

@penCHORD_UoE
@peninsula_ARC



Module 4 : Network Analysis
Session 4D : Advanced Network Analysis (Part 2)
Elliott Coyne
"Flying The Flag"



#hsma5isalive

@penCHORD_UoE
@peninsula_ARC



Module 4 : Network Analysis
Session 4D : Advanced Network Analysis (Part 2)
Elliott Coyne
"Flying The Flag"



#hsma5isalive

Learning Objectives

- Understand how graph algorithms seek to represent data
- Know how to visualise graphs using two different approaches
- Be able to customise a graph visualisation to improve its readability

How graph visualisations work

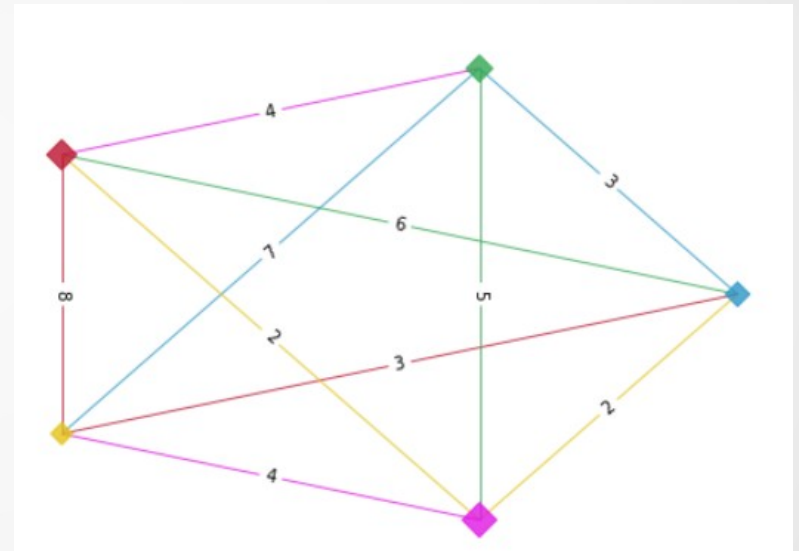
Network graphs provide a flexible way to visualise multiple attributes of the data. The three main components of the graph that can be customised to represent different aspects of the data are:

- Nodes
- Edges
- Layout

How graph visualisations work

Node customisation

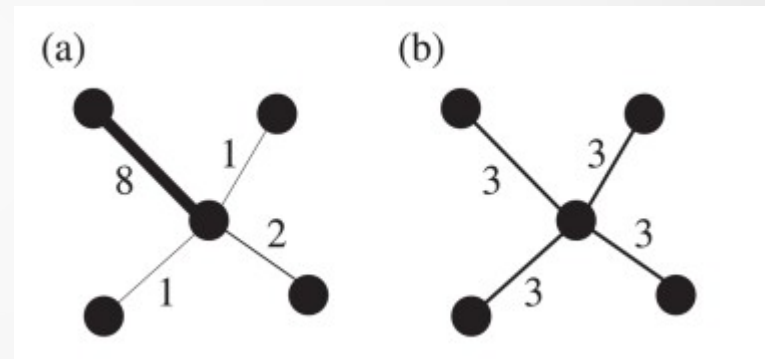
- **Size** - for *continuous* attributes (i.e. eigenvector centrality)
- **Colour** - for *discrete* attributes (i.e. categorical data) NB could also be used for continuous attributes also.
- **Shape** - for *discrete* attributes
- **Opacity** - for *continuous* attributes
- **Label** - for *discrete or continuous* attributes



How graph visualisations work 2

Edge customisation

- **Thickness** - for *continuous* attributes (i.e., usually weight)
- **Colour** - for *discrete* attributes (i.e., edge the same colour as source in directed graph)
- **Label** - for *continuous* or *discrete* attributes

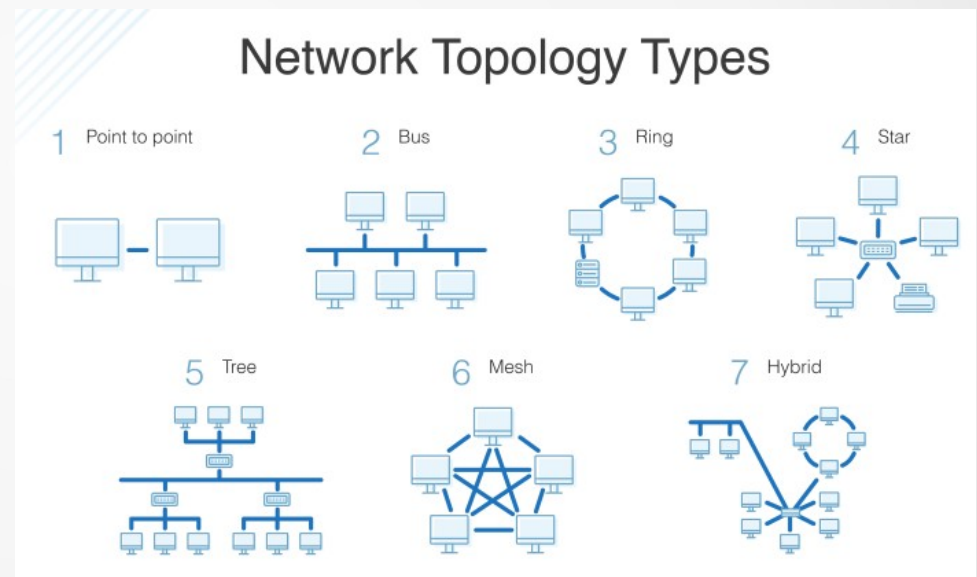


How graph visualisations work 3

Layout customisation

The basic layout of the graph is determined by the structure of the network, also called the 'network topology'. Some common network structures are:

- Hierarchical
- Tree/Forest (*c.f. Family Tree*)
- Ring
- Mesh (*All interconnected*)
- Star (*Central Point*)
- Bus



Only work for highly organised systems i.e., power networks, computer networks, etc

How graph visualisations work 4

Simple and highly structured systems might conform to a simple topology.

Complex systems (i.e., health systems) are often a hybrid of two or more topologies making them more difficult to visualise. There are algorithms that help to visualise complex networks with non-standard topologies.

How graph visualisations work 5

NetworkX core layouts....

- **Circular layout** - nodes positioned in a circle (*Most simple*)
- **Kamada kawai layout** - nodes positioned using a path-length cost-function
- **Random layout** - nodes positioned uniformly at random (i.e., laid out in a block)
- **Spring layout/Fruchterman-Reingold force directed layout** - a physics based algorithm using attraction and repulsion and gravity based parameters
- **Spectral layout** - nodes are positioned using the eigenvectors of the Laplacian matrix
- **Spiral layout** - nodes positioned in a spiral

For detailed layout documentation see

<https://networkx.org/documentation/stable/reference/drawing.html>

How graph visualisations work 6

Dash Cytoscape standard layouts

- **Preset** - nodes positioned by pre-determined coordinates (*i.e. geographic coordinates – locations of facilities, service users, etc*)
- **Random** - nodes are randomly positioned
- **Grid** - nodes are positioned in a square or rectangular grid
- **Circle layout** - nodes positioned in a circle
- **Concentric** - nodes are positioned in multiple concentric circles
- **Breadthfirst** - nodes are positioned in a hierarchical tree structure
- **Cose** - a physics based spring simulation (arguably the most useful for health care systems – similar to Spring Layout in NetworkX)

For detailed layout information see

<https://dash.plotly.com/cytoscape/reference>

<https://js.cytoscape.org/#layouts>

How graph visualisations work 7

Cytoscape external layout

Requires the line of code `cyto.load_extra_layouts()`

- **Cose-bilkent** - extended cose layout for nested structures
- **Cola** - force directed physics layout with good generic applicability
- **Euler** - force directed physics layout with good generic applicability
- **Spread** - force directed physics layout aiming for an even spread
- **Dagre** - for use with directed acyclic graphs and trees
- **Klay** - discrete layout algorithm with good generic applicability

For detailed layout information see

<https://dash.plotly.com/cytoscape/reference>

<https://js.cytoscape.org/#layouts>

Visualising graphs with NetworkX

NetworkX

i.e. spring, random, circle...

```
pos = nx.layout('graph_object')
```

```
draw_networkx('graph_object', position_dict)
```

Or

```
# One-step without creating dictionary separately  
draw_networkx('graph_object', nx.layout, **kwds)
```

Visualising graphs with NetworkX 2

NetworkX key word arguments (kwds/kwargs)

Allowing us to represent additional attributes within graph

- `arrows, arrowstyle, arrowsize`
- `with_labels, node_size, node_color, node_shape, cmap`
- `width, edge_color, edge_cmap, style`
- `font_size, font_color, font_weight, font_family`

NetworkX Example

Let's look at a simple example of producing a network visualisation in NetworkX. Open the '**networkx_vis_example.py**' file.

The code breaks down as:

- Create some data
- Function to create the graph
- Define the layout
- Define the node and edge attributes: note the number of values for each attribute
- Draw the plot

Visualising graphs with Plotly Cytoscape

Plotly Dash is a python framework for building web applications.

Cytoscape (found in Dash) is a powerful network graph library for creating interactive graphs within web applications.

The two examples that we looked at in the introduction to network analysis were both built using Dash and Cytoscape (i.e., to render network graphs): GoT & Systematic Review apps.

We will now look at how to create a simple Dash web-application that contains a Cytoscape network graph.

The structure of a Dash application

A simple Dash application is constructed from **six** core sections (order specific):

1)Python library imports

2)Persistent local variable creation

Any functions or vars created when the app first starts – only run once!

3)Application initialisation

4)The application layout

Structure of visual application

5)Application callback functions

Enables us to interact with things in the application (i.e., drop down menu)

6)The application run command

Let the magic commence!

Python library imports

Python library imports

Like with any python script we need to import the required libraries at the beginning of our script

The Dash and Cytoscape libraries most commonly needed are:

```
import dash # core library
import dash_html_components as html
import dash_core_components as dcc
import dash_cytoscape as cyto
import networkx as nx # allows analysis of graph prior to web-app viz
from dash.dependencies import Input, Output, State # for callbacks
```

All (or certainly more) will be revealed as we progress through the following slides.

Persistent local variable creation

Any code that is run before the app initialisation section **is only run once** when the app is first loaded.

This can be useful for simple applications that don't require lots of interactivity or where you have local variables that never change. (Avoids the needs for more complex reactive programming i.e., use of callbacks).

In the example we will look at, all of the data upload and preprocessing is done when the app first loads.

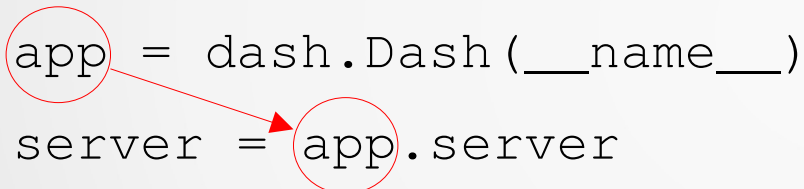
We also set the style options for the network graph in a dictionary known as a *stylesheet* (nomenclature from web development - css stylesheets). ([CSS](#) is the coding framework that is used to style web pages).

Application initialisation

Once we have our local variables (previous step) we can then initialize the app.

Creating the application object requires only two lines of code to setup an application object and a server object.

```
app = dash.Dash(__name__)  
server = app.server
```



The first line creates a Dash application object

The second line creates the server object linked to the application object

Application layout

The application layout contains the information on how the objects used in the application are to be structured and setup on app initialisation.

This information is passed to the **layout** component of the application object.

The layout is based on HTML components and syntax which has been adapted for use in python.

```
app.layout = html.Div([
    html.Div(className='site-title text-center', children=[
        html.H1(className='padheader', children=[
            'Network of Thrones'
        ]),
    ]),
    html.Div(className='text-center', children=[
        html.P(children=[
            'Data from https://networkofthrones.wordpress.com/'
        ]),
    ]),
    html.Div(className='row', children=[
        html.Div(className='col-3', children=[
            html.Div(className='center-items', children=[
                html.Div(className='text-center', children=[
                    html.H3(children=['Select season']),
                ]),
                html.Div(className='slider-container', children=[
                    dcc.Slider(className='slider', id='season-slider',
```

Div – Divider Object – Containers of different objects within a single unit that CSS styling can be passed to. Helps to structure the web layout itself i.e., spacing, columns and rows.

Html.H1 – Header Size 1 (Largest)

Html.P – Paragraph

Application layout 2

It is in the layout section that we build up the layout structure using

- HTML components,
- Dash core components and
- additional component libraries such as *cytoscape* and *dash bootstrap*.

HTML component examples Non-Interactive / Static

- **Div** - Divider/container
- **H1** - Headers, sizes 1-5
- **P** - Paragraph
- **A** - Hyperlink
- **UL** - Unordered list
- **Img** – Image

Dash core component examples (Interactive)

- Dropdown
- Input
- Slider
- Button
- Graph
- Checklist

• Examples of above **here**.

Application callback functions

Callback functions come after the layout section.

These functions take inputs from the app components defined in the layout and return outputs to other objects in the layout. The syntax is an adapted version of the standard python user defined function syntax.

```
@app.callback(Output('dynamic-net', 'elements'),
               [Input('hidden-elements-dict', 'children'),
                Input('graph-slider', 'value')])
def dynamic_network_select(data, selection):
    if data != None:
        el_dict = json.loads(data)
        selection = str(selection)
        selected = el_dict[selection]
        #elements = list(selected)

        return selected
    else:
        return list()
```

Number of UDF return objects should match number of Output at the top

To start a callback function we declare @app.callback - this links callback function directly to the layout.

Pass in:

- Output(s) – as a list, if multiple
First **which** then **where**
- Inputs - as a list, as multiple
UDF inputs correspond to these inputs
- states

Application run command

The application run command does just that. It comes at the end of the script and runs the app server in the development environment

```
if __name__ == "__main__":  
    app.run_server(debug=False)
```

The Cytoscape graph object

The main Cytoscape graph objects arguments can be summarised as:

- **id** = *str* - a unique identifier within the app layout
- **className** = *str* - a CSS class with standardised styling options
- **elements** = *list of dicts* - nodes and edge data – **See next slide**
- **style** = *dict* - style options alternately placed in the stylesheet
- **layout** = *dict* - graph layout name and the layout attributes
- **stylesheet** = *dict* - node and edge style options

The Cytoscape graph object 2

An example of a very simple Cytoscape graph object in the app layout.

As is common in web dev – objects are wrapped in containers

```
app.layout = html.Div(children=[
    cyto.Cytoscape(
        id='cyto-graph',
        className='net-obj',
        elements=elements,
        style={'width': '80%', 'height': '600px'},
        layout={'name': 'cose',
                'padding': '200',
                'nodeRepulsion': '7000',
                'gravityRange': '6.0',
                'nestingFactor': '0.8',
                'edgeElasticity': '50',
                'idealEdgeLength': '200',
                'nodeDimensionsIncludeLabels': 'true',
                'numIter': '6000',
                },
        stylesheet=default_stylesheet
    )
])
```

Parent

Children – Always contained within Parent Object.
Makes this explicit.

Pass in list of objects that belong to that element.

Objects given a unique ID so they can be identified within the web app and callback function – the **must** be unique.

CSS Class name to provide formatting to object (not essential but useful when trying to make things look pretty!)

Elements – where data for graph will go. (All the pre-processing will end up here)

Style contains a dict of display attributes

Layout attributes for Cytoscape graph is declared

See next slide

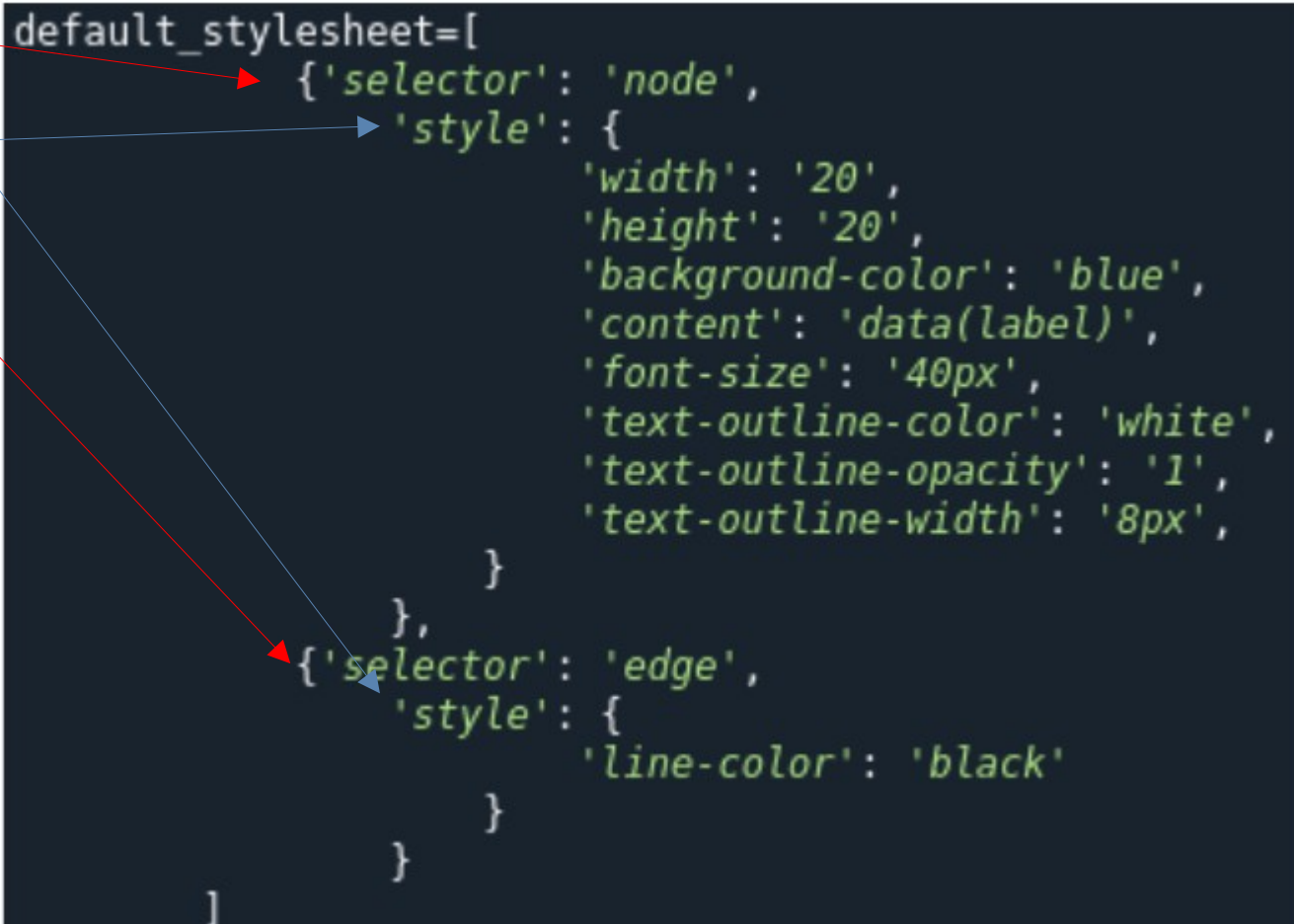
The Cytoscape graph object - Stylesheet

An example of a stylesheet...

Selector level – one for each **node**
and another for **edge**

Then **style** dict of different attributes

```
default_stylesheet=[
  {'selector': 'node',
   'style': {
     'width': '20',
     'height': '20',
     'background-color': 'blue',
     'content': 'data(label)',
     'font-size': '40px',
     'text-outline-color': 'white',
     'text-outline-opacity': '1',
     'text-outline-width': '8px',
   }
  },
  {'selector': 'edge',
   'style': {
     'line-color': 'black'
   }
  }
]
```



A simple Cytoscape visualisation

Let's now have a look at a simple Dash app displaying a network graph made using Cytoscape. Open the '***cytoscape_vis_simple.py***' file. This example imports two csv files for the node and edge data so you will need to set your working directory accordingly.

Over to Spyder....

A more advanced Cytoscape visualisation

Now let's look at a more advanced version of the simple Dash app.

This app not only displays the network graph made using cytoscape but also incorporates an analysis and visualises this as node size and colour. This example uses a callback to enable interactivity in the app.

Open the '***cytoscape_vis_advanced.py***' file. This example also imports the same two csv files for the node and edge data so you will need to set your working directory accordingly.

Resources

- <https://dash.plotly.com/cytoscape>
- <https://dash.plotly.com/introduction>
- <https://networkx.org/documentation/stable/index.html>

Further Info & Acknowledgement

Slides adapted from Sean Manzi

Checkout <https://www.project-nom.com> for more information and training on the use of network-based operational modelling for whole system modelling in healthcare