# Table of Contents



# CS4423-Networks: Week 12 (2+3 April 2025)

# Part 1: Bow Tie Components


Niall Madden, School of Mathematical and Statistical Sciences
University of Galway


This Jupyter notebook, and PDF and HTML versions, can be found at https://www.niallmadden.ie/2425-CS4423/#Week12

*This notebook was adapted by Niall Madden from one developed by Angela Carnevale.*

In [1]:
```
import networkx as nx
import numpy as np
opts = { "with_labels": True,  "node_color": "#224488", "font_color": "white", "arrows

import matplotlib.pyplot as plt

np.set_printoptions(precision=2)    # just display arrays to 2 decimal places
np.set_printoptions(suppress=True)

from collections import Counter
```

## Digraphs and Mathematical Relations

When a directed graph $G$ is regarded as a **relation** on the set $X$, strongly connected components can be described as the **equivalence classes** of an equivalence relation, as we'll now explain.

First recall that

- $x \to y$ means that there is a (directed) edges from $x$ to $y$
- $x \rightsquigarrow y$ means that there is a path from $x$ to $y$.

We can see (right?) that the relation $x \rightsquigarrow y$ is the reflexive and transitive closure of the edge relation $x \to y$.

Thus, by construction it is reflexive and transitive.

(There might be nodes $x$ and $y$ with $x \rightsquigarrow y$ and $y \rightsquigarrow x$, though it won't be all of them).

So this allows us to define a new relation as follows.

**Definition.** We set $x \equiv y$ if $x \rightsquigarrow y$ *and* $y \rightsquigarrow x$.

This **is** an equivalence relation we get equivalence classes that partition our graph. These equivalence classes are the **strongly connected components** of $G$. We denote the class of $x \in X$ by $[x]$.

Moreover, there is a partial order relation $\leq$ (a relation which is reflexive, transitive and anti-symmetric) on the set of equivalence classes:

$[x] \leq [y]$ if $x \rightsquigarrow y$.

We can say something about the structure of the WWW in terms of these equivalence classes and of the partial order on them.

## The Bow-Tie Structure of the WWW

Like the giant component in a simple graph, it turns out that a directed graph with sufficiently many edges has a **giant SCC**.

The remainder of the graph consists of four more sets of components of nodes, as follows:

1. IN: upstream components, the set of all components $C$ with $C <$ SCC.
2. OUT: downstream components, the set of all components $C$ with $C >$ SCC.
3. tendrils: the set of all components $C$ with either $C >$ IN and $C \not<$ OUT or $C <$ OUT and $C \not>$ IN; and tubes: components $C$ with $C >$ IN, $C <$ OUT but $C \not<$ SCC.
4. disconnected components.

Thus, in any directed graph with a distinguished SCC, the WCC in which it is contained necessarily has the following global bow-tie structure:

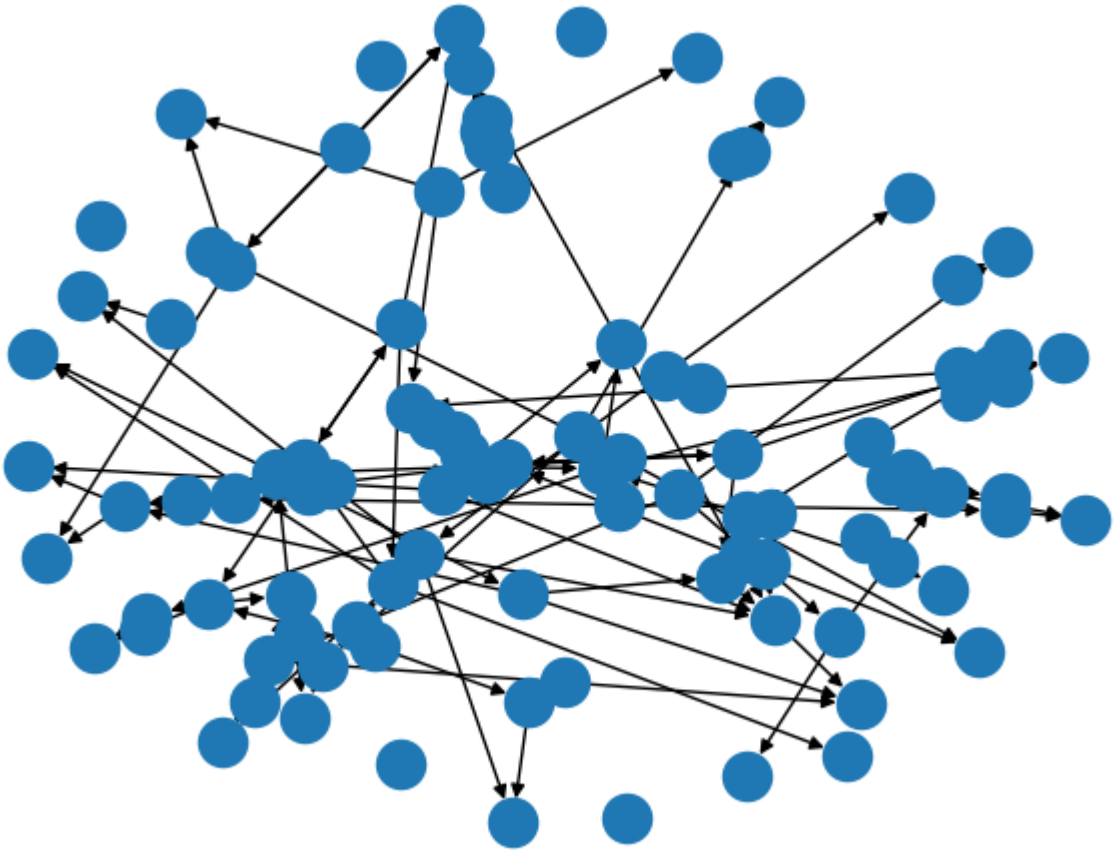
Research on available data on the Web in 1999 has confirmed this bow tie structure for the World Wide Web, with a **large Giant SCC** covering less than $\frac{1}{3}$ of the vertex set, and the three parts IN, OUT and the tendrils and tubes roughly of the same size. One can assume that this proportion has not changed much over time, although the advent of social media has brought many new types of links and documents to the Web.

# Computing Bow-Tie Components

**Example.** Let's start with a reasonably large random **directed graph**, using the Erdős-Rényi $G(n, m)$ model:

In [2]:
```
n, m = 100, 120
G = nx.gnm_random_graph(n, m, directed=True)
```

In [3]:
```
nx.draw(G)
```



## Weakly Connected Components

The weakly connected components of a directed graph $G$ can be determined by BFS, as before, counting as "neighbors" of a node $x$ **both** its *successors* and it *predecessors* in the graph.

A single component, the weakly connected component of node $x$, is found as follows.

In [4]:
```
def weak_component(G, x):
    nodes = {x}
    queue = [x]
    for y in queue:
        G.nodes[y]["seen"] = True
        for z in set(G.successors(y)) | set(G.predecessors(y)): ## preds+succs are the
            if z not in nodes:
                nodes.add(z)
                queue.append(z)
    return nodes
```

The list of all weakly connected components is computed by looping over all the nodes of `G`, computing the components of "unseen" nodes and collecting them in a list. The final result is sorted by decreasing length before it is returned.

In [5]:
```
def weak_components(G):
    wccs = []     # initialize
```

```
    # find each node's wcc
    for x in G:
        if not G.nodes[x].get("seen"):
            wccs.append(weak_component(G, x))

    # clean up after yourself
    for x in G:
        del G.nodes[x]["seen"]

    # return sorted list of wccs
    return sorted(wccs, key=len, reverse=True)
```

Note let's check the number of Weakly Connected Components, and their order:

In [6]:
```
wccs = weak_components(G)
print(f"G has {len(wccs)} weakly connected components")
```

```
G has 7 weakly connected components
```

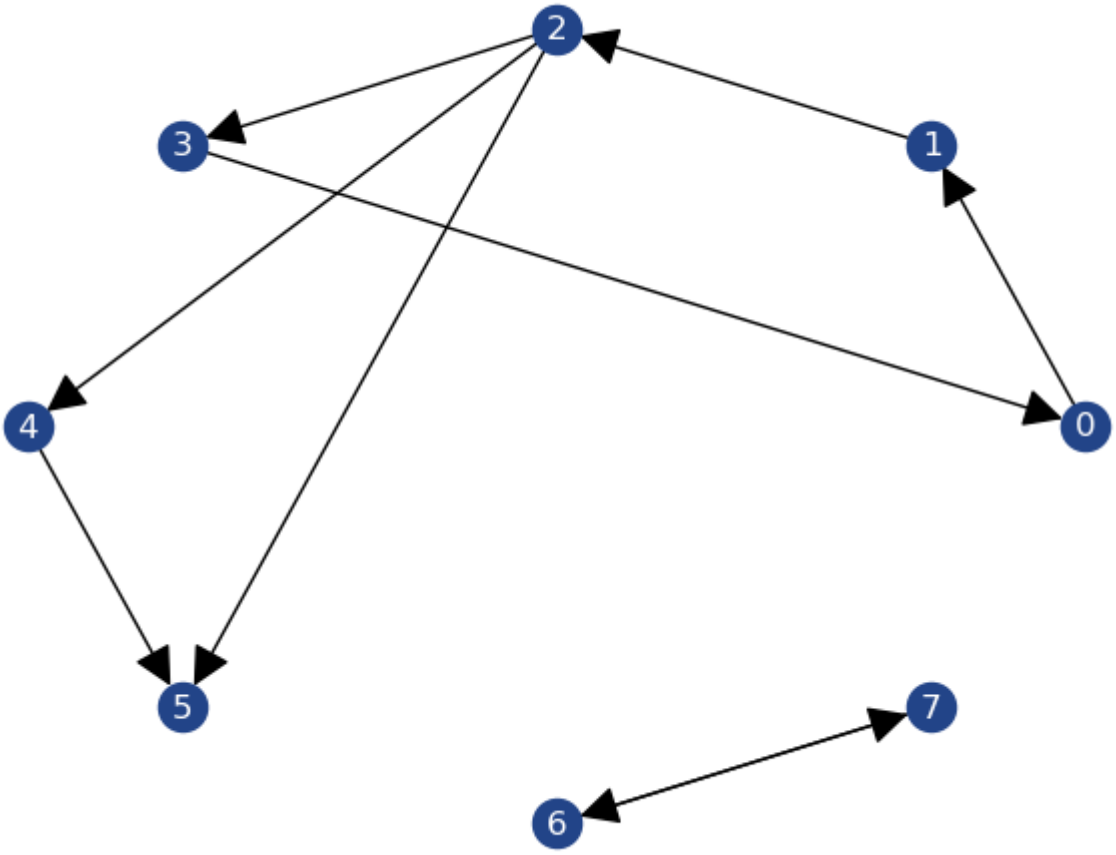
In [7]:
```
[len(c) for c in wccs]
```

Out[7]:
```
[93, 2, 1, 1, 1, 1, 1]
```

## Computing WCCs and SCCs in `networkx`

As you might expect, there are algorithms for doing this in `networkx`. Let's try them, but first recall a variant on an examples from last week:

In [8]:
```
G = nx.DiGraph([(0, 1), (1, 2), (2, 3), (3,0), (2,4),(2,5),(4,5), (6,7), (7,6)])
nx.draw_circular(G, **opts)
```




## WCCS

We should have been able to see there are 2 WCCs:

In [9]:
```
C = nx.weakly_connected_components(G)   # returns and iterable
print(f"There are {len(list(C))} WCCs:")
for c in nx.weakly_connected_components(G):
    print(c)
```

```
There are 2 WCCs:
{0, 1, 2, 3, 4, 5}
{6, 7}
```

## Strongly Connected Components

**Strongly** connected components are efficiently found by DFS. Tarjan's Algorithm cleverly uses recursion and an additional stack for this.

Have a look at the Wiki page. We'll use `networkx`:

In [10]:
```
C = nx.strongly_connected_components(G)   # returns and iterable
print(f"There are {len(list(C))} WCCs:")
for c in nx.strongly_connected_components(G):
    print(c)
```

```
There are 4 WCCs:
{5}
{4}
{0, 1, 2, 3}
{6, 7}
```

## $G_{ER}$

We'll finish by checking the size of the components of a graph in $G_{ER}(n,p)$

In [11]:
```
n, m = 100, 120
G = nx.gnm_random_graph(n, m, directed=True)
```

In [12]:
```
C = nx.weakly_connected_components(G)   # returns and iterable
print(f"There are {len(list(C))} WCCs:")
k=-1
for c in sorted(nx.weakly_connected_components(G), key=len, reverse=True):
    if (len(c)>1):
        k+=1
        print(f"Component {k} has {len(c)}")
print(f"Other {n-k-1} components have order 1")
```

```
There are 16 WCCs:
Component 0 has 83
Component 1 has 2
Component 2 has 2
Other 97 components have order 1
```

For SCCs, we get:

In [13]:
```
C = nx.strongly_connected_components(G)   # returns and iterable
print(f"There are {len(list(C))} SCCs:")
k=-1
for c in sorted(nx.strongly_connected_components(G), key=len, reverse=True):
    if (len(c)>1):
        k+=1
        print(f"Component {k} has {len(c)}")
print(f"Other {n-k-1} components have order 1")
```

```
There are 95 SCCs:
Component 0 has 6
Other 99 components have order 1
```