LAB 5: WORKING WITH GRAPHS & ADVANCED DEBUGGING

University of Washington

ECE 241

Author: Jimin Kim (jk55@uw.edu)
Version: v1.7.0

OUTLINE

Part 1: Introduction to graphs

- Fundamental graph elements
- Graph degrees
- Paths and connectedness
- Removing or Adding vertices/edges

Part 2: Application of graph data structure

- Brain connectome
- Social network
- Power grid
- Knowledge graph

Part 3: Working with Graph data

- Graphs as a data structure
- Visualizing graphs
- Computing graph properties
- Removing or adding edges

Part 4: Advanced debugging

- Introduction to Python Debugger
- Python Debugger Basics
- Useful Commands

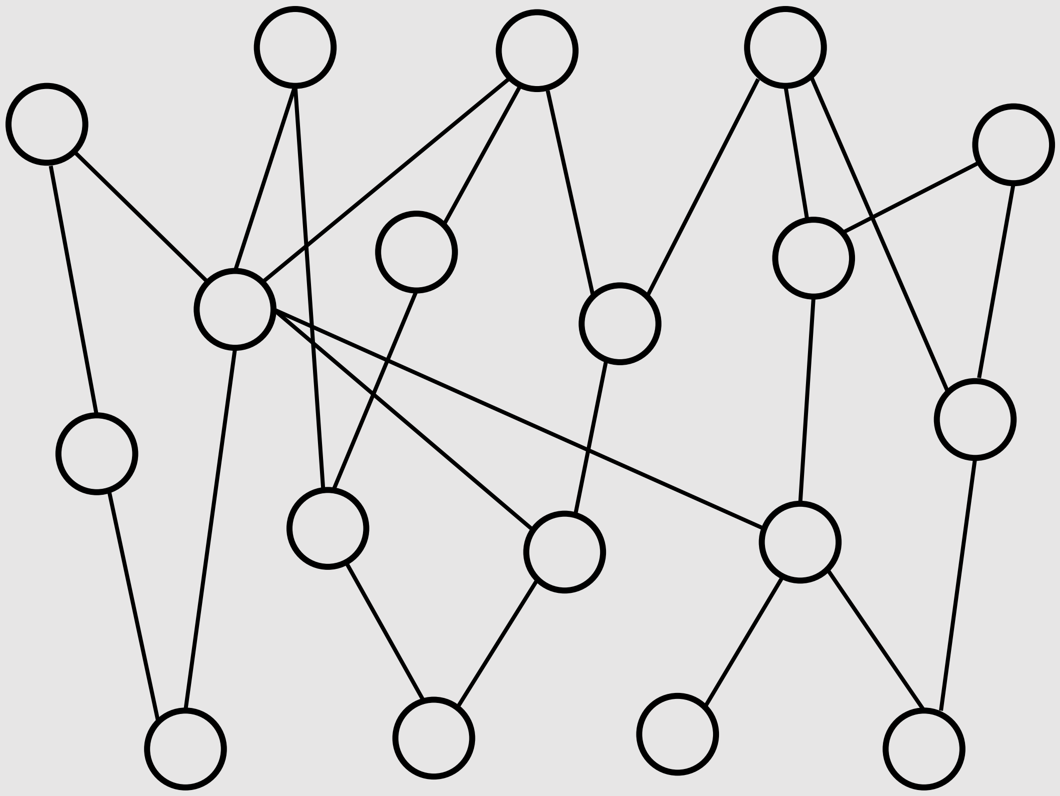
Part 5: Lab Assignments

- **Exercise 1 – 5**

PART 1: INTRODUCTION TO GRAPHS

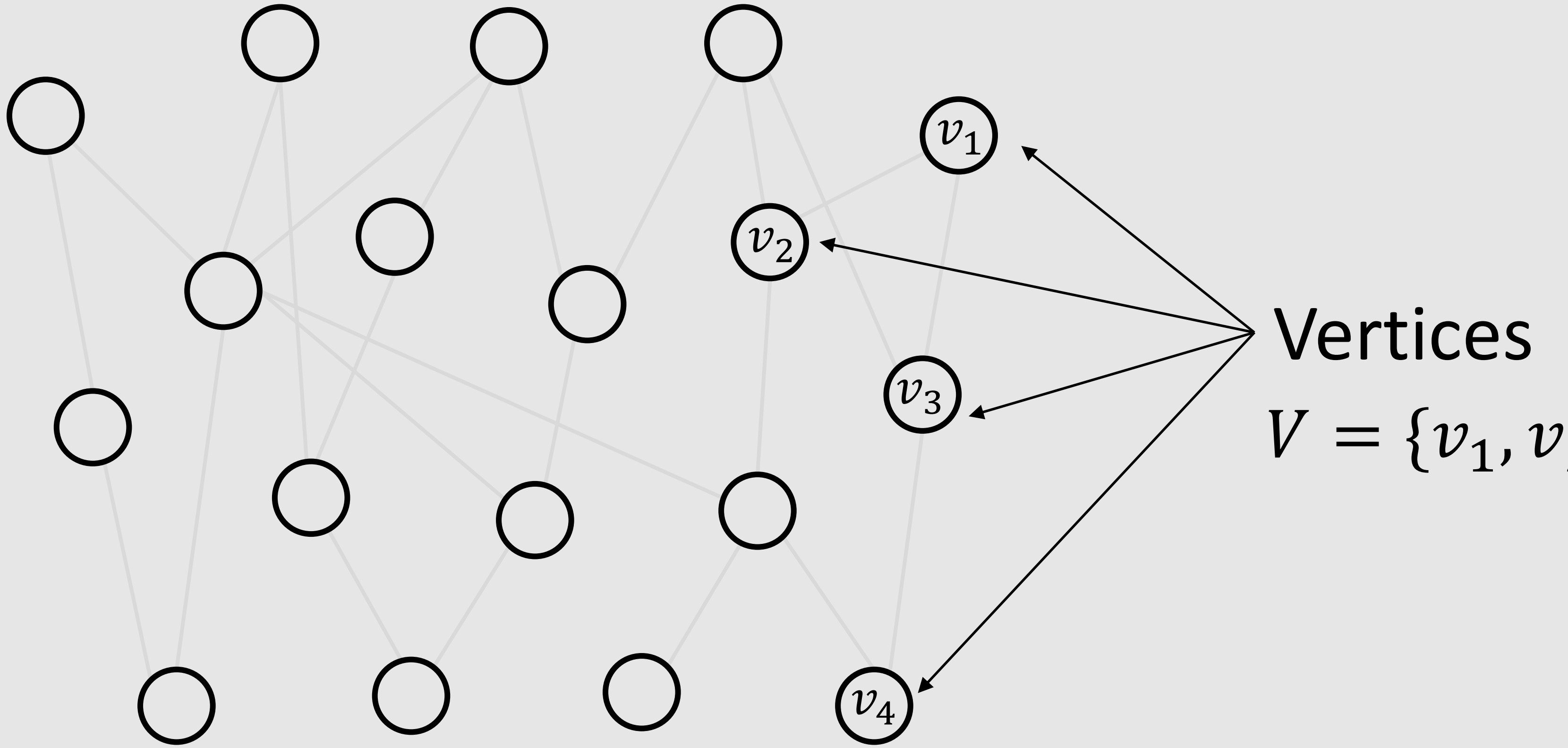
FUNDAMENTAL GRAPH ELEMENTS

$$G = (V, E)$$



FUNDAMENTAL GRAPH ELEMENTS

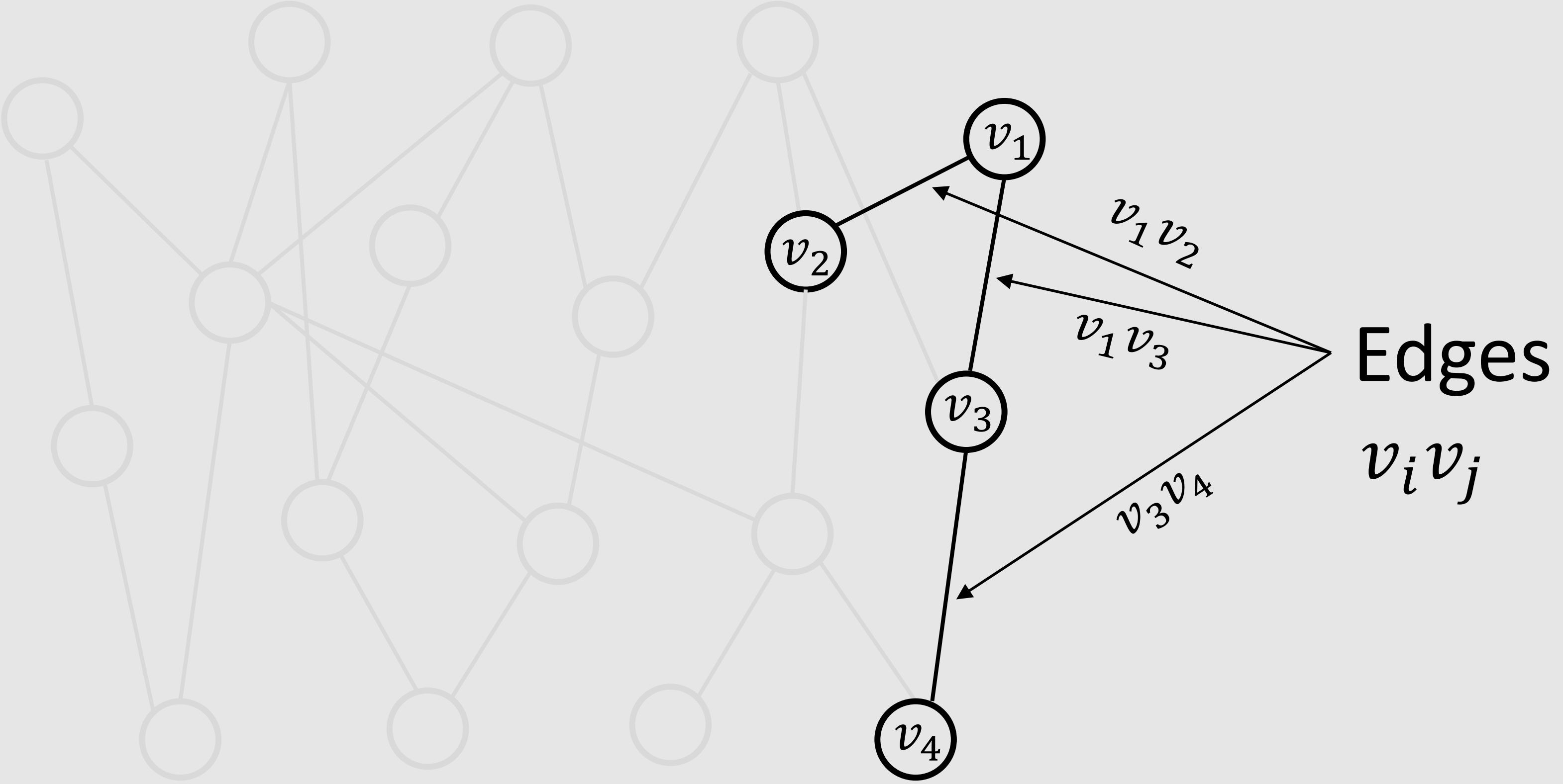
$G = (V, E)$



$V = \{v_1, v_2, \dots, v_n\}$

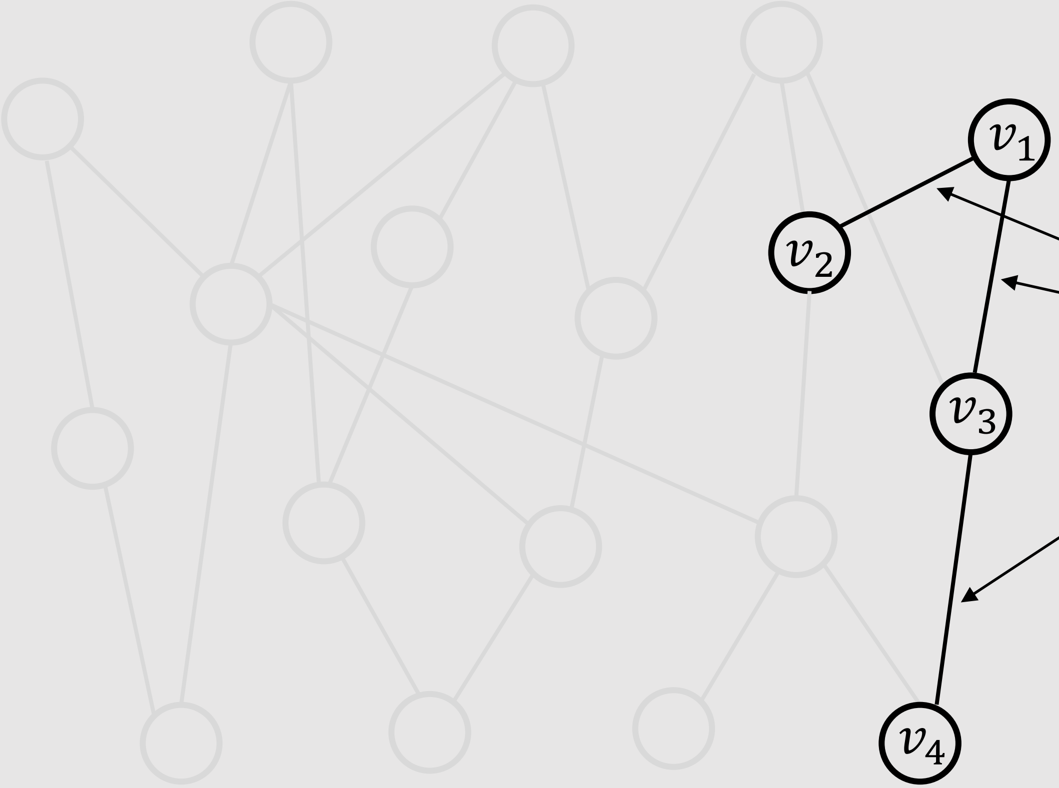
FUNDAMENTAL GRAPH ELEMENTS

$$G = (V, E)$$



FUNDAMENTAL GRAPH ELEMENTS

$$G = (V, E)$$



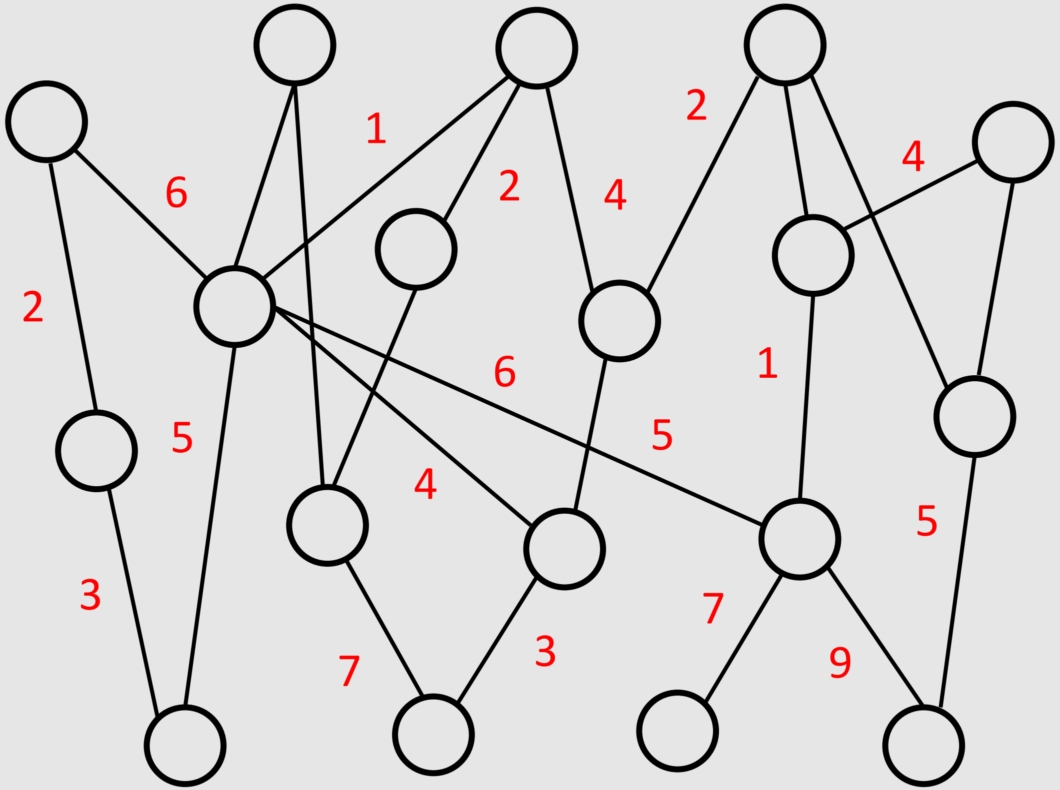
Edges

$v_i v_j$

$E = \{v_1 v_2, v_1 v_3, \ldots, v_m v_n\}$

FUNDAMENTAL GRAPH ELEMENTS

$$G = (V, E)$$

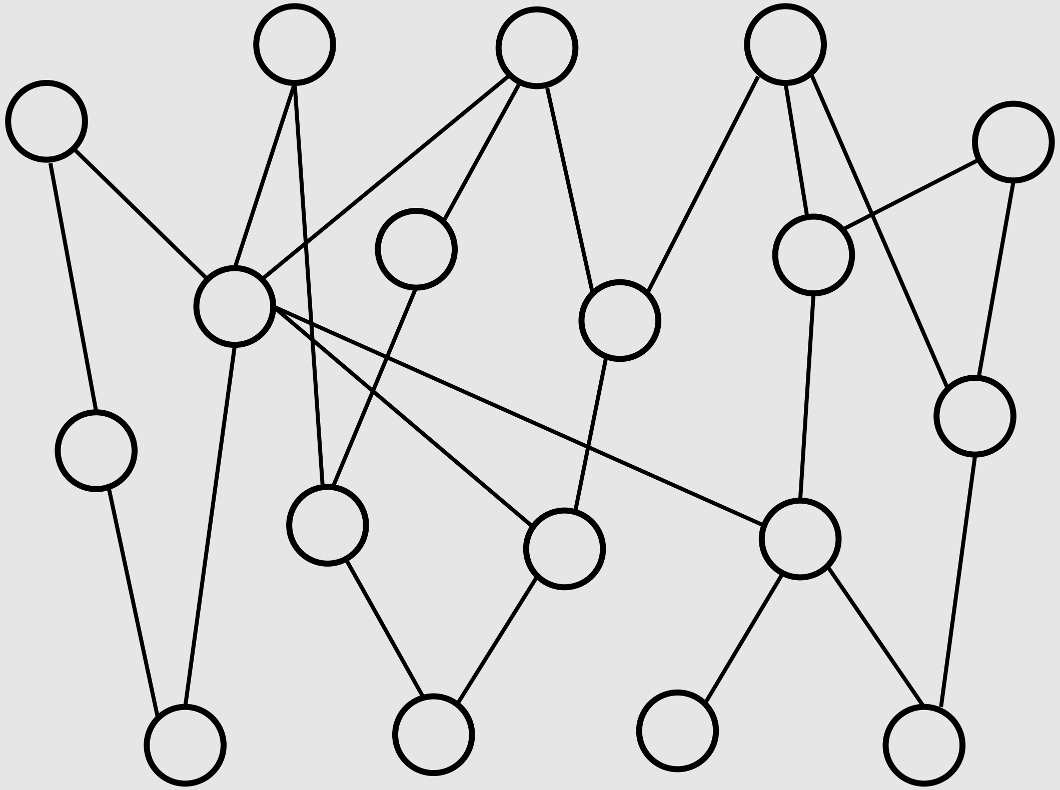


Edges can have values attached to them to represent:

- Number or weight of connections between vertices
- Probability of an event from another event (vertex = event)
- Relationship between vertices (i.e. predicate)
- etc…

FUNDAMENTAL GRAPH ELEMENTS

$$G = (V, E)$$

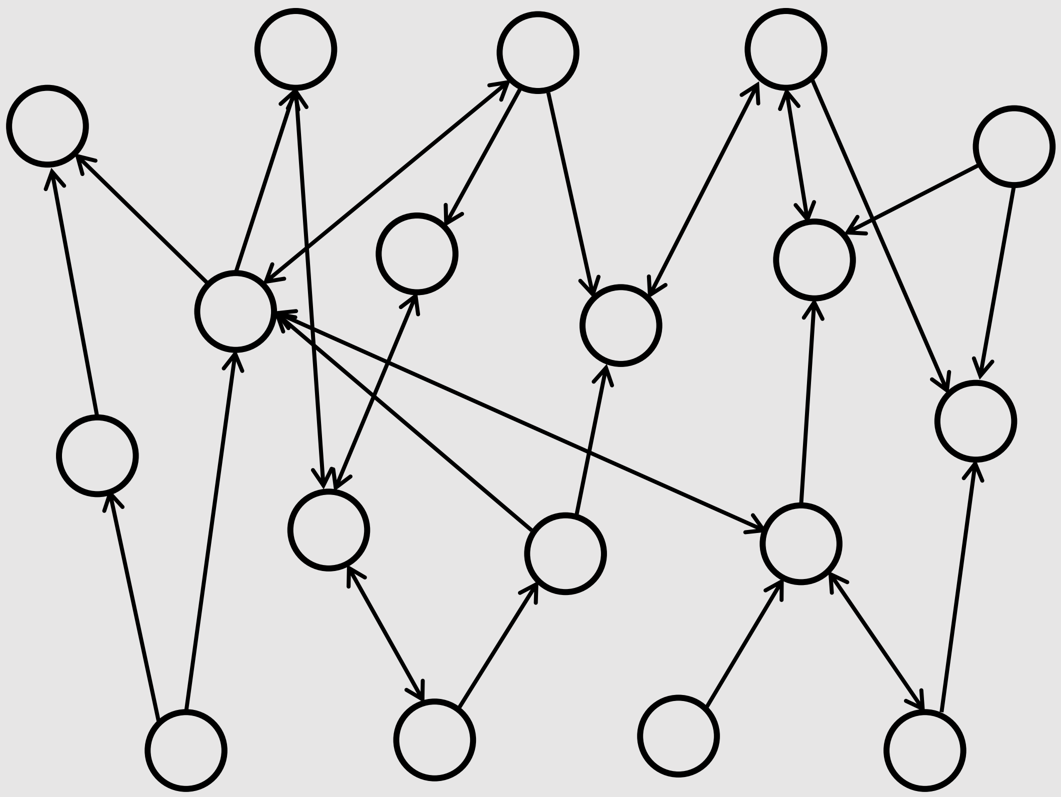


Edges can be:

- Undirected (i.e. bidirectional)

FUNDAMENTAL GRAPH ELEMENTS

$$G = (V, E)$$



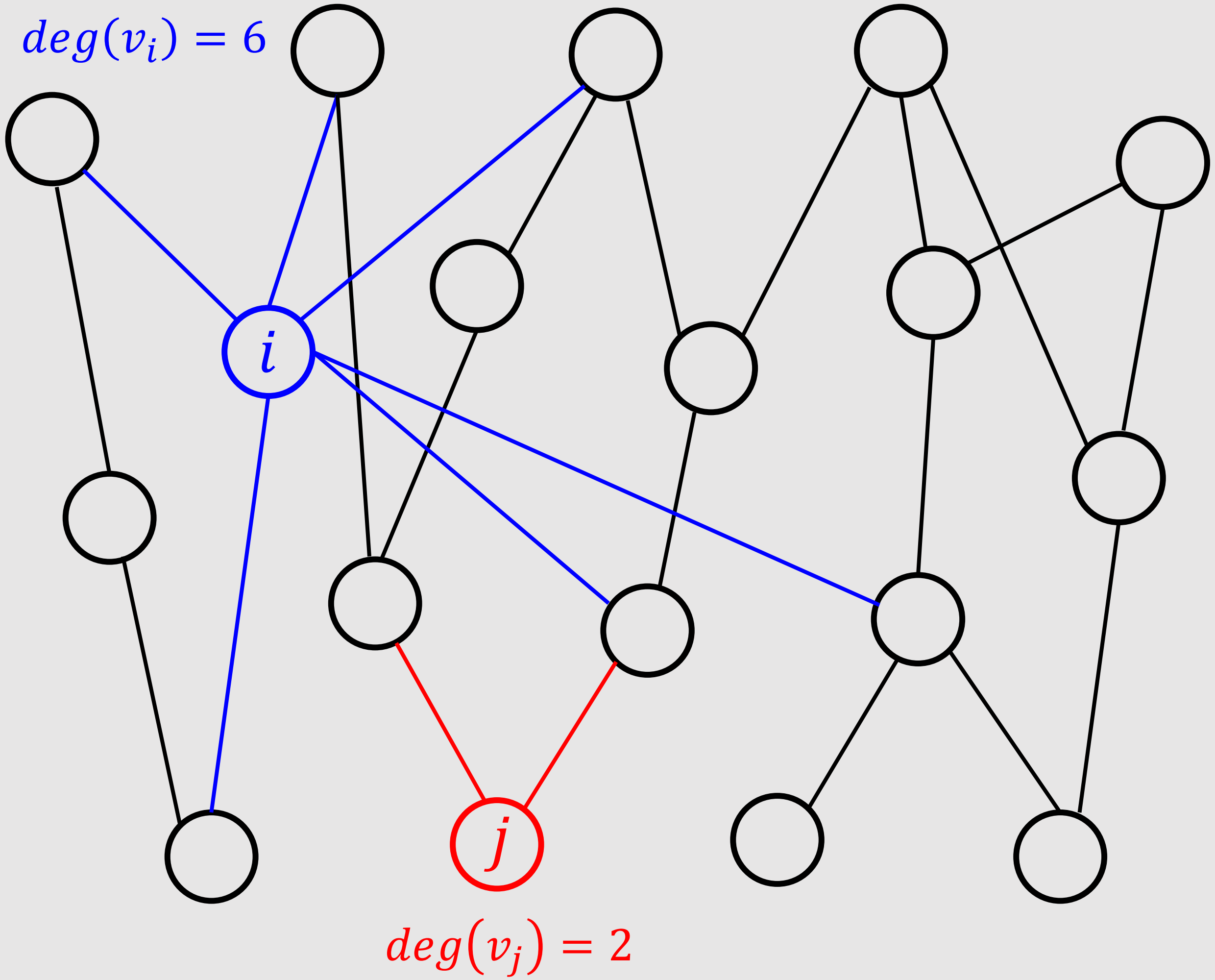
Edges can be:

- Undirected (i.e. bidirectional)
- Directed (i.e. unidirectional)

Note:

Directed graph can have bidirectional connection between two vertices via having two edges A → B and B → A

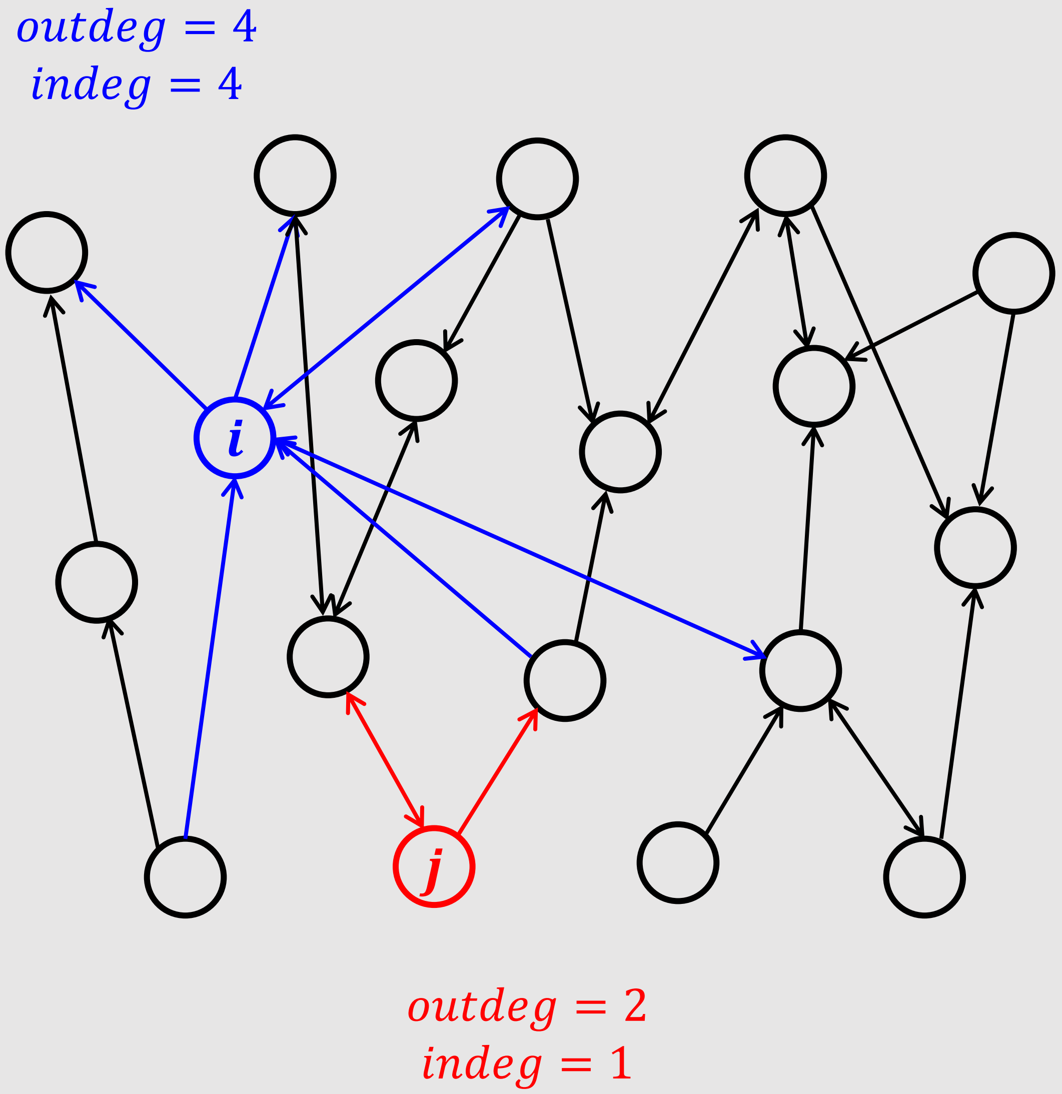
GRAPH DEGREES



Undirected graphs

Degree of a vertex v_i = number of edges or sum of edge weights **incident** to the vertex

GRAPH DEGREES

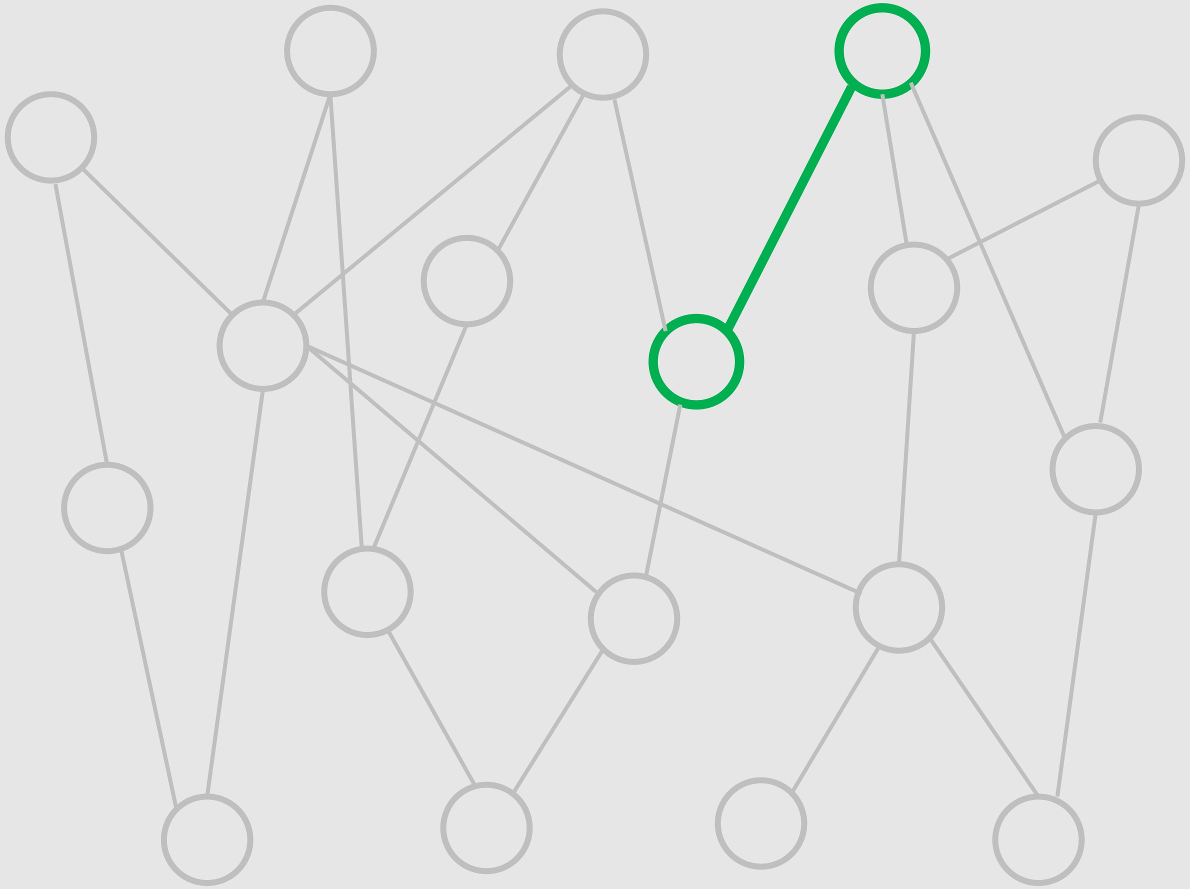


Directed graphs

Out-degree: number of incident tail ends or outgoing edge weights

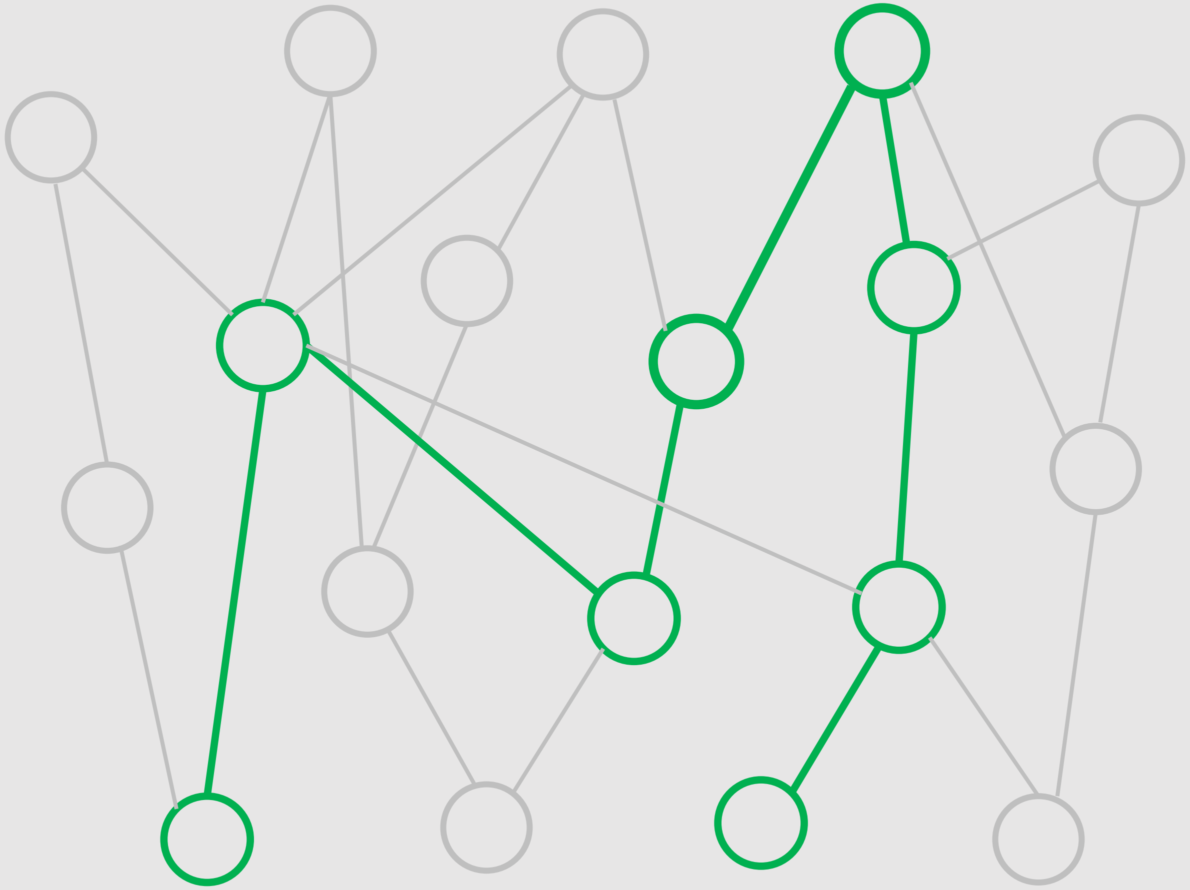
In-degree: number of incident head ends or incoming edge weights

PATHS AND CONNECTEDNESS



Two vertices connected by an edge are said to be **adjacent**

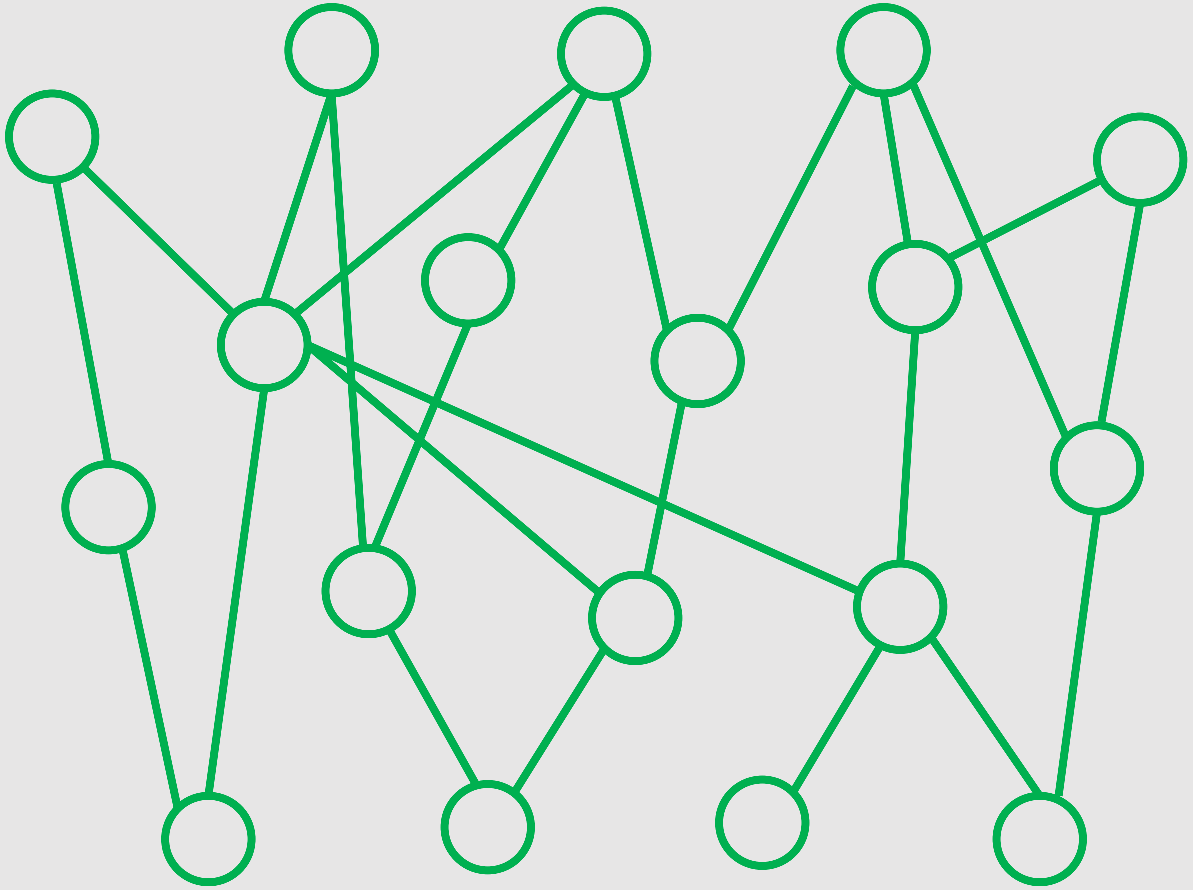
PATHS AND CONNECTEDNESS



A **path** of length ***m*** in ***G***:

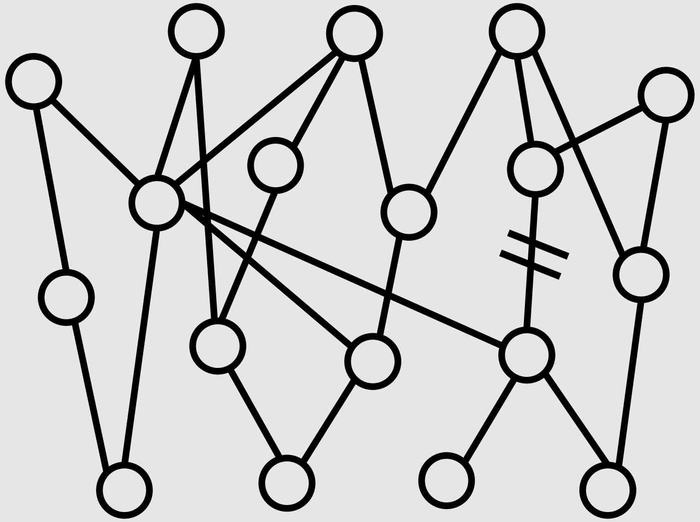
A sequence of distinct adjacent vertices

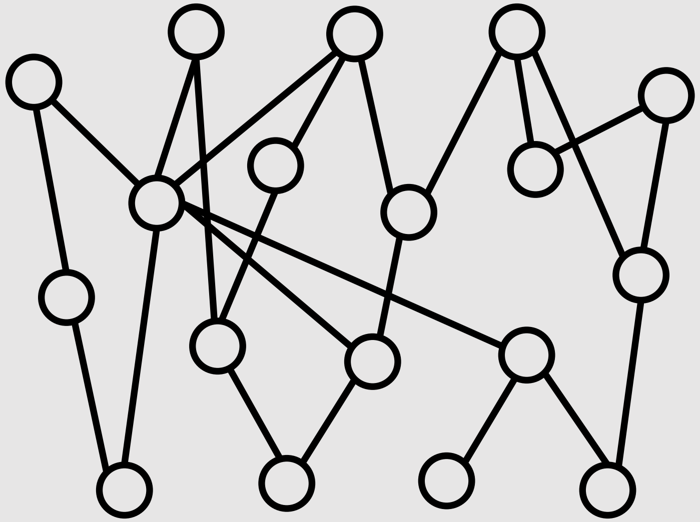
PATHS AND CONNECTEDNESS



A graph is **connected** if **all pairs of vertices** are connected by a **path**

REMOVING VERTICES/EDGES

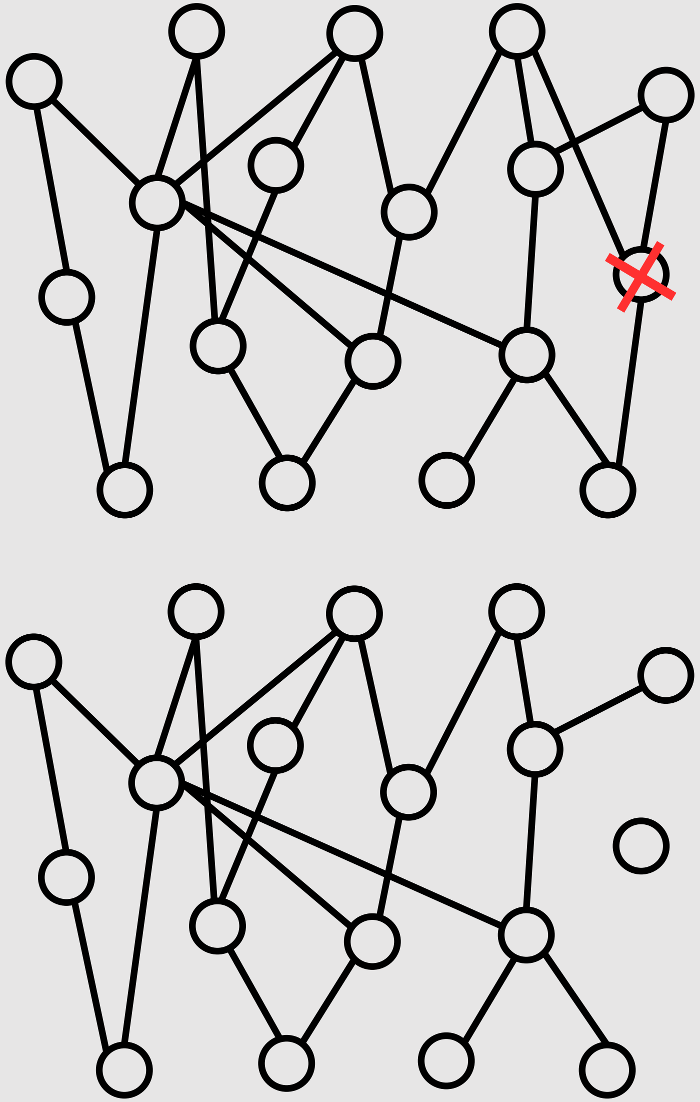




Both **vertices** and **edges** can be removed from a graph

Removing an edge **disconnects a connection** between 2 vertices

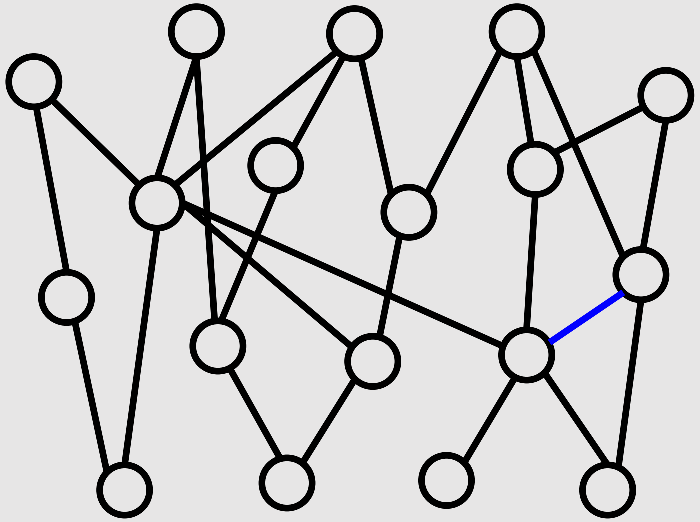
REMOVING VERTICES/EDGES

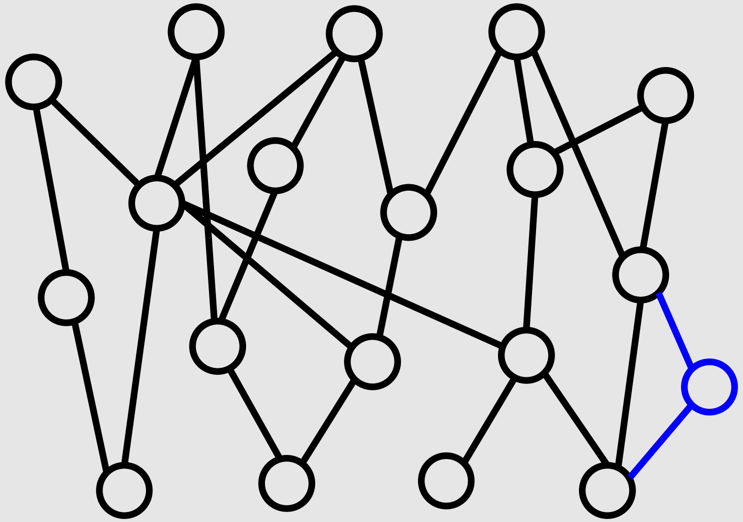


Both **vertices** and **edges** can be removed from a graph

Removing a vertex removes the **all the edges** (both **incoming and outgoing** if directed) that connects to it

ADDING VERTICES/EDGES

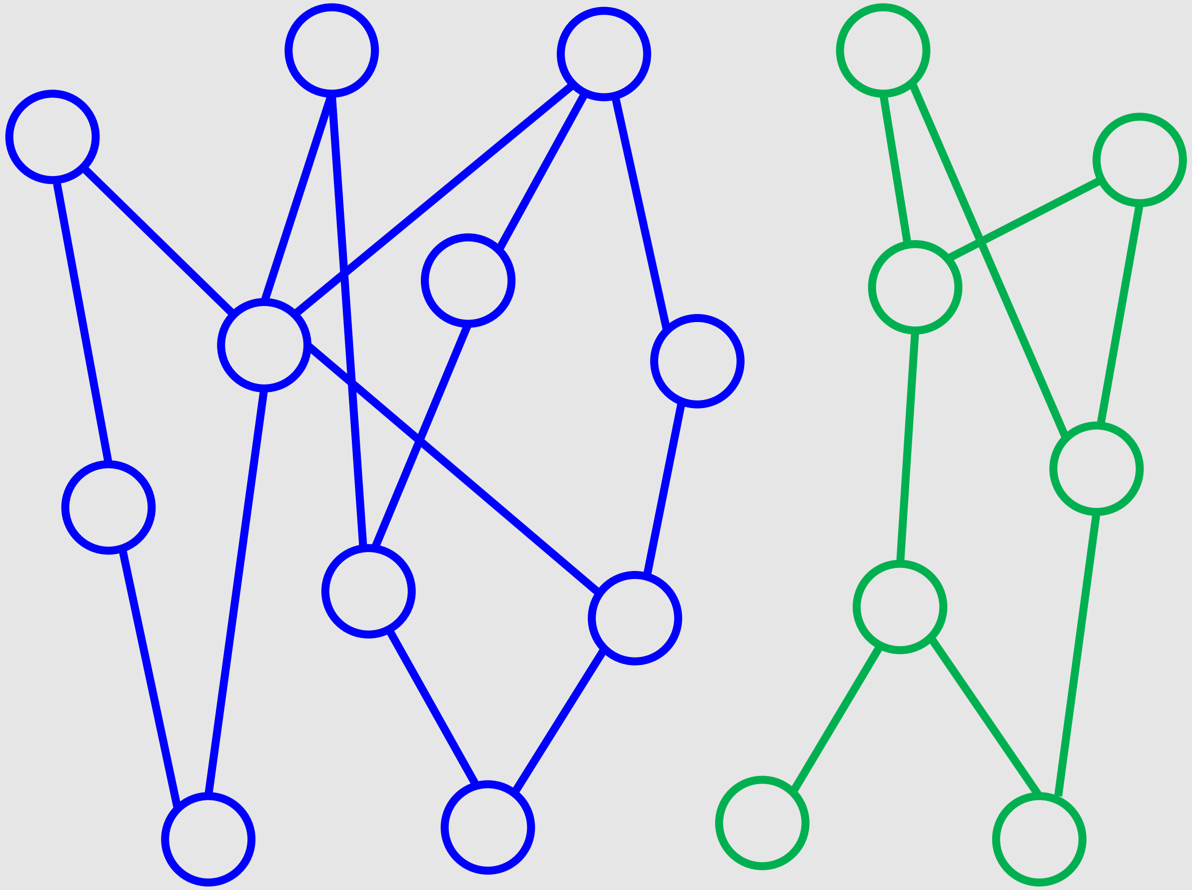




One can also ADD new **edges** or **vertices** into a graph

Adding a vertex **also adds edges** that connects the new vertex to the graph

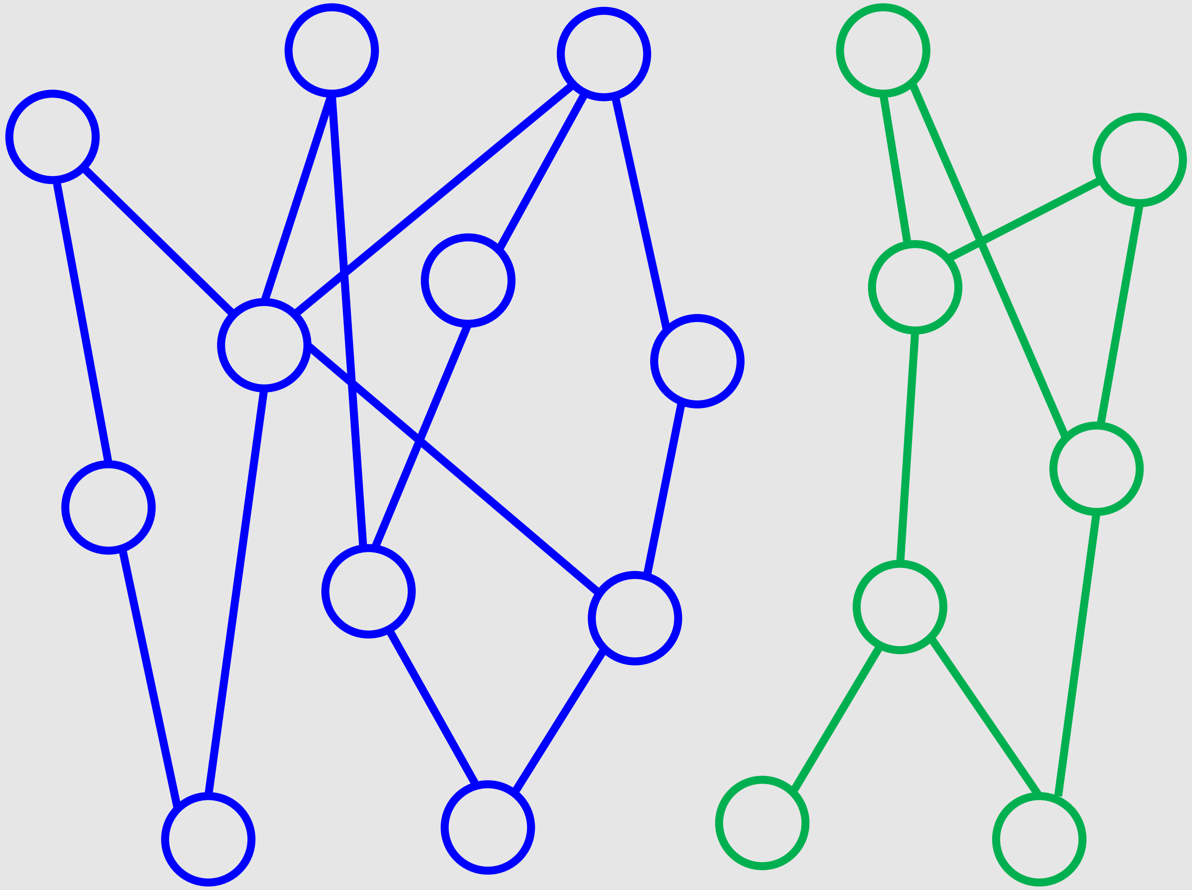
PATHS AND CONNECTEDNESS



Removing edges or vertices may result in two separate graphs that are **not connected by any path**

In such a case, a graph is said to be **Disconnected**

PATHS AND CONNECTEDNESS



Component 1 Component 2

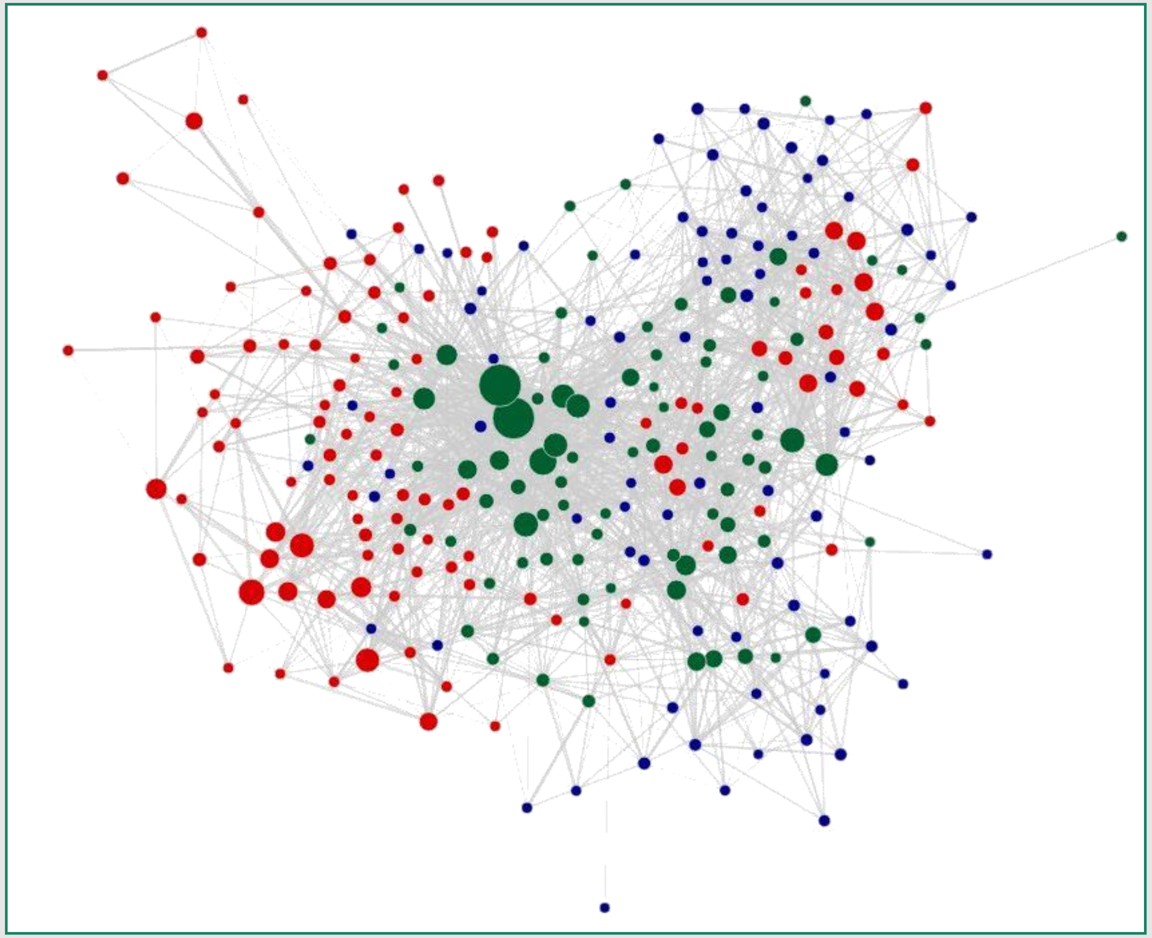
Removing edges or vertices may result in two separate graphs that are **not connected by any path**

In such a case, a graph is said to be **Disconnected**

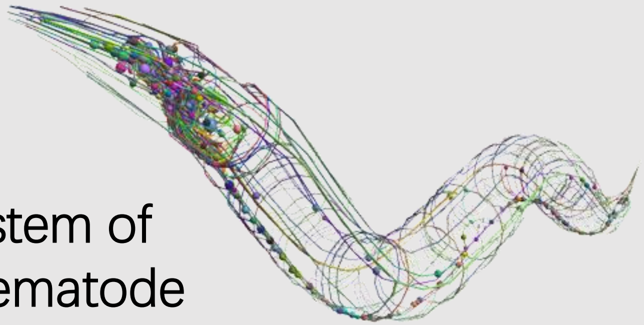
Each disconnected graph is called a **'Component'**

PART 2:
APPLICATIONS OF GRAPH STRUCTURES

BRAIN CONNECTOME



Synaptic Connectome
(Weighted Directed Network)



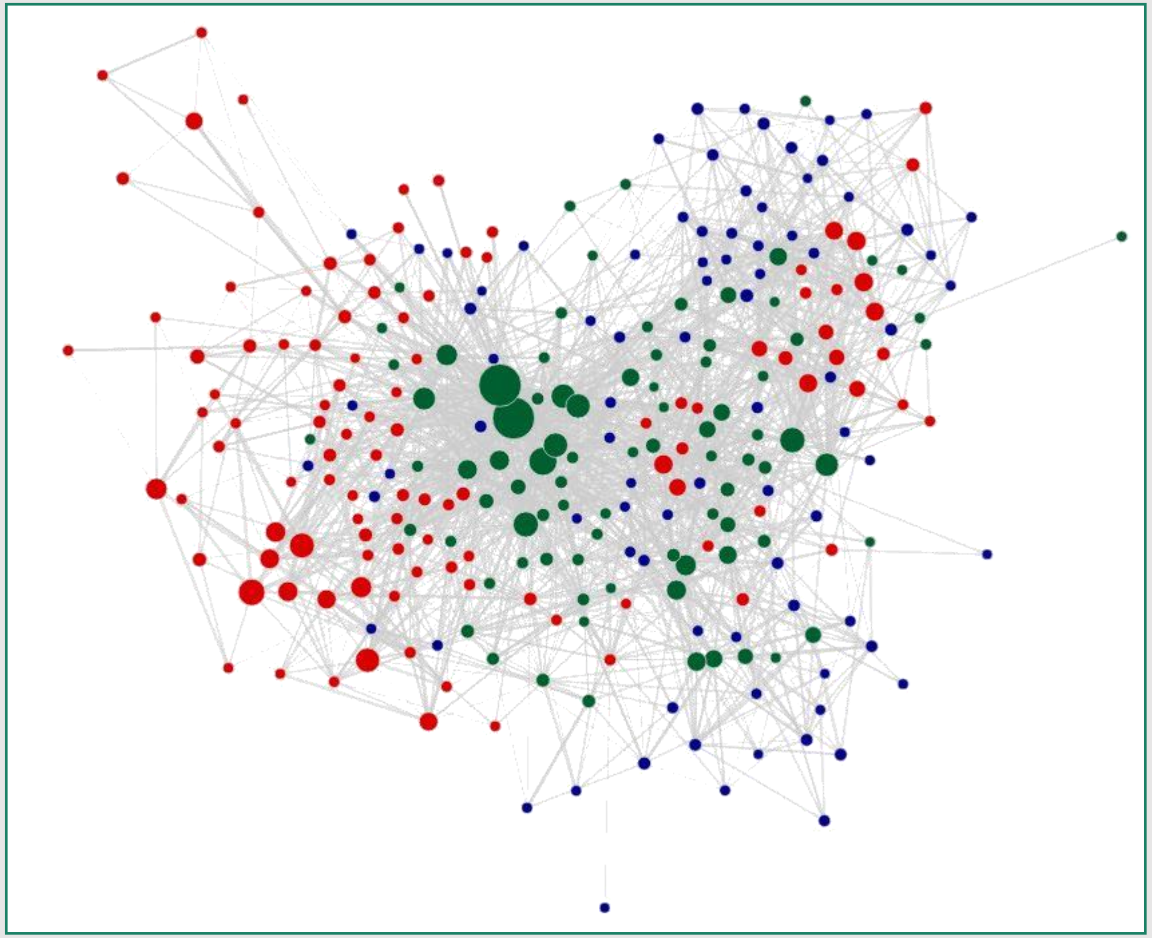
Nervous System of
C. elegans Nematode

Vertices:
Neurons

Edges:
Synaptic connection

Image credit: Jimin Kim, Will Leahy, Eli Shlizerman,
Neural Interactome: Interactive Simulations of Neuronal Networks
Frontiers in Computational Neuroscience, 2019

BRAIN CONNECTOME



Synaptic Connectome
(Weighted Directed Network)

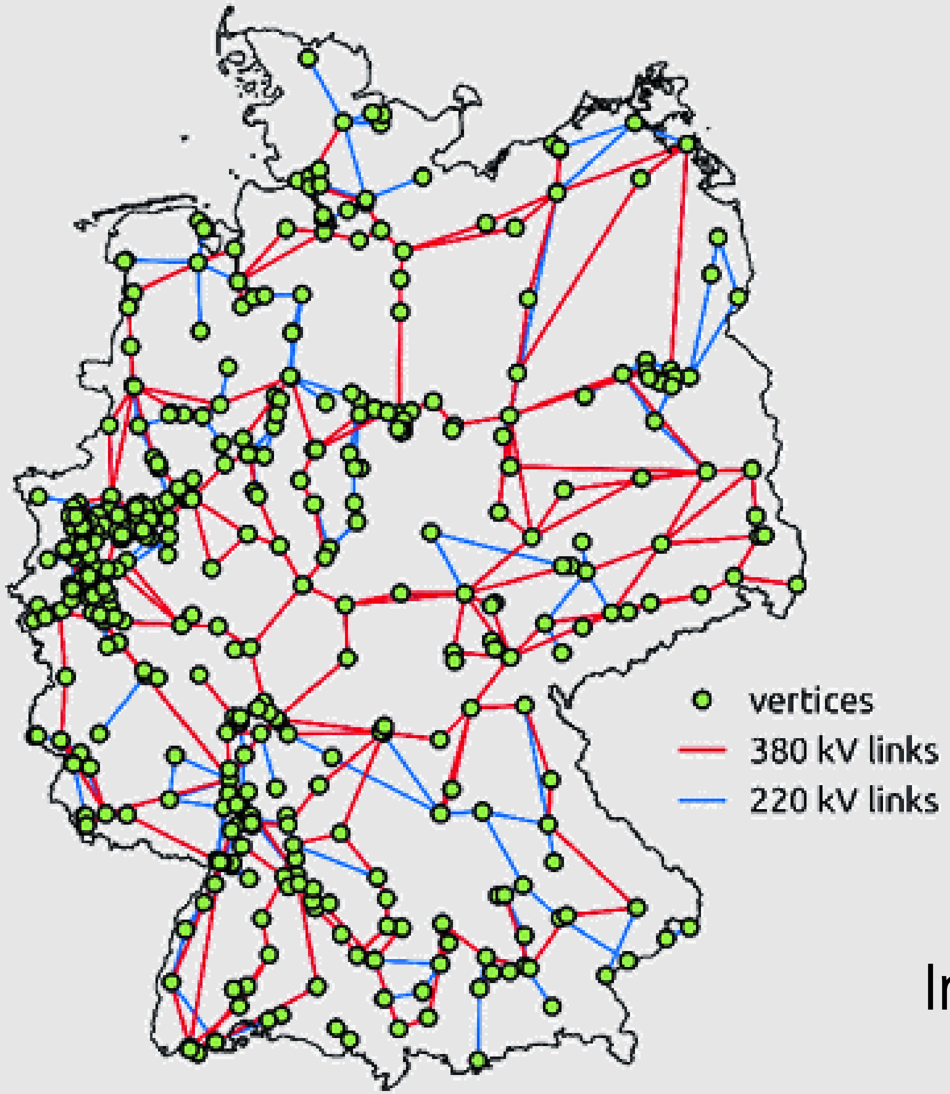
In-degree of a vertex:
Number of synaptic connections a neuron receives from others

Out-degree of a vertex:
Number of synaptic connections a neuron makes to others

Removing an edge:
Disconnecting a synaptic connection between two neurons

Removing a vertex:
Removing a neuron and its incoming + outgoing synaptic connections from the brain

POWER GRID



Vertices:

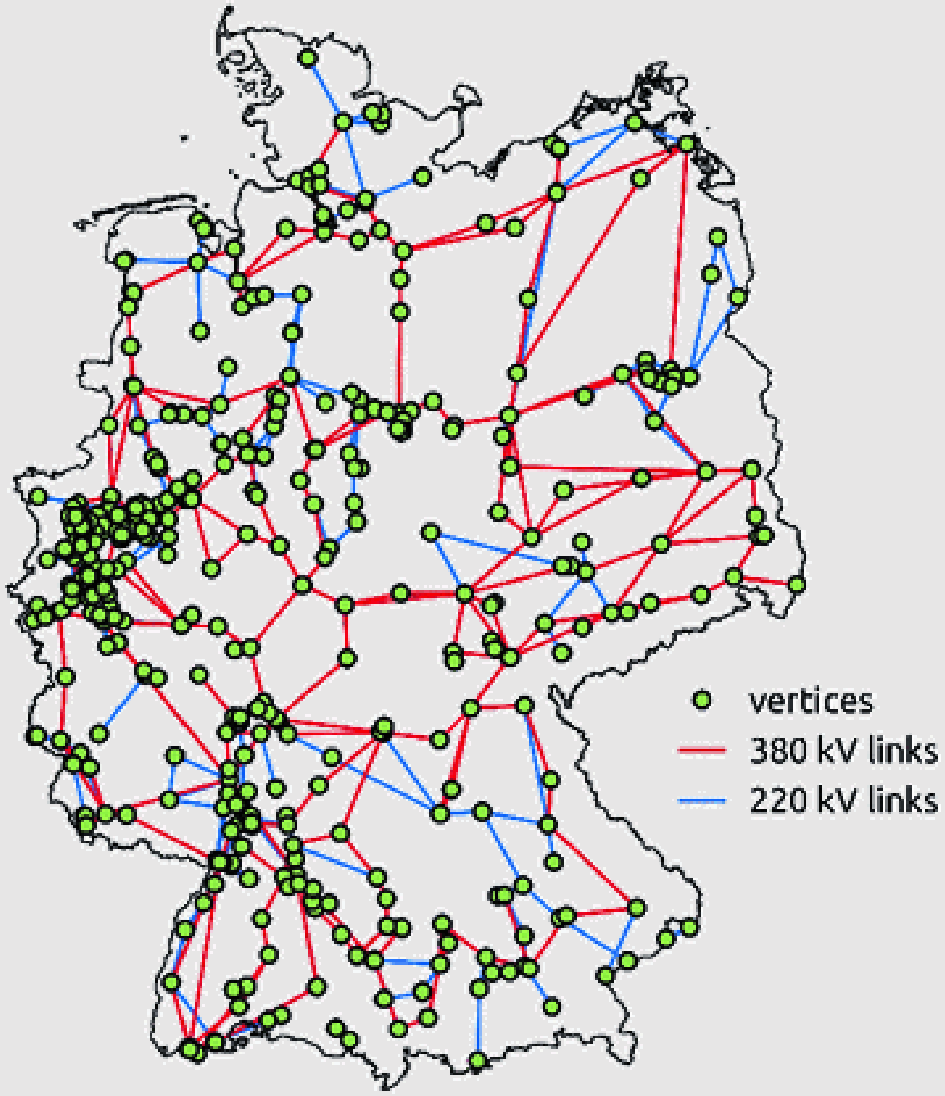
Power stations

Edges:

Power connections

Image credit: Germany's power grid

POWER GRID



Degree of a vertex:
Number of power links a station is connected

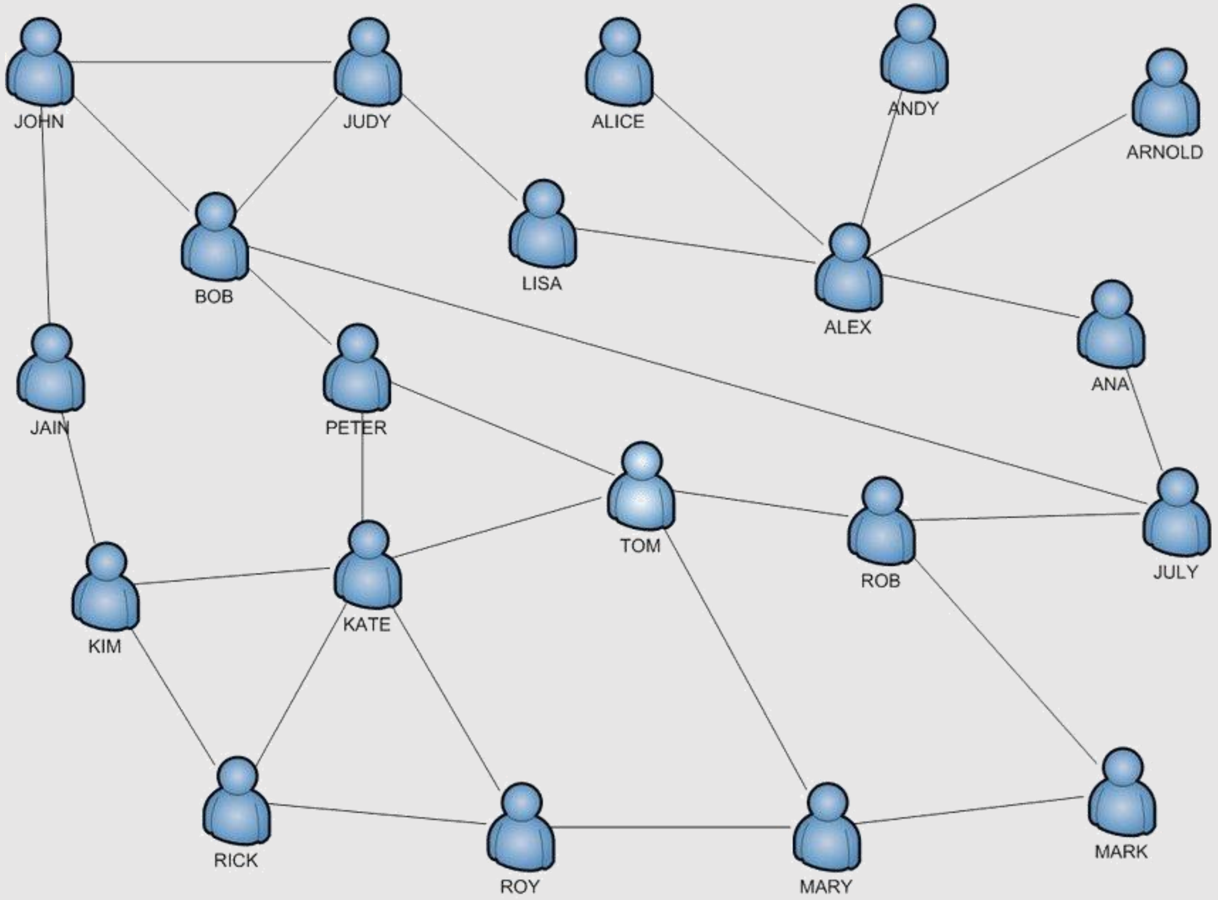
Removing an edge:
Disconnecting a power link between two stations

Removing a vertex:
Removing a station and its power links from the grid

Adding an edge
Adding a new power link between two stations

Adding a vertex
Adding a new power station and power links that connect to grid

SOCIAL NETWORK

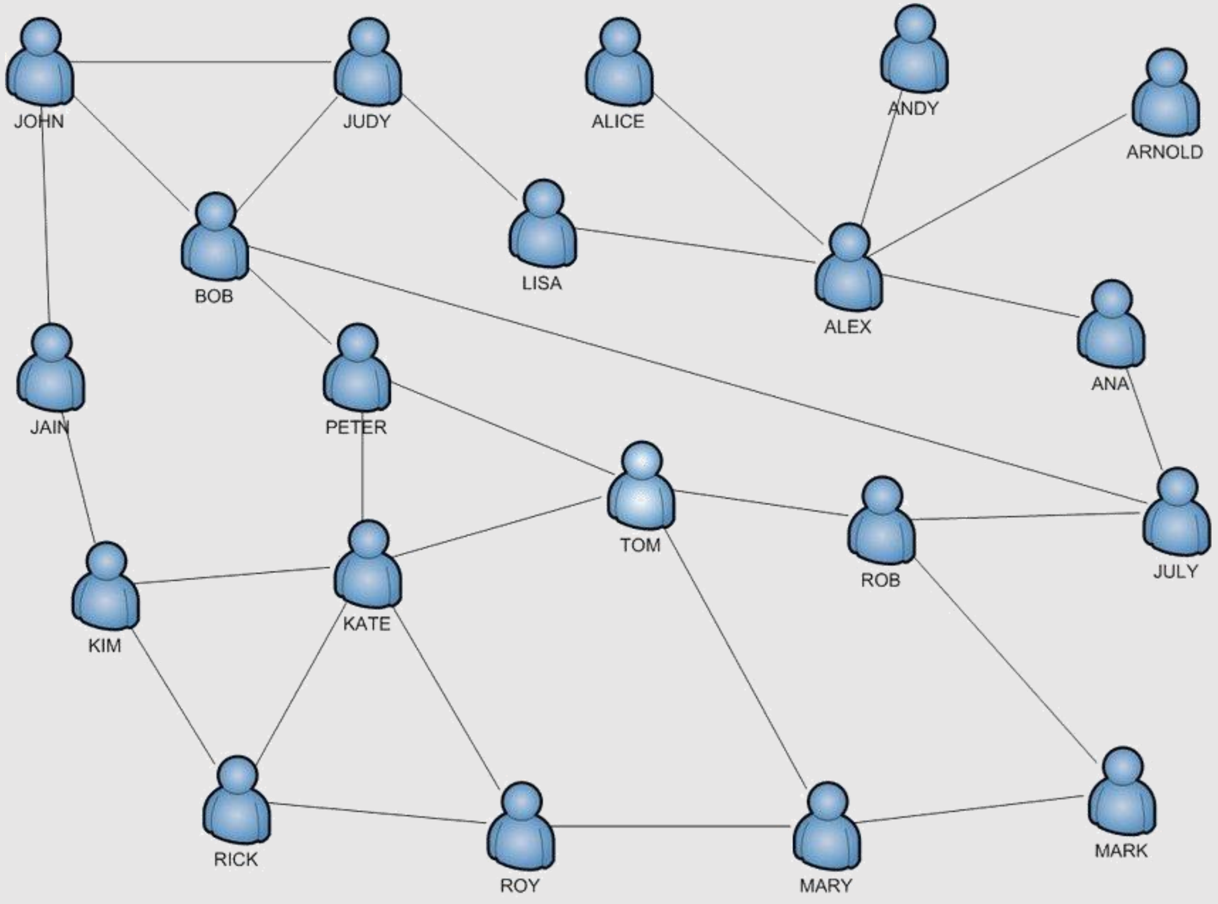


Vertices:
People

Edges:
Friend, follow status, etc

Image credit: Tomasz Filipowski
Web-based knowledge exchange through social links in the workplace
Behavior and Information Technology, 2012

SOCIAL NETWORK



Degree of a vertex:
Number of social connections (e.g. friends) a person (vertex) has

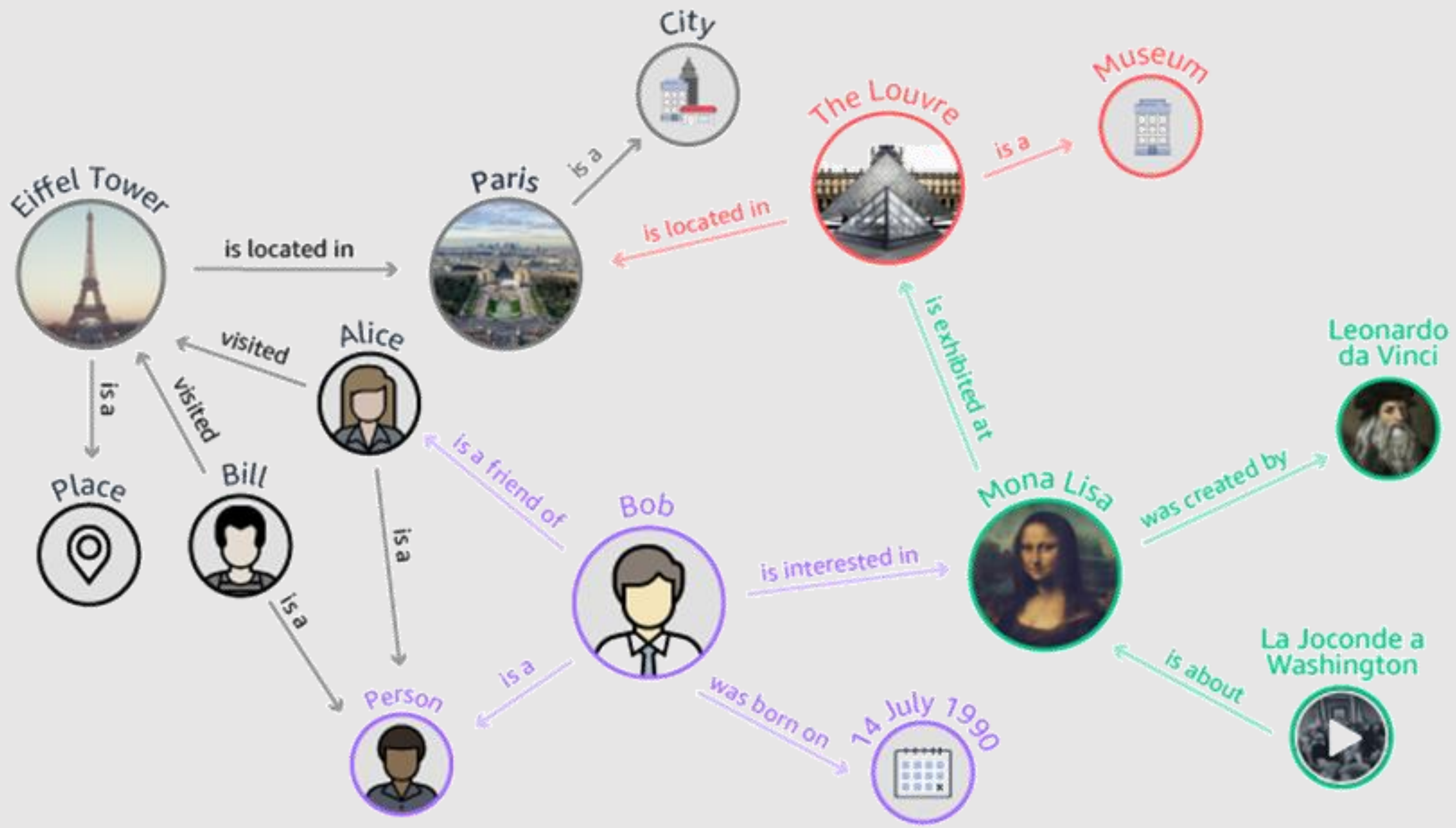
Removing an edge:
Disconnecting a social connection between two people (e.g. un-follow)

Removing a vertex:
Removing a person and his/her social connections from the network (e.g. deleting account)

Adding an edge
Adding a friend, following an account

Adding a vertex
Newly registering to the social network

KNOWLEDGE GRAPH

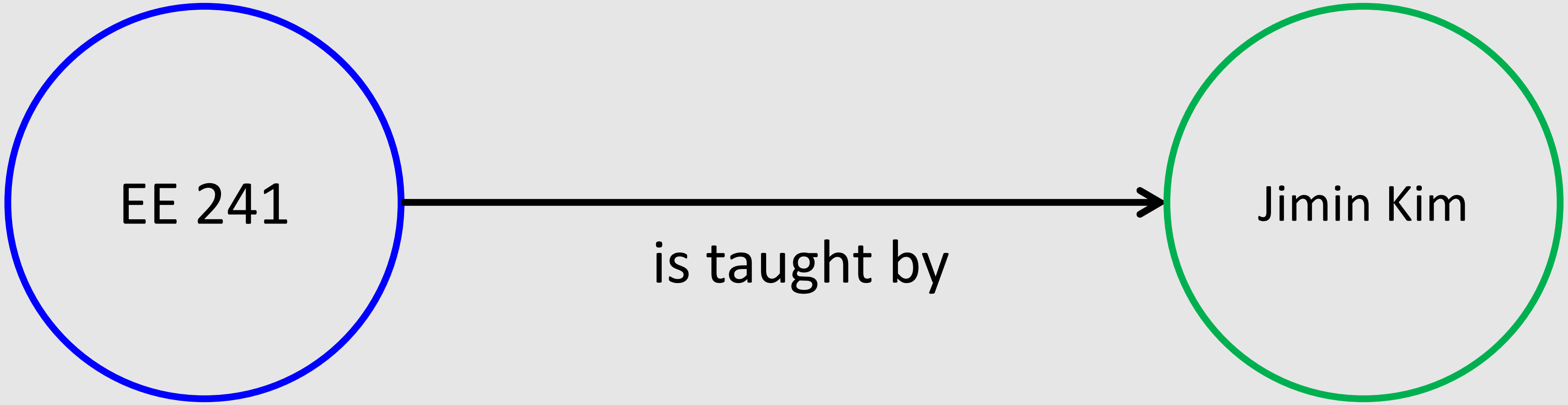


Vertices:
Subjects/Objects

Edges:
Predicates

Image credit: Amazon AWS

KNOWLEDGE GRAPH: TRIPLES



SUBJECT
Source

PREDICATE
Edge

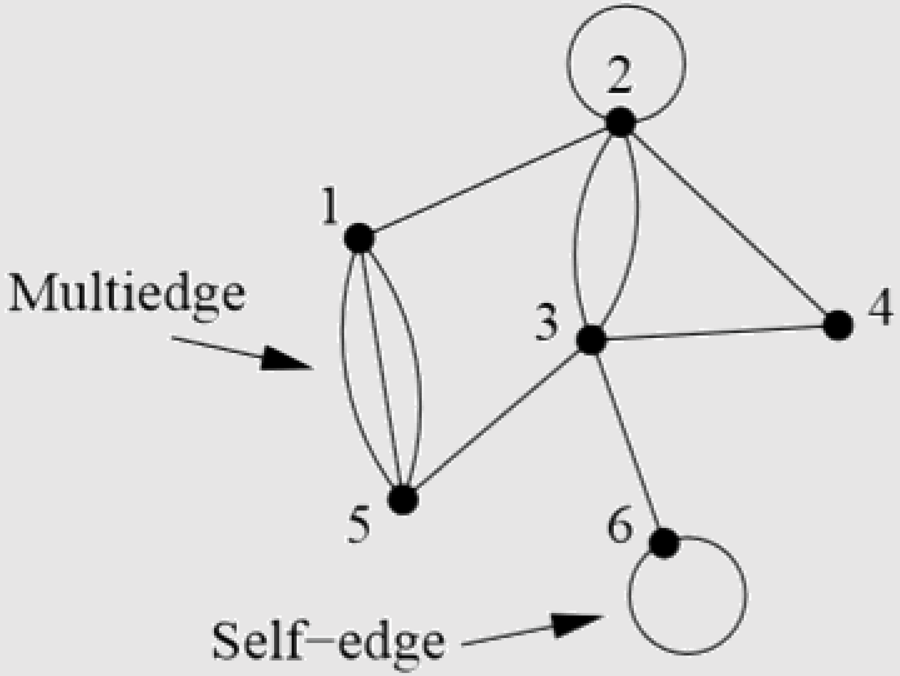
OBJECT
Target

Q: What would these mean in the context of knowledge graph?

- In-degree
- Out-degree
- Removing/adding an edge
- Removing/adding a vertex

PART 3: WORKING WITH GRAPH DATA

GRAPH AS A DATA STRUCTURE: ADJACENCY MATRIX



Undirected Graph

Target

Vertex index

# of edges

$$\begin{array}{rcl} & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} & \\ \mathbf{A} = & \begin{pmatrix} 0 & 1 & 0 & 0 & 3 & 0 \\ 1 & 2 & 2 & 1 & 0 & 0 \\ 0 & 2 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 3 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 2 \end{pmatrix} & \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \end{array}$$

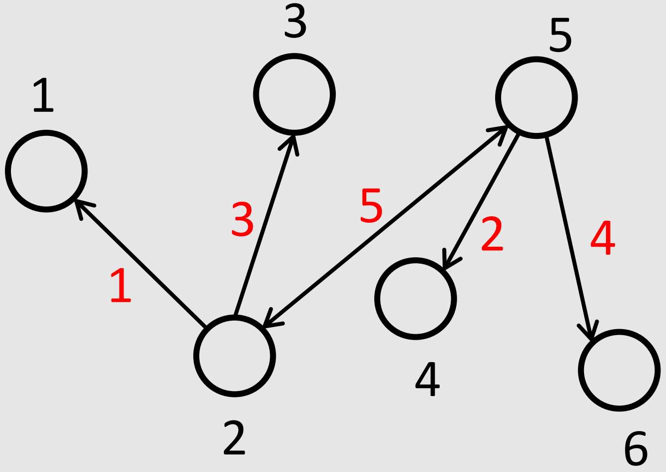
Source

Adjacency Matrix

NOTE: Undirected graph has ***Symmetric*** Adjacency Matrix (i.e. $A = A^T$)

Image credit: Newman, Mark. *Network: An Introduction*. Oxford University Press, 2018.

GRAPH AS A DATA STRUCTURE: ADJACENCY MATRIX



Directed Graph

Weight of the edge

$$
A = \begin{array}{c} \text{Target} \\ \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \left(\begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 3 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 2 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right) \end{array} \begin{array}{c} \\ \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \text{Source}
$$

Adjacency Matrix

NOTE: Directed graph is **NOT** necessarily ***Symmetric***

NOTE: Weight of the edge is often equivalent to # of edges

GRAPH AS A DATA STRUCTURE: TRIPLES ARRAY

Loading a graph as adjacency matrix could be problematic if the graph is:

- Sparse (i.e. many zeros)
- Very large (e.g. social network)

$$
\mathbf{A} = \begin{array}{c c}
 & \begin{array}{cccccc} v_0 & v_1 & \text{Target} & & & \end{array} \\
\begin{array}{c} v_0 \\ v_1 \\ \\ \\ \\ \\ \end{array} &
\begin{pmatrix}
0 & 1 & 0 & 0 & 3 & 0 \\
1 & 2 & 2 & 1 & 0 & 0 \\
0 & 2 & 0 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 \\
3 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 2
\end{pmatrix} \text{Source}
\end{array}
$$

Example of a sparse adjacency matrix

One way to address this is to represent graphs as lists of triples arrays

- Saves memory via only recording non-zero edges

	Source	Target	Weight
edge 1	0	1	1
edge 2	0	4	3

Vertex v_0

Source	Target	Weight
1	0	1
1	1	2
1	2	2
1	3	1

Vertex v_1

. . .

GRAPH AS A DATA STRUCTURE

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 0 & 3 & 0 \\ 1 & 2 & 2 & 1 & 0 & 0 \\ 0 & 2 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 3 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 2 \end{pmatrix}$$

Adjacency Matrix

.csv

.xlsx

.npy

⋮

GRAPH AS A DATA STRUCTURE

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 0 & 3 & 0 \\ 1 & 2 & 2 & 1 & 0 & 0 \\ 0 & 2 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 3 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 2 \end{pmatrix}$$

.csv

.xlsx

Load with Pandas

.npy

Load with Numpy

⋮

LOADING GRAPH DATA

```
directed_adj_mat = pd.read_excel('directed_sample.xlsx')
directed_adj_mat = np.array(directed_adj_mat)
print(directed_adj_mat)
```

```
[[0 6 0 0 0 0 6]
 [0 0 6 5 0 0 6]
 [5 0 0 0 0 0 0]
 [6 6 0 0 5 0 6]
 [0 0 0 5 0 0 0]
 [0 0 0 5 5 0 0]
 [0 6 0 6 5 0 0]]
```

Loading an adjacency matrix in .xlsx form

```
directed_adj_mat = pd.read_csv('directed_sample.csv')
directed_adj_mat = np.array(directed_adj_mat)
print(directed_adj_mat)
```

```
[[0 6 0 0 0 0 6]
 [0 0 6 5 0 0 6]
 [5 0 0 0 0 0 0]
 [6 6 0 0 5 0 6]
 [0 0 0 5 0 0 0]
 [0 0 0 5 5 0 0]
 [0 6 0 6 5 0 0]]
```

Loading .csv form

```
directed_adj_mat = np.load('directed_sample.npy')
print(directed_adj_mat)
```

```
[[0 6 0 0 0 0 6]
 [0 0 6 5 0 0 6]
 [5 0 0 0 0 0 0]
 [6 6 0 0 5 0 6]
 [0 0 0 5 0 0 0]
 [0 0 0 5 5 0 0]
 [0 6 0 6 5 0 0]]
```

Loading .npy form

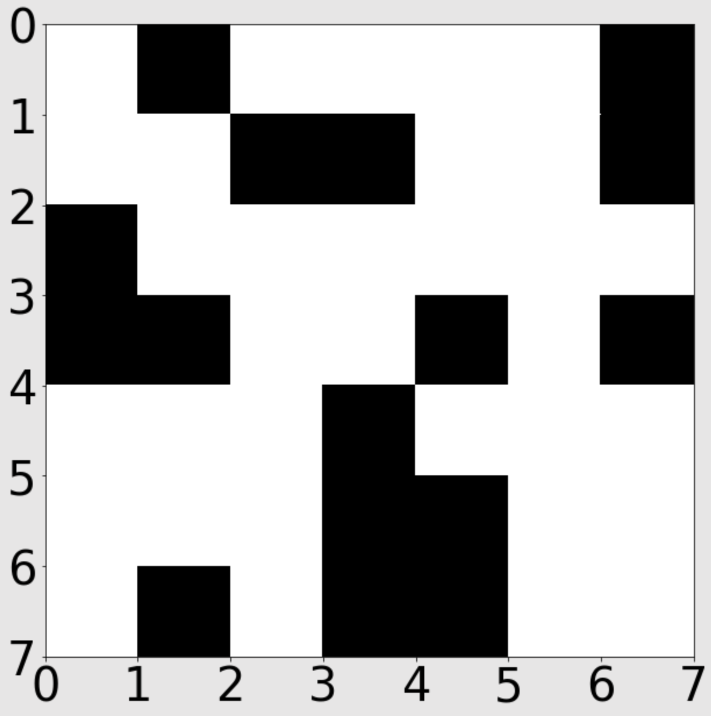
VISUALIZING AN ADJACENCY MATRIX

```
fig = plt.figure(figsize=(10,10))

plt.pcolor(directed_adj_mat, cmap = 'Greys', vmin = 0, vmax = 1)
plt.ylim(len(directed_adj_mat), 0)
plt.xticks(fontsize=40)
plt.yticks(fontsize=40)
plt.show()
```

Use plt.pcolor() to visualize the adjacency matrix

Use vmin = 0, vmax = 1 to highlight all non-zero edges

Invert the y-axis so that the first row starts with vertex 0




```
[[0 6 0 0 0 0 6]
 [0 0 6 5 0 0 6]
 [5 0 0 0 0 0 0]
 [6 6 0 0 5 0 6]
 [0 0 0 5 0 0 0]
 [0 0 0 5 5 0 0]
 [0 6 0 6 5 0 0]]
```

VISUALIZING A GRAPH USING NetworkX

What is NetworkX?

- A Python package for creating, manipulating and analyzing networks
- Provides tools for the study of the network structure
- Graph visualization tools built with matplotlib
- Works seamlessly with Numpy
- Included in Anaconda3

NetworkX
Network Analysis in Python

VISUALIZING A GRAPH (Undirected)

```
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
```

Import pandas, networkx, matplotlib

```
undirected_adj_mat_panda = pd.read_excel('undirected_sample.xlsx')
undirected_adj_mat_np = np.array(undirected_adj_mat_panda)
undirected_adj_mat_nx = nx.from_numpy_array(undirected_adj_mat_np)
```

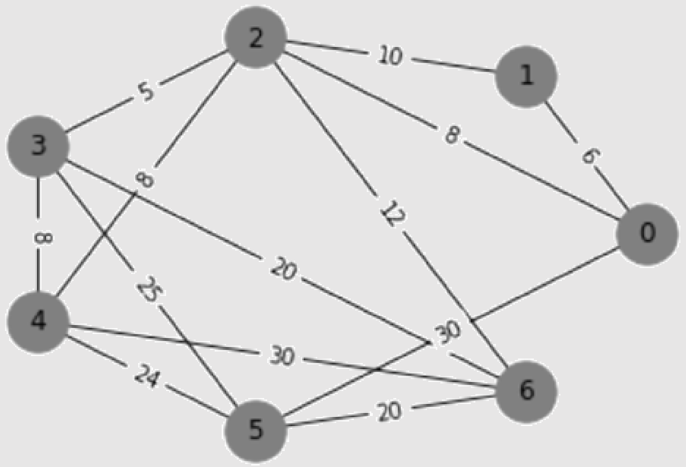
Load adjacency matrix with pandas

Convert loaded matrix → numpy array → networkx graph object

```
pos=nx.circular_layout(undirected_adj_mat_nx)

nx.draw_networkx(undirected_adj_mat_nx, pos, with_labels = True, node_size = 750, node_color='grey')
labels = nx.get_edge_attributes(undirected_adj_mat_nx,'weight')
nx.draw_networkx_edge_labels(undirected_adj_mat_nx, pos, edge_labels=labels)

plt.axis('off')
plt.show()
```

Obtain positions of vertex according to circular graph layout

Use nx.draw_networkx() to plot the graph



Use nx.get_edge_attributes() to obtain the edge weights according to adjacency matrix

Use nx.draw_networkx_edge_labels() to label the edges with the weights

VISUALIZING A GRAPH (Directed)

```
directed_adj_mat_panda = pd.read_excel('directed_sample.xlsx')
directed_adj_mat_np = np.array(directed_adj_mat_panda)
directed_adj_mat_nx = nx.from_numpy_array(directed_adj_mat_np, create_using=nx.DiGraph())
```

Load adjacency matrix with pandas

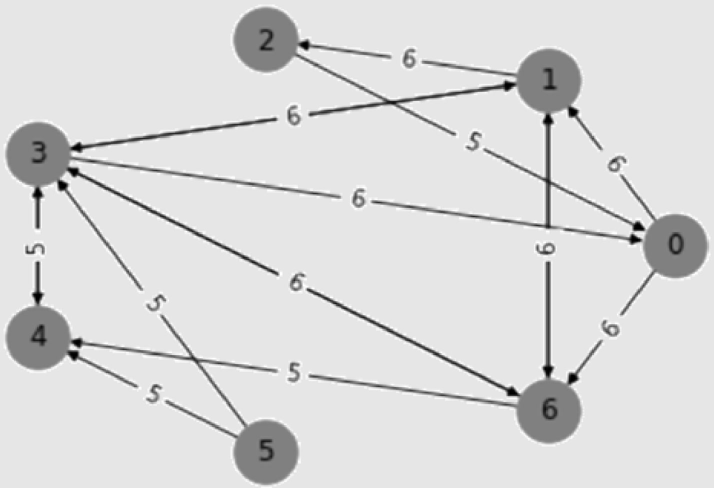
Convert loaded matrix → numpy array → networkx graph object

Add create_using=nx.DiGraph() to specify that the graph is directed

```
pos=nx.circular_layout(directed_adj_mat_nx)

nx.draw_networkx(directed_adj_mat_nx, pos, with_labels = True, node_size = 750, node_color='grey')
labels = nx.get_edge_attributes(directed_adj_mat_nx,'weight')
nx.draw_networkx_edge_labels(directed_adj_mat_nx, pos,edge_labels=labels)

plt.axis('off')
plt.show()
```

Obtain positions of vertex according to circular graph layout

Use nx.draw_networkx(), nx.get_edge_attributes(), nx.draw_networkx_edge_labels() to plot the graph



NOTE: By default, NetworkX uses **rows = source, columns = target** orientation for adjacency matrix

COMPUTING DEGREES (UNDIRECTED)

```
undirected_adj_mat = pd.read_excel('undirected_sample.xlsx')
undirected_adj_mat_np = np.array(undirected_adj_mat)
```

```
print(undirected_adj_mat_np)
```

Load the included undirected adjacency matrix sample

```
[[ 0  6  8  0  0 30  0]
 [ 6  0 10  0  0  0  0]
 [ 8 10  0  5  8  0 12]
 [ 0  0  5  0  8 25 20]
 [ 0  0  8  8  0 24 30]
 [30  0  0 25 24  0 20]
 [ 0  0 12 20 30 20  0]]
```

We will compute the degree of the vertex 3

```
np.sum(undirected_adj_mat_np[3, :])
```

```
58
```

Summing over the row (or column) gives the degree

```
np.sum(undirected_adj_mat_np, axis = 1)
```

```
array([44, 16, 43, 58, 70, 99, 82], dtype=int64)
```

Summing over all the columns gives a list of degrees for all the neurons

NOTE: Due to adjacency matrix being ***symmetric***, the operation will yield identical degrees when computed in respect to **columns** instead of **rows**

COMPUTING DEGREES (DIRECTED)

```
directed_adj_mat = pd.read_excel('directed_sample.xlsx')
directed_adj_mat_np = np.array(directed_adj_mat)
```

Load the included undirected adjacency matrix sample

```
print(directed_adj_mat_np)
```

```
[[0 6 0 0 0 0 6]
 [0 0 6 5 0 0 6]
 [5 0 0 0 0 0 0]
 [6 6 0 0 5 0 6]
 [0 0 0 5 0 0 0]
 [0 0 0 5 5 0 0]
 [0 6 0 6 5 0 0]]
```

out-degrees of vertex 3

in-degrees of vertex 3

We will compute both **in-degree and out-degree** of the vertex 3

COMPUTING DEGREES (DIRECTED)

```
np.sum(directed_adj_mat_np[3, :])
```

```
23
```

Summing over all numbers along the columns with respect to a single vertex gives **out-degree of a vertex**

```
np.sum(directed_adj_mat_np, axis = 1)
```

```
array([12, 17,  5, 23,  5, 10, 17], dtype=int64)
```

Summing over all the columns gives **out-degrees of all the vertices**

```
np.sum(directed_adj_mat_np[:, 3])
```

```
21
```

Summing over all numbers along the rows with respect to a single vertex gives **in-degree of a vertex**

```
np.sum(directed_adj_mat_np, axis = 0)
```

```
array([11, 18,  6, 21, 15,  0, 18], dtype=int64)
```

Summing over all the rows gives **in-degrees of all the vertices**

NOTE: We are using **rows = source, columns = target** orientation for adjacency matrix

REMOVING EDGES

```
directed_adj_mat = pd.read_excel('directed_sample.xlsx')
directed_adj_mat = np.array(directed_adj_mat)
directed_adj_mat[0, 1] = 0
directed_adj_mat[1, 2] = 0
```

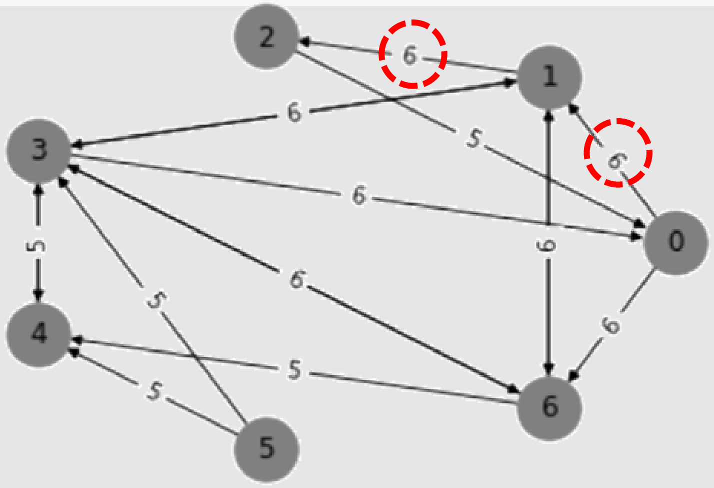
Load the included directed adjacency matrix sample

Set following edges to zero

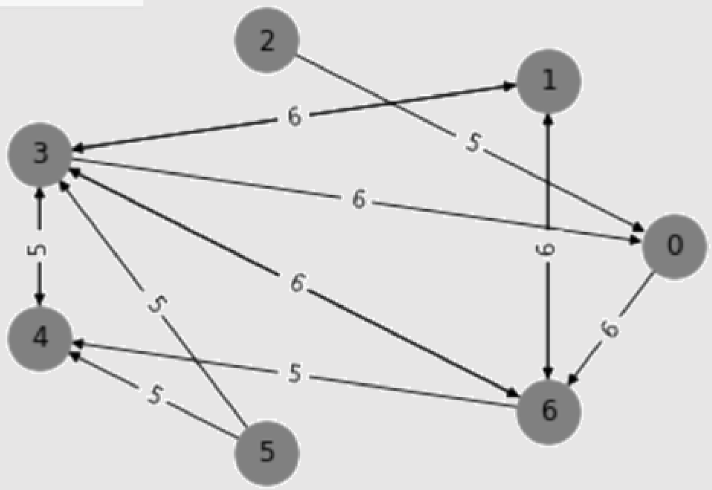
- vertex 0 → vertex 1
- vertex 1 → vertex 2

```
directed_graph_sample = nx.from_numpy_array(directed_adj_mat, create_using=nx.DiGraph())
pos=nx.circular_layout(directed_graph_sample)

nx.draw_networkx(directed_graph_sample,pos,with_labels = True, node_size = 750, node_color='grey')
labels = nx.get_edge_attributes(directed_graph_sample,'weight')
nx.draw_networkx_edge_labels(directed_graph_sample,pos,edge_labels=labels)

plt.axis('off')
plt.show()
```

Plot the graph with removed edges




```
[[0 1 0 0 0 0 6]
 [0 0 1 5 0 0 6]
 [5 0 0 0 0 0 0]
 [6 6 0 0 5 0 6]
 [0 0 0 5 0 0 0]
 [0 0 0 5 5 0 0]
 [0 6 0 6 5 0 0]]
```



Q: How would you expand the operation to remove a vertex?

ADDING EDGES

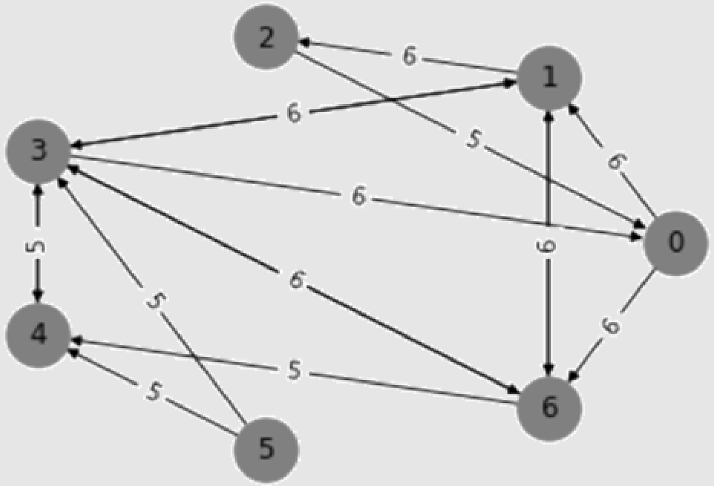
```
directed_adj_mat = pd.read_excel('directed_sample.xlsx')
directed_adj_mat = np.array(directed_adj_mat)
directed_adj_mat[2, 3] = 4
```

Load the included directed adjacency matrix sample

Add a new edge with weight = 4

- vertex 2 → vertex 3

```
directed_graph_sample = nx.from_numpy_array(directed_adj_mat, create_using=nx.DiGraph())
pos=nx.circular_layout(directed_graph_sample)

nx.draw_networkx(directed_graph_sample,pos,with_labels = True, node_size = 750, node_color='grey')
labels = nx.get_edge_attributes(directed_graph_sample,'weight')
nx.draw_networkx_edge_labels(directed_graph_sample,pos,edge_labels=labels)

plt.axis('off')
plt.show()
```

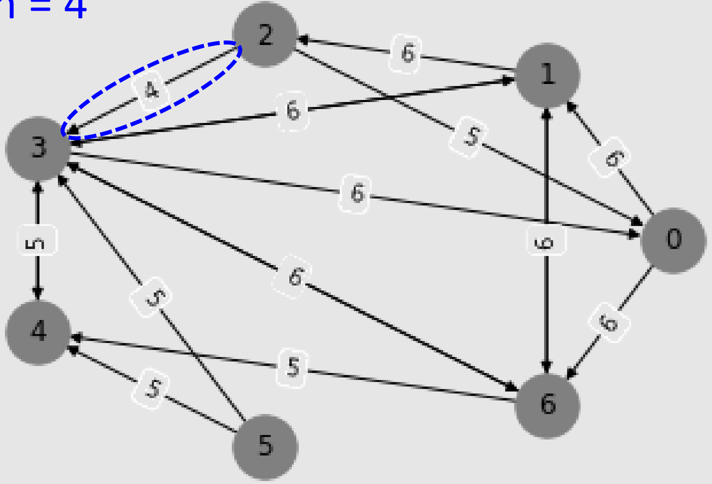


New connection = 4

v_3

v_2

```
[[0 6 0 0 0 0 6]
 [0 0 6 5 0 0 6]
 [5 0 0 0 0 0 0]
 [6 6 0 0 5 0 6]
 [0 0 0 5 0 0 0]
 [0 0 0 5 5 0 0]
 [0 6 0 6 5 0 0]]
```



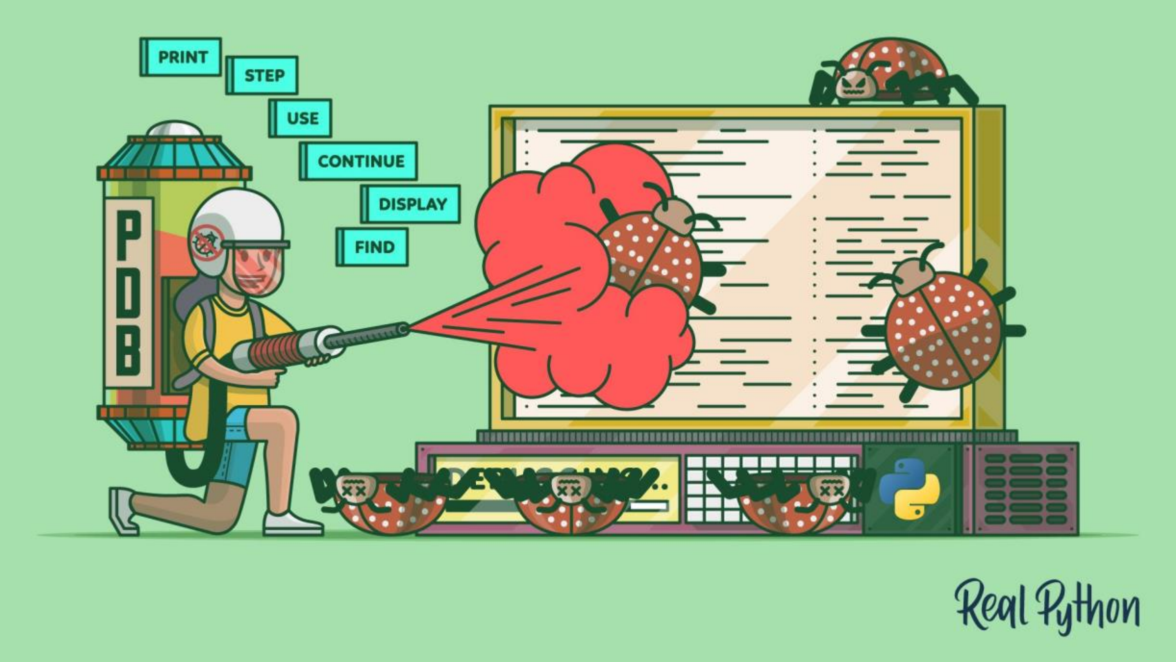
Q: How would you expand the operation to add a vertex?

PART 4: ADVANCED DEBUGGING

INTRODUCTION TO PYTHON DEBUGGER

What is Python Debugger (PDB)?

- Debugging tool built in Python
- No installation required
- Provides Interactive environment
- Useful when debugging using logging is problematic



BASICS

```
def divide(divisor):

    breakpoint()

    val = 1/divisor

    return val
```

Upon execution of the code, activates Python debugger console at this line

```
divide(0)
```

```
> <ipython-input-24-03e5b3dd8265>(5)divide()
      3     breakpoint()
      4 
----> 5     val = 1/divisor
      6 
      7     return val
```

```
ipdb>
```

Python debugger console

USEFUL COMMANDS

```
> <ipython-input-24-03e5b3dd8265>(5)divide()
      3     breakpoint()
      4
----> 5     val = 1/divisor
      6
      7     return val

ipdb>
```

- h: help
- w: where
- n: next
- c: continue
- p: print
- l: list
- q: quit

USEFUL COMMANDS: help

```
divide(0)
```

```
> <ipython-input-24-03e5b3dd8265>(5)divide()
      3     breakpoint()
      4
----> 5     val = 1/divisor
      6
      7     return val

ipdb> h

Documented commands (type help <topic>):
========================================
EOF    commands   enable    ll        pp       s                until
a      condition  exit      longlist  psource  skip_hidden      up
alias  cont       h         n         q        skip_predicates  w
args   context    help      next      quit     source           whatis
b      continue   ignore    p         r        step             where
break  d          interact  pdef      restart  tbreak
bt     debug      j         pdoc      return   u
c      disable    jump      pfile     retval   unalias
cl     display    l         pinfo     run      undisplay
clear  down       list      pinfo2    rv       unt

Miscellaneous help topics:
==========================
exec  pdb

ipdb> h l
Print lines of code from the current stack frame

ipdb>
```

h: help

w: where

n: next

c: continue

p: print

l: list

q: quit

Provide documentation for the command

USEFUL COMMANDS: where

```
divide(2)
```

```
> <ipython-input-1-f4a9c3842530>(7)divide()
      3     val = 1/divisor
      4
      5     breakpoint()
      6
----> 7     return val

ipdb> w
    [... skipping 27 hidden frame(s)]

  <ipython-input-2-cfdb0f794c84>(1)<module>()
----> 1 divide(2)

> <ipython-input-1-f4a9c3842530>(7)divide()
      3     val = 1/divisor
      4
      5     breakpoint()
      6
----> 7     return val

ipdb>
```

h: help

w: where

n: next

c: continue

p: print

l: list

q: quit

Print a stack trace (i.e. the order of code being executed), with the most recent frame at the bottom.

USEFUL COMMANDS: next

```
divide(2)

> <ipython-input-1-03e5b3dd8265>(5)divide()
      3     breakpoint()
      4 
----> 5     val = 1/divisor
      6 
      7     return val

ipdb> n
> <ipython-input-1-03e5b3dd8265>(7)divide()
      3     breakpoint()
      4 
      5     val = 1/divisor
      6 
----> 7     return val

ipdb>
```

Executes the next line of code

h: help
w: where
n: next
c: continue
p: print
l: list
q: quit

USEFUL COMMANDS: continue

```
divide(2)

> <ipython-input-1-a669feb31165>(5)divide()
      3     breakpoint()
      4
----> 5     val = 1/divisor
      6
      7     breakpoint()

ipdb> c
> <ipython-input-1-a669feb31165>(9)divide()
      5     val = 1/divisor
      6
      7     breakpoint()
      8
----> 9     return val

ipdb>
```

h: help

w: where

n: next

c: continue

p: print

l: list

q: quit

Run the code until the next breakpoint()

USEFUL COMMANDS: print

```
divide(2)
```

```
> <ipython-input-1-f4a9c3842530>(7)divide()
      3     val = 1/divisor
      4 
      5     breakpoint()
      6 
----> 7     return val

ipdb> p val
0.5

ipdb>
```

h: help
w: where
n: next
c: continue
p: print
l: list
q: quit

Print the value of the desired variable

USEFUL COMMANDS: list

```
divide(2)
```

```
> <ipython-input-1-f4a9c3842530>(7)divide()
      3     val = 1/divisor
      4 
      5     breakpoint()
      6 
----> 7     return val

ipdb> l
      2 
      3     val = 1/divisor
      4 
      5     breakpoint()
      6 
----> 7     return val

ipdb> 
```

h: help
w: where
n: next
c: continue
p: print
l: list
q: quit

Similar to where but print the lines of code from the current stack frame

USEFUL COMMANDS: quit

```
divide(2)
```

```
> <ipython-input-1-f4a9c3842530>(7)divide()
      3     val = 1/divisor
      4 
      5     breakpoint()
      6 
----> 7     return val

ipdb> q

---------------------------------------------------------------------------
BdbQuit                                   Traceback (most recent call last)
<ipython-input-2-cfdb0f794c84> in <module>
----> 1 divide(2)

<ipython-input-1-f4a9c3842530> in divide(divisor)
      5     breakpoint()
      6 
----> 7     return val

<ipython-input-1-f4a9c3842530> in divide(divisor)
      5     breakpoint()
      6 
----> 7     return val

~\anaconda3\lib\bdb.py in trace_dispatch(self, frame, event, arg)
     86             return # None
     87         if event == 'line':
---> 88             return self.dispatch_line(frame)
     89         if event == 'call':
     90             return self.dispatch_call(frame, arg)

~\anaconda3\lib\bdb.py in dispatch_line(self, frame)
    111         if self.stop_here(frame) or self.break_here(frame):
    112             self.user_line(frame)
--> 113             if self.quitting: raise BdbQuit
    114         return self.trace_dispatch
    115 

BdbQuit: 
```

h: help
w: where
n: next
c: continue
p: print
l: list
q: quit

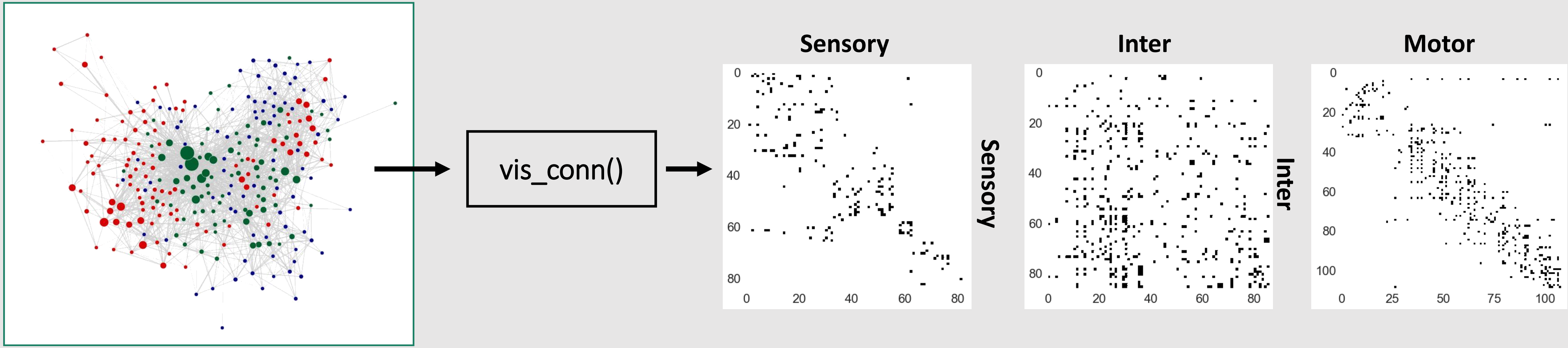
Exit the debugger console

Useful video tutorial of basic usage of Python debugger:
https://www.youtube.com/watch?v=aZJnGOwzHtU

LAB ASSIGNMENTS

Download ipynb template in Canvas page:
Assignments/Lab 5 report -> click "Lab 5 Report Templates"

EXERCISE 1: Visualize brain connectomes

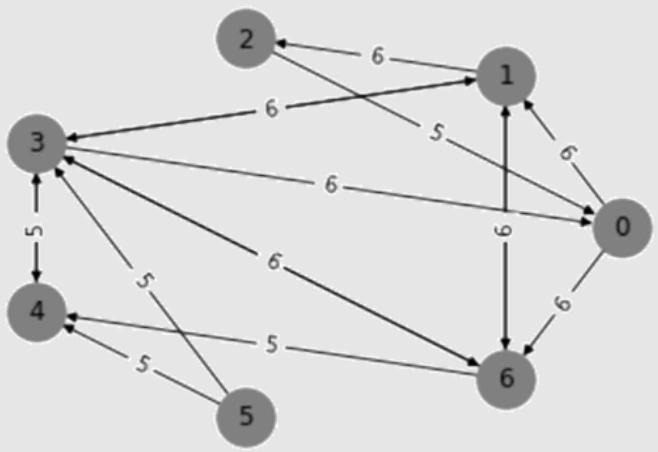


- Recalling from slides 22-23, create a function **vis_conn()** which takes adjacency matrix of the real *C. elegans* synaptic connectome, list of neuron classes (i.e. labels) for each neuron (sensory, inter, motor) and outputs **1 x 3 subplots of three adjacency matrices** corresponding to **sub-networks of Sensory, Inter and Motorneurons** (e.g. sub-network of sensory neurons is a wiring map between sensory neurons only).

- The function should accept following parameters
  - syn_conn – the adjacency matrix of the *C. elegans* synaptic connectome (279 * 279)
  - neuron_classes – A list of neuron classes for each neuron (length = 279). Follows the same order of neurons as syn_conn.

- Use plt.pcolor() with cmap = 'Greys' and vmin = 0, vmax = 1.
- Don't forget to invert the y-axis so that the first row starts with vertex 0 (See slide 37)
- Include appropriate x and y labels for each sub-network plot.

EXERCISE 2: Locating the most connected vertices



find_hub_vertices()

Highest in-degree vertices	Highest out-degree vertices
3	3
1	1
4	0
.	.
.	.

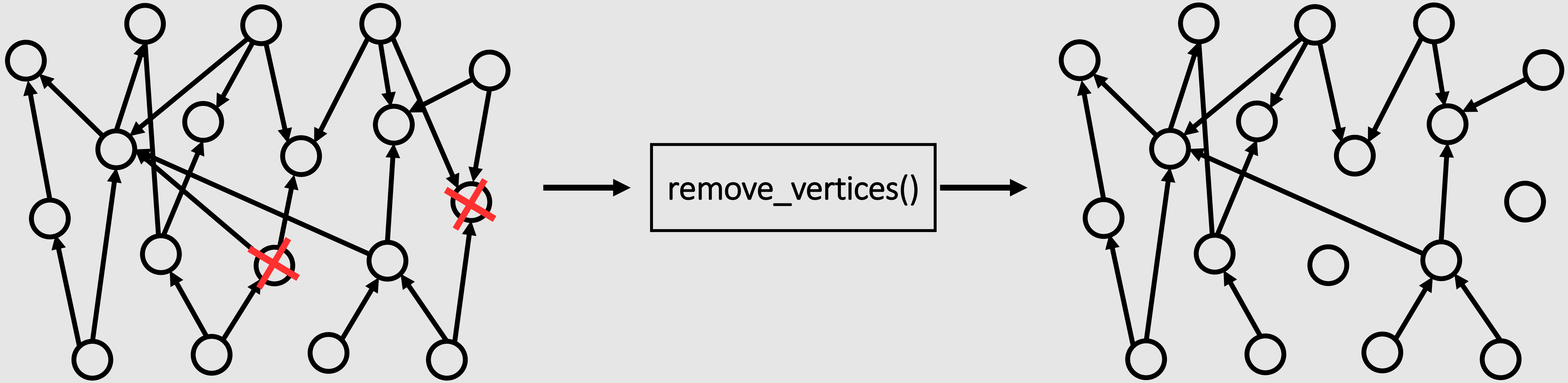
- Create a function **find_hub_vertices()** which takes adjacency matrix of a directed graph and outputs two lists where the **first list: indices of vertices with the highest in-degree** and the **second list: indices of vertices with the highest out-degree.**

- The function should accept following parameters
  - adj_mat – the input adjacency matrix
  - num_vertices - the number of the highest degree vertices to find for each list

- The output lists should be ordered such that the vertices with highest degree comes at the top, second highest comes at the second etc.

- Test you function with ***C. elegans* synaptic connectome** and a **sample social network graph**.

EXERCISE 3: Removing vertices from a graph



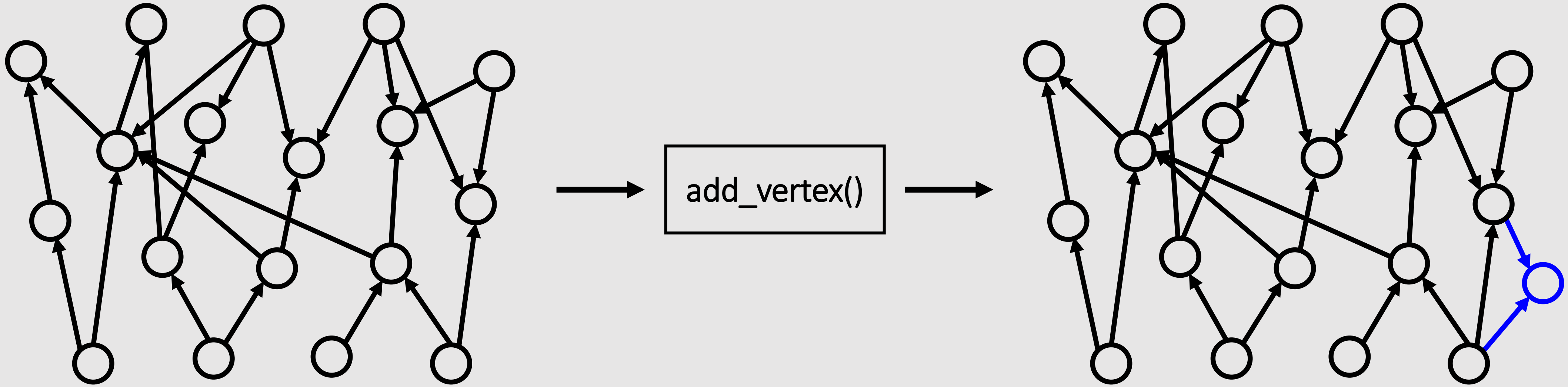
Expanding the concept of removing edges from a graph, write a function named **remove_vertices()** which takes **adjacency matrix** of an existing graph, **list of vertices to be removed** and returns a new adjacency matrix with removed vertices.

The function should take following parameters as inputs:

- adj_mat - the input adjacency matrix
- vertices_2b_removed – list of vertices to be removed from the graph

Assume that the adjacency matrix follows **row = source, column = to orientation.**
Removing a vertex is equivalent to removing all the incoming/outgoing edges from that vertex.
Test your function with provided adjacency matrix and lists of vertices to be removed.

EXERCISE 4: Adding a new vertex to a graph



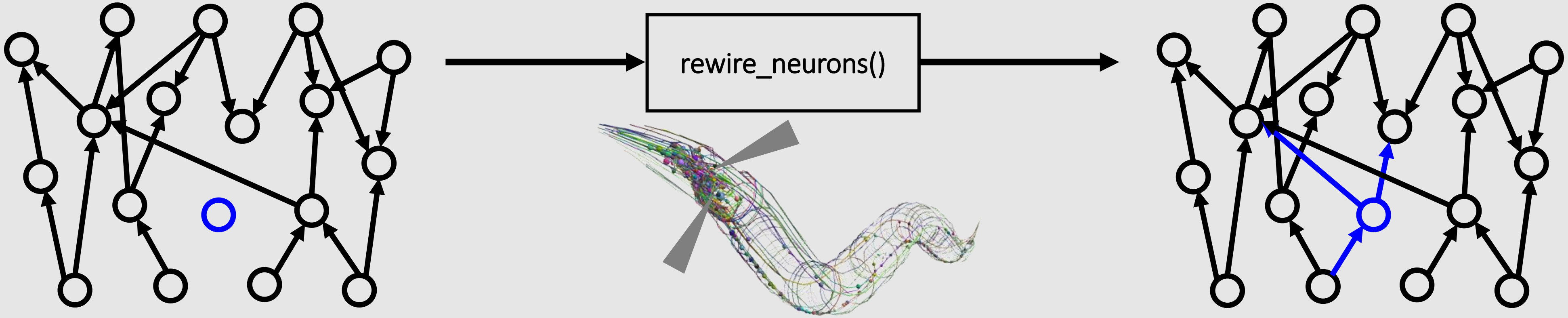
Expanding the concept of adding edges to the graph, write a function named **add_vertex()** which takes adjacency matrix of an existing graph and returns a new adjacency matrix with an added vertex.

The function should take following parameters as inputs:

- adj_mat - the input adjacency matrix.
- outgoing_edges – list of vertices in which the new vertex connects to.
- incoming_edges – list of vertices in which the new vertex receives connections from.

Assume that the adjacency matrix follows **row = source, column = target orientation**. Test your function with provided adjacency matrix and lists of outgoing and incoming edges.

EXERCISE 5: Re-wire neurons to restore behavior of *C. elegans*



In this problem we will work with the **mathematical model of *C. elegans'* nervous system and body** which can simulate the **organism's brain, muscles and the body** according to its real neural wiring data and its interactions with muscles [1].

Two neurons - '**AVAL**' and '**AVAR**' play essential roles during **the worm's escape maneuver via forward crawling** upon a gentle touch on its tail. Unfortunately, the worm has damaged its brain and now lost all the synaptic connections (both incoming and outgoing) for these two neurons. Your task is to **re-wire these two neurons** to the rest of the nervous system to recover the worm's forward crawling function.

Write a function called **rewire_neurons()** which take following parameters as inputs:

- damaged_brain_adj_matrix – Adjacency matrix of the damaged synaptic connectome
- rewiring_instructions_AVAL – Triples array describing the re-wiring instructions for AVAL
- rewiring_instructions_AVAR – Triples array describing the re-wiring instructions for AVAR

[1] Whole integration of neural connectomics, dynamics and bio-mechanics for identification of behavioral sensorimotor pathways in *Caenorhabditis elegans*, *Jimin Kim, Julia Santos, Mark Alkema, Eli Shlizerman, BioArxiv, 2019*

Use rewiring instructions for AVAL, AVAR to **correctly re-wire each neuron to the rest of the neurons** in the system. Note that instructions contain both outgoing and incoming connections for both neurons. Your output should be an **adjacency matrix** of the same dimension as the input adjacency matrix which now contains new connections for AVAL and AVAR. See **slide 33** to review the concept of triples array.

Run **test_brain_repair**() provided in lab template to check if your repair operation was successful.