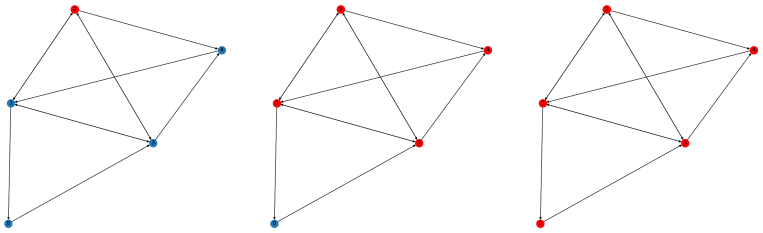
Homework #1

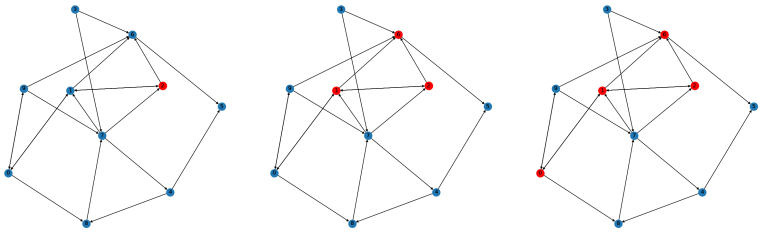
AA 597: Networked Dynamics Systems
Prof. Mehran Mesbahi
Due: Jan 20, 2024 11:59pm
Soowhan Yi

P1. Write a code that implements breadth first search to determine if two nodes in a graph are connected by a walk. The input to the code is a list adjacency of the graph (so a dictionary that has the list of nodes and and their respective neighbors), and the desired pair of nodes. The output of the code should come back with the answer that lists either the sequence of adjacent nodes that allows a walk from one node to the next or a message that these two nodes are not connected. Show how the algorithm works on a few generated graphs on five nodes.

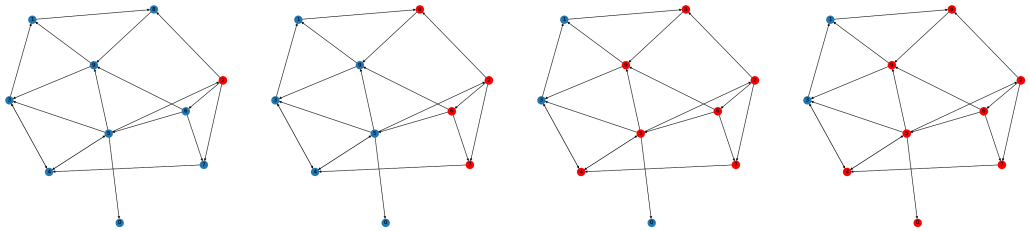
```
"""Breath First Search
    Args:
        G (nx.Digraph): an directed graph with n nodes.
        desired_nodes ([nx.Digraph.node, nx.Digrapoh.node]):
        [0]: source node, [1]: destination node
    Returns:
        Tuple(np.list, np.list):
        [0]: Either the sequence of walk to destination or the trial path with statement for failure.
        [1]: list of nodes for BFS
    """
import util
def bfs_search(G, desired_nodes):
    source = desired_nodes[0]
    destination = desired_nodes[1]
    stateQueue = util.Queue()
    pathQueue= util.Queue()
    visitedSet = []
    visitedSet = set(visitedSet)
    pathList = []
    while source != destination:
        if source not in visitedSet:
            visitedSet.add(source)
            successors = G[source]
            for successor in successors:
                stateQueue.push(successor)
                pathQueue.push(pathList + [[source, successor]])
        if stateQueue.isEmpty():
            print("state queue empty two nodes are not connected.")
            break
        source = stateQueue.pop()
        pathList = pathQueue.pop()
    nodeList = util.np.zeros(len(pathList)+1)
    nodeList[0] = pathList[0][0]
    for i in range(0, len(pathList)):
        nodeList[i+1] = pathList[i][1]
    return pathList, nodeList
```



(a) Source = 2 (b) [2,3] (c) [3,4]



(a) Source =2 (b) frame 1 (c) frame 2



(a) Source =2 (b) frame 1 (c) frame 2 (d) frame 3

Above figures show how the breath first search works. Those graphs are randomly generated and their edges have directions. Utility file, which includes codes for randomly generated graphs and class for Queue, is also included at the end of the document. Starting from node 2, the search agent looks for the neighbors and put those neighbors into a queue. This queue(first in first out) then pops the next node to search and puts its neighbors to the queue again. This way we can traverse through the graph by searching for the closest neighbor to get to the destination. Here are the links to the video of above simulation.
https://github.com/SoowhanYi94/ME597/tree/ebb5167094092be938096bbc1a82e4b2dce9d7b3/HW1

P2. Show the trace $A(G)^3$ (the cube of the adjacency matrix) is six times the number of triangles (cycles with length 3) in the graph G.

Let $A(G) = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{11} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$ and $A(G)_{ij} = a_{ij}$.

$$A(G)^k_{ij} = \text{number of walks of length k between node i and node j}$$

$$\begin{aligned} Trace(A(G)^3) &= \sum_{i=1}^{n} A(G)^3_{ii} \\ &= \sum_{i=1}^{n} \text{number of walks of length 3 between node i and node i} \\ &= \sum_{i=1}^{n} \text{number of triangle from node i} \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} a_{ij}a_{jk}a_{ki} \\ &= \text{total number of triangle} * 6 \qquad \text{if the graph is undirected} \\ &= \text{total number of triangle} * 3 \qquad \text{if the graph is directed} \end{aligned}$$

From above equations, we see that the number of triangles from every node is added up. However, this is triple counting the triangles since there can be 3 triangles when triangle is formed. It counts the [1,2][2,3][3,1] and [2,3][3,1][1,2] differently, even though they are the same triangle. If we were to find number of k-degree triangle(I am sticking with this terminology for simplicity)in the graph, we would have to divide $\sum_{i=1}^{n}$ number of triangle from node i by k, since it would count the same triangle for every node in the triangle. Therefore $Trace(A(G)^3)$ is six times the number of triangles if the graph is undirected and 3 times the number of tirangles if the graph is directed. Below is code for counting triangles in generated random graph.

```
def main():
    n_nodes = 5
    edge_probability = 2/n_nodes
    G = util.random_graph(edge_probability, n_nodes)
    A_G = util.nx.to_numpy_array(G)
    m = range(len(A_G))
    count =0
    for i in m:
        for j in m:
            for k in m:
                if A_G[i][j] and A_G[j][k] and A_G[k][i]:
                    count +=1

    print("number of trianlge: ", count/3)
    util.nx.draw(G, font_weight='bold', with_labels=True)
    util.plt.show()
    util.plt.savefig('graph.png')

if __name__=="__main__":
    main()
```

P3. (MEBook2010 Exercise 2.11). Show that any graph on n vertices that has more than n - 1 edges contains a cycle.
Unless a vertex is not connected with the graph, it needs one edge to stay connected to the graph. Lets say we use one vertex as a root node. Then it needs n-1 edges to connect n-1 nodes, no matter how the graph looks(except for the disconnection). Therefore we need at least n-1 edges to connect n vertices, and extra edges

would result in choosing same node pairs as there is no other option. Therefore any graph on n vertices that has more than n - 1 edges contains a cycle. Also a cycle graph is 2 regular, meaning that nodes in cycle graph each has 2 edges. IF we have more than n-1 edges, a pair of the nodes need to have at least 2 edges, therefore forming a cycle. Another explaination is that if we think about the adjacency matrix, each component in a row or column, we need at least one compnent to be 1 in order to make the corresponding vertex connected to the graph. If we were to use n number of edges on n vertices, there would be at least one component in diagonal of adjacency matrix that has value of 1, meaning that it has a cycle.

P4. (MEBook2010 Exercise 2.2) The degree sequence for a graph is a listing of the degrees of its nodes; thus K3 has the degree sequence 2, 2, 2. Is there a graph with the degree sequence 3, 3, 3, 3, 5, 6, 6, 6, 6, 6, 6? How about with the degree sequence 1, 1, 3, 3, 3, 3, 5, 6, 8, 9?
Using the Havel-Hakimi theorum, the graphs for first and second degree sequences ([3, 3, 3, 3, 5, 6, 6, 6, 6, 6, 6] and [1, 1, 3, 3, 3, 3, 5, 6, 8, 9]) do not always exist. But it exists for [6 ,3, 3, 3, 3, 2, 2, 2, 2, 1, 1].

```
def Havel_Hakimi(sequence=list):
    if len(sequence) ==0:
        print("length of degree sequence is 0")
    sequence.sort(reverse=True)
    flag = False
    print(sequence)
    while (len(sequence) != 0):
        num = sequence.pop(0)
        print(sequence)
        if len(sequence) == 0:
            flag = True
            print("The graph does not always exist with this degree sequence.")
        for i in range(num):
            sequence[i] -=1
            if sequence[i] < 0:
                flag = True
                print("The graph does not always exist with this degree sequence.")
                break
        print(sequence)
        if flag: break
        sequence.sort(reverse=True)
        if sum(sequence) == 0:
            print("There exists a graph with the given degree sequence.")
            break
```

$[3,3,3,3,5,6,6,6,6,6,6]$

$[6,6,6,6,6,6,5,3,3,3,3] \rightarrow [6,6,6,6,6,5,3,3,3,3] \rightarrow [5,5,5,5,5,4,3,3,3,3]$

$[5,5,5,5,5,4,3,3,3,3] \rightarrow [5,5,5,5,4,3,3,3,3] \rightarrow [4,4,4,4,3,3,3,3,3]$

$[4,4,4,4,3,3,3,3,3] \rightarrow [4,4,4,3,3,3,3,3] \rightarrow [3,3,3,2,3,3,3,3]$

$[3,3,3,3,3,3,3,2] \rightarrow [3,3,3,3,3,3,2] \rightarrow [2,2,2,3,3,3,2]$

$[3,3,3,2,2,2,2] \rightarrow [3,3,2,2,2,2] \rightarrow [2,2,1,2,2,2]$

$[2,2,2,2,2,1] \rightarrow [2,2,2,2,1] \rightarrow [1,1,2,2,1]$

$[2,2,1,1,1] \rightarrow [2,1,1,1] \rightarrow [1,0,1,1]$

$[1,1,1,0] \rightarrow [1,1,0] \rightarrow [0,1,0]$

$[1,0,0] \rightarrow [0,0] \rightarrow [-1,0]$

$---$

$[1,1,3,3,3,3,5,6,8,9]$

$[9,8,6,5,3,3,3,3,1,1] \rightarrow [8,6,5,3,3,3,3,1,1] \rightarrow [7,5,4,2,2,2,2,0,0]$

$[7,5,4,2,2,2,2,0,0] \rightarrow [5,4,2,2,2,2,0,0] \rightarrow [4,3,1,1,1,1,-1,0]$

$---$

$[6,3,3,3,3,2,2,2,2,1,1]$

$[6,3,3,3,3,2,2,2,2,1,1] \rightarrow [3,3,3,3,2,2,2,2,1,1] \rightarrow [2,2,2,2,1,1,2,2,1,1]$

$[2,2,2,2,2,2,1,1,1,1] \rightarrow [2,2,2,2,2,1,1,1,1] \rightarrow [1,1,2,2,2,1,1,1,1]$

$[2,2,2,1,1,1,1,1,1] \rightarrow [2,2,1,1,1,1,1,1] \rightarrow [1,1,1,1,1,1,1,1]$

$[1,1,1,1,1,1,1,1] \rightarrow [1,1,1,1,1,1,1] \rightarrow [0,1,1,1,1,1,1]$

$[1,1,1,1,1,1,0] \rightarrow [1,1,1,1,1,0] \rightarrow [0,1,1,1,1,0]$

$[1,1,1,1,0,0] \rightarrow [1,1,1,0,0] \rightarrow [0,1,1,0,0]$

$[1,1,0,0,0] \rightarrow [1,0,0,0] \rightarrow [0,0,0,0]$

$[0,0,0,0]$

Therefore first two degree sequences are not graphical but the third one is graphical.

P5. (MEBook2010 Exercise 2.12) Show that the graph and its complement cannot both be disconnected. The complement of the graph G is a graph that has the edges that are not in G but not the edges that are in G.

Lets say there is a set of connected vertices V, with a set of edges E in graph S, and a graph G is a subset of S. Lets assume that the graph G and its complement are both disconnected. Then at least one vertex from each of them (G and its complement) are disconnected from them respectively. This means that those two vertices are disconnected in graph S, which should be connected by definition. There is a contradiction in the statement and therefore the graph and its complement cannot both be disconnected.

P6. (MEBook2010 Exercise 2.6) Show that any graph on n vertices with more than (n-1)(n-2)/2 edges is connected.

Lets say that we have a complete graph with n-1 vertices. Each vertex would have n-2 degree. Then the total number of edge would be $\frac{(n-1)(n-2)}{2}$, since each vertex would have n-2 edges and we have n-1 vertices in K_{n-1} graph. Also we are dividing by 2 because we are double counting the same edge. Finally we need $\frac{(n-1)(n-2)}{2}$ for K_{n-1} graph. If we were to have one more edge, then we would need to connect one more vertex and those n vertices would never become disconnected.

P7. Write a code that implements the consensus protocol $\dot{x} = -L(G)x$, where L(G) is the Laplacian matrix of the graph, from arbitrary (random) initial conditions. Use networkX or something similar to run the consensus dynamics on a cycle, path, star, and complete graphs. Find the second smallest eigenvalues of these graphs and see if the convergence properties of consensus can be matched to these second smallest eigenvalues. Then explore the convergence as a function the number nodes in these graphs- again using networkX or something similar, choose graphs of sizes 5, 10, 20, and 50 for your computational experiments.

```
import random
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from matplotlib import animation
import numpy as np
from networkx.drawing.nx_agraph import graphviz_layout
## util.py
class Queue:
"A container with a first-in-first-out (FIFO) queuing policy."
    def __init__(self):
        self.list = []

    def push(self,item):
        "Enqueue the 'item' into the queue"
        self.list.insert(0,item)

    def pop(self):
        """
            Dequeue the earliest enqueued item still in the queue. This
            operation removes the item from the queue.
        """
        return self.list.pop()

    def isEmpty(self):
        "Returns true if the queue is empty"
        return len(self.list) == 0


def random_graph(prob =0.3, n_nodes = 10):

    G = nx.DiGraph()

    G.add_nodes_from(range(n_nodes))

    for u in G.nodes:
        for v in G.nodes:
            if random.random() < prob and u!=v :
                G.add_edge(u, v)
    return G
```