CSCE 5013 - Graph Theory

010646947

Zack Fravel

1 Dynamic Processes

Exercise 1

The claim that the number of balanced triads can never decrease is False. A very simple counterexample can illustrate why; consider a graph with the only initial condition being it is an unbalanced triad with at least one positive edge joining two of the nodes. If we wish to add a new node to this triad graph where both edges are positive, thus only introducing new balanced triads, then if we run this algorithm we will inadvertently unbalance one of our new triads in as a means to the end of balancing the original triad. This makes this algorithm insufficient for balancing signed networks.

2 Experiment

Exercise 2

The following are the results from the dynamic process program I wrote:

```
100 out of 100 graphs ended up balanced

Average number of iterations was  1084.47

Process finished with exit code 0
```

It can be observed that the dynamic process managed to balance all 100 out of 100 graphs with no problem, taking on average 1084 iterations. The python code I wrote using NetworkX is given at the end of the report. To summarize the functionality, the program first initializes a random complete graph with 10 nodes. After that, I use a decision function that uses python's random library to randomly assign (+) or (-) to each edge. After that, I use a multitude of NetworkX functions to find all triangles and cliques, and then isolate to only triads. I then run the dynamic processes algorithm on the list of triads and use that list to verify the status of the graph. Once the graph is balanced, I note the number of iterations and then repeat the process 100 times. At the end, I summarize the fraction of balanced graphs and the average number of iterations .

3 Evolving Graphs

Exercise 3

It is not possible to add a node D that forms signed edges with all nodes but doesn't also form unbalanced triangles in the configuration given by the figure. For instance, if we add a node D and begin with the joining with nodes B and C, we know one edge must be positive (+) and one must be negative (-) since the only balance configuration involving a (-) are the configurations [(-, -, +), (-, +, -), and (+, -, -)]. Since AB and AC are both

(+), the next step is symmetrical. Say we make edge CD (-). This gives us a balanced triangle BCD (-, -, +). With edge CD (-) and BD (+), to balance triangle ABD we must make edge AD (+) to form a balanced (+, +, +) configuration. However, this unavoidably creates an unbalanced triangle ACD with the configuration (+, +, -). Therefore, it is not possible.

Exercise 4

It is not possible to add a node X to an unbalanced complete signed graph and have X not be a part of an unbalanced triangle. If we extend the logic from exercise 3, but starting from the (-, -, -) configuration for triangle ABC, we can prove for all complete unbalanced graphs that it is not possible since that is the only other simple unbalanced configuration. If we attempt to add a node D to this new (-, -, -) ABC triangle and attempt to place it in only balanced triangles, we will have trouble. Again, symmetry makes this next step work for both possibilities. If we make edge CD (+) and edge BD (-), we create a balanced BCD triangle with configuration (-, +, -). However, in order to balance both triangles ABD and ACD, we have a contradiction where the AD edge would need to be (+) and (-) at the same time. Therefore, any node X added to any complete unbalanced signed graph will inevitably be a part of an unbalanced triangle, it is unavoidable.

4 Structural Balance of Roll Call Votes

Exercise 5

Converting my previous program to create signed graphs was luckily fairly trivial since most of the work had been done previously. I added a function that analyzes the weight of all the edges in the graph and sets the sign based on a given "n" value between 6 and 10.

I was able to generate signed graphs using a few new functions I wrote, as well as detecting cycles and balance within the cycles. I was however unfortunately unable to yield any more meaningful results than that because of time constraints. I have included the code I did write with the report.

5 Experiment Code

```
# --------------------------------------
# Zack Fravel
# 010646947
# CSCE 5014 Applications of graph Theory
# --------------------------------------
# experiment.py
#
# Simulation of randomized signed
# complete networks
# --------------------------------------

import random
import networkx as nx


# Function for generating a random boolean value
def decision(probability):

    return random.random() < probability


# Assigns random signs to all edges in graph using decision function
# True  = (+)
# False = (-)
```

```
def assign_signs():

    for u, v in g.edges():
        g[u][v]['sign'] = decision(.5)

    pass


# Chooses two random nodes in a triad and flips the sign of the edge joining them
def flip_random_sign(t):

    # Pick two random nodes
    random_a = random.choice(t)
    random_b = random.choice(t)

    # Avoid choosing the same random node
    while random_b == random_a:
        random_b = random.choice(t)

    # Flip the sign of the edge
    g[random_a][random_b]['sign'] = not g[random_a][random_b]['sign']

    pass


# Function to check if a triad is balanced
def balance_check(triad):

    a = triad[0]
    b = triad[1]
    c = triad[2]

    # (+, +, +)
    if (g[a][b]['sign'] is True) and (g[a][c]['sign'] is True) and (g[b][c]['sign'] is True):
        return True
    # (+, -, -)
    elif (g[a][b]['sign'] is True) and (g[a][c]['sign'] is False) and (g[b][c]['sign'] is False):
        return True
    # (-, +, -)
    elif (g[a][b]['sign'] is False) and (g[a][c]['sign'] is True) and (g[b][c]['sign'] is False):
        return True
    # (-, -, +)
    elif (g[a][b]['sign'] is False) and (g[a][c]['sign'] is False) and (g[b][c]['sign'] is False):
        return True
    else:
        return False


# Function to check if a graph is balanced using list of triads
def balanced_graph_check(triads):

    if (all(balance_check(triad) is True for triad in triads)) is True:
        # print("Balanced Graph")
        return True
    else:
        # print("Graph is not balanced")
        return False
```

```
# Dynamic processes algorithm described in homework
# 1. Pick triad at random
# 2. If it's balanced, do nothing
# 3. Otherwise, choose one of the edges uniformly at random and flip its sign (balancing the triad)
def triad_analysis(triads):

    t = random.choice(triads)

    # If balanced
    if balance_check(t) is True:
        # Do nothing
        pass
    # If unbalanced, choose an edge at random and flip sign
    else:
        flip_random_sign(t)
        pass

    pass


process_count = 0                                   # Process count
process_limit = 100                                 # Process limit
iteration_count = 0                                 # Iteration count
iteration_limit = 1000000                           # Iteration limit
balanced_graphs = 0                                 # Balanced graphs count
iteration_average = 0                               # Iteration average

while process_count < process_limit:

    g = nx.complete_graph(10)  # Initialize complete graph with 10 nodes

    assign_signs()  # Create random signed graph

    tri = nx.triangles(g)  # Find triangles

    cliques = nx.enumerate_all_cliques(g)  # Find cliques (all nodes degree n-1)

    triad_cliques = [l for l in cliques if len(l) == 3]  # Stores edge list of all triads

    # Run the dynamic process for a maximum of 1,000,000 iterations
    while iteration_count < iteration_limit:

        # Check if graph is balanced
        if balanced_graph_check(triad_cliques) is True:
            # print("Balanced graph found!")
            # print("Iterations: ", iteration_count)
            iteration_average += iteration_count
            iteration_count = 0
            balanced_graphs += 1
            break
        else:
            triad_analysis(triad_cliques)

        iteration_count += 1
```

```
    process_count += 1

print('\n', balanced_graphs, "out of", 100, "graphs ended up balanced")
print('\n', "Average number of iterations was ", iteration_average/100)
```

6 Roll Call Votes Code

```
# -------------------------------------
# Zack Fravel
# 010646947
# CSCE 5014 Applications of graph Theory
# -------------------------------------
# HW3.py
#
# Analysis of  U.S. Senate roll call
# voting data using NetworkX
# -------------------------------------

# ##################################################################### #
# IMPORTANT: HW3.py must be in the same directory as roll_call_votes  #
# ##################################################################### #

# Program takes a directory of US roll call votes specified by folder number (101 - 115-2).
# The program parses the xml documents and generates a graph where each node
# represents a US senator and each edge corresponds to Senators voting the same way.
# The weight of an edge is the number of times the Senators voted the same way.
# The thickness of the edge on the produced plot is also an indication of cooperation.

import os
import networkx as nx
import matplotlib.pyplot as plt
import xml.etree.ElementTree as et


# Returns the path to all the .xml documents within one folder
def get_xml_path(filepath):

    xml_path_list = []

    for filename in os.listdir(filepath):

        if filename.endswith(".xml"):
            xml_path_list.append(os.path.join(filepath, filename))

    return xml_path_list


# Create signed graph from given graph of senators
def create_signed(g, n):
    for u, v in g.edges():
        # Voted the same
        if g[u][v]['weight'] >= n:
            # Positive Edge
            g[u][v]['sign'] = True
        # Voted differently greater than n times
        if (10 - g[u][v]['weight']) > n:
```

```
            # Negative Edge
            g[u][v]['sign'] = False

    pass


# Function for detecting balance
def detect_balance(g, a, b, c):
    # (+, +, +)
    if (g[a][b]['sign'] is True) and (g[a][c]['sign'] is True) and (g[b][c]['sign'] is True):
        return True
    # (+, -, -)
    elif (g[a][b]['sign'] is True) and (g[a][c]['sign'] is False) and (g[b][c]['sign'] is False):
        return True
    # (-, +, -)
    elif (g[a][b]['sign'] is False) and (g[a][c]['sign'] is True) and (g[b][c]['sign'] is False):
        return True
    # (-, -, +)
    elif (g[a][b]['sign'] is False) and (g[a][c]['sign'] is False) and (g[b][c]['sign'] is False):
        return True
    else:
        return False


# Function used to detect cycles and perform analysis
def detect_cycles(g):

    cycles = nx.cycle_basis(g)

    balanced = 0
    not_balanced = 0

    for cyc in cycles:

        a = cyc[0]
        b = cyc[1]
        c = cyc[2]

        if detect_balance(g, a, b, c) is True:
            # print("Graph is balanced")
            balanced += 1
        else:
            # print("Graph is not balanced")
            not_balanced += 1

    pass


# Add edges for a given senator on a given vote
def add_edges(g, member):

    global edge_width

    # Check all nodes
    for nodes in g.nodes():

        # Compare votes with current node
```

```
        if member[5].text == g.node[nodes]['vote']:

            # Edge already exists
            if g.has_edge(member[0].text, nodes):

                # Modify edge attributes (increase thickness and weight)
                g[member[0].text][nodes]['thickness'] += 0.01
                g[member[0].text][nodes]['weight'] += 1

                # If from opposing parties
                if g.node[member[0].text]['party'] != g.node[nodes]['party']:
                    # +1 to cooperation_level on each node
                    g.node[member[0].text]['cooperation_level'] += 1
                    g.node[nodes]['cooperation_level'] += 1

            # Edge doesn't exist
            else:

                # Add Edge
                g.add_edge(member[0].text, nodes, thickness=0.01, weight=1, opposed=False, sign=None)

                # Indicate opposition if Senators are in different parties
                if g.node[member[0].text]['party'] != g.node[nodes]['party']:
                    g[member[0].text][nodes]['opposed'] = True
                    # Add +1 to cooperation_level on each node
                    g.node[member[0].text]['cooperation_level'] += 1
                    g.node[nodes]['cooperation_level'] += 1

        # Remove all edges between senators and themselves
        if member[0].text == nodes:
            if g.has_edge(member[0].text, nodes):
                g.remove_edge(member[0].text, nodes)

    # Assign width attribute for each edge to the edge_width list
    for u, v in g.edges():
        edge_width = g[u][v]['thickness']

    pass


# Create a graph given a folder of roll call votes (XML documents)
def initialize_graph(g, folder):

    ry = 0  # Y-location variables for R nodes on plot
    dy = 0  # Y-location variables for D nodes on plot
    modify_flag = False

    # Loop through each roll call vote in each folder
    for file in folder:

        # Set XML file in folder to be parsed
        tree = et.parse(file)
        root = tree.getroot()
        print('\n', file, '\n')

        # Check for root structure
        if root[5].tag == 'modify_date':
```

```
    member_level = 17
    modify_flag = True
else:
    member_level = 16

# Loop through each senator in roll call vote
for members in root[member_level]:

    # If node doesn't exist
    if members[0].text not in senators:

        # Add senator to list
        senators.append(members[0].text)
        # Add node
        g.add_node(members[0].text, position=(0, 0), experience=1, party='n/a', vote='n/a', cooper

        # Check for Republican or Democrat
        if members[3].text == 'R':

            # Set node position
            g.node[members[0].text]['position'] = (0, ry)
            # Set node party affiliation
            g.node[members[0].text]['party'] = 'R'
            # Set node color
            color_map.append('red')
            # Increment position variable
            ry += 0.3

        else:

            # Set node position
            g.node[members[0].text]['position'] = (1, dy)
            # Set node party affiliation
            g.node[members[0].text]['party'] = 'D'
            # Set node color
            color_map.append('blue')
            # Increment position variable
            dy += 0.3

    # If node already exists
    else:

        # Add experience point
        g.node[members[0].text]['experience'] += 1

    # Record vote and store to node
    if members[5].text == 'Yea':
        g.node[members[0].text]['vote'] = 'Yea'
    elif members[5].text == 'Nay':
        g.node[members[0].text]['vote'] = 'Nay'
    else:
        pass

    # Add edges to node
    add_edges(g, members)

# Reset vote on each node so as to avoid incorrect edge creation
```

```
        for node in g.nodes():
            g.node[node]['vote'] = 'n/a'

    if modify_flag is True:
        print("This analysis contains modified votes")

    pass


# Create lists and graph
g = nx.Graph()      # Create graph
color_map = []      # Colors List for party affiliation
edge_width = []     # Use to record weight of each edge
senators = []       # List of senators

# Ask for user input
requested_folder = input("Enter folder number (101, 102, ... 115, 115-2): ")
folder_string = "roll_call_votes/"+requested_folder

# Initialize graph from the folder specified by the user
initialize_graph(g, get_xml_path(folder_string))

# Modify graph to be signed
# Input 'n' value here (6, 7, 8, 9, 10)
create_signed(g, 6)

# Detect cycles in graph and check edges
detect_cycles(g)
```