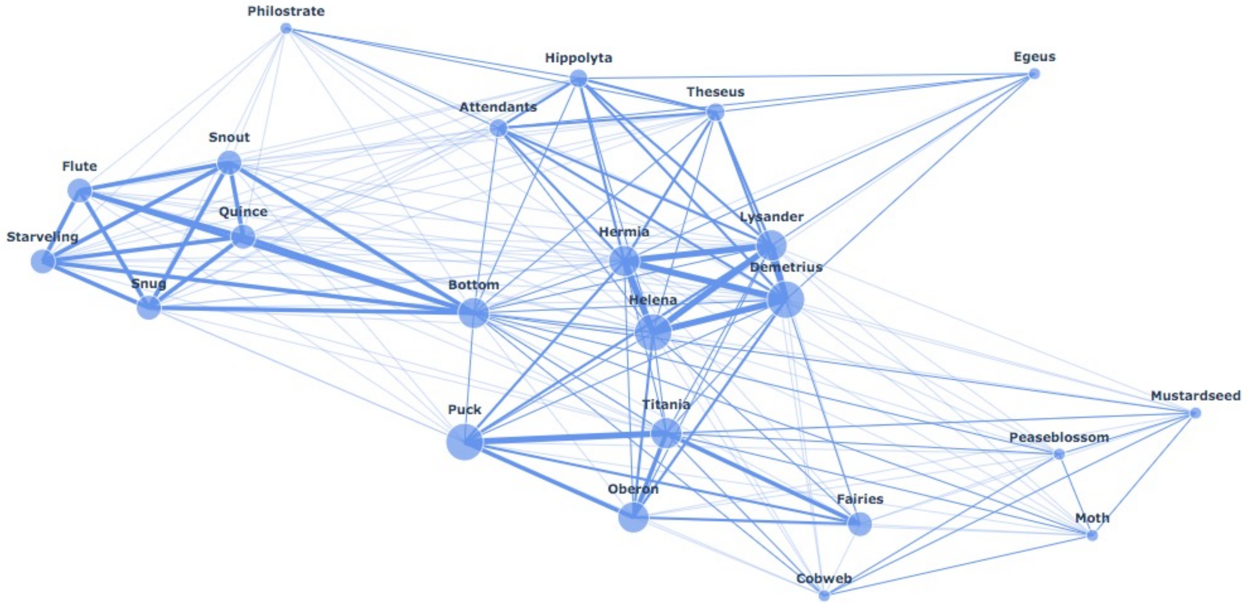
Intro to Graphs

NetworkX

Fulvio Corno
Giuseppe Averta
Carlo Masone
Francesca Pistilli



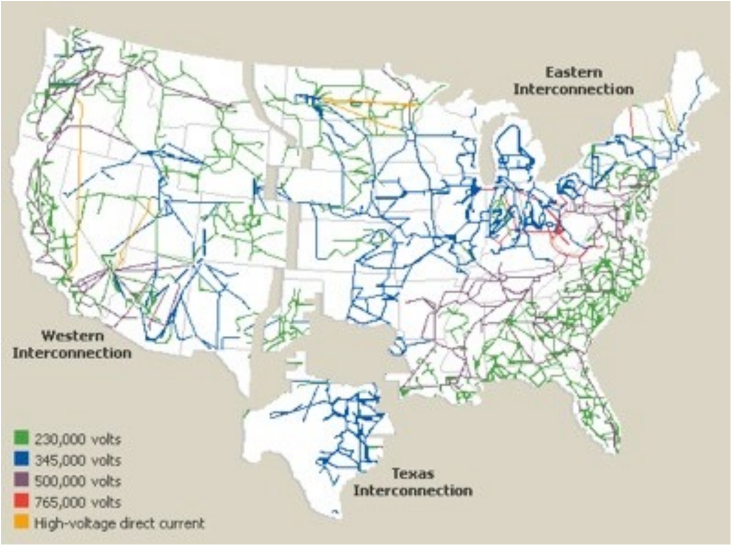
INTRODUCTION TO NETWORKX

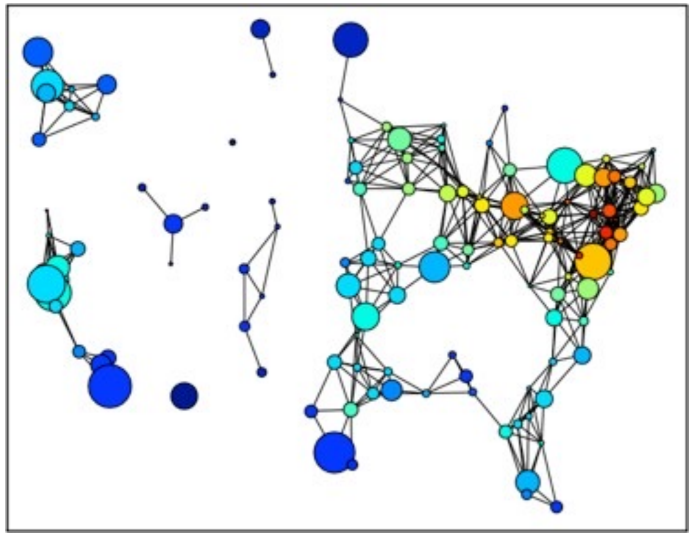
Introduction to NetworkX - network analysis

- Vast amounts of network data are being generated and collected
- Sociology: web pages, mobile phones, social networks
- Technology: Internet routers, vehicular flows, power grids

How can we analyse these networks?

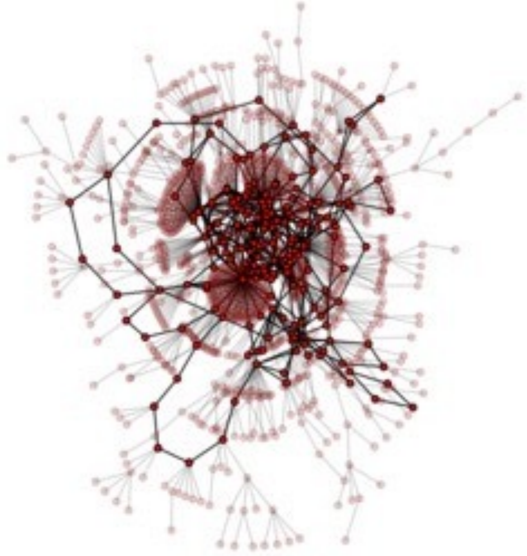
Python + NetworkX!



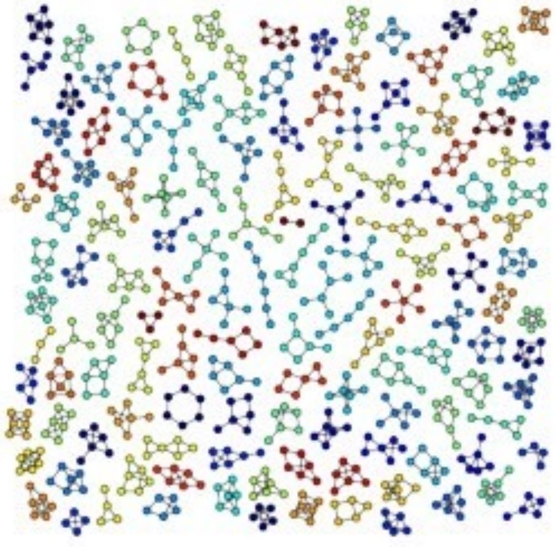


Introduction to NetworkX

“Python package for the creation, manipulation and study of the structure, dynamics and functions of complex networks.”



- Data structures for representing many types of data in the form of graphs
- Nodes can be any (hashable) Python object, edges can contain arbitrary data
- Flexibility ideal for representing networks found in many different fields
- Easy to install on multiple platforms
- Online up-to-date documentation
- First public release in April 2005



Introduction to NetworkX - design requirements

- Tool to study the structure and dynamics of social, biological, and infrastructure networks
- Ease-of-use and rapid development
- Open-source tool base that can easily grow in a multidisciplinary environment with non-expert users and developers
- An easy interface to existing code bases written in C, C++, and FORTRAN
- To painlessly slurp in relatively large nonstandard data sets

Introduction to NetworkX - object model

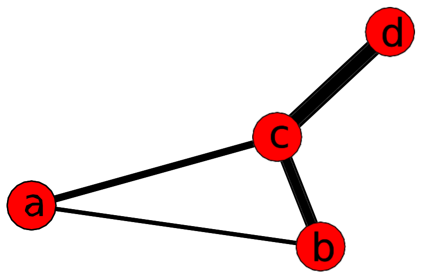
NetworkX defines no custom node objects or edge objects

- node-centric view of network
- nodes can be any hashable object, while edges are tuples with optional edge data (stored in dictionary)
- any Python object is allowed as edge data and it is assigned and stored in a Python dictionary (default empty)

Introduction to NetworkX - quick example

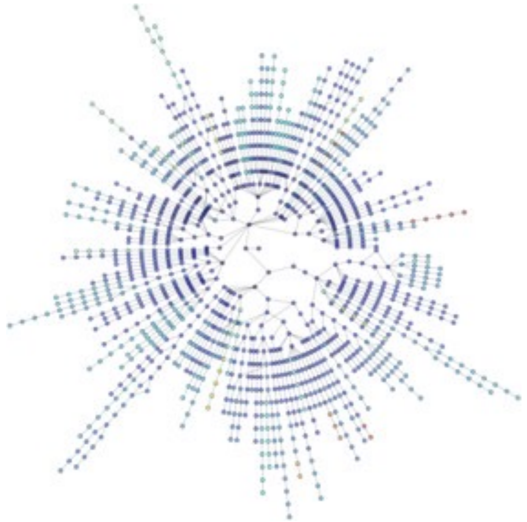
- Search for the shortest path in a weighted and unweighted network:

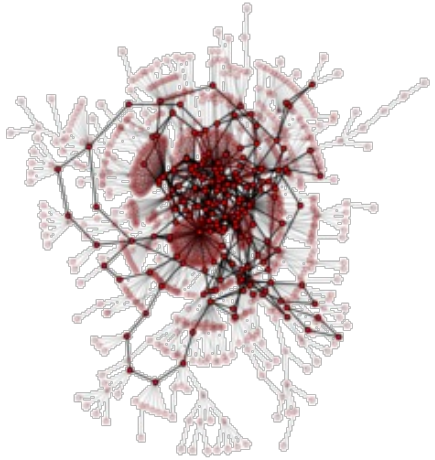
```python
import networkx as nx
g = nx.Graph()
g.add_edge("a","b",weight=1)
g.add_edge("b","c",weight=100)
g.add_edge("a","c",weight=1)
g.add_edge("c","d",weight=1)
print(nx.shortest_path(g,"b","d"))
print(nx.dijkstra_path(g, "b", "d", weight='weight'))
```

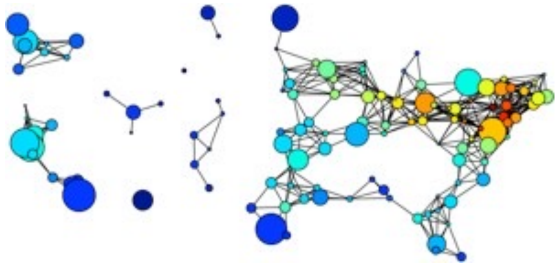


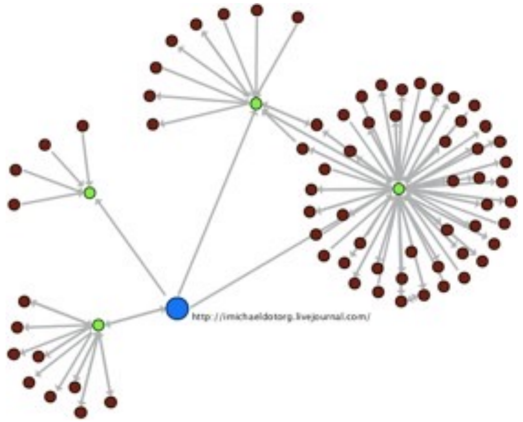
Introduction to NetworkX - drawing and plotting

- It is possible to draw small graphs within NetworkX and to export network data and draw with other programs (i.e., GraphViz, matplotlib)



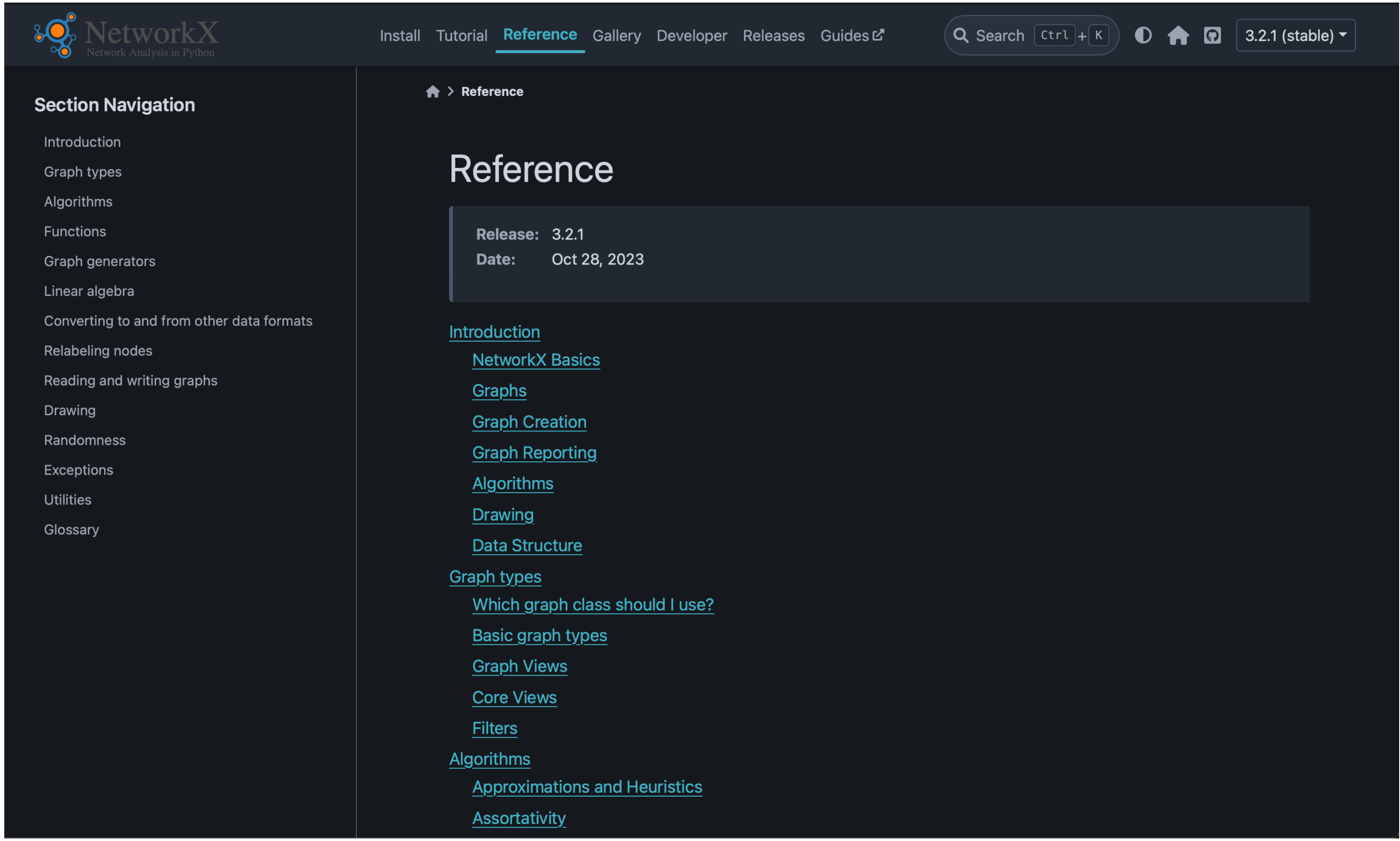






Introduction to NetworkX - official website

- https://networkx.org/



GETTING STARTED WITH PYTHON AND NETWORKX

Getting started – import NetworkX

- NetworkX supports many different graph types, like:
  - `nx.Graph()` – undirected
  - `nx.DiGraph()` – directed
  - `nx.MultiGrap()` – supports multiple edges between nodes
  - `nx.MultiGrap()` – directed multigraph
- Also provide implementation of notable graphs (like heawood)

```python
import networkx as nx
import math
import flet as ft

g = nx.heawood_graph()
print(g.nodes, g.edges)

g.add_node(math.cos)  # cosine functionx
g.add_node(ft.Text("Pippo"))
g.add_edge("math.cos", 3)
print(g.nodes, g.edges)
```

Getting started – build a graph

- Nodes could be (almost) anything
  - Numbers, strings
  - Objects
  - Functions
  - Flet containers
- Edges connect nodes (even heterogeneous)
- Nodes and edges could have attributes

```python
import networkx as nx
import math
import flet as ft

g = nx.Graph()
g.add_edge(1, 2)  # default edge data=1
g.add_edge(2, 3, weight=0.9)  # specify edge data

g.add_edge('y', 'x', function=math.cos)
g.add_node(math.cos)  # any hashable can be a node

elist = [(1, 2), (2, 3), (1, 4), (4, 2)]
g.add_edges_from(elist)
elist = [('a', 'b', 5.0), ('b', 'c', 3.0), ('a', 'c', 1.0), ('c', 'd', 7.3)]
g.add_weighted_edges_from(elist)
g.add_node(ft.Text("Pippo"))

print(g.nodes())
print(g.edges())
print(g.get_edge_data('a','b'))
```

Getting started – Data Structure

- A graph is essentially a "dictionary of dictionaries of dictionaries"
- The keys are the nodes
- Indeed, `g[n]` yields a dictionary where keys are all the nodes connected with n (adjacency) and values are the edges params (like weight)

```
import networkx as nx

g = nx.Graph()
g.add_edge(1, 2)  # default edge data=1
g.add_edge(2, 3, weight=0.9)  # specify edge data

elist = [(1, 2, 1), (2, 3, 1), (1, 4, 1), (4, 2, 1),
         ('a', 'b', 5.0), ('b', 'c', 3.0), ('a', 'c', 1.0), ('c', 'd', 7.3)]
g.add_weighted_edges_from(elist)

print(g[2])
```

```
{1: {'weight': 1}, 3: {'weight': 1}, 4: {'weight': 1}}

Process finished with exit code 0
```

Getting started – Data Structure

- `g[u][v]` yields the edge attributes
- `n in g` tests if node `n` is in `g`
- `for n in g:` iterates through the graph
- `for nbr in g[n]:` iterates through the neighbors of n
- Data struct for direct graphs is only slightly more complex (two dics, one for successors and one for predecessors)
- You can also use `g.nodes()` and `g.edges()` to get corresponding data
- Edges can have arbitrary attributes

```
import networkx as nx

g = nx.Graph()
g.add_edge(1, 2)  # default edge data=1
g.add_edge(2, 3, weight=0.9)  # specify edge data

elist = [(1, 2, 1), (2, 3, 1), (1, 4, 1), (4, 2, 1),
         ('a', 'b', 5.0), ('b', 'c', 3.0), ('a', 'c', 1.0), ('c', 'd', 7.3)]
g.add_weighted_edges_from(elist)
g.add_edge(2,5,arbitraryAttr = "foo")

print(g[2])
print("---------")
print(g['a']['b'])
print("---------")
print('e' in g)
print("---------")
for n in g:
    print (n)
print("---------")
for nbr in g[2]:
    print(nbr)
print("---------")
print(g[2][5]['arbitraryAttr'])
```

Getting started – Directed and Multi

- Graphs can be directed, therefore differentiating neighbors in predecessors and successors
- Two nodes can have more than one edge

```
import networkx as nx

dg = nx.DiGraph()
dg.add_weighted_edges_from([(1,4,0.5), (3,1,0.75)])

print([s for s in dg.successors(1)])
print([p for p in dg.predecessors(1)])

mg = nx.MultiGraph()

mg.add_weighted_edges_from([(1,2,.5), (1,2,.75),
(2,3,.5)])

print(mg[1][2])
```

Getting started - graph operators

Classic graph operations

- `subgraph(G, nbunch)` - induce subgraph of G on nodes in nbunch
- `union(G1,G2)` - graph union
- `disjoint_union(G1,G2)` - graph union assuming all nodes are different
- `compose(G1,G2)` - combine graphs identifying nodes common to both
- `complement(G)` - graph complement
- `create_empty_copy(G)` - return an empty copy of the same graph class
- `convert_to_undirected(G)` - return an undirected representation of G
- `convert_to_directed(G)` - return a directed representation of G

License

- These slides are distributed under a Creative Commons license "**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**"
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for commercial purposes.
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- https://creativecommons.org/licenses/by-nc-sa/4.0/