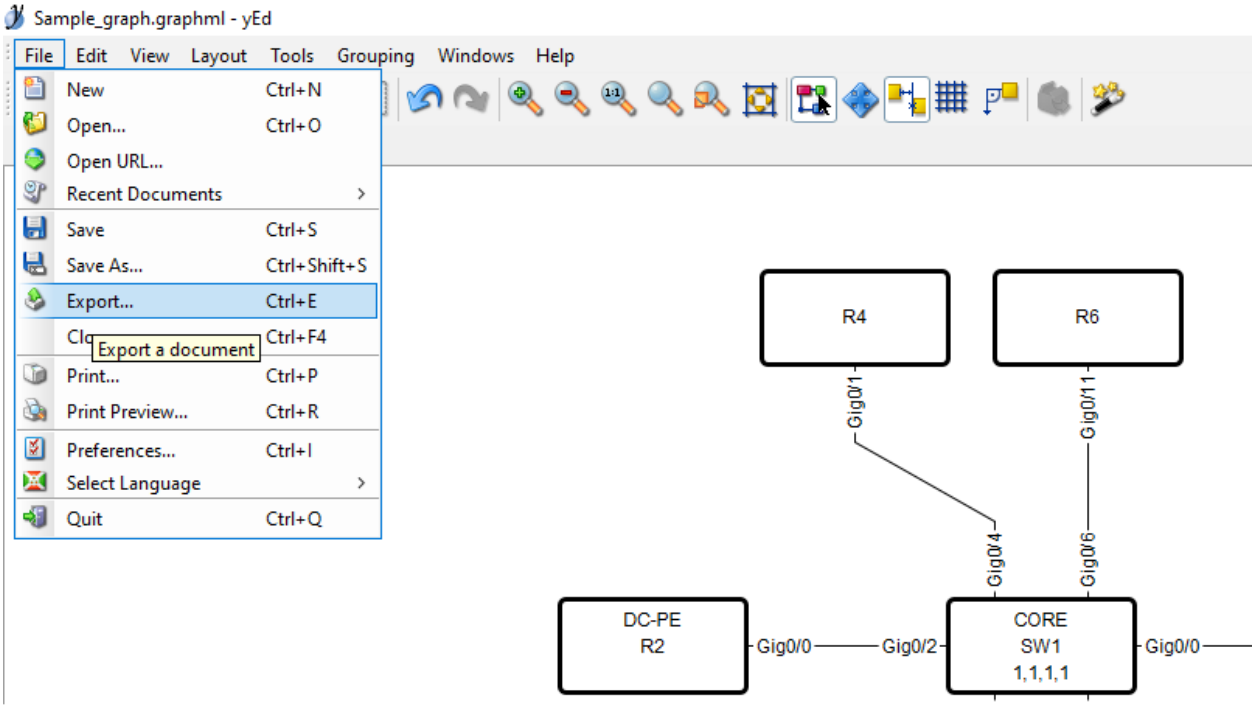
This viewer needs to have Flask installed:

```
pip install flask
```

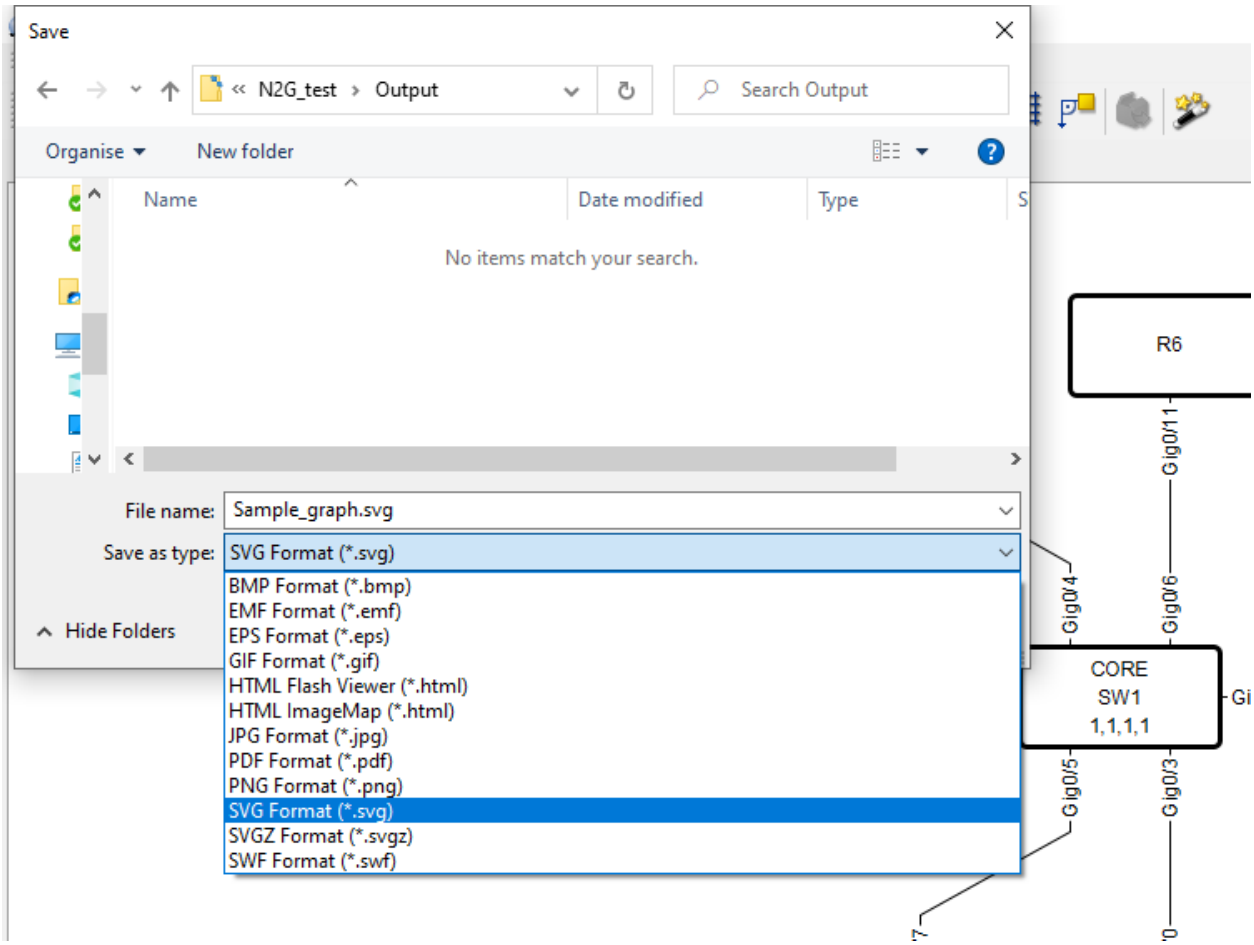
Flask installed as part of `full` extras as well.

Tutorial How to Make SVG Diagrams

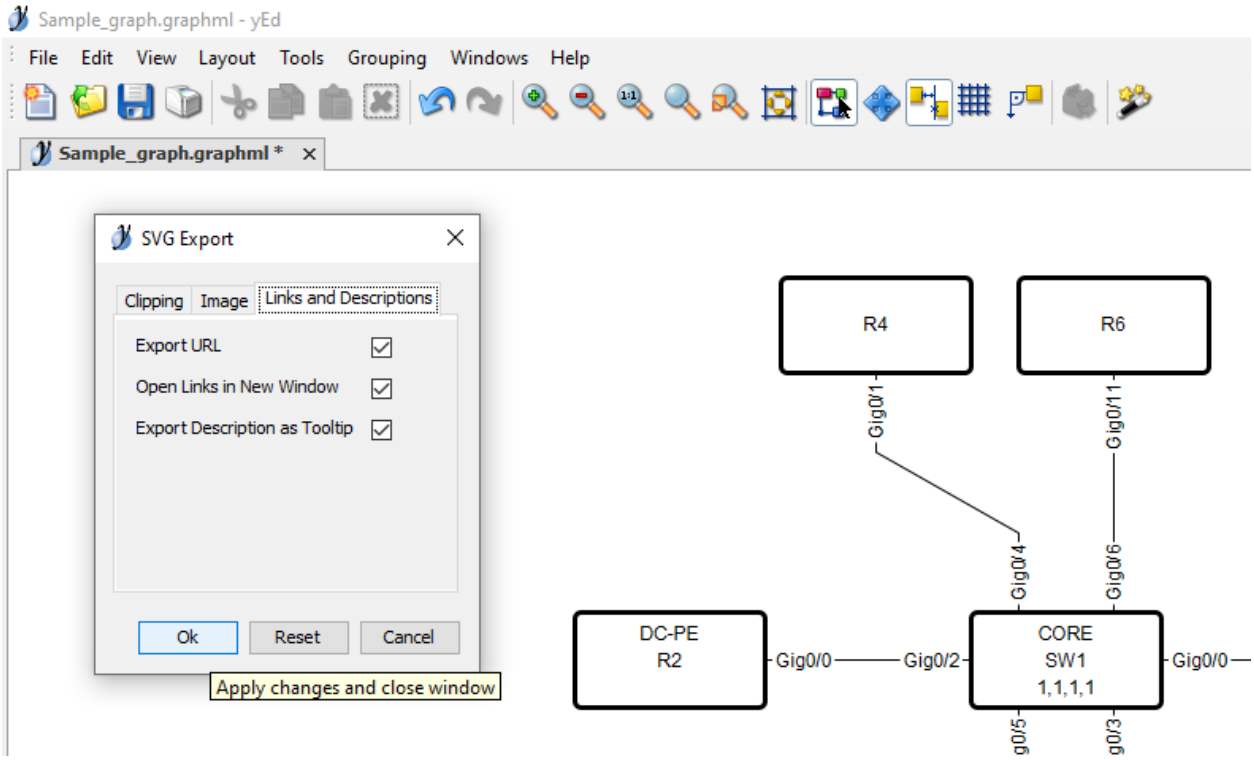
Open diagram in yEd graph editor application and navigate to `File -> Export`:



Choose SVG format and click save:



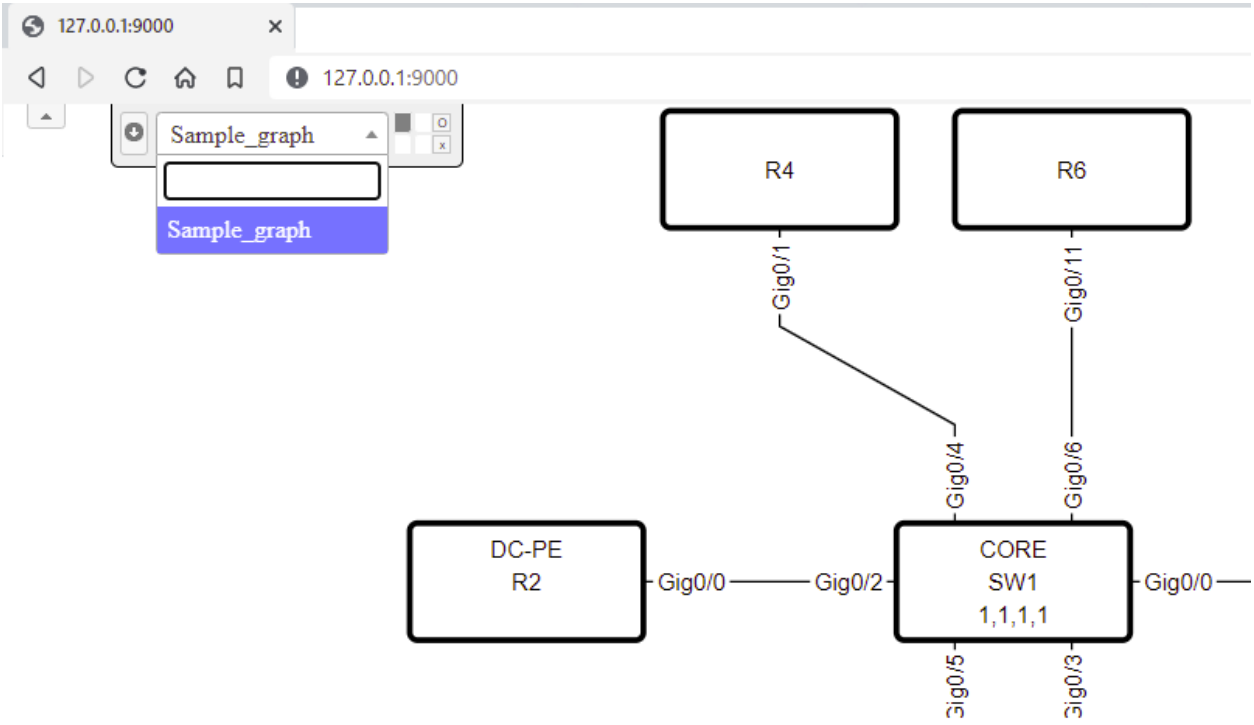
In export menu, make sure to select “Export URL” and “Export Description as Tooltip”, press “Ok” button:



Navigate to folder with exported SVG file and run yEd SVG viewer using N2G CLI tool:

```
N2G --yed-svg-viewer --diagrams-dir "."
```

Access WEB UI application via URL `http://127.0.0.1:9000` using your browser.



By default Flask server starts and listens on all operating system interfaces, but specific IP address and port number

can be specified as required using `--ip` and `--port` N2G CLI arguments.

OS path to directory with diagram files can be specified using `--diagrams-dir` N2G CLI tool argument or using `N2G_DIAGRAMS_DIR` environment variable. If no `--diagrams-dir` argument provided, N2G attempts to retrieve diagrams directory path using `N2G_DIAGRAMS_DIR` environment variable.

5.2 V3D Diagrams Viewer

V3D (Vasturiano 3D) Diagrams Viewer allows to start simple Flask WEB UI application to visualize network data in 3D using force-3d-graph library.

This viewer needs to have Flask installed:

```
pip install flask
```

Flask installed as part of `full` extras as well.

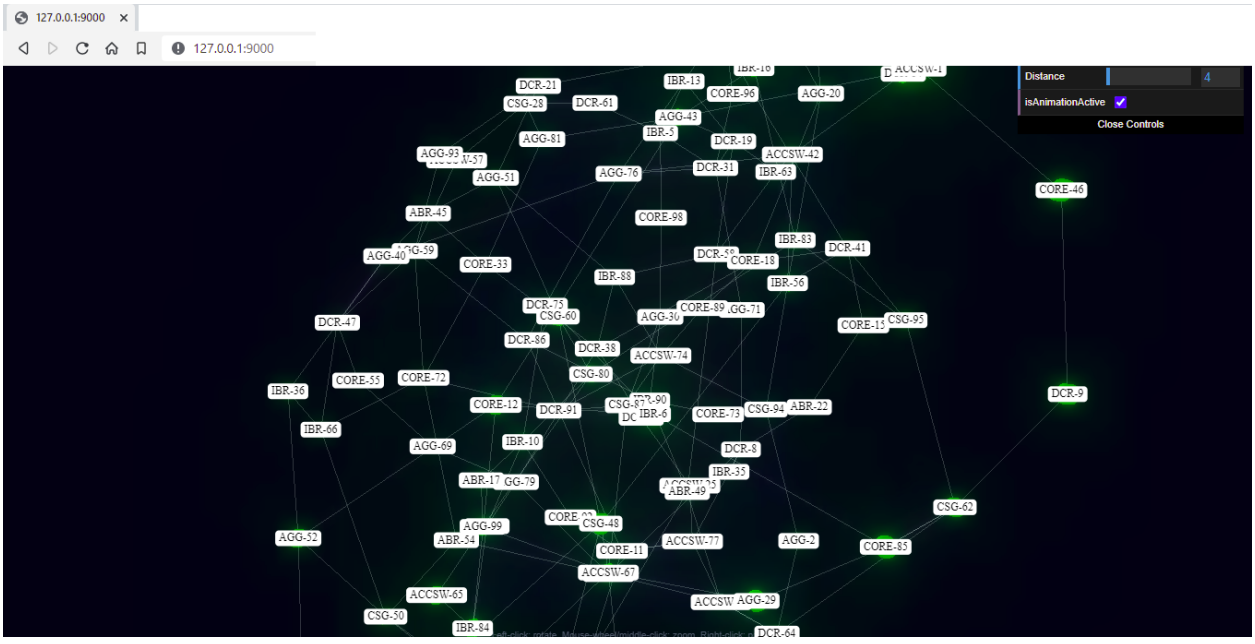
First, produce JSON file using N2G V3D diagram module using preferred data plugin, L2 in this case:

```
N2G -d ./Data/ -m v3d -L2 -fn sample_v3d_viewer_file
```

Next, run V3D viewer application using N2G CLI tool:

```
N2G --v3d-viewer --diagram-file Output/sample_v3d_viewer_file.txt
```

Access WEB UI application via URL `http://127.0.0.1:9000` using your browser. It should look similar to this:



By default Flask server starts and listens on all operating system interfaces, but specific IP address and port number can be specified as required using `--ip` and `--port` N2G CLI arguments.

Where `sample_v3d_viewer_file.txt` file content should contain JSON data conforming to force-3d-graph input JSON syntax format for example:

```
{
    "nodes": [
        {
            "id": "id1",
            "name": "name1",
        },
        {
            "id": "id2",
            "name": "name2",
        }
    ],
    "links": [
        {
            "source": "id1",
            "target": "id2"
        }
    ]
}
```

CHAPTER

SIX

N2G CLI TOOL

This tool allows to use N2G module capabilities from command line interface.

To produce diagram, N2G will need source data to work with, for data plugins source data usually comes in the form of directories structure with text files containing show commands output for devices.

After source data provided, CLI tool need to know what it needs to do, hence next comes the options of various Data Plugins, such as L2 - layer 2 data plugin.

And finally, results need to be saved somewhere on the local file system using filename and folder options.

Supported options:

```
Parsing order is: CDP/LLDP (L2)  => IP => OSPF => ISIS

-d,   --data         OS path to data folder with files or file
-of,  --out-folder   Folder where to save result, default ./Output/
-fn,  --filename     Results filename, by default filename based on current time
-m,   --module       Module to use - yed, drawio or v3d
-ipl, --ip_lookup    Path to CSV file for IP lookups, first column header must be ``ip``
--no-data            Do not add any data to links or nodes
--layout             Name of iGraph layout algorithm to run for the diagram e.g. "kk",
↪"tree" etc.
--log-level          Logging level, default is ``ERROR``
--port               Port number to run viewer server (V3D, yED) on, default is 9000
--ip                 IP address to run viewer server (V3D, yED) on, default is ``0.0.0.
↪0``

V3D Module arguments:
--run                Run built in test web server to display topology instead of saving␣
↪to file

XLSX data adapter. -d should point to ".xlsx" spreadsheet file.
-nt,     --node-tabs           Comma separate list of tabs with nodes data
-lt,     --link-tabs           Comma separate list of tabs with links data
-nm,     --node-headers-map    JSON dictionary structure for node headers translation
-lm,     --link-headers-map    JSON dictionary structure for link headers translation

CDP and LLDP L2 Data Plugin options:
-L2                 Parse CDP and LLDP data
-L2-add-lag         Add LAG/M-LAG information and delete member links
-L2-group-links     Group links between nodes
-L2-add-connected   Add all connected nodes
```

(continues on next page)

(continued from previous page)

```
-L2-combine-peers   Combine CDP/LLDP peers behind same interface
-L2-platforms       Comma separated list of platforms to parse

IP Data Plugin:
-IP                 Parse IP subnets
-IP-group-links     Group links between nodes
-IP-lbl-intf        Add interfaces names to link labels
-IP-lbl-vrf         Add VRF names to link labels
-IP-add-arp         Add ARP cache IPs to the diagram

OSPF LSDB Data Plugin:
-OSPF               Diagram OSPFv2 LSDB data
-OSPF-add-con       Add connected subnets to diagram

ISIS LSDB Data Plugin:
-ISIS               Diagram ISIS LSDB data
-ISIS-add-con       Add connected subnets to diagram

yED SVG Viewer:
--yed-svg-viewer    Run yED SVG Viewer
--diagrams-dir      OS Path to directory with diagrams svg files

V3D Diagram Viewer:
--v3d-viewer        Run V3D JSON files viewer
--diagram-file      OS Path to JSON file with diagram data
```

Sample Usage

To make L2 (CDP and LLDP) diagram in yEd format and save it into `./Output/diagram_1.graphml` file grouping L2 links:

```
n2g -d ./path/to/data/ -m yed -L2 -L2-group-links -fn diagram_1.graphml -of ./Output/
```

PYTHON MODULE INDEX

n

INDEX