SYSM 6302 - Lab 2

Jonas Wagner, jrw200000

In [265…

```
from IPython.display import Image
import numpy as np
import matplotlib.pyplot as plt
```

Section 6.9: Degree

Good Example: An example of networks where a node having a high degree is a good thing may be a social network in which edges represent freindships. In such a network, having a high degree indicates you have many friends.

Bad Example: An example of networks where a node having a high degree is a bad thing may be a highway system in which nodes are interchanges of highways. In such a network, having a high degree indicates a lot of highways being connected through that interchange (in other words, a lot of traffic).

Low-Degree Importance Example: An example of a low degree node that may be important would be a single node that solely connects two highly connected (technically not) components. This node may have a low degree, but is vital for allowing paths between two subsets of a graph.

Equation Explination: Equation (6.25) in Newman is the pair of summations for in and out degrees of a directed network.

The in-degree equation is given as:

$$k_i^{in} = \sum_{j=1}^{n} A_{ij}$$

The out-degree equation is given as:

$$k_i^{out} = \sum_{i=1}^{n} A_{ij}$$

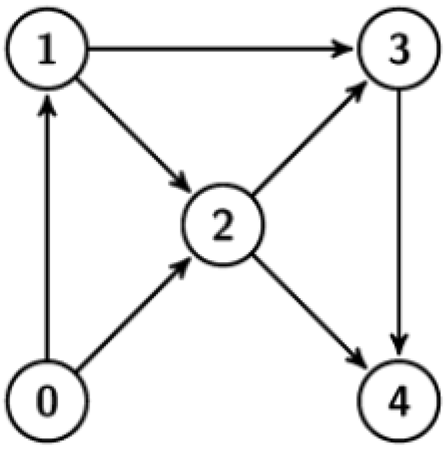
All these equation are describing is the sum of either the collum or row of the adjacency matrices, which can be equivently thought of as the sum of incomming or outgoing edges respectivly.

Section 6.10-6.11: Paths & Components

In [266…

```
Image(filename = 'fig/ExamplePathNetwork.png')
```

Out[266…



The network shown in the previous figure can be described by the folloing adjacency matrix.

In [267…

```
A = np.array([  [0, 0, 0, 0, 0],
                [1, 0, 0, 0, 0],
                [1, 1, 0, 0, 0],
                [0, 1, 1, 0, 0],
                [0, 0, 1, 1, 0]])
print('A = ')
print(A)
```

```
A =
[[0 0 0 0 0]
 [1 0 0 0 0]
 [1 1 0 0 0]
 [0 1 1 0 0]
 [0 0 1 1 0]]
```

A^2 Calculations

In [268…

```
print('A^2 = A A =\n')
print(A)
print('\n*\n')
print(A)
print('\n=\n')
print(A.dot(A))
```

```
A^2 = A A =

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

*

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

=
[[0 0 0 0 0]
 [0 0 0 0 0]
 [1 0 0 0 0]
 [2 1 0 0 0]
 [1 2 1 0 0]]
```

Each nonzero element in A^2 represents the number of paths of length 2 from node j to i.

$A_{0,2}$ = 0 -> 1 -> 2

$A_{0,3}$ = 0 -> 1 -> 3 and 0 -> 2 -> 3

$A_{0,4}$ = 0 -> 2 -> 4

$A_{1,3}$ = 1 -> 2 -> 3

$A_{1,4}$ = 1 -> 2 -> 4 and 1 -> 3 -> 4

$A_{2,4}$ = 2 -> 3 -> 4

x Vector Propogation Calculations

In [269…

```
x = np.array([[1,0,0,0,0]]).T
print(x)
```

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

In [270…

```
print('Ax=\n')
print(A.dot(x))
```

```
Ax=

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

In [271…

```
print('A^2x=\n')
print(A.dot(A.dot(x)))
```

```
A^2x=

[[0]
 [0]
 [1]
 [2]
 [1]]
```

In [272…

```
print('A^3x=\n')
print(A.dot(A.dot(A.dot(x))))
```

```
A^3x=
[[0]
 [0]
 [0]
 [1]
 [3]]
```

Each resulting vector for $A^r x$ represents the potential end points for all paths of length r propogating from the initial state x.

In [273…

```
x = np.array([[0,0,1,0,0]]).T
print(x)
```

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

In [274…

```
print('Ax=\n')
print(A.dot(x))
```

```
Ax=

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

In [275…

```
print('A^2x=\n')
print(A.dot(A.dot(x)))
```

```
A^2x=

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

In [276…

```
print('A^3x=\n')
print(A.dot(A.dot(A.dot(x))))
```

```
A^3x=

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

This observation can be generalized, as each one of these solutions are also indicative of the final locations for paths originating at node 2.

Snub Code

The previous figure can be described by the following adjacency matrix:

In [277…

```
import networkx as nx
```

```python
import sys
sys.path.append('../d3networkx/')
import d3networkx as d3nx
from d3graph import D3Graph, D3DiGraph
from numpy import *
from time import time
import asyncio

def square_grid(n,d3,G,x0=100,y0=100,w=50):
    if G is None:
        G = D3Graph()
    # find the dimensions for the grid that are as close as possible
    num_rows = int(floor(sqrt(n)))
    while n % num_rows != 0:
        num_rows += 1
    num_cols = int(n/num_rows)

    # Add all the nodes
    G.add_nodes_from(range(n))

    # Add the edges and position the nodes
    for i in range(num_rows):
        for j in range(num_cols):
            n = num_cols*i + j
            d3.position_node(n,x0+i*w,y0+j*w)
            if i < num_rows-1:
                G.add_edge(n,n+num_cols) # add edge down
            if j < num_cols-1:
                G.add_edge(n,n+1) # add edge right

async def propagate(G,d3,x,steps,slp=0.5,keep_highlights=False,update_at_end=False):
    interactive = d3.interactive
    d3.set_interactive(False)
    A = nx.adjacency_matrix(G).todense().T  # adjacency matrix
    d3.highlight_nodes_by_index(list(where(x>0)[0]))
    d3.update()
    await asyncio.sleep(slp)
    cum_highlighted = sign(x)
    for i in range(steps): # the brains
        x = sign(dot(A,x)) # the brains
        cum_highlighted = sign(cum_highlighted+x)
        if not update_at_end:
            if not keep_highlights:
                d3.clear_highlights()
            d3.highlight_nodes_by_index(list(where(x>0)[0]))
            d3.update()
            await asyncio.sleep(slp)
    if update_at_end:
        if not keep_highlights:
            d3.clear_highlights()
            d3.highlight_nodes_by_index(list(where(x>0)[0]))
        else:
            d3.highlight_nodes_by_index(list(where(cum_highlighted>0)[0]))
        d3.update()
    d3.set_interactive(interactive)
    if keep_highlights:
        return cum_highlighted
    else:
        return x
```

This next line starts up the visualizer. It will start some background code that sends data to the visualizer and then it will open a new browser window where the visualizer will live. Once you have the visualizer running, you can leave it running for the entire session, so don't re-run this block. If you close the `visualizer.html` (or hit refresh), you will need to reestablish this connection. In this case, you should click the refresh button in the Jupyter notebook (not for the webpage) to restart the kernel (which will clear your variables and Python environment).

In [278…
```
d3 = await d3nx.create_d3nx_visualizer()
#d3 = await d3nx.create_d3nx_visualizer(canvas_size=(1200,1000))
```

```
websocket server started...networkx connected...
```

Grid Network

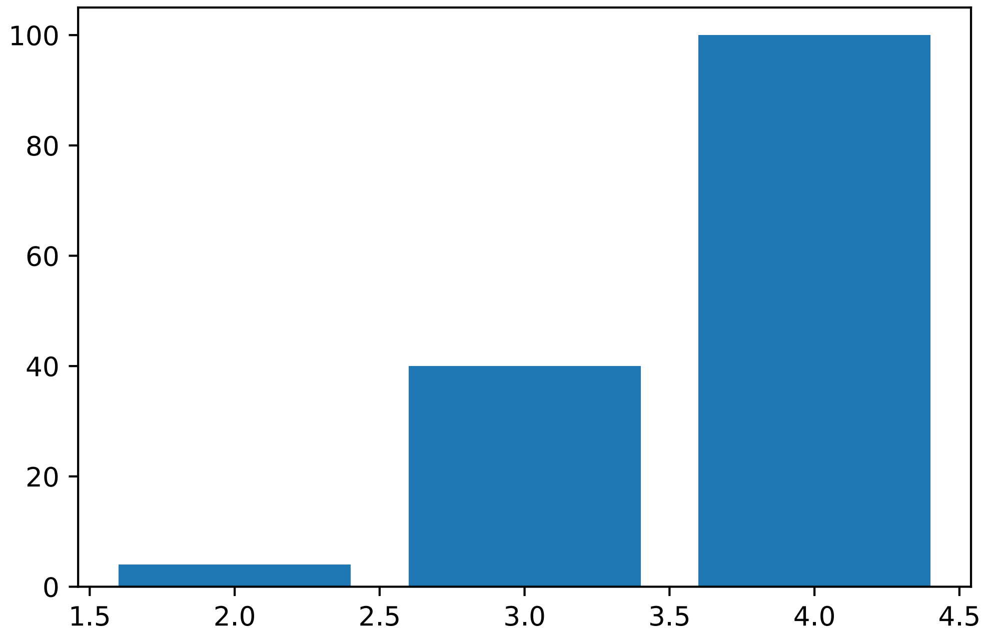
In [279…
```
d3.clear()
d3.set_interactive(False)
G = D3Graph()
d3.set_graph(G)
square_grid(144,d3,G,x0=75,y0=70)
d3.update()
```

Node Degrees

The 12x12 lattice has many nodes with degrees ranging from 2-4. The 4 corner nodes have a degree of 2, the other exterior nodes have a degree of 3, and the interior nodes have degree of 4.

In [280…
```
x = [2,3,4]
y = [4,40,100]
plt.bar(x,y)
```

Out[280…
```
<BarContainer object of 3 artists>
```



Propogate

In [281…
```
x = zeros((G.number_of_nodes(),1))
x[0] = 1
await propagate(G,d3,x,10,slp=1);
```

The propogation is propogating the state of individual nodes to their neighbors for 10 time steps.

It stops becouse 10 time steps have pased, and the result is the set of nodes that can be reaced by paths of length 10.

The propogation can be furthered by adjusting the number of steps it takes:

In [282…
```
steps = 15
await propagate(G,d3,x,steps,slp=1);
```

Directed Network

In [283…
```
d3.clear()
G = D3DiGraph(nx.read_weighted_edgelist('lab2.edgelist',create_using=nx.DiGraph))
d3.set_graph(G)
d3.update()
d3.set_interactive(True)
```

Propogate

In [284…
```
x = zeros((G.number_of_nodes(),1))
x[0] = 1
x_end = await propagate(G,d3,x,10,slp=1)
print('x_end =')
print(x_end)
```

```
x_end =
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]]
```

The end result, x_end, is a vector containing the states of the nodes after 10 time steps. This is also the set of nodes that are reachable by paths of length 10. In this case the result is all zeros, wich is related to the acyclic nature of the graph.

In [285…
```
A = nx.adjacency_matrix(G).todense().T
print('A = ')
print(A)
```

```
A =
[[0 0 0 0 0]
 [1 0 0 0 0]
 [1 1 0 0 0]
 [0 1 1 0 0]
 [0 0 1 1 0]]
```

In [286…
```
A_10 = A**10
print('A^10 = ')
print(A_10)
```

```
A^10 =
[[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]]
```

In [287…
```
x_end_mat = A_10.dot(x)
print('x_end = ')
print(x_end_mat)
```

```
x_end =
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]]
```

From the linear algebra perspective, $A^r x$, is the state of the network after r time steps. This is why the result of $A^{10} x$ is the same as propogating the network 10 times. These propogation matrices (also known as transfer matrices) are able to

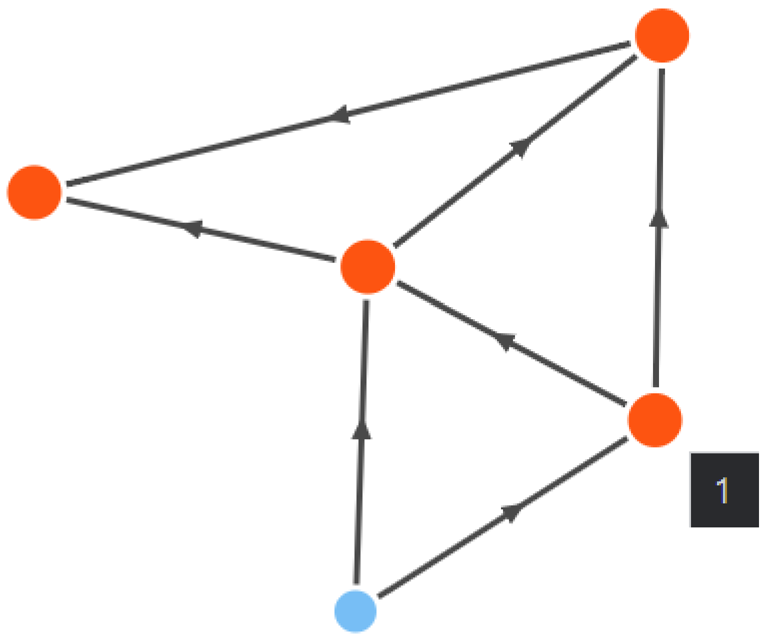
Out Component

In [288…
```
x = zeros((G.number_of_nodes(),1))
x[1] = 1
x_end = await propagate(G,d3,x,20,update_at_end=True,keep_highlights=True)
print(x_end)
```

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

In [289…
```
Image(filename = 'fig/out-component.png')
```

Out[289…



The out-component is as expected.

In-component: a method using the standard propogation function to finding the in-component would be to do an exhaustive search of nodes checking which ones include the target node within their out-component.

Strongly-Connected: The largest strongly-connected component in this network is a single node. This is evident by the acyclic nature of the graph.

E. coli Protein Network

In [290…

```
d3.clear()
G = D3DiGraph(nx.read_weighted_edgelist('ecoli.edgelist',create_using=nx.DiGraph))
d3.set_interactive(False)
d3.set_graph(G)
d3.set_interactive(True)
d3.update()
print('Ecoli has %i nodes.' % G.number_of_nodes())
```

```
Ecoli has 418 nodes.
```

Out-component

Node 2

In [291…

```
x = zeros((G.number_of_nodes(),1))
x[2] = 1
steps = G.number_of_nodes()-1
x_end = await propagate(G,d3,x,steps,update_at_end=True,keep_highlights=True)
print('Out-component of node 2 has %i nodes.' % size(where(x_end>0)[0]))
```

```
Out-component of node 2 has 3 nodes.
```

Node 16

In [292…

```
d3.clear_highlights()
d3.update()
x = zeros((G.number_of_nodes(),1))
x[16] = 1
steps = G.number_of_nodes()-1
x_end = await propagate(G,d3,x,steps,update_at_end=True,keep_highlights=True)
print('Out-component of node 16 has %i nodes.' % size(where(x_end>0)[0]))
```

```
Out-component of node 16 has 22 nodes.
```

It is certain to find all the out components if you propogate at least $n - 1$ times as there are a total of n nodes and a worse case scenario for calculating the out-component would be for the entire graph to be reachable along a single path, which would take $n - 1$ steps to propogate through.

Diameter Calculation

In [293…

```
def diameter2(G):
    spaths = dict(nx.all_pairs_shortest_path(G))

    diameter = 1
    path = []
    for u in G:
        for v in spaths[u].keys():
            if size(spaths[u][v]) > diameter:
                diameter = size(spaths[u][v])
                path = spaths[u][v]
                # print(size(spaths[u][v]), ': ', u,v,spaths[u][v])
    return diameter, path

# use the new diameter function here
```

In [294…

```
diam, path = diameter2(G)
print('Diameter: %i' % diam)
print('Path:', path)
```

```
Diameter: 5
Path: ['190', '292', '136', '137', '133']
```

In [295…

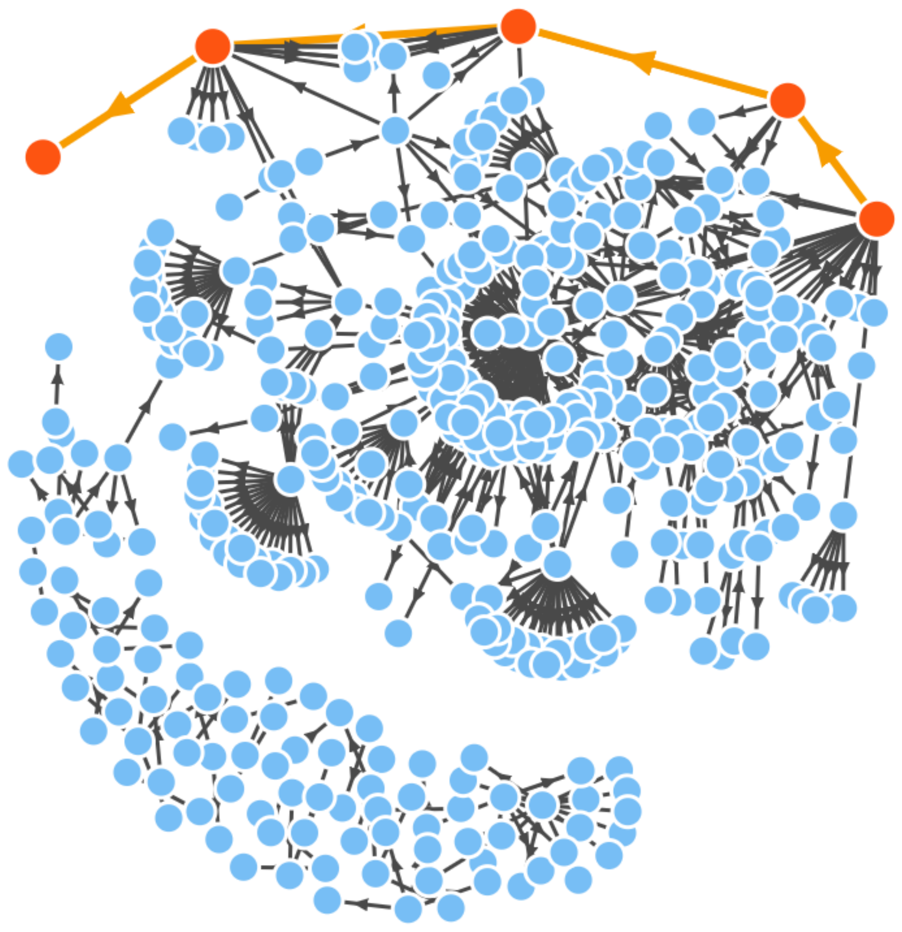
```
d3.clear_highlights()
d3.highlight_nodes(path)
edges = []
for i in range(size(path)-1):
    edges.append((path[i],path[i+1]))
d3.highlight_edges(edges)
d3.update()
```

In [296…

```
Image(filename = 'fig/Diameter.png')
```

Out[296…

Gravity:
Force:
Link length:



Section 6.12: Flows & Cut Sets

Augmenting Path Efficency

Worst-Case: The worst case scenario for the augmented path algorithm is a graph that is extreamly dense becouse it would result in $O(n^3)$.

In [297…

```
def worst_graph(n):
    G = nx.Graph()
    for i in range(n):
        G.add_node(i)
        for j in G.nodes():
            G.add_edge(i,j)
            G[i][j]['capacity'] = 1
    return G
```

Best-Case: The best case is a very sparse network resulting in $O((m+n)m/n)$.

In [298…
```
def best_graph(n):
    G = nx.Graph()
    G.add_node(0)
    for i in range(1,n):
        G.add_node(i)
        G.add_edge(i,i-1)
        G[i][i-1]['capacity'] = 1
    return G
```

Time Testing

In [299…
```
n = 1000
s = 45
t = 990
## Worst Graph
G = worst_graph(n)
start_time = time()
nx.minimum_cut(G,s,t)
print('min cut took %1.2f seconds' % (time() - start_time))
```

```
min cut took 5.46 seconds
```

In [300…
```
## Best Graph
G = best_graph(n)
start_time = time()
nx.minimum_cut(G,s,t)
print('min cut took %1.2f seconds' % (time() - start_time))
```

```
min cut took 0.03 seconds
```

Gantt Chart Network

A grouping of activities within a gantt chart should be an acyclic directed graph becouse thoruought a project it should strieve to progress and eventually complete it. If any cycles were to exist it would mean a looping would continue around on a activies that are ultimently repeated many times.

In [301…
```
G = nx.read_gml('pert.gml','name')

path = nx.bellman_ford_path(G,'Lead time','Leave site')
length = 0
for i in range(size(path)-1):
    length += G[path[i]][path[i+1]]['weight']

print('Critical Path Length:', length)
print('Path:',path)
```

```
Critical Path Length: 48.0
Path: ['Lead time', 'Obtain valves', 'Fit valves', 'Finish valve chambers', 'Leave sit
e']
```

In [302…
```
cut_value, partition = nx.minimum_cut(G,'Lead time','Leave site','weight')
reachable, non_reachable = partition

cutset = set()
```

```
for u, nbrs in ((n,G[n]) for n in reachable):
    cutset.update((u,v) for v in nbrs if v in non_reachable)


print('Cut set weight:', cut_value)
print('Cut set:', sorted(cutset))
```

```
Cut set weight: 11.0
Cut set: [('Clean up', 'Leave site'), ('Finish valve chambers', 'Leave site')]
```

Identifying the cutset within an activity network may be useful for finding the primary bottlenecks of a project. Specifically, the min cutset are the bottlenecks that are scheduled to take the least amount of time and therefore could lead to relativly larger delays if off schedule.

```
In [ ]:
```