Lab 1

MECH 6317.001: Dynamics of Complex Networks and Systems

Tanner Kogel tjk190000

bring networkx library into code as nx to use

bring all other libraries needed into code

In [2]:
```
import networkx as nx
import numpy
import itertools
```

create the graph G that models the network shown in figure 6.1a

In [3]:
```
G = nx.Graph() # initialize G as a graph

G.add_nodes_from(range(1,7)) # create nodes 1,2,...,6 in order

# add all edges shown in figure 6.1a
G.add_edge(1,2)
G.add_edge(1,5)
G.add_edge(2,3)
G.add_edge(2,4)
G.add_edge(3,4)
G.add_edge(3,5)
G.add_edge(3,6)
```

print proof that G is correctly defined: number of nodes, number of edges 3-4 edge exists 4-6 edge does not exist

In [4]:
```
print('#Nodes: %i, #Edges: %i' % (G.number_of_nodes(),G.number_of_edges())) # quantity of nodes & edges
print('Has 3-4 edge: %s' % G.has_edge(3,4))                                 # 3-4 edge exists?
print('Has 4-6 edge: %s' % G.has_edge(4,6))                                 # 4-6 edge exists?
```

```
#Nodes: 6, #Edges: 7
Has 3-4 edge: True
Has 4-6 edge: False
```

print out the corresponding adjacency matrix

In [5]:
```
A = nx.to_numpy_array(G)
print(A)
```

```
[[0. 1. 0. 0. 1. 0.]
 [1. 0. 1. 1. 0. 0.]
 [0. 1. 0. 1. 1. 1.]
 [0. 1. 1. 0. 0. 0.]
 [1. 0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]]
```

ensure that adjacency matrix is symetric (A=A.T)

In [6]:
```
print('Adjacency matrix symmetry: %s' % (A==A.T).all())
```

```
Adjacency matrix symmetry: True
```

Section 6.3: Weighted Networks

create the weighted network shown in figure 6.1b

In [7]:
```
G_b = nx.Graph() # initialize G_b as a graph

G_b.add_nodes_from(range(1,7)) # add nodes 1,2,...,6 in order

# add all edges shown in figure 6.1b
G_b.add_edge(1,2)
G_b.add_edge(1,5,weight=3)
G_b.add_edge(2,2)
G_b.add_edge(2,3,weight=2)
G_b.add_edge(2,4)
G_b.add_edge(3,4)
G_b.add_edge(3,5)
G_b.add_edge(3,6)
G_b.add_edge(6,6)

# correct self-edges to weight 2
G_b[2][2]['weight'] = 2
G_b[6][6]['weight'] = 2
```

print adjacency matrix of the weighted network

In [8]:
```
A_b = nx.to_numpy_array(G_b)
print(A_b)
```

```
[[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.]]
```

Section 6.4: Directed Networks

create the directed network shown in figure 6.2

In [9]:
```
G_d = nx.DiGraph() # initialize G_d as a directed graph

G_d.add_nodes_from(range(1,7)) # add nodes 1,2,...,6 in order

# add all directed edges shown in figure 6.2
G_d.add_edge(1,3)
G_d.add_edge(2,6)
G_d.add_edge(3,2)
G_d.add_edge(4,1)
G_d.add_edge(4,5)
G_d.add_edge(5,3)
G_d.add_edge(6,4)
G_d.add_edge(6,5)
```

print adjacency matrix of the directed network

In [10]:
```
A_d = nx.to_numpy_array(G_d)
print(A_d)
```

```
[[0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1.]
 [0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 1. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 1. 0.]]
```

the adjacency matrix shown above is the transpose of the adjacency matrix shown in the textbook

ensure that the adjacency matrix is in the format of the textbook by taking the transpose

In [11]:
```
A_d = A_d.T
print(A_d)
```

```
[[0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [1. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 1. 0. 1.]
 [0. 1. 0. 0. 0. 0.]]
```

Section 6.4.1: Cocitation & Bibliographic Coupling

create a function to transform a directed network into a cocitation network

In [12]:
```
def cocitation(G):

    G_cocitation = nx.Graph()              # initialize cocitation graph
    G_cocitation.add_nodes_from(G.nodes()) # ensure all nodes match

    for u,v in itertools.combinations(G.nodes(),2):  # loop for every combination of nodes
        w = 0                                        # reset weight to be zero
        for i in G.predecessors(u):                  # loop for each predecessor in node u
            for j in G.predecessors(v):              # loop for each predecessor in node v
                if i == j:                           # cocitation exists
                    if 'weight' in G[i][u]:          # additional weight in u edge exists
                        u_weight = G[i][u]['weight'] # add appropriate weight to u edge
                    else:                            # weight of 1 in u exists
                        u_weight = 1                 # assign weight of 1 to u edge
                    if 'weight' in G[i][v]:          # additional weight in v exists
                        v_weight = G[i][v]['weight'] # add appropriate weight to v edge
                    else:                            # weight of 1 in v exists
                        v_weight = 1                 # assign weight of 1 to v edge
                    w = w + (u_weight*v_weight)      # add weight
        if w > 0:                                    # edge exists
            G_cocitation.add_edge(u,v,weight=w)      # create edge with appropriate weight

    return G_cocitation # return cocitation graph
```

load ProofWiki network

In [13]:
```
G_pw = nx.read_gml('proofwikidefs_la.gml','name')
```

create cocitation graph of ProofWiki network

In [14]:
```
A = nx.to_numpy_array(G_pw).T # create adjacency matrix of proofwiki network in textbook format
C1 = numpy.dot(A,A.T)         # use linear algebra definition to create cocitation matrix
# loop added to ensure that the diagonal values are set to 0
for i in range(len(G_pw.nodes())):
    for j in range(len(G_pw.nodes())):
        C1[i,i] = 0
Gc = cocitation(G_pw)                                          # use cocitation algorithm to create an undirect
C2 = nx.to_numpy_array(Gc).T                                   # compute the adjacency matrix for the undirecte
Cdiff = C1-C2                                                  # compare two cocitation matrices
print('Difference between cocitation methods: %i' % Cdiff.sum().sum()) # print quantity of differencs between function
```

```
Difference between cocitation methods: 0
```

print out neighbors of the node "Linear Combination" in the cocitation network with edge weights

In [15]:
```
print(Gc['Linear Combination'])
```

```
{'Vector (Euclidean Space)': {'weight': 10.0}, 'Set of All Linear Transformations': {'weight': 1.0}, 'Ordered Basis':
{'weight': 3.0}, 'Linearly Independent/Sequence/Real Vector Space': {'weight': 4.0}, 'Linearly Dependent/Sequence/Real
Vector Space': {'weight': 6.0}, 'Linear Span': {'weight': 6.0}, 'Linear Combination of Subset': {'weight': 10.0}, 'Line
ar Combination of Sequence': {'weight': 8.0}, 'Linear Combination of Empty Set': {'weight': 6.0}, 'Matrix': {'weight':
1.0}, 'Basis (Linear Algebra)': {'weight': 2.0}, 'Matrix Product (Conventional)': {'weight': 1.0}, 'Module': {'weight':
8.0}, 'Linearly Independent/Set/Real Vector Space': {'weight': 1.0}, 'Linearly Dependent/Set/Real Vector Space': {'weig
ht': 2.0}, 'Linearly Independent/Set': {'weight': 1.0}, 'Linearly Independent/Sequence': {'weight': 1.0}, 'Linearly Ind
ependent Set': {'weight': 6.0}, 'Linearly Independent Sequence': {'weight': 10.0}, 'Linearly Independent': {'weight':
2.0}, 'Linearly Dependent/Set': {'weight': 2.0}, 'Linearly Dependent/Sequence': {'weight': 2.0}, 'Linearly Dependent Se
t': {'weight': 2.0}, 'Linearly Dependent Sequence': {'weight': 8.0}, 'Linearly Dependent': {'weight': 1.0}, 'Zero Vecto
r': {'weight': 10.0}, 'Zero Scalar': {'weight': 3.0}, 'Unitary Module': {'weight': 9.0}, 'Vector Space': {'weight': 3.
0}, 'Linear Transformation': {'weight': 7.0}, 'Vector Subspace': {'weight': 1.0}, 'Vector (Linear Algebra)': {'weight':
3.0}}
```

This means that for each neighbor with edge weight "n" printed on the above list, There exists "n" ProofWiki entries that site "Linear Combination" as well as that neighbor.

print the the neighbors of "Linear Combination" from the original graph and compare

In [16]:
```
print(G_pw['Linear Combination'])
```

```
{'Linear Transformation': {'weight': 1.0}, 'Linear Span': {'weight': 1.0}, 'Linear Combination of Subset': {'weight':
1.0}, 'Linear Combination/Empty Set': {'weight': 1.0}, 'Linear Combination of Sequence': {'weight': 1.0}, 'Linear Combi
nation/Subset': {'weight': 1.0}, 'Module': {'weight': 3.0}, 'Linear Combination/Sequence': {'weight': 1.0}}
```

create a new function to produce the bibliographic coupling network of the ProofWiki graph

In [17]:
```
def bibliographic_coupling(G):

    G_Bcoupling = nx.Graph()              # initialize cocitation graph
    G_Bcoupling.add_nodes_from(G.nodes()) # ensure all nodes match

    for u,v in itertools.combinations(G.nodes(),2):  # loop for every combination of nodes
        w = 0                                        # reset weight to be zero
        for i in G.successors(u):                    # loop for each successor in node u
            for j in G.successors(v):                # loop for each successor in node v
                if i == j:                           # bibliographic coupling exists
                    if 'weight' in G[u][i]:          # additional weight in u edge exists
                        u_weight = G[u][i]['weight'] # add appropriate weight to u edge
                    else:                            # weight of 1 in u exists
                        u_weight = 1                 # assign weight of 1 to u edge
                    if 'weight' in G[v][i]:          # additional weight in v exists
                        v_weight = G[v][i]['weight'] # add appropriate weight to v edge
                    else:                            # weight of 1 in v exists
                        v_weight = 1                 # assign weight of 1 to v edge
                    w = w + (u_weight*v_weight)      # add weight
        if w > 0:                                    # edge exists
            G_Bcoupling.add_edge(u,v,weight=w)       # create edge with appropriate weight

    return G_Bcoupling # return bibliographical coupled graph
```

test bibliographic coupling method

In [18]:
```
A = nx.to_numpy_array(G_pw).T # create adjacency matrix of proofwiki network in textbook format
B1 = numpy.dot(A.T,A)         # use linear algebra definition to create bibliographic coupling matrix
# loop added to ensure that the diagonal values are set to 0
for i in range(len(G_pw.nodes())):
    for j in range(len(G_pw.nodes())):
        B1[i,i] = 0
Gbc = bibliographic_coupling(G_pw)                                                  # use bibliographic coupling algorith
B2 = nx.to_numpy_array(Gbc).T                                                       # compute the adjacency matrix for th
Bdiff = B1-B2                                                                       # compare two bibliographic coupling
print('Difference between bibliographic couling methods: %i' % Bdiff.sum().sum()) # print quantity of differencs betwee
```

```
Difference between bibliographic couling methods: 0
```

Concept Question:

What is the original directed network and corresponding bibliographic coupling matrix for the network which has cocitation matrix:

In [19]:
```
C = [[0,4,4,4,4,4],
     [4,0,4,4,4,4],
     [4,4,0,4,4,4],
     [4,4,4,0,4,4],
     [4,4,4,4,0,4],
     [4,4,4,4,4,0]]
```

the following code block creates a system G that satisfies and proves this criteria

In [20]:
```
G = nx.DiGraph() # initialize G as a directed graph

G.add_nodes_from(range(1,7)) # add nodes 1,2,...6 in order

# create a network where every node points to every node that is not itself
for i in range(1,7):
    for j in range(1,7):
        if i == j:
            continue
        G.add_edge(i,j)

# create a cocitation network based on the graph G
G_c = cocitation(G)
C_test = nx.to_numpy_array(G_c).T

print('G is a directed system that has cocitation matrix C: %s' % (C==C_test).all()) # output result of test
```

```
G is a directed system that has cocitation matrix C: True
```

What is the corresponding bibliographic coupling matrix for this network

In [21]:
```
G_bc = bibliographic_coupling(G)
B = nx.to_numpy_array(G_bc).T
print(B)
```

```
[[0. 4. 4. 4. 4. 4.]
 [4. 0. 4. 4. 4. 4.]
 [4. 4. 0. 4. 4. 4.]
 [4. 4. 4. 0. 4. 4.]
 [4. 4. 4. 4. 0. 4.]
 [4. 4. 4. 4. 4. 0.]]
```

Concept Question:

Is it possible that two different (potentially weighted) original graphs G1 and G2 have the same cocitation and same bibliographic coupling graphs (e.g., C1=C2 and B1=B2)? If so, give an example.

The following code block creates two networks that have identical cocitation matrices and bibliographic matrices

In [22]:
```
# initialize two directed graphs
G1 = nx.DiGraph()
G2 = nx.DiGraph()

# add nodes 1,2,...,6 in order to both graphs
G1.add_nodes_from(range(1,7))
G2.add_nodes_from(range(1,7))

# G1: each node points to the node in front of it (+1)
# G2: each node points to the node behind it (-1)
for i in range(1,7):
    for j in range(1,7):
        if i+1 == j:
            G1.add_edge(i,j)
            G2.add_edge(j,i)
        elif i+5 == j:
            G1.add_edge(j,i)
            G2.add_edge(i,j)

# compute adjacency matrix for each graph
A1 = nx.to_numpy_array(G1)
A2 = nx.to_numpy_array(G2)

print('G1 and G2 are the same network: %s' % (A1==A2).all()) # print equivalency of adjacency matrices

# create cocitation and bibliographic coupling networks based on the original graphs G1 & G2
G_1c = cocitation(G1)
C1 = nx.to_numpy_array(G_1c)
G_2c = cocitation(G2)
C2 = nx.to_numpy_array(G_2c)
G_1b = bibliographic_coupling(G1)
B1 = nx.to_numpy_array(G_1b)
G_2b = bibliographic_coupling(G2)
B2 = nx.to_numpy_array(G_2b)
```

```
print('G1 and G2 have identical cocitation matrices: %s' % (C1==C2).all())            # print equivalency of cocitatio
print('G1 and G2 have identical bibliographic coupling matrices: %s' % (B1==B2).all()) # print equivalency of bibliogra
```

```
G1 and G2 are the same network: False
G1 and G2 have identical cocitation matrices: True
G1 and G2 have identical bibliographic coupling matrices: True
```

Section 6.4.2: Acyclic Networks

create a function that determines if a network is acyclic

In [23]:
```
def is_acyclic(G):

    has_removed_node = False # initialize false value

    while G.number_of_nodes() > 1:              # stop loop if all nodes have been removed
        if has_removed_node:                    # node was removed in previous loop
            G.remove_node(node_r)               # remove node without a successor
        has_removed_node = False                # reset node removed for this loop
        for i in G.nodes():                     # test every node in network
            if len(list(G.successors(i))) == 0: # find node without a successor
                node_r = i                      # get name of node without successsor
                has_removed_node = True         # node has been removed in this for loop
                break                           # move to the next line of code
        if has_removed_node:                    # ensure node has been removed
            continue                            # go to begining of while loop
        else:                                   # node has not been removed
            return False                        # system is not acyclic

    return True # system is acyclic
```

test if given networks are cyclic or acyclic

In [24]:
```
# create graphs for each given network
G1 = nx.read_weighted_edgelist('acyclic1.edgelist',create_using=nx.DiGraph)
G2 = nx.read_weighted_edgelist('acyclic2.edgelist',create_using=nx.DiGraph)
G3 = nx.read_weighted_edgelist('acyclic3.edgelist',create_using=nx.DiGraph)

# output result of acyclic test
print('System 1 is acyclic: %s' % is_acyclic(G1))
print('System 2 is acyclic: %s' % is_acyclic(G2))
print('System 3 is acyclic: %s' % is_acyclic(G3))
```

```
System 1 is acyclic: True
System 2 is acyclic: True
System 3 is acyclic: False
```

Section 6.5: Hypergraphs

I have read the Hypergraph section of the book

Section 6.6: Bipartite Networks

import the 2013 IMDB bipartite network

In [25]:
```
B = nx.read_gml('2013-actor-movie-bipartite.gml','name')
```

I believe that bibliographic coupling would be best to define the one-mode projection, because of the similarities of eq (6.17) & (6.10)

use the bibliographic coupling function to create a one-mode projection of the network & print the neighbors of Will Ferrell and Jason Statham

In [26]:
```
P = bibliographic_coupling(B)                                  # create one-mode projection using bibliographic coupli
print('Will Ferrell: %s' % list(P.neighbors('Will Ferrell')))   # print neighbors of Will Ferrell
print('Jason Statham: %s' % list(P.neighbors('Jason Statham'))) # print neighbors of Jason Statham
```

```
Will Ferrell: ['Brad Pitt', 'Matt Damon', 'Bradley Cooper', 'Mark Wahlberg', 'Melissa McCarthy', 'Ben Affleck', 'Dwayne
Johnson', 'Natalie Portman', 'Tina Fey', 'Steve Carell', 'Seth Rogen', 'Amy Adams', 'Ben Stiller', 'Jonah Hill', 'Paul
Rudd', 'Julianne Moore', 'Rachel McAdams', 'Kristen Wiig', 'Owen Wilson', 'Jason Bateman']
Jason Statham: ['Brad Pitt', 'Tom Cruise', 'Mark Wahlberg', 'Robert De Niro', 'Javier Bardem', 'Chris Evans', 'Charlize
Theron', 'Bruce Willis', 'Jamie Foxx', 'Sylvester Stallone', 'Liam Hemsworth']
```

This informaion could be used to learn more about these actors because you are able to see what costars they had, and what costars they had in common

What are the number of neighbors for Zac Efron & Clint Eastwood

In [27]:
```
print('Zac Efron Costars: %s' % list(P.neighbors('Zac Efron')))            # print list of Zac Efron costars
print('Degree: %i' % P.degree('Zac Efron'))                                # print degree of Zac Efron node (number of
print('Clint Eastwood Costars : %s' % list(P.neighbors('Clint Eastwood'))) # print list of Clint Eastwood costars
print('Degree: %i' % P.degree('Clint Eastwood'))                           # print degree of Clint Eastwood node (numbe
```

```
Zac Efron Costars: ['Robert De Niro']
Degree: 1
Clint Eastwood Costars : ['Meryl Streep']
Degree: 1
```

it is surprising that these famous actors have only 1 degreee in this network

Section 6.7: Trees

The difference between a directed tree and an directed acyclic graph is that a directed tree would still be a tree if the directions were ignored, meaning that there are no loops, even without directions, but a directed acyclic graph could have loops that exist if directions are ignored.

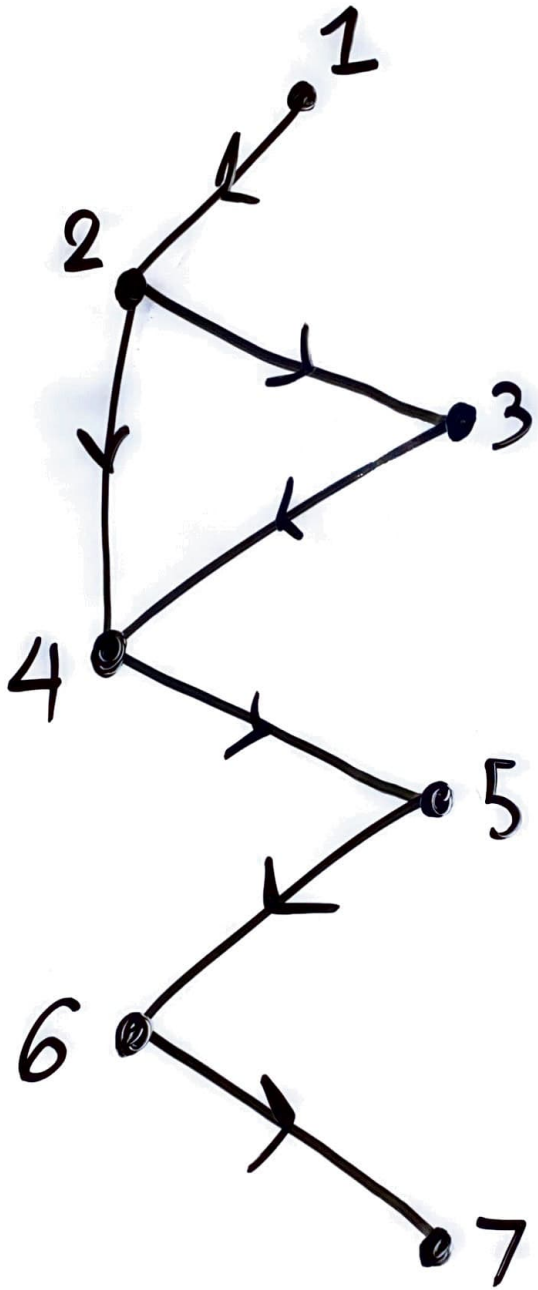
an examplee of a directed acyclic graph which is not a directed tree is shown in the PDF document

the example is a directed acyclic graph because all edges point downward, but if the directions are ignore, there is a loop containing nodes 2,3, & 4, therefore it is not a directed tree graph

Section 6.8: Planar Networks

there are no questions in this section

I have read the planar networks section of the book


1
2
3
4
5
6
7