CS4423-Assignment-1

February 25, 2025

Table of Contents

1 CS4423 Assignment 1: Solution Template

This is a template for your solution to the `networkx` questions on Assignment 1.

1.0.1 Collaboration Policy

This is a homework assignment. You are welcome to collaborate with class-mates if you wish. Please note: * You may collaborate with at most two other people; * Each of you must submit your own copy of your work; * The file(s) you submit must contain a statement on the collaboration: who you collaborated with, and on what part of the assignment. * The use of any AI tools, such as ChatGPT or DeepSeek is prohibited, and will be subject to disciplinary procedures.

1.0.2 Instructions

This assignment involves a mix of questions, some of which require use of the `networkx` Python module, and some which you solve by hand. You can decide the best way to submit your work (e.g., do everything in Jupyter, or a combination of hand-written work and Jupyter notebook). However: * Any file you submit must include your name and ID number. * All files must be in PDF format. To convert your notebook to `pdf` the easiest method maybe to first export as 'html', then open that in a browser, and then print to pdf.

1.0.3 Preliminaries

Name: Andrew Hayes

ID Number: 21321503

1.0.4 Usual list of Python modules

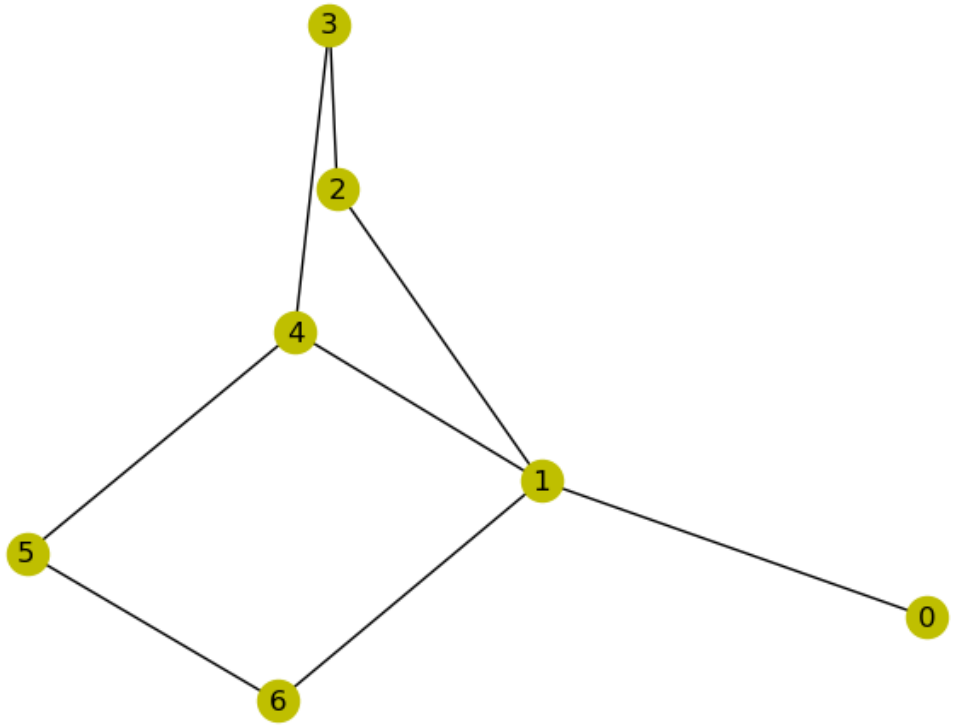
[5]:
```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
opts = { "with_labels": True, "node_color": 'y' } # show labels; yellow noodes
```

1.1 Q1: Bipartite Graphs

1.1.1 Define and draw the following graph

Let G_1 be the graph on the nodes $\{0, 1, 2, 3, 4, 5, 6\}$ with edges $0-1$, $1-2$, $1-4$, $1-6$, $2-3$, $3-4$, $4-5$, $5-6$.
Define this graph in `networkx` and draw it.

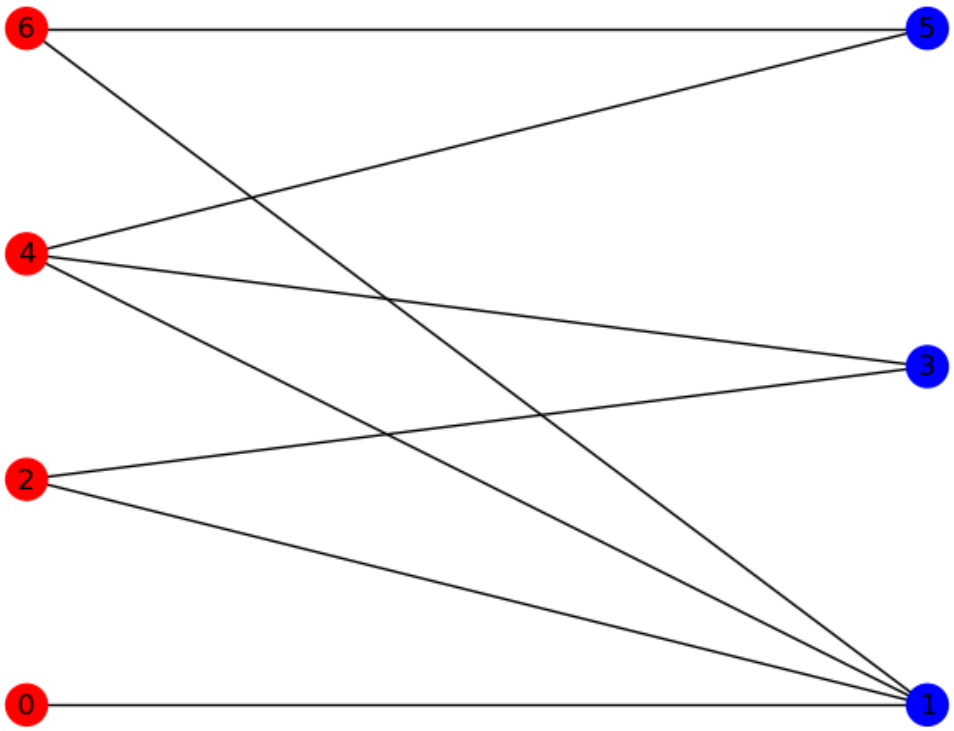
[8]:
```
G1 = nx.Graph()
G1.add_edges_from([(0,1), (1,2), (1,4), (1,6), (2,3), (3,4), (4,5), (5,6)])
nx.draw(G1, **opts)
```



1.1.2 Determine if G_1 is bipartite.

If G_1 is bipartite, draw it in `networkx` with a two-colouring of the nodes. If not, explain why it is not bipartite.

[17]:
```
if nx.is_bipartite(G1):
    partition1, partition2 = nx.bipartite.sets(G1)
    nodes = list(G1.nodes())
    colours = ['r' if node in partition1 else 'b' for node in nodes]
    positions = nx.bipartite_layout(G1, partition1) # optional extra, computing␣
↪easy-to-understand positions for nodes in the bipartite graph
    nx.draw(G1, node_color=colours, pos=positions, with_labels=True)
```



1.2 Q2: A Network of friends

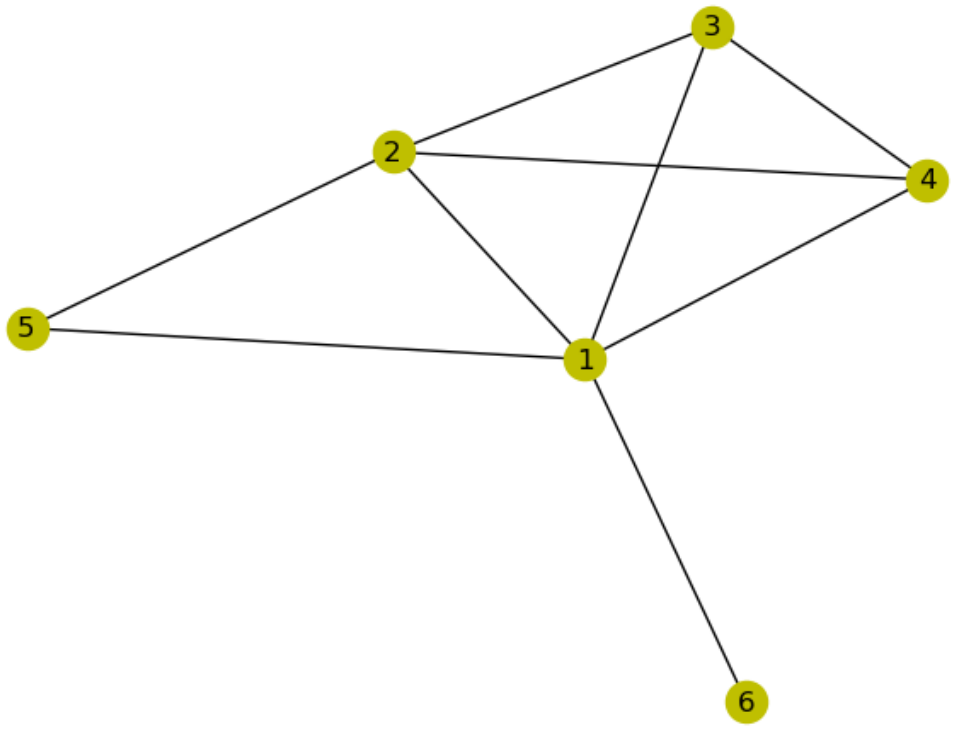
At a party with $n = 6$ people, some people know each other already while others don't. Each of the 6 guests is asked how many friends they have at this party.
One person says they know all of the others.
One person says they know four of the others.
Two report that they know three of the others.
One person agrees they know two of the other guests, while
one person says they know only one other.

1.2.1 Define a graph in `networkx` that represents this scenario.

[25]:
```
p1 = [(1,2), (1,3), (1,4), (1,5), (1,6)] # person who knows everyone else
p2 = [(2,1), (2,3), (2,4), (2,5)] # person who knows four of the others
p3 = [(3,1), (3,2), (3,4)] # knows three of the others
p4 = [(4,1), (4,2), (4,3)] # knows three
p5 = [(5,1), (5,2)] # knows two
p6 = [(6,1)]

edges = p1 + p2 + p3 + p4 + p5 + p6
```

```
G2 = nx.Graph()
G2.add_edges_from(edges)
nx.draw(G2, **opts)
```



1.2.2 Verify that the graph has the correct properties by displaying the diagonal of the square of the graphs adjacency matrix.

Hint: `np.diag(X)` returns the entries on the main diagonal of X.

[44]:
```
A = nx.adjacency_matrix(G2).toarray()
diagonal_of_square = np.diag(A@A)

print(diagonal_of_square)

# the diagonal of the square of the adjacency matrix tell you how many paths of␣
 ↪length 2 start and end at each matrix
# in this situation, it tells us how many people each person knows
# we can throw an error if the diagonal of the square does not have the value we␣
 ↪expect.
# we expect 5 connections for the first person, 4 for the second, 3, 3, 2, and 1
```

```
assert all(a == b for a,b in zip(diagonal_of_square, [5, 4, 3, 3, 2, 1]))
```

```
[5 4 3 3 2 1]
```

1.3 Q3: Isomorphic graphs

In `networkx` we can check if two (smallish) graphs, G and H, are **isomorphic** by using the `nx.is_isomorphic()` function: `nx.is_isomorphic(G,H)` evaluates as `True` if they are isomorphic.

1.3.1 Self-complementary cycle graph

Use `networkx` to check which of the cycle graphs C_3, C_4, ..., C_{10} are isomorphic to its own complement.
Notes: * You can use the constructor `nx.cycle_graph(n)` to make C_n * You can use the method `nx.complement(G)` to make construct the complement of the graph G.

[49]:
```
for n in range(3,11):
    graph = nx.cycle_graph(n)
    complement = nx.complement(graph)
    print(f"Is C{n} isomorphic? {nx.is_isomorphic(graph, complement)}")
```

```
Is C3 isomorphic? False
Is C4 isomorphic? False
Is C5 isomorphic? True
Is C6 isomorphic? False
Is C7 isomorphic? False
Is C8 isomorphic? False
Is C9 isomorphic? False
Is C10 isomorphic? False
```

1.3.2 Graphs that are isomorphic to their line graphs.

Use `networkx` to check that all cycle graphs C_3, C_4, ..., C_{10} are isomorphic to their line graphs.

You can use the `nx.line_graph()` function.

[53]:
```
for n in range(3,11):
    graph = nx.cycle_graph(n)
    line_graph = nx.line_graph(graph)
    print(f"Is C{n} isomorphic to its line graph? {nx.is_isomorphic(graph,␣
 ↪line_graph)}") # a cycle graph is always isomorphic to its line graph
```

```
Is C3 isomorphic to its line graph? True
Is C4 isomorphic to its line graph? True
Is C5 isomorphic to its line graph? True
Is C6 isomorphic to its line graph? True
Is C7 isomorphic to its line graph? True
Is C8 isomorphic to its line graph? True
Is C9 isomorphic to its line graph? True
Is C10 isomorphic to its line graph? True
```

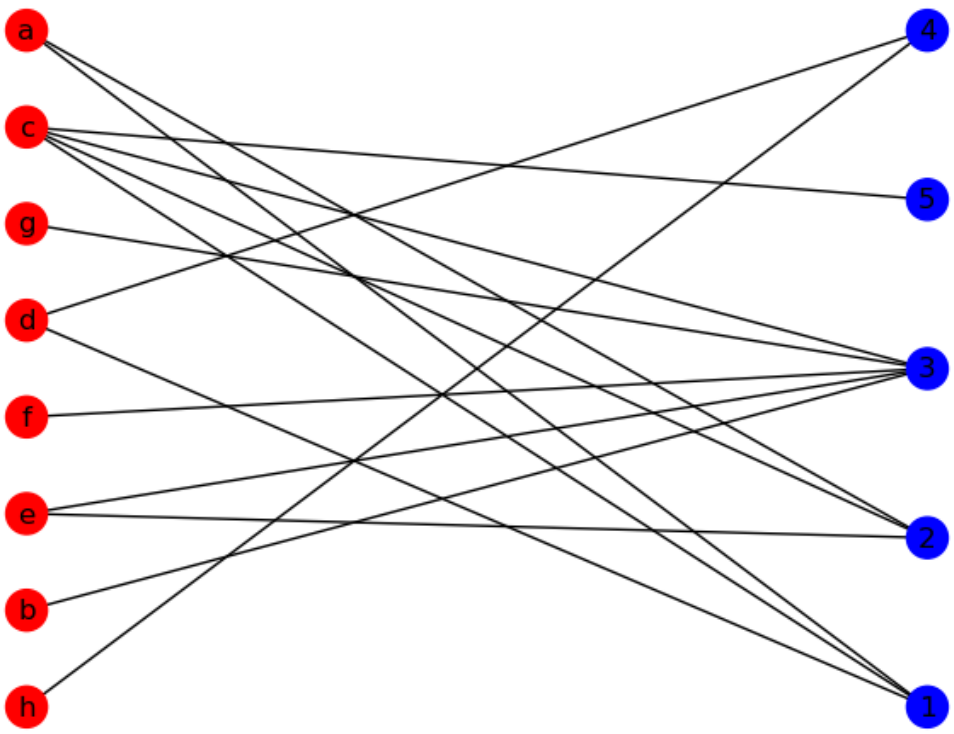
1.4 Q4: Bipartite Graphs

Consider the following affiliation network, G_4, with 8 people labelled a to h, and five foci ("focal points" of interaction) labelled 1 to 5:

1. Create this graph in `networkx` and draw it with a two-colouring.
2. Compute the adjacency matrix of G.
3. Draw the projection on (just) the people, in which two people are joined by an edge if they have a common focus. (You can do this by hand, or in `networkx`.)

[59]:
```
G4 = nx.Graph()
G4.add_edges_from([('a', 1), ('a', 2), ('b', 3), ('c', 1), ('c', 5), ('c', 3),␣
↪('c', 2), ('d', 1), ('d', 4), ('e', 3), ('e', 2), ('f', 3), ('g', 3), ('h',␣
↪4)])


partition1, partition2 = nx.bipartite.sets(G4)
nodes = list(G4.nodes())
colours = ['r' if node in partition1 else 'b' for node in nodes]
positions = nx.bipartite_layout(G4, partition1)
nx.draw(G4, node_color=colours, pos=positions, with_labels=True)
```

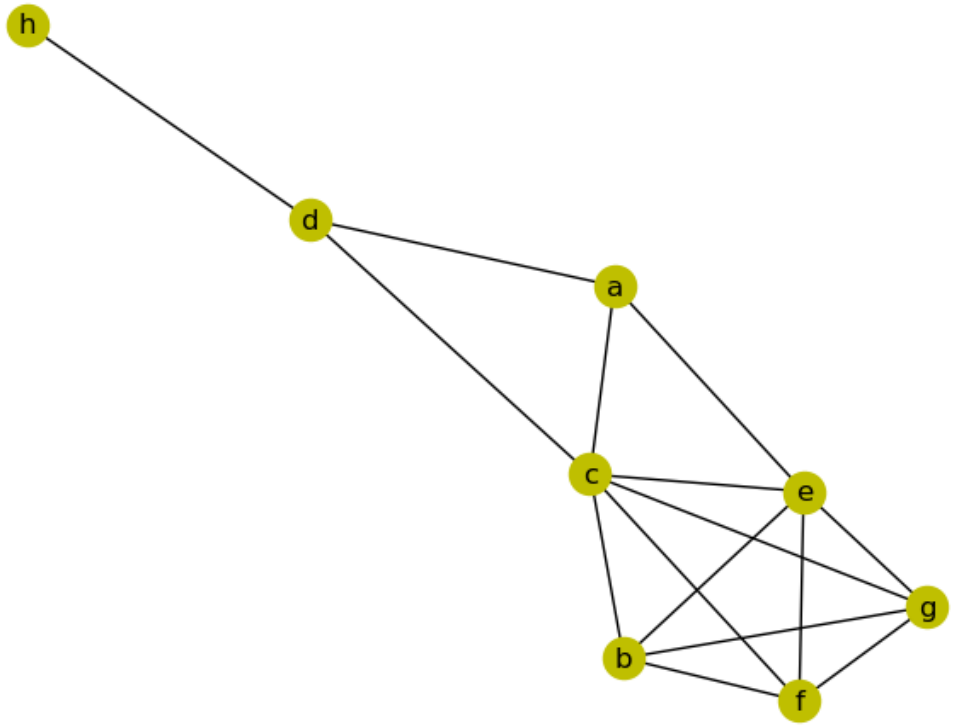


[64]:
```
# 2. Adjacency matrix
A = nx.adjacency_matrix(G4).toarray()
print(A)
```

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

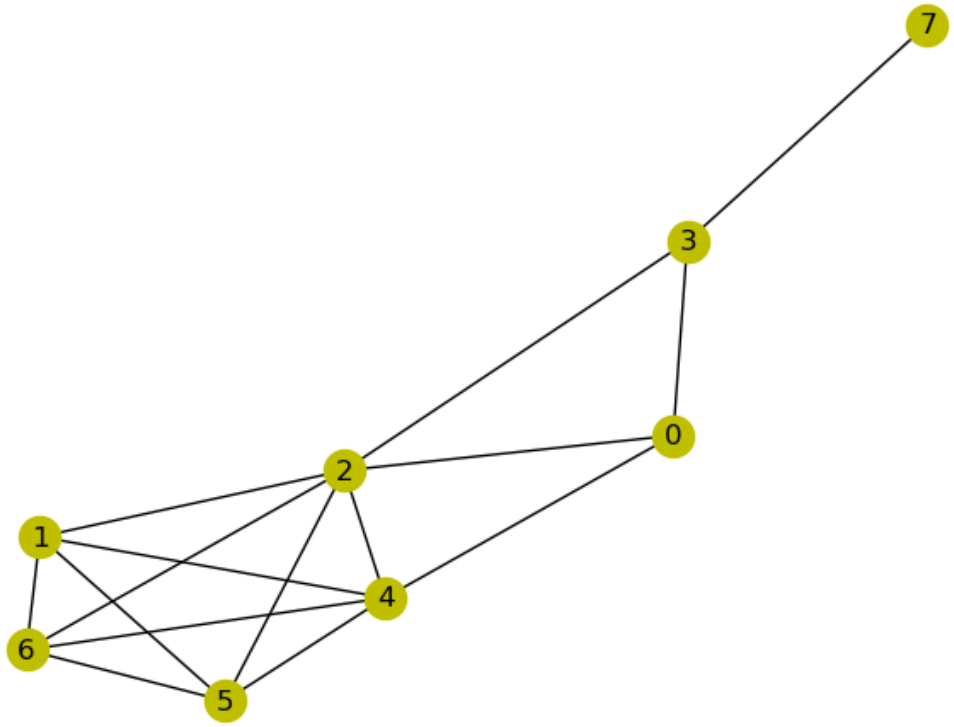
[78]:
```
# 3. Compute and draw the projection
# easy way
people = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
foci = [i for i in range (1, 6)]

G_V1 = nx.projected_graph(G4, people)
nx.draw(G_V1, **opts)
```



[77]:
```
## hard way
B = nx.bipartite.biadjacency_matrix(G4, people, foci).toarray()
C = B@B.T

# make graph simple
C[C>0] = 1 # make edges either exist or not
np.fill_diagonal(C, 0) # remove self adjacency
G_V1_hard = nx.from_numpy_array(C)
nx.draw(G_V1_hard, **opts)
```



1.5 Q5: Directed Graphs

For this question, the use of `networkx` *is optional. You may write out your solution if you prefer.*

So far in CS4423 we have only considered **undirected graphs**. That is, the edge $a - b$ is the same as the edge $b - a$.

Now I want you to think about *directed graphs* (also called *digraphs*): where the edge $a \to b$ is not the same as the edge $b \to a$. One can think of such edges as "one way streets": an edge that can be used to get from a to b can't be used to get from b to a.

There are numerous differences between directed and undirected graphs, including: * When you draw a digraph you add arrows to edges to indicate its direction. * If there is an edge $u \to v$ and $v \to u$, this can be represented by either having two edges between these nodes (with arrows in opposite directions), or by adding two arrows to a single edge. * The adjacency matrix is not necessarily symmetric. * The graph may have a path from node u to node v, but not from v to u. * We talk of a digraph being * **Strongly Connected** meaning there is a path between every pair of nodes * **Weakly Connected** meaning, for every pair of nodes, u and v, there is a path from u to v, or from v to u. * **Disconnected** (same as the usual meaning of disconnected). * In `networkx` we construct a directed graph with the `nx.DiGraph()` constructor.

Here is an example of a digraph in `networkx` which is strongly connected:

[ ]:
```
G5a = nx.DiGraph(["ab", "bc", "cd", "da"])
nx.draw(G5a, **opts)
```

And here is one that is weakly connected: there is no path from c to a, for example (since d is a "dead end").

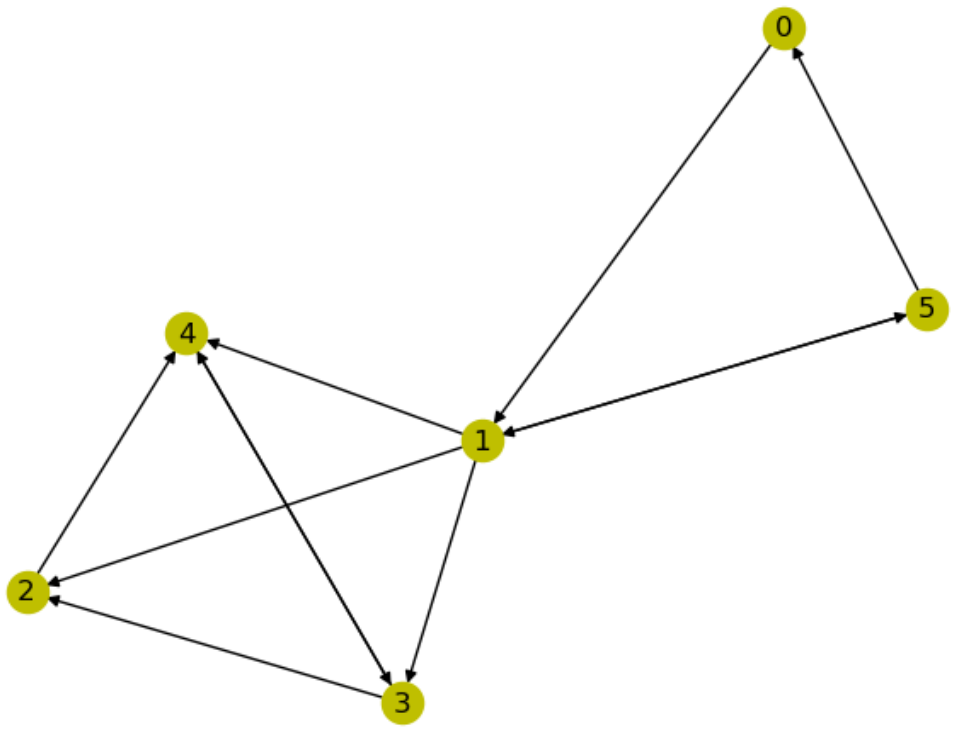
[ ]:
```
G5b = nx.DiGraph(["ab", "bc", "cd", "ad"])
nx.draw(G5b, **opts)
```

1.5.1 Construct and draw a digraph

Let G_5 be the directed graph on the nodes 0, 1, 2, 3, 4 and 5, with edges $0 \to 1$, $1 \to 2$, $1 \to 3$, $1 \to 4$, $1 \to 5$, $2 \to 4$, $3 \to 2$, $3 \to 4$, $4 \to 3$, $5 \to 0$ and $5 \to 1$.

Either by hand, or in `networkx`, draw G_5.

[103]:
```
## A drawing of the digraph
G5 = nx.DiGraph(["01", "12", "13", "14", "15", "24", "32", "34", "43", "50",␣
 ↪"51"])
nx.draw(G5, **opts)
```



1.5.2 G_5 is not strongly connected.

Show that this digraph is *not* strongly connected (i.e., find a pair of nodes between which there is no path).

[98]:
```
# easy (cheeky) way
print(f"Is G5 strongly connected? {nx.is_strongly_connected(G5)}")

# proper way (breadth first search)
from collections import deque

def bfs(start, target, graph):
    visited = set()
    queue = deque([start])

    while queue:
        node = queue.popleft()
        if node == target:
            return True
        if node not in visited:
            visited.add(node)
            queue.extend(graph.neighbors(node))
    return False

for node1 in G5.nodes():
    for node2 in G5.nodes():
        if node1 != node2 and not bfs(node1, node2, G5):
            print(f"There is no path between node {node1} and node {node2}")
```

```
Is G5 strongly connected? False
There is no path between node 2 and node 0
There is no path between node 2 and node 1
There is no path between node 2 and node 5
There is no path between node 3 and node 0
There is no path between node 3 and node 1
There is no path between node 3 and node 5
There is no path between node 4 and node 0
There is no path between node 4 and node 1
There is no path between node 4 and node 5
```

1.5.3 Permuting the adjacency matrix.

Suppose that A is a the adjacency matrix of a digraph. Say there is a permutation matrix, P, such that

$$P^T AP = \begin{pmatrix} X & Y \\ O & Z \end{pmatrix}$$

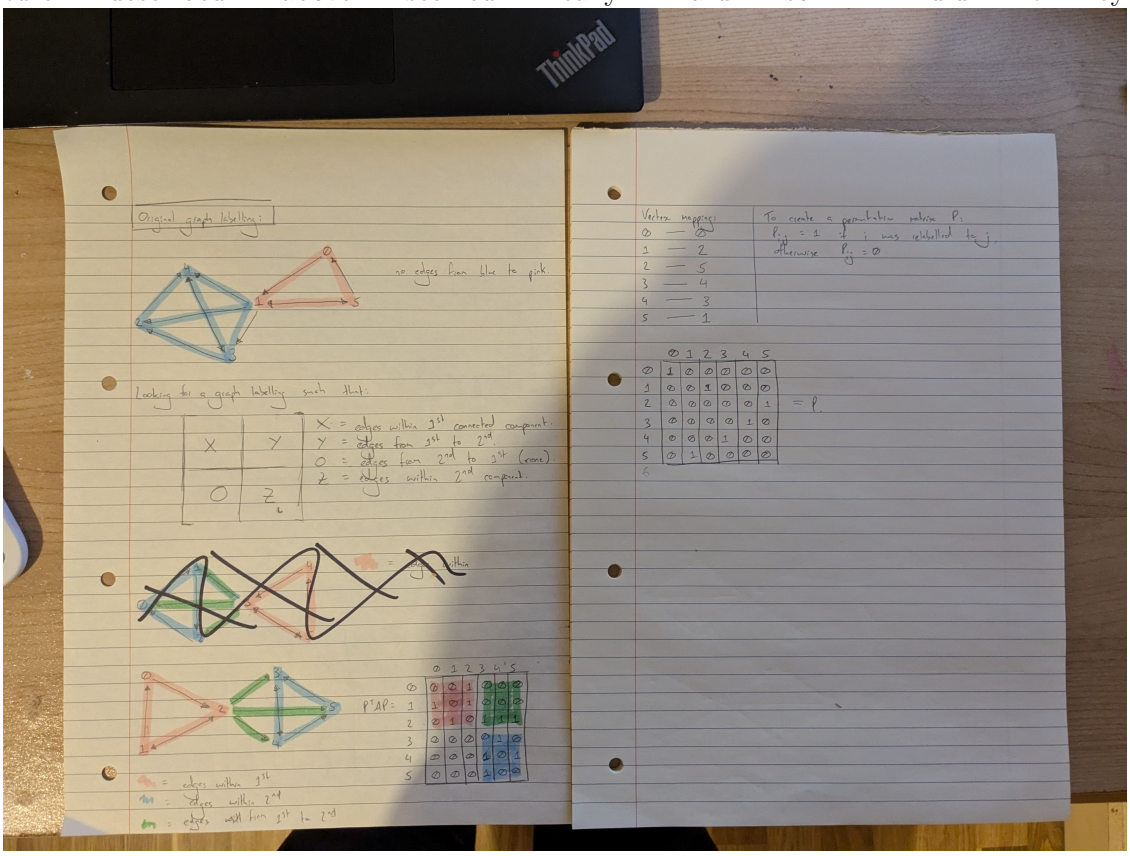
where X and Z are square matrices and O is a zero matrix.

Explain why, if there is such a P, the graph is not strongly connected.

Write down the adjacency matrix for G_5, and also a permutation matrix P such that P^TAP has the structure described above.

Answer: If there exists a permutation P such that $P^TAP = \begin{pmatrix} X & Y \\ O & Z \end{pmatrix}$ where X and Z are square matrices and O is a zero matrix, then we know that the graph is **not strongly connected** because: - The block structure indicates that the vertex set can be partitioned into two non-empty subsets, V_1 and V_2, each corresponding to the blocks X and Z. - The zero block O indicates that there are no edges from any vertex in V_2 any vertex in V_1, implying that at least one of the two subsets is not reachable from the other. - Since every vertex must be reachable from every other vertex for a graph to be **strongly connected**, we know that the graph cannot be strongly connected.

Finding a permutation matrix P such that P^TAP has the structure described above seemed really hard so I did it by hand:



[102]:
```
A = nx.adjacency_matrix(G5).toarray()
print(A)
```

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

[109]:
```
PTAP = np.array([
    [0, 0, 1, 0, 0, 0],
    [1, 0, 1, 0, 0, 0],
    [0, 1, 0, 1, 1, 1],
    [0, 0, 0, 0, 1, 0],
    [0, 0, 0, 1, 0, 1],
    [0, 0, 0, 1, 0, 0]
])

P = np.array([
    [1,0,0,0,0,0],
    [0,0,1,0,0,0],
    [0,0,0,0,0,1],
    [0,0,0,0,1,0],
    [0,0,0,1,0,0],
    [0,1,0,0,0,0]
])

print(P.T@A@P)

np.testing.assert_array_equal(PTAP, P.T@A@P) # check that they're equal
```

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

[ ]: