Humza Salman mhs180007

In [7]:
```
import networkx as nx
import numpy as np
```

Section 6.1-6.2 Networks

In [8]:
```
g = nx.Graph()
```

In [9]:
```
g.add_nodes_from(range(1,7))
```

In [10]:
```
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)
```

In [11]:
```
print('HIGHLIGHTED QUESTION:-')
print(f'#Nodes: {g.number_of_nodes()} , #Edges: {g.number_of_edges()}')
```

```
HIGHLIGHTED QUESTION:-
#Nodes: 6 , #Edges: 7
```

In [12]:
```
print('HIGHLIGHTED QUESTION:-')
print(f'Has 3-4 edge: {g.has_edge(3,4)}')
```

```
HIGHLIGHTED QUESTION:-
Has 3-4 edge: True
```

In [13]:
```
print('HIGHLIGHTED QUESTION:-')
print(f'Has 4-6 edge: {g.has_edge(4,6)}')
```

```
HIGHLIGHTED QUESTION:-
Has 4-6 edge: False
```

In [15]:
```
A = nx.to_numpy_array(g)
print('HIGHLIGHTED QUESTION:- Print out the corresponding adjacency matrix:')
print(f'Adjacency Matrix: {A}')
```

```
HIGHLIGHTED QUESTION:- Print out the corresponding adjacency matrix:
Adjacency Matrix: [[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.]]
```

In [16]:
```
print('HIGHLIGHTED QUESTION:- running the command A == A.transpose()')
print(f'Adjacency Matrix Symmetric?\n {A == A.transpose()}')
```

```
HIGHLIGHTED QUESTION:- running the command A == A.transpose()
Adjacency Matrix Symmetric?
 [[ True  True  True  True  True  True]
 [ True  True  True  True  True  True]
 [ True  True  True  True  True  True]
 [ True  True  True  True  True  True]
 [ True  True  True  True  True  True]
 [ True  True  True  True  True  True]]
```

Section 6.3 Weighted Networks

In [17]:
```
del g
```

In [18]:
```
g = nx.Graph()
```

In [19]:
```
g.add_nodes_from(range(1,7))
```

In [20]:
```
g.add_edge(1, 2, weight=1)
g.add_edge(1, 5, weight=3)
g.add_edge(2, 2, weight=1) # will change weight to 2 manually later on
g.add_edge(2, 3, weight=2)
g.add_edge(2, 4, weight=1)
g.add_edge(3, 4, weight=1)
g.add_edge(3, 5, weight=1)
g.add_edge(3, 6, weight=1)
g.add_edge(6, 6, weight=1) # will change weight to 2 manually later on
```

In [21]:
```
A = nx.to_numpy_array(g)
print('HIGHLIGHTED QUESTION:- print the corresponding adjacency matrix')
print(f'Adjacency Matrix Before Fixing Weight of Self-Loops:')
print(A)
```

```
HIGHLIGHTED QUESTION:- print the corresponding adjacency matrix
Adjacency Matrix Before Fixing Weight of Self-Loops:
[[0. 1. 0. 0. 3. 0.]
 [1. 1. 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. 1.]]
```

In [22]:
```
g[2]
```

Out[22]:
```
AtlasView({1: {'weight': 1}, 2: {'weight': 1}, 3: {'weight': 2}, 4: {'weight': 1}})
```

In [23]:
```
g[2][2]['weight'] = 2
g[6][6]['weight'] = 2
```

In [24]:
```
A = nx.to_numpy_array(g)
print('HIGHLIGHTED QUESTION:- print the corresponding adjacency matrix')
print(f'Adjacency Matrix After Fixing Weight of Self-Loops:')
print(A)
```

```
HIGHLIGHTED QUESTION:- print the corresponding adjacency matrix
Adjacency Matrix After Fixing Weight of Self-Loops:
[[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

In [25]:
```
del g
g = nx.DiGraph()
```

In [26]:
```
g.add_nodes_from(range(1,7))
```

In [27]:
```
g.add_edge(1,3)
g.add_edge(2,6)
g.add_edge(3,2)
g.add_edge(4,1)
g.add_edge(4,5)
g.add_edge(5,3)
g.add_edge(6,4)
g.add_edge(6,5)
```

In [29]:
```
A = nx.to_numpy_array(g)
print('HIGHLIGHTED QUESTION:- compare the adjacency matrix to the one in the textbook'
print('Adjacency Matrix:')
print(A)
print('The adjacency matrix in the book is is the transpose of the matrix we have calc
```

```
HIGHLIGHTED QUESTION:- compare the adjacency matrix to the one in the textbook
Adjacency Matrix:
[[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 in the book is is the transpose of the matrix we have calculated
```

Section 6.4.1 EXAMPLE -- Cocitation & Bibliographic Coupling -- IGNORE

In [30]:
```
# Example for the benefit of creating the cocitation network algorithm
exg = nx.DiGraph()

exg.add_nodes_from(range(1,8))

exg.add_edge(1,7, weight=1)
exg.add_edge(3,1, weight=2)
exg.add_edge(3,4, weight=3)
exg.add_edge(4,1, weight=0.1)
exg.add_edge(4,2, weight=10)
exg.add_edge(4,3, weight=4)
exg.add_edge(5,1, weight=1)
exg.add_edge(5,2, weight=1)
exg.add_edge(6,2, weight=1)
exg.add_edge(7,2, weight=1)
```

```
# exg.add_edge(1,7, weight=1)
# exg.add_edge(3,1, weight=1)
# exg.add_edge(3,4, weight=1)
# exg.add_edge(4,1, weight=1)
# exg.add_edge(4,2, weight=1)
# exg.add_edge(4,3, weight=1)
# exg.add_edge(5,1, weight=1)
# exg.add_edge(5,2, weight=1)
# exg.add_edge(6,2, weight=1)
# exg.add_edge(7,2, weight=1)
print(exg)
```

```
DiGraph with 7 nodes and 10 edges
```

In [31]:

```
encg = nx.Graph()
for node in exg:
    encg.add_node(node)

for node1 in exg:
    incoming_node1 = list(exg.predecessors(node1))

    for node2 in exg:

        if (node1 != node2):
            if encg.has_edge(node1, node2):
                continue

            incoming_node2 = list(exg.predecessors(node2))

            intersection = list(set(incoming_node1) & set(incoming_node2))

            weighted_product_sum = 0

            for i in intersection: # for every node in the intersection
                product = 1
                print(node1, node2, i)
                product = 1
                for j in exg[i]: # for every node being pointed to
                    # print(i, j)
                    # product = 1
                    if 'weight' in exg[i][j] and (j == node1 or j == node2):
                        print(i, j)
                        product *= exg[i][j]['weight']

                weighted_product_sum += product

            if intersection:

                encg.add_edge(node1, node2, weight=weighted_product_sum)

# print(encg)
```

```
1 2 4
4 1
4 2
1 2 5
5 1
5 2
1 3 4
4 1
4 3
1 4 3
3 1
3 4
2 3 4
4 2
4 3
```

In [32]:

```
A = nx.to_numpy_array(exg).T
C_algebraic = np.dot(A, A.transpose())
np.fill_diagonal(C_algebraic, 0)
print(C_algebraic)
G_cocitation = encg
C_graph = nx.to_numpy_array(encg).T
print(C_graph)
C_diff = C_algebraic - C_graph
print(C_algebraic.shape)
print(C_graph.shape)
print(f'Difference between cocitation methods: {C_diff.sum().sum()}')
```

```
[[ 0.   2.   0.4  6.   0.   0.   0. ]
 [ 2.   0.  40.   0.   0.   0.   0. ]
 [ 0.4 40.   0.   0.   0.   0.   0. ]
 [ 6.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0. ]]
[[ 0.   2.   0.4  6.   0.   0.   0. ]
 [ 2.   0.  40.   0.   0.   0.   0. ]
 [ 0.4 40.   0.   0.   0.   0.   0. ]
 [ 6.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0. ]]
(7, 7)
(7, 7)
Difference between cocitation methods: 0.0
```

Section 6.4.1 Cocitation & Bibliographic Coupling

In [34]:

```
def cocitation(g):
    cg = nx.Graph() # new undirected graph

    # add all nodes ot new cocitation graph
    for node in g:
        cg.add_node(node)

    # loop through possible permutations of different pairs of nodes
    for node1 in g:

        incoming_node1 = list(g.predecessors(node1)) # get predecessors for first node
```

```
        for node2 in g:

            if (node1 != node2): # make sure nodes are not the same

                # check to see if edge already exists, if so then skip
                if cg.has_edge(node1, node2):
                    continue

                incoming_node2 = list(g.predecessors(node2)) # get predecessors for se

                intersection = list(set(incoming_node1) & set(incoming_node2))

                weighted_product_sum = 0

                for i in intersection: # for every node in the intersection
                    product = 1

                    for j in g[i]: # for every node being pointed to

                        if 'weight' in g[i][j] and (j == node1 or j == node2): # make
                            product *= g[i][j]['weight']

                    weighted_product_sum += product # sum product for those nodes

                if intersection:
                    cg.add_edge(node1, node2, weight=weighted_product_sum)

    return cg
```

In [35]:

```
del g
g = nx.read_gml('proofwikidefs_la.gml', 'name')
```

In [36]:

```
A = nx.to_numpy_array(g).T
C_algebraic = np.dot(A, A.transpose())
np.fill_diagonal(C_algebraic, 0)
G_cocitation = cocitation(g)
C_graph = nx.to_numpy_array(G_cocitation).T
C_diff = C_algebraic - C_graph
print('HIGHLIGHTED QUESTION:-')
print(f'Difference between cocitation methods: {C_diff.sum().sum()}')
```

```
HIGHLIGHTED QUESTION:-
Difference between cocitation methods: 0.0
```

In [37]:

```
print("HIGHLIGHTED QUESTION:- The sum should be zero - but it isn't! Why not? What did
print('Initially the difference of the cocitation methods was not 0 because we calcula
      'This creates 0 entries in the diagonal matrix, so we had to adjust our Algebrai
```

```
HIGHLIGHTED QUESTION:- The sum should be zero - but it isn't! Why not? What did we mi
ss?

Initially the difference of the cocitation methods was not 0 because we calculated ou
r cocitation graph assuming a simple graph.
This creates 0 entries in the diagonal matrix, so we had to adjust our Algebraic meth
od adjacency matrix to have diagonal entries of 0.
```

In [41]:

```
print('HIGHLIGHTED QUESTION:- Print out the neighbors of this node in the cocitation r
      'phrase that captures the meaning of these neighbors. Compare these with the in-
      'printing both.\n')
```

```
print('Cocitation Network neigbors of Linear Combination:')
for n in G_cocitation['Linear Combination']:
    print('\t', n, G_cocitation[n]['Linear Combination']['weight'])

print()
print('These neighbors in the cocitation network represent an edge between two nodes t
      'The weight of this edge represents the number of terms referencing these nodes
print()


print('In-Neighbors of Linear Combination:')
for n in g.predecessors('Linear Combination'):
    print('\t', n, g[n]['Linear Combination']['weight'])

print()
print('The in-neighbors of Linear Combination represent the nodes/terms that reference
      'the neighbors of the Linear Combination node in the cocitation network. These i
```

```
HIGHLIGHTED QUESTION:- Print out the neighbors of this node in the cocitation network
along with the weights of the edges they share. Write a concise
phrase that captures the meaning of these neighbors. Compare these with the in-neighb
ors of “Linear Combination” in the original graph by
printing both.

Cocitation Network neigbors of Linear Combination:
         Vector (Euclidean Space) 10.0
         Set of All Linear Transformations 1.0
         Ordered Basis 3.0
         Linearly Independent/Sequence/Real Vector Space 4.0
         Linearly Dependent/Sequence/Real Vector Space 6.0
         Linear Span 6.0
         Linear Combination of Subset 10.0
         Linear Combination of Sequence 8.0
         Linear Combination of Empty Set 6.0
         Matrix 1.0
         Basis (Linear Algebra) 2.0
         Matrix Product (Conventional) 1.0
         Module 8.0
         Linearly Independent/Set/Real Vector Space 1.0
         Linearly Dependent/Set/Real Vector Space 2.0
         Linearly Independent/Set 1.0
         Linearly Independent/Sequence 1.0
         Linearly Independent Set 6.0
         Linearly Independent Sequence 10.0
         Linearly Independent 2.0
         Linearly Dependent/Set 2.0
         Linearly Dependent/Sequence 2.0
         Linearly Dependent Set 2.0
         Linearly Dependent Sequence 8.0
         Linearly Dependent 1.0
         Zero Vector 10.0
         Zero Scalar 3.0
         Unitary Module 9.0
         Vector Space 3.0
         Linear Transformation 7.0
         Vector Subspace 1.0
         Vector (Linear Algebra) 3.0

These neighbors in the cocitation network represent an edge between two nodes that ar
e being referenced by the same node/term(s) in the original graph.
The weight of this edge represents the number of terms referencing these nodes as wel
l as the weight associated with those references.

In-Neighbors of Linear Combination:
         Spanning Set 1.0
         Linearly Dependent/Sequence/Real Vector Space 1.0
         Linear Span 1.0
         Linear Combination/Subset 1.0
         Linear Combination/Sequence 1.0
         Linear Combination/Empty Set 1.0
         Linear Combination of Subset 1.0
         Linear Combination of Sequence 1.0
         Linear Combination of Empty Set 1.0
         Generator/Module/Spanning Set 1.0
         Relative Matrix 1.0
         Linearly Independent/Sequence 1.0
         Linearly Independent Sequence 1.0
         Linearly Independent 1.0
```

```
Linearly Dependent/Sequence 2.0
Linearly Dependent Sequence 2.0
Linearly Dependent 2.0
```

```
The in-neighbors of Linear Combination represent the nodes/terms that reference the L
inear Combination node and potentially the other nodes seen in
the neighbors of the Linear Combination node in the cocitation network. These influen
ce the connections made in the cocitation network.
```

In [42]:
```python
def bibliographic_coupling(g):
    bg = nx.Graph() # new undirected graph

    # add all nodes ot new cocitation graph
    for node in g:
        bg.add_node(node)

    # loop through possible permutations of different pairs of nodes
    for node1 in g:

        incoming_node1 = list(g.successors(node1)) # get successors for first node

        for node2 in g:

            if (node1 != node2): # make sure nodes are not the same

                # check to see if edge already exists, if so then skip
                if bg.has_edge(node1, node2):
                    continue

                incoming_node2 = list(g.successors(node2)) # get successors for second

                # print(node1, incoming_node1, node2, incoming_node2)

                intersection = list(set(incoming_node1) & set(incoming_node2))

                weighted_product_sum = 0

                if intersection:
                    # print(node1, node2, intersection)
                    product = 0
                    for i in intersection:
                        product += g[node1][i]['weight'] * g[node2][i]['weight']
                        # print(g[node1][i]['weight'])
                        # print(g[node2][i]['weight'])
                    weighted_product_sum += product
                    bg.add_edge(node1, node2, weight=weighted_product_sum)


    return bg
```

In [43]:
```python
A = nx.to_numpy_array(g).T
BC_algebraic = np.dot(A.transpose(), A)
np.fill_diagonal(BC_algebraic, 0)
G_bibliographic = bibliographic_coupling(g)
BC_graph = nx.to_numpy_array(G_bibliographic).T
BC_diff = BC_algebraic - BC_graph
print('HIGHLIGHTED QUESTION:-')
print(f'Difference between bibliographic methods: {BC_diff.sum().sum()}')
```

```
HIGHLIGHTED QUESTION:-
Difference between bibliographic methods: 0.0
```

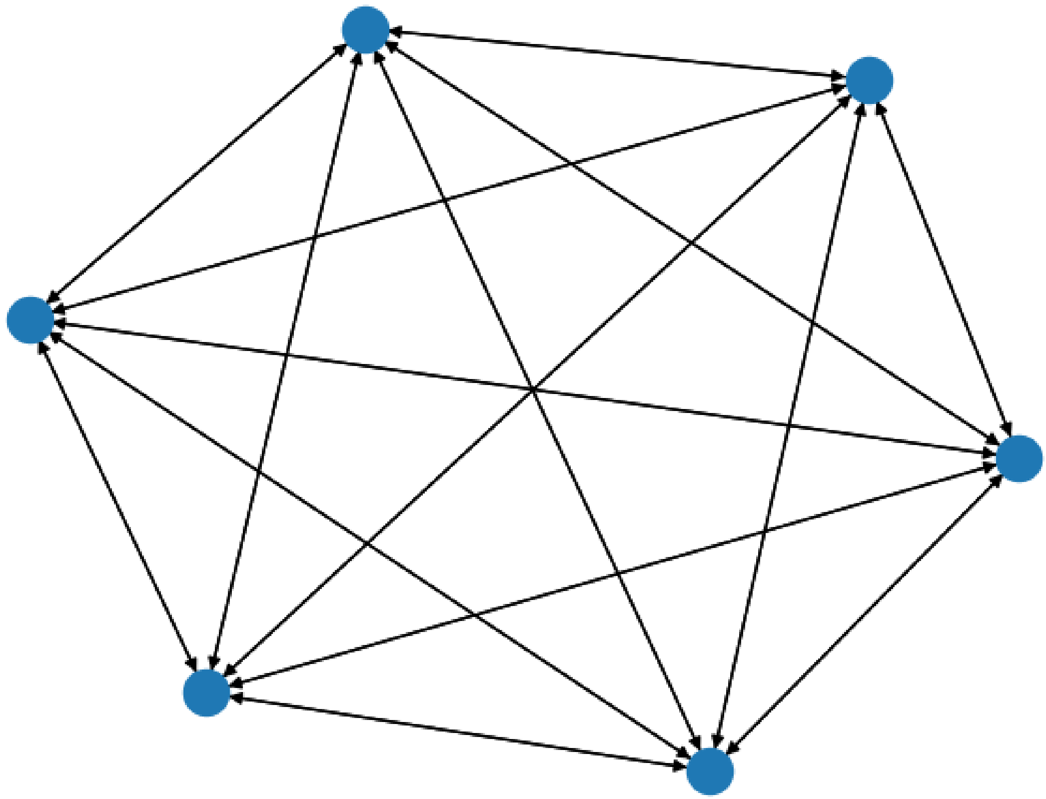
In [ ]:
```
test = nx.DiGraph()
test.add_nodes_from(range(1,7))
for n1 in test.nodes():
    for n2 in test.nodes():
        if n1 != n2:
            test.add_edge(n1, n2, weight=1)
test_cocitation = cocitation(test)
print(test_cocitation)
print(test)


test_graph = nx.to_numpy_array(test_cocitation).T
print(test_graph)
```

In [69]:
```
print('HIGHLIGHTED QUESTION:-')
print('What is the original directed network (draw or describe) that has a cocitation
print(test_graph)
print('The original graph will be a K6 complete graph with edge weights of 1')
nx.draw(test)
print()
print('What is the corresponding bibliographic coupling matrix for this network?')
print('The corresponding matrix would be the same since C = (A)(A.T) and B = (A.T)(A),
print(test_graph.T)
```

```
HIGHLIGHTED QUESTION:-
What is the original directed network (draw or describe) that has a cocitation matrix
given by:
[[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 original graph will be a K6 complete graph with edge weights of 1

What is the corresponding bibliographic coupling matrix for this network?
The corresponding matrix would be the same since C = (A)(A.T) and B = (A.T)(A), so B
= C.T and since the matrix given is symmetric, then B = C in this case
[[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.]]
```



In [70]:
```
print('HIGHLIGHTED QUESTION:- Is it possible that two different (potentially weighted)
      'bibliographic coupling graphs (e.g., C1=C2 and B1=B2)? If so, give an example.'
print('Yes. Take the example weighted graph in slide 14 of the Class 02 slides. If we
      'predecessor between them. In the original graph edge(3,1) has a weight of 2 and
      'weights we would have: 1. a different graph and 2. the same result in a cocitat
      'and sum of products remained the same.')
```

```
HIGHLIGHTED QUESTION:- Is it possible that two different (potentially weighted) origi
nal 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.

Yes. Take the example weighted graph in slide 14 of the Class 02 slides. If we take n
odes 1, 3, and 4 as an example we find that 3 is the common
predecessor between them. In the original graph edge(3,1) has a weight of 2 and edge
(3,4) has a weight of 3. If we simply flipped these two edge
weights we would have: 1. a different graph and 2. the same result in a cocitation ne
twork and bibliographic network since the connections
and sum of products remained the same.
```

Section 6.4.2 Acyclic Networks

In [74]:
```
def is_acyclic(g):
    nodes = list(g.nodes())

    i = 0
    while i < len(nodes): # i in range(len(nodes)):

        if len(list(g.successors(nodes[i]))) == 0:
            g.remove_node(nodes[i])
            del nodes[i]
```

```
            i = 0
            continue
        i += 1

    if len(g.nodes()) != 0:
        return False

    return True
        # print(n)
```

In [75]:
```
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)
```

In [76]:
```
print('Checking to see true answers:')
print(nx.is_directed_acyclic_graph(g1))
print(nx.is_directed_acyclic_graph(g2))
print(nx.is_directed_acyclic_graph(g3))
```

```
Checking to see true answers:
True
True
False
```

In [77]:
```
print('HIGHLIGHTED QUESTION:- Implement the simple algorithm introduced in this sectic
      'Run your algorithm on the three networks supplied: acyclic1.edgelist, acyclic2.
print('Predicted Answers:')
print('acyclic1:', is_acyclic(g1))
print('acyclic2:', is_acyclic(g2))
print('acyclic3:', is_acyclic(g3))
```

```
HIGHLIGHTED QUESTION:- Implement the simple algorithm introduced in this section (as
a new function) to determine whether a network is acyclic or not.
Run your algorithm on the three networks supplied: acyclic1.edgelist, acyclic2.edgeli
st, and acyclic3.edgelist

Predicted Answers:
acyclic1: True
acyclic2: True
acyclic3: False
```

Section 6.5 Hypergraphs

In [79]:
```
print('These are graphs with hyperedges where a hyperedge is capable of joining more t
```

```
These are graphs with hyperedges where a hyperedge is capable of joining more than tw
o nodes at a time.
```

Section 6.6 Bipartite Networks

In [80]:
```
B = nx.read_gml('2013-actor-movie-bipartite.gml','name')
```

In [82]:
```
g = bibliographic_coupling(B)
```

In [83]:
```
print('HIGHLIGHTED QUESTION:-If we wanted to find the one-mode projection of this bipa
      'or bibliographic coupling? Write a sentence to justify your choice\n')
print('We would use bibliographic coupling to find a one-mode projection of this graph
      'of nodes Actors and Nodes; so, we can create connections between actors by seei
```

```
HIGHLIGHTED QUESTION:-If we wanted to find the one-mode projection of this bipartite
network onto the actors/actresses, would we use cocitation
or bibliographic coupling? Write a sentence to justify your choice

We would use bibliographic coupling to find a one-mode projection of this graph since
Actors are linked to Movies. The graph has two distinct types
of nodes Actors and Nodes; so, we can create connections between actors by seeing whi
ch movies they have worked in together.
```

In [84]:

```
A = nx.to_numpy_array(g).T
print(A)
P = np.dot(A, A.transpose())
print(P)
```

```
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
[[23.  0.  0. ...  0.  0.  0.]
 [ 0. 21.  0. ...  0.  0.  0.]
 [ 0.  0.  3. ...  0.  0.  0.]
 ...
 [ 0.  0.  0. ...  0.  0.  0.]
 [ 0.  0.  0. ...  0.  0.  0.]
 [ 0.  0.  0. ...  0.  0.  0.]]
```

In [87]:

```
print('HIGHLIGHTED QUESTION:- Will Ferrell and Jason Statham. Print out their immediat
print(f"WILL FERREL: {list(g.neighbors('Will Ferrell'))}")
print(f"JASON STATHAM: {list(g.neighbors('Jason Statham'))}")
```

```
HIGHLIGHTED QUESTION:- Will Ferrell and Jason Statham. Print out their immediate neig
hbors in the one-mode projection

WILL FERREL: ['Brad Pitt', 'Matt Damon', 'Bradley Cooper', 'Mark Wahlberg', 'Melissa
McCarthy', 'Ben Affleck', 'Dwayne Johnson', 'Natalie Portman', 'Tina Fey', 'Steve Car
ell', '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 S
tallone', 'Liam Hemsworth']
```

In [89]:

```
print("HIGHLIGHTED QUESTION:- Suppose you don't know who Will Ferrell and Jason Statha
      "in the one-mode projection to learn more about them?\n")

print('You can use this information to learn the names of people linked to Will Ferrel
      'them then you could see that they are all names of actors/actresses that know e
```

```
HIGHLIGHTED QUESTION:- Suppose you don't know who Will Ferrell and Jason Statham are
- how could you use this information about their neighbors
in the one-mode projection to learn more about them?

You can use this information to learn the names of people linked to Will Ferrel and J
ason Statham, so if you recognize any of the names linked to
them then you could see that they are all names of actors/actresses that know each ot
her by having worked together.
```

In [90]:
```
print('HIGHLIGHTED QUESTION:- What are the number of neighbors of newcomers like Zac E
      'Is it surprising that they have low degrees in this network?\n')

print(f"ZAC EFRON worked with {len(list(g.neighbors('Zac Efron')))} actor(s)/actress(e
print(f"CLINT EASTWOOD EFRON worked with {len(list(g.neighbors('Clint Eastwood')))} ac

print()

print('Since they are new and old comers it is not surprising that in 2013 these actor
      'they did not work on a lot of movies at this time due to them being new and not
```

```
HIGHLIGHTED QUESTION:- What are the number of neighbors of newcomers like Zac Efron a
nd old-timers like Clint Eastwood?
Is it surprising that they have low degrees in this network?

ZAC EFRON worked with 1 actor(s)/actress(es): ['Robert De Niro']
CLINT EASTWOOD EFRON worked with 1 actor(s)/actress(es): ['Meryl Streep']

Since they are new and old comers it is not surprising that in 2013 these actors did
not have as many connections/degrees in the network because
they did not work on a lot of movies at this time due to them being new and not havin
g as many opportunities, or being past their move-making prime.
```

Section 6.7 Trees

In [93]:
```
print('HIGHLIGHTED QUESTION:-  Describe how a directed tree is different from an acycl
print('A directed tree can only have one path to the vertices it is legally allowed to
      'to another. A directed tree must also have n-1 edges, whereas an acyclic networ
```

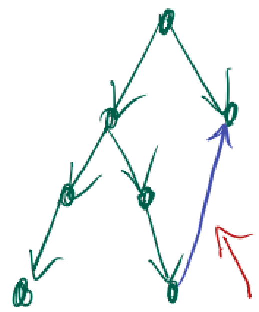
```
HIGHLIGHTED QUESTION:-  Describe how a directed tree is different from an acyclic net
work
A directed tree can only have one path to the vertices it is legally allowed to visi
t, whereas an acyclic network can have multiple paths from one vertex
to another. A directed tree must also have n-1 edges, whereas an acyclic network is n
ot restricted by these bounds.
```

In [95]:
```
print('HIGHLITED QUESTIONS:- Draw a 7 node directed acyclic graph that is not a direct
      'to make the graph a directed tree. Draw this graph so that all edges are pointi

print('The graph depicts a directed tree in green and the blue edge weight is what it
      'If we remove the blue edge we get a directed tree. If we are talking about the
      'We could have drawn the blue edge between the bottom two leaves instead and sti
```

```
HIGHLITED QUESTIONS:- Draw a 7 node directed acyclic graph that is not a directed tre
e, then highlight which edges you would need to remove
to make the graph a directed tree. Draw this graph so that all edges are pointing dow
nwards or sideways. Is this set of edges always unique?

The graph depicts a directed tree in green and the blue edge weight is what it would
be if it was an acyclic graph.
If we remove the blue edge we get a directed tree. If we are talking about the acycli
c graph, this set of edges is not always unique.
We could have drawn the blue edge between the bottom two leaves instead and still ach
eived the same result.
```



Directed Tree
↳ n-1 nodes
↳ only one path b/w vertices

DAG
↳

remove this edge to make a directed tree

Section 6.8 Planar Networks

In [96]:
```
print('Cool fact: A graph can be planar but still have edges that cross each other. If
```

```
Cool fact: A graph can be planar but still have edges that cross each other. If it ca
n be drawn such that edges do not cross, then it is planar.
```

In [ ]:
```

```