HW5 - Graph Partitioning

Logan Bartels

November 8, 2020

Note: This assignment was completed in Google Collab. All code shown in this report is copy-pasted from the notebook used to complete this assignment. All relevant code is under the section "HW5." Here is the link to the notebook: `https://colab.research.google.com/drive/1IyTntJkuAC_1TGNQC7cEzKzxOPk4_v4o?usp=sharing`

Step 1

Answer

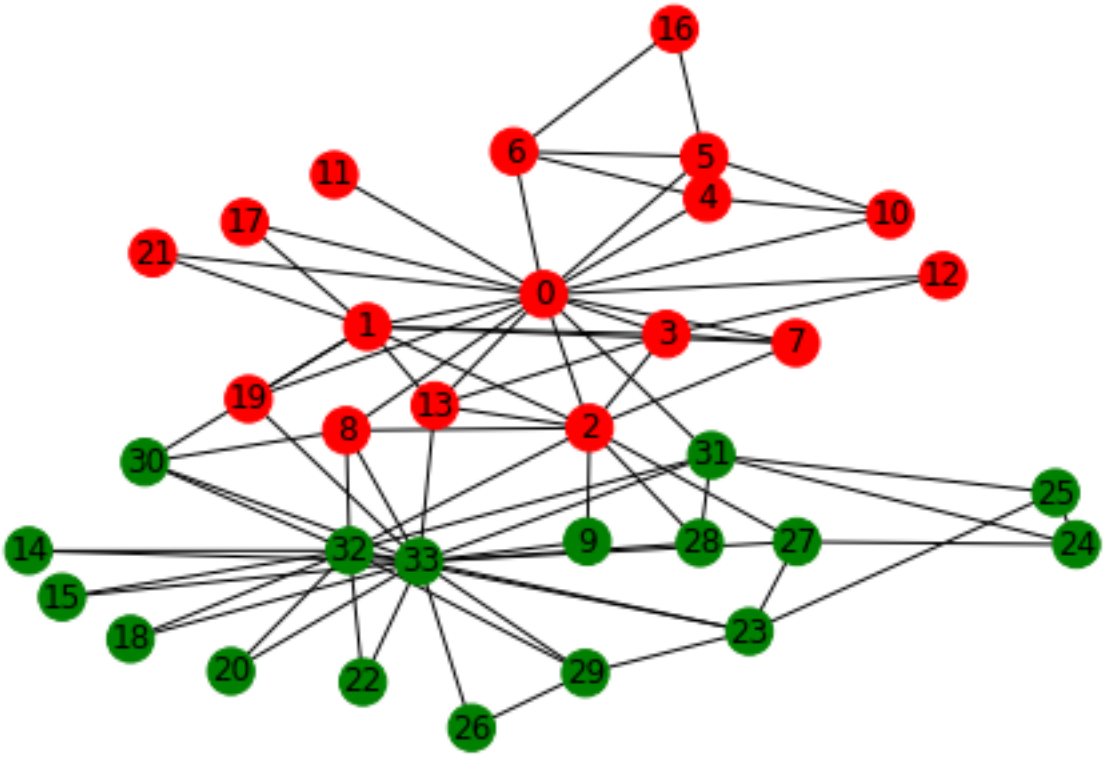

Figure 1: Force-directed original Karate Club graph.

```
import matplotlib.pyplot as plt
import networkx as nx

G = nx.karate_club_graph()
#G.nodes[33]["club"]

# Step 1
```

```
8  # Make all nodes the same color
9  color = ["#1f78b4"] * 34
10 # Loop to color nodes according to their faction
11 for node in G.nodes:
12   # If node belongs to Mr. Hi, color red
13   if G.nodes[node]["club"] == 'Mr. Hi':
14     color[node] = 'red'
15   # Node belongs to John A. color green
16   else:
17     color[node] = 'green'
18 nx.draw_kamada_kawai(G, with_labels=True, node_color=color)
19 # plt.show()
```

Listing 1: Python code used to complete Step 1.

Discussion

The first thing in the code to note is line 5. Originally, that line printed out the faction that the node belonged to. I tested it on node 0 (Mr. Hi) and node 33 (John A.). The results were "Mr. Hi" and "Officer," respectively. On line 9, I made every node in the graph the same color blue. This was a precaution to catch any nodes that would not be colored by my coloring method. Luckily, all nodes were colored. The "for" loop starting on line 11 checks what "club" or faction the node belongs to. If the node belongs to Mr. Hi, then it is colored red. If the node does not belong to Mr. Hi, then it belongs to John A. or, in this case, Officer. The node is then colored green. The graph is force-directed, and the two factions are easily distinguishable. The graph for this step is displayed in figure 1.

Step 2

Algorithm

```
# Full algorithm
# Calculate edge betweenness of all edges
edges = nx.edge_betweenness_centrality(G)
# Sort the returned dictionary by its values in descending order
sortedEdges = sorted(edges, key=edges.__getitem__, reverse=True)
# Remove the edge with the highest betweenness
G.remove_edge(sortedEdges[0][0], sortedEdges[0][1])
# Add the removed edge to a list
removedEdges.append(sortedEdges[0])
nx.draw_kamada_kawai(G, with_labels=True, node_color=color)
# plt.show()
# Check if the graph is connected
print(nx.is_connected(G))
# Show the number of connected components
print(nx.number_connected_components(G))
```

Listing 2: Algorithm used to remove edges from the graph.

Graphs

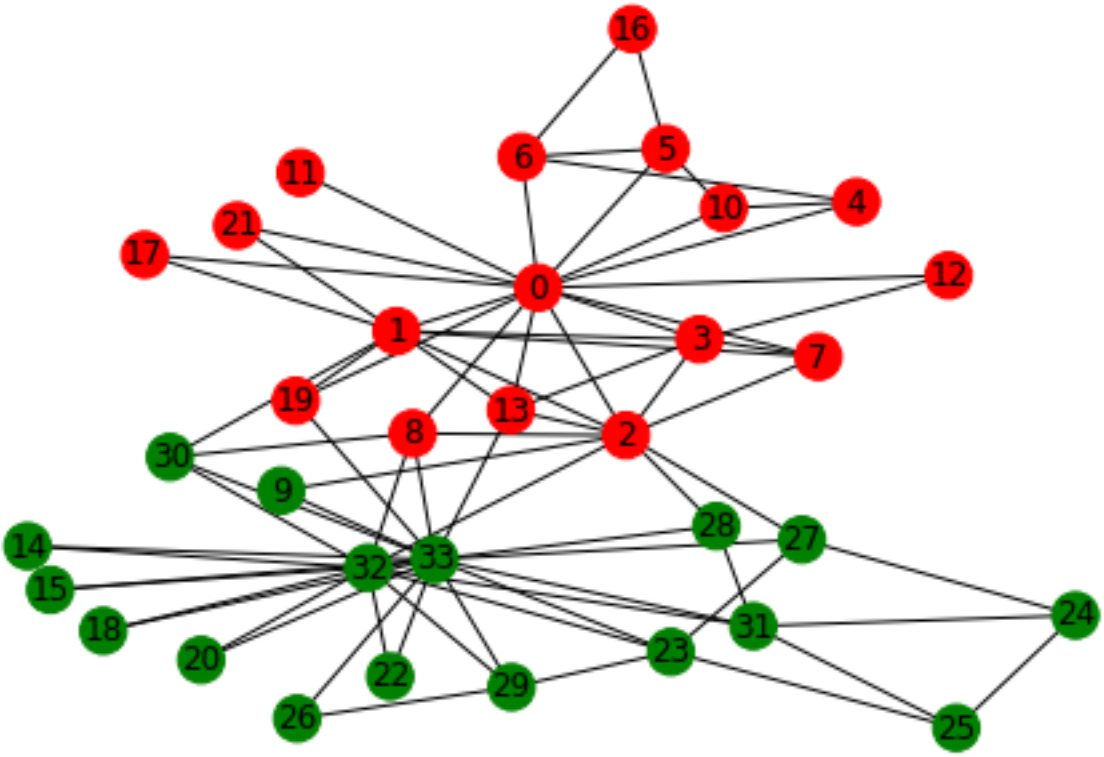

Figure 2: The graph after the first iteration of my Girvan-Newman-like algorithm.

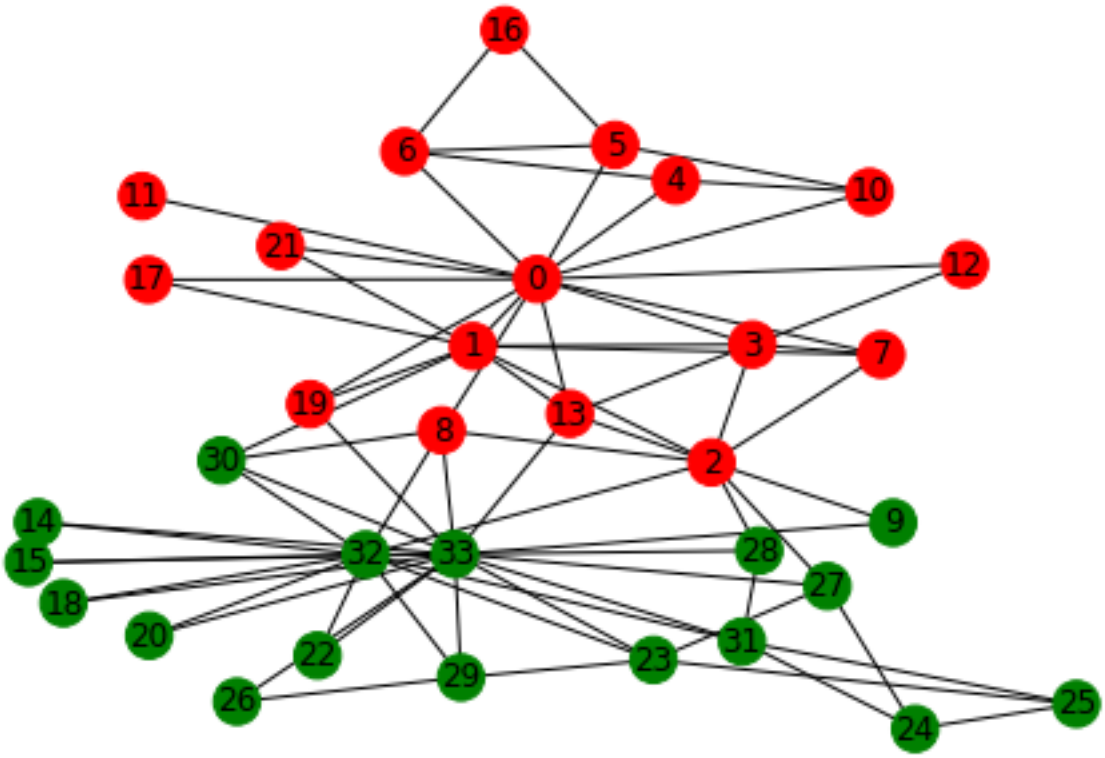

Figure 3: The graph after the second iteration of my Girvan-Newman-like algorithm.

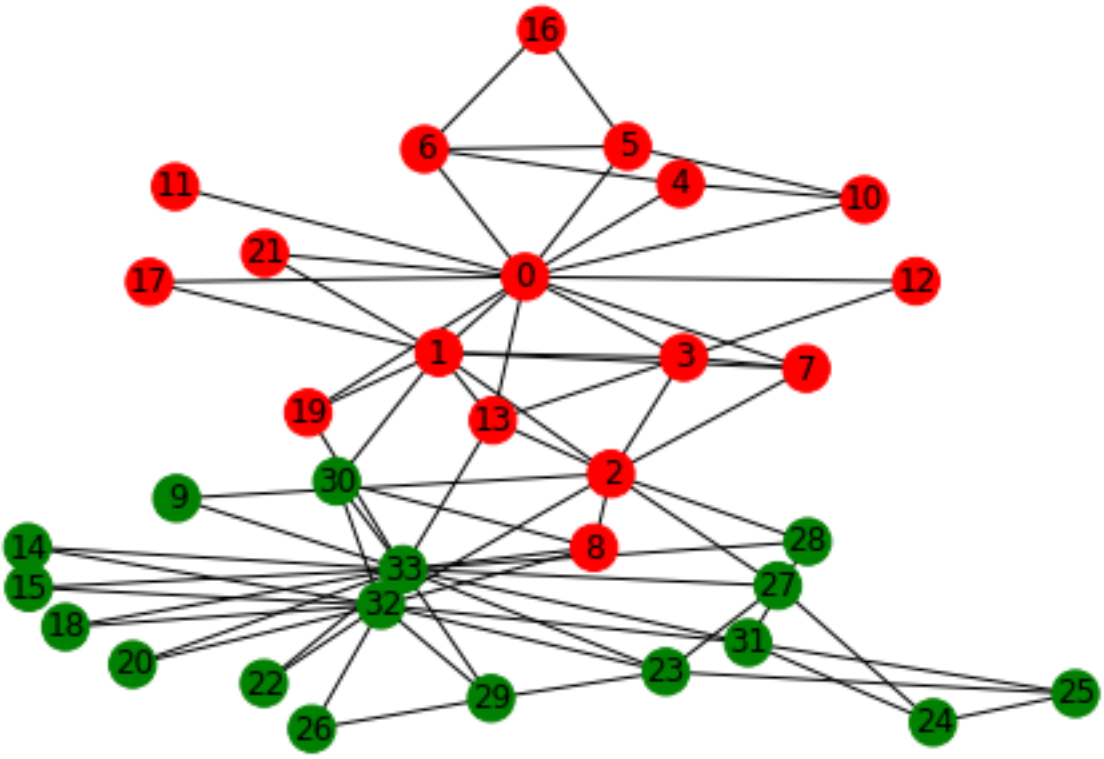

Figure 4: The graph after the third iteration of my Girvan-Newman-like algorithm.

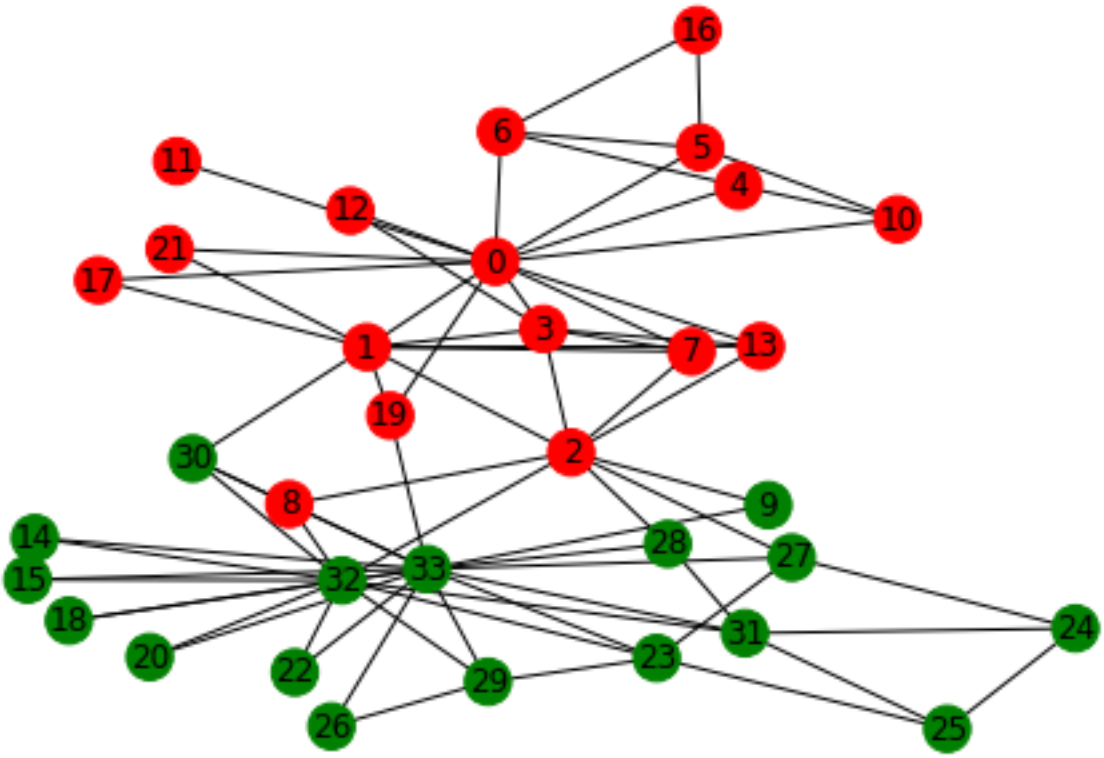

Figure 5: The graph after the fourth iteration of my Girvan-Newman-like algorithm.

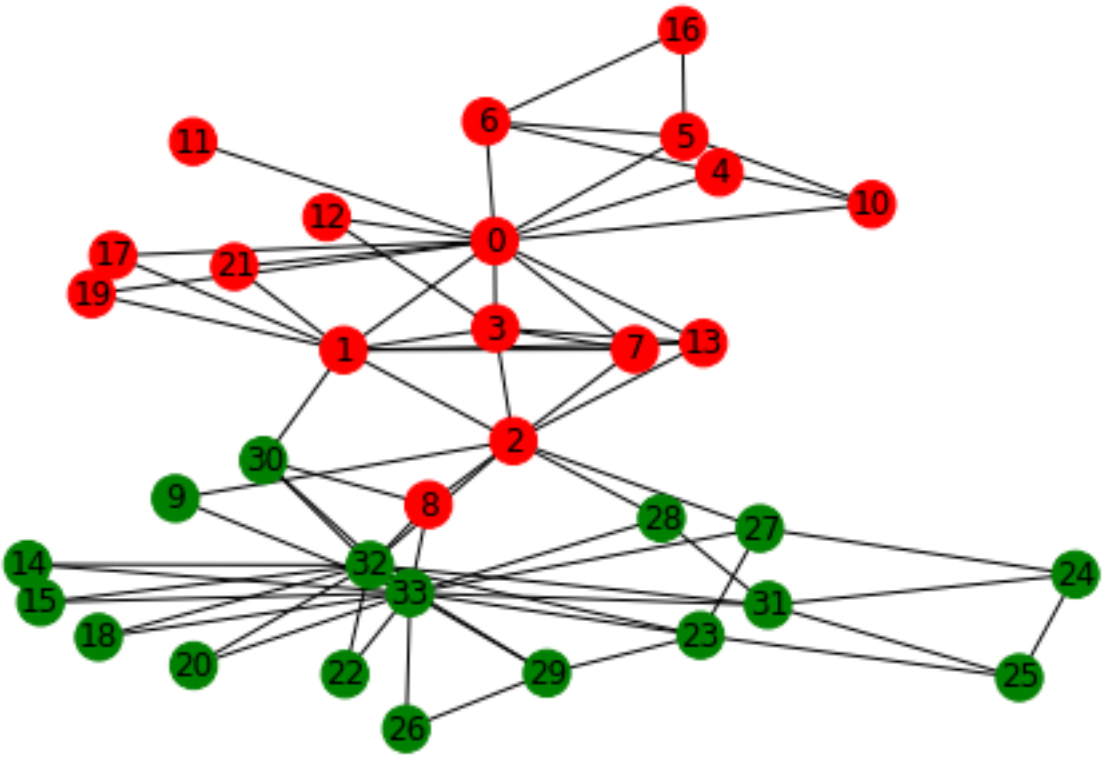

Figure 6: The graph after the fifth iteration of my Girvan-Newman-like algorithm.

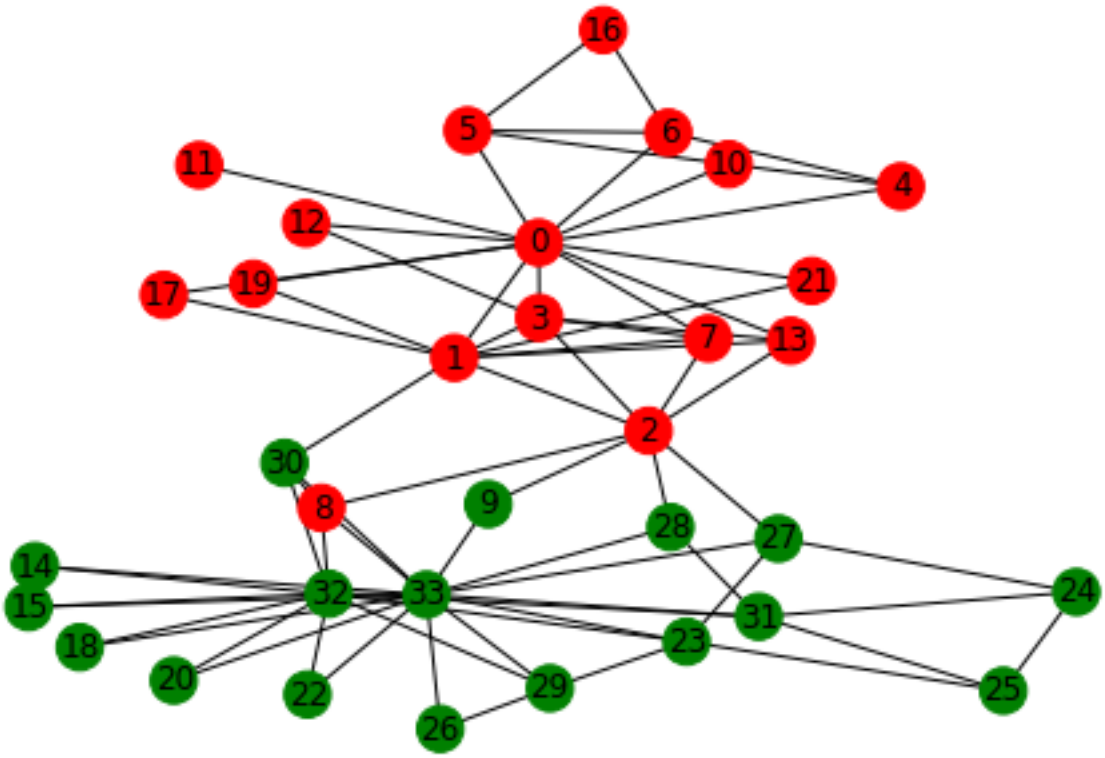

Figure 7: The graph after the sixth iteration of my Girvan-Newman-like algorithm.

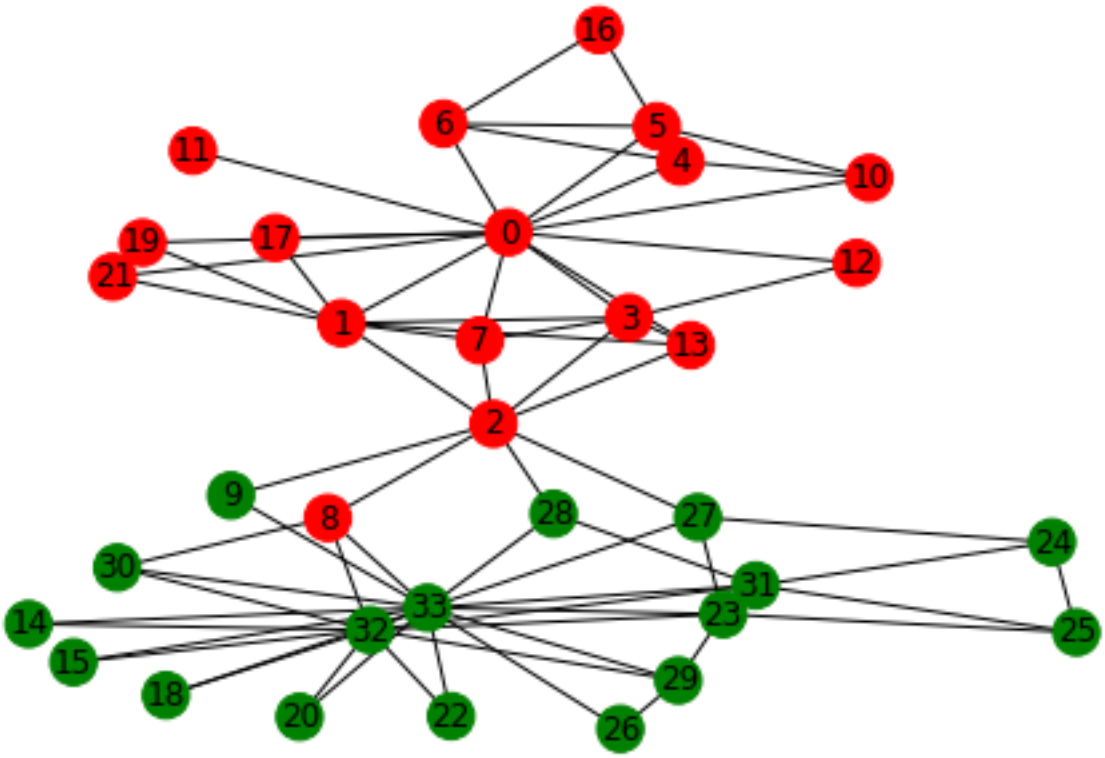

Figure 8: The graph after the seventh iteration of my Girvan-Newman-like algorithm.

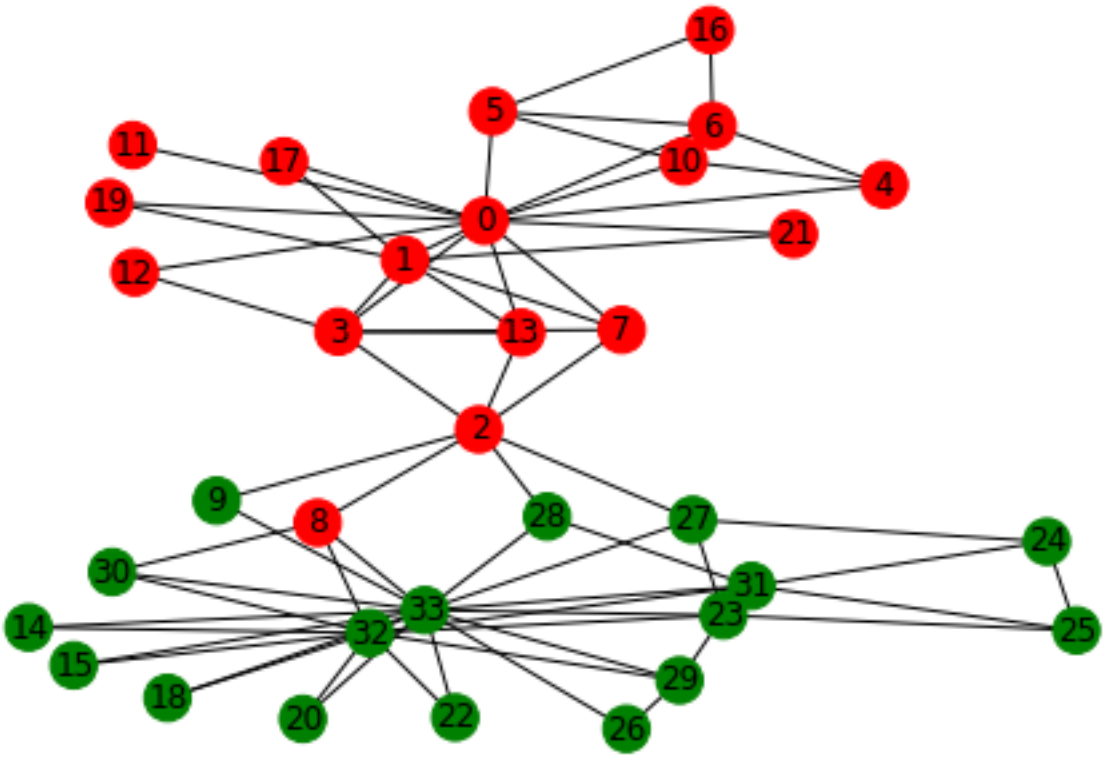

Figure 9: The graph after the eighth iteration of my Girvan-Newman-like algorithm.

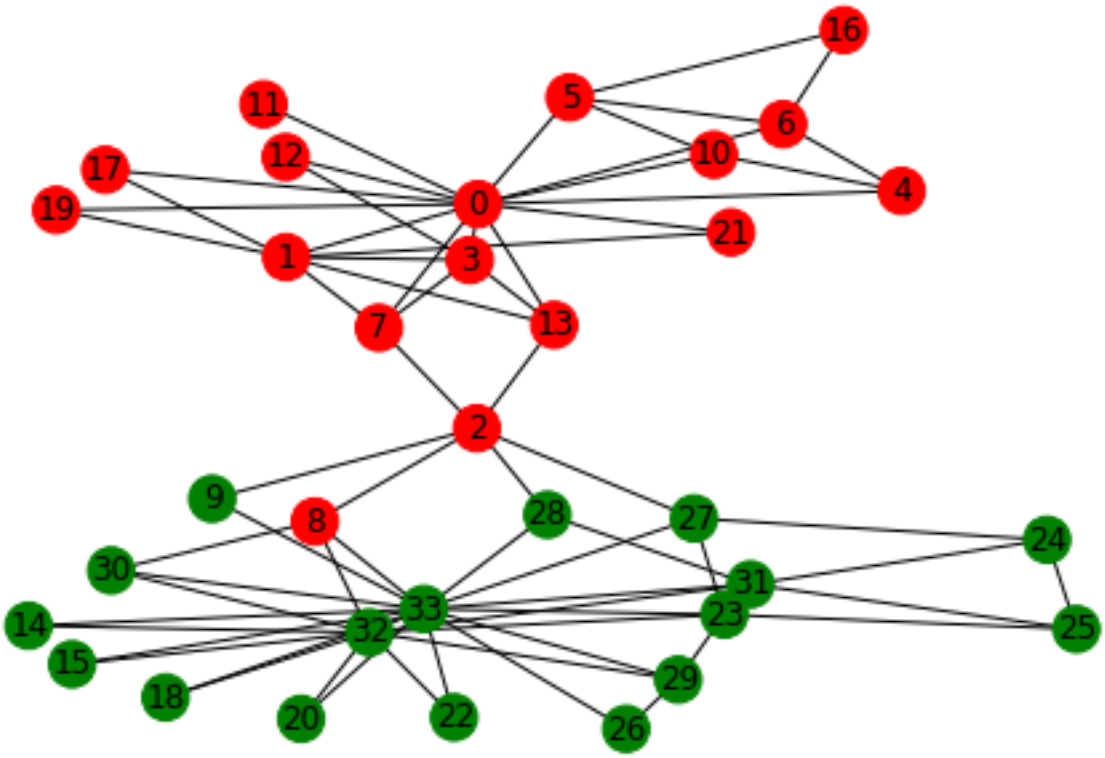

Figure 10: The graph after the ninth iteration of my Girvan-Newman-like algorithm.

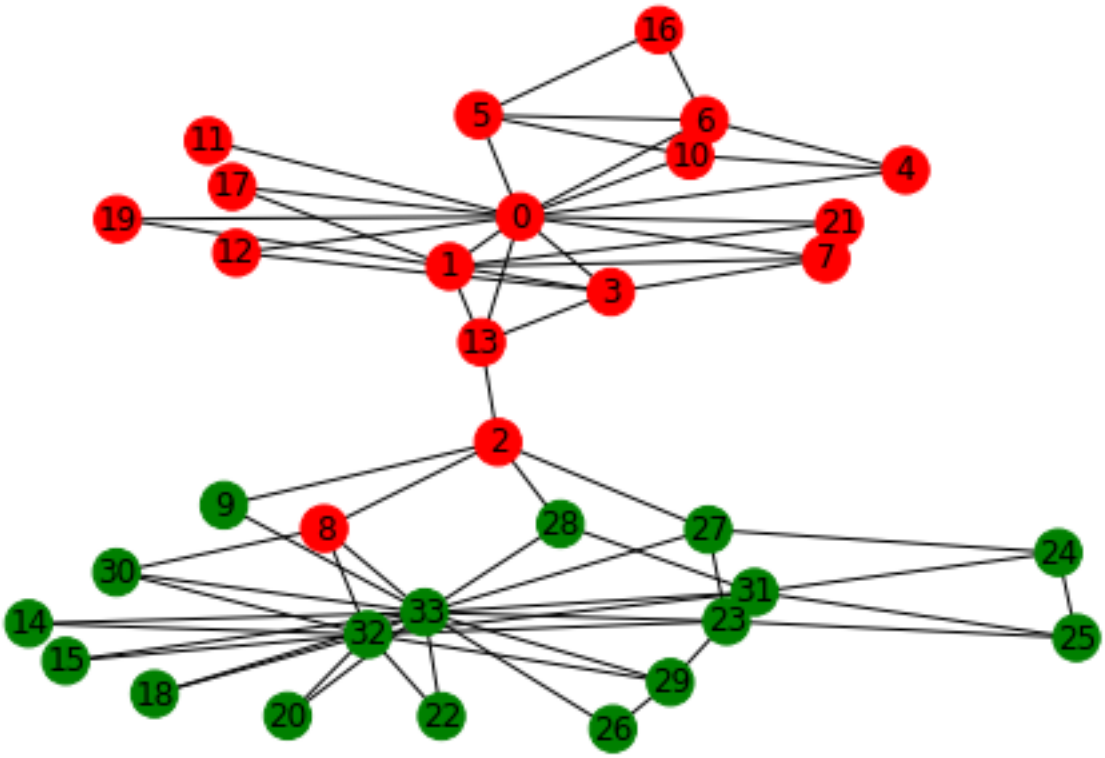

Figure 11: The graph after the tenth iteration of my Girvan-Newman-like algorithm.

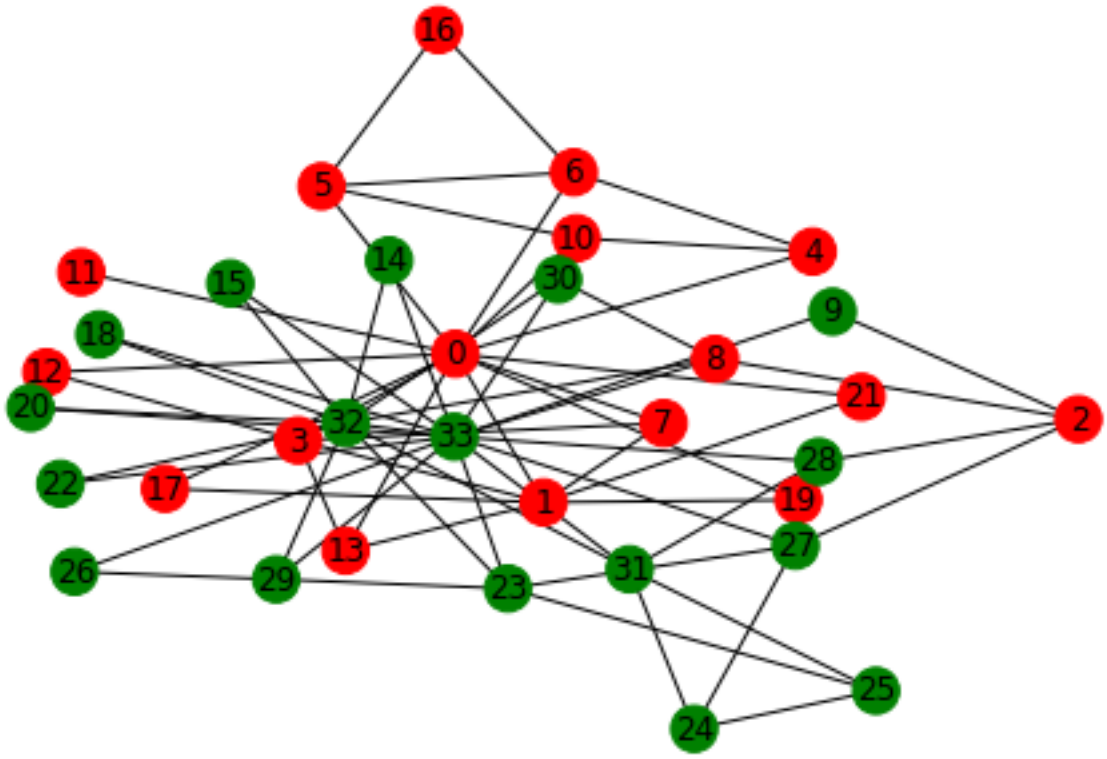

Figure 12: The graph after the eleventh iteration of my Girvan-Newman-like algorithm.

Because the graph got rather messy after the eleventh iteration, I inserted a circular version of the step1 graph and a circular version of the iteration11 graph on the next two pages. The circular step1 graph is for comparison to the circular version of the iteration11 graph. I did attempt to run more iterations in the hope of the graph looking neater to no avail, hence the circular graphs.

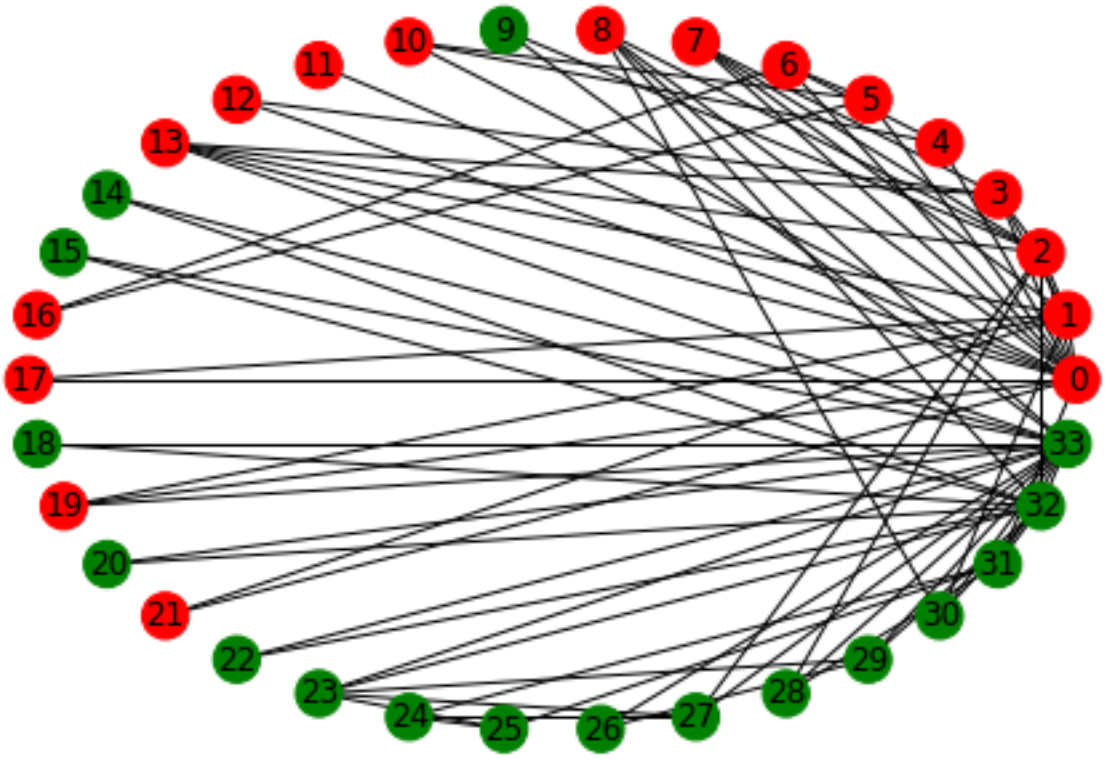

Figure 13: A circular version of the original Karate Club graph used in step1 for comparison to the circular version of iteration11.

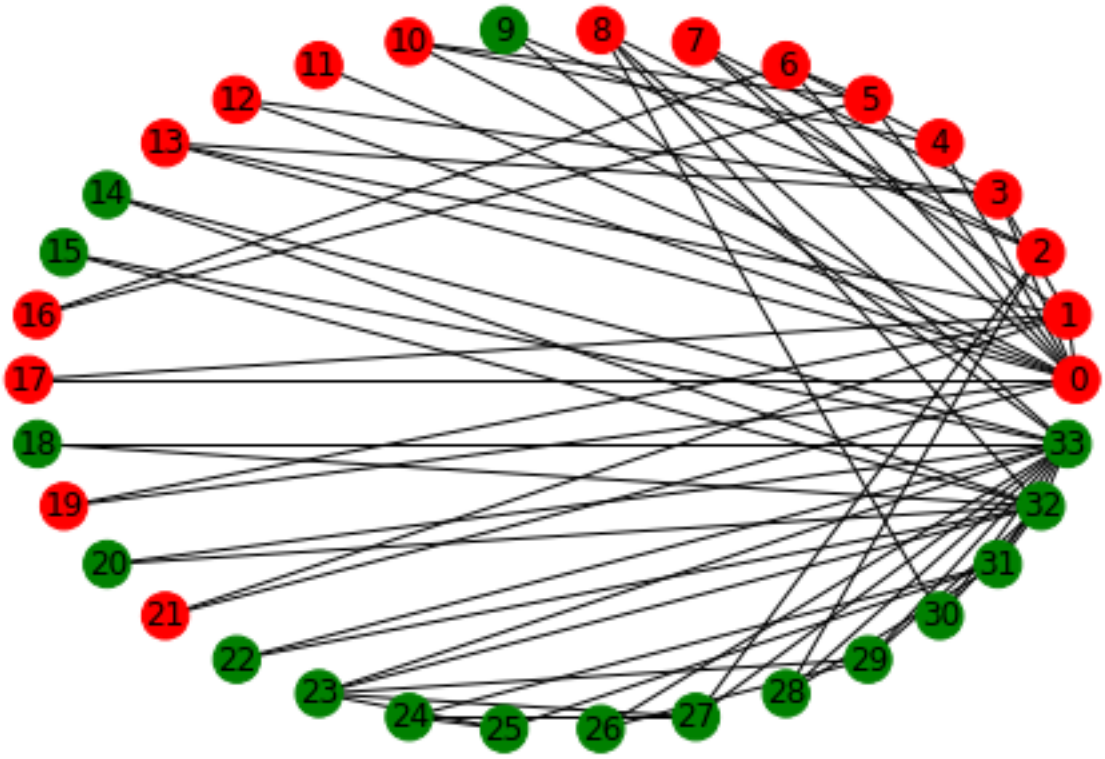

Figure 14: A circular version of the graph after the eleventh iteration.

Discussion

Starting with my edge-removal algorithm in listing 2, the edge betweenness of every edge is calculated by the "edge_betweenness_centrality" function on line 3. The dictionary returned by the function is stored in the "edges" variable. The dictionary stored in the "edges" variable was copy-pasted into "edges.txt." The file contents are displayed in listing 3. **Please note that the dictionary displayed consists of the betweenness of every edge of the original graph;** no edges have been removed. It is only meant to illustrate the data container. On line 5, the keys (edges) in the "edges" dictionary are sorted by their values (betweenness) in descending order, and stored in the "sortedEdges" variable. The resulting list of tuples stored in the "sortedEdges" variable was copy-pasted into "sortedEdges.txt." The file contents are displayed in listing 4. Like "edges.txt," no edges have been removed. On line 7, the first edge in "sortedEdges" (the edge with the highest betweenness) is removed. Something to note: since this was done in Google Collab, lines 3 and 5 are redone every time the code is executed. So after an edge is removed, the betweenness values are recalculated and re-sorted. This assures that the edge with the highest betweenness is removed every iteration. When an edge is removed, it is appended to a list called "removedEdges." Note that in the Collab notebook, "removedEdges" was declared as an empty list in the cell above the algorithm, and it is not shown in the report. The content of "removedEdges" was copy-pasted into "removedEdges.txt." The contents of "removedEdges.txt" are displayed in listing 5. The graph is then drawn. Then the value (true or false) of the "is_connected" function is printed, along with the number of connected components of the graph using the "number_connected_components" function.

The algorithm was executed manually until the "is_connected" function returned "false." This took eleven iterations. Once the graph was no longer one connected component, the "number_connected_components" function printed out "2." This meant the graph consisted of two connected components. Because it was difficult to see the disconnected components of the graph in figure 12, I plotted a circular version in figure 14. I also took the liberty of providing a circular version of the graph in figure 1 for comparison. It is shown in figure 13.

```
1 {(0, 1): 0.025252525252525245,
2  (0, 2): 0.0777876807288572,
3  (0, 3): 0.02049910873440285,
4  (0, 4): 0.0522875816993464,
5  (0, 5): 0.07813428401663694,
6  (0, 6): 0.07813428401663695,
7  (0, 7): 0.0228206434088787,
8  (0, 8): 0.07423959482783014,
9  (0, 10): 0.0522875816993464,
10 (0, 11): 0.058823529411764705,
11 (0, 12): 0.04652406417112298,
12 (0, 13): 0.04237189825425121,
13 (0, 17): 0.04012392835922248,
14 (0, 19): 0.045936960642843,
15 (0, 21): 0.040123928359222474,
16 (0, 31): 0.1272599949070537,
```

```
 (1, 2): 0.023232323232323233,
 (1, 3): 0.0077243018419489,
 (1, 7): 0.007422969187675069,
 (1, 13): 0.01240556828792123,
 (1, 17): 0.01869960105254222,
 (1, 19): 0.014633732280791102,
 (1, 21): 0.01869960105254222,
 (1, 30): 0.032280791104320514,
 (2, 3): 0.022430184194890075,
 (2, 7): 0.025214328155504617,
 (2, 8): 0.009175791528732704,
 (2, 9): 0.030803836686189627,
 (2, 13): 0.007630931160342923,
 (2, 27): 0.04119203236850296,
 (2, 28): 0.02278244631185807,
 (2, 32): 0.06898678663384543,
 (3, 7): 0.003365588659706307,
 (3, 12): 0.012299465240641705,
 (3, 13): 0.01492233256939139,
 (4, 6): 0.0047534165181224,
 (4, 10): 0.0029708853238265,
 (5, 6): 0.0029708853238265003,
 (5, 10): 0.0047534165181224,
 (5, 16): 0.029411764705882353,
 (6, 16): 0.029411764705882353,
 (8, 30): 0.00980392156862745,
 (8, 32): 0.0304416716181422,
 (8, 33): 0.04043657867187279,
 (9, 33): 0.029615482556659026,
 (13, 33): 0.06782389723566191,
 (14, 32): 0.024083977025153497,
 (14, 33): 0.03473955238661121,
 (15, 32): 0.024083977025153497,
 (15, 33): 0.03473955238661121,
 (18, 32): 0.024083977025153497,
 (18, 33): 0.03473955238661121,
 (19, 33): 0.05938233879410351,
 (20, 32): 0.024083977025153497,
 (20, 33): 0.03473955238661121,
 (22, 32): 0.024083977025153493,
 (22, 33): 0.03473955238661121,
 (23, 25): 0.019776193305605066,
 (23, 27): 0.010536739948504653,
 (23, 29): 0.00665478312537136,
 (23, 32): 0.022341057635175278,
 (23, 33): 0.03266983561101209,
 (24, 25): 0.0042186571598336305,
```

```
 (24, 27): 0.018657159833630418,
 (24, 31): 0.040106951871657755,
 (25, 31): 0.04205783323430383,
 (26, 29): 0.004532722179781003,
 (26, 33): 0.0542908072319837,
 (27, 33): 0.030477039300568713,
 (28, 31): 0.0148544266191325,
 (28, 33): 0.024564977506153975,
 (29, 32): 0.023328523328523323,
 (29, 33): 0.029807882749059215,
 (30, 32): 0.01705288175876411,
 (30, 33): 0.02681436210847975,
 (31, 32): 0.04143394731630026,
 (31, 33): 0.05339388280564752,
 (32, 33): 0.008225108225108224}
```

Listing 3: Dictionary with edges as keys and their betweenness as values

```
[(0, 31),
 (0, 6),
 (0, 5),
 (0, 2),
 (0, 8),
 (2, 32),
 (13, 33),
 (19, 33),
 (0, 11),
 (26, 33),
 (31, 33),
 (0, 4),
 (0, 10),
 (0, 12),
 (0, 19),
 (0, 13),
 (25, 31),
 (31, 32),
 (2, 27),
 (8, 33),
 (0, 17),
 (0, 21),
 (24, 31),
 (14, 33),
 (15, 33),
 (18, 33),
 (20, 33),
 (22, 33),
 (23, 33),
 (1, 30),
```

```
 (2, 9),
 (27, 33),
 (8, 32),
 (29, 33),
 (9, 33),
 (5, 16),
 (6, 16),
 (30, 33),
 (0, 1),
 (2, 7),
 (28, 33),
 (14, 32),
 (15, 32),
 (18, 32),
 (20, 32),
 (22, 32),
 (29, 32),
 (1, 2),
 (0, 7),
 (2, 28),
 (2, 3),
 (23, 32),
 (0, 3),
 (23, 25),
 (1, 17),
 (1, 21),
 (24, 27),
 (30, 32),
 (3, 13),
 (28, 31),
 (1, 19),
 (1, 13),
 (3, 12),
 (23, 27),
 (8, 30),
 (2, 8),
 (32, 33),
 (1, 3),
 (2, 13),
 (1, 7),
 (23, 29),
 (4, 6),
 (5, 10),
 (26, 29),
 (24, 25),
 (3, 7),
 (5, 6),
```

```
 (4, 10)]
```

Listing 4: List of tuples that consists of edges sorted by their betweenness in descending order.

```
[(0, 31),
 (0, 2),
 (0, 8),
 (13, 33),
 (19, 33),
 (2, 32),
 (1, 30),
 (1, 2),
 (2, 3),
 (2, 7),
 (2, 13)]
```

Listing 5: List of tupes that contains the removed edges of the Karate Club graph.

Step 3

Note: I will be comparing figures 13 and 14.

The first difference to note is the step1 graph has only one connected component. The graph after eleven iterations has two connected components. Something I noticed between the two graphs is after eleven iterations, there were edges removed between red nodes, red and green nodes, but not green nodes. Comparing the list of removed edges to the nodes in the graph corroborates this. I counted six edges between red nodes removed, and five edges between red and green nodes removed. Given the amount of edges removed between nodes belonging to Mr. Hi (red nodes), I'm not sure that this particular split could have been predicted by the weighted graph of social interactions. I would think that like-minded people (in this case, people that belong to Mr. Hi's club; red nodes) would have a stronger connection than than those who think differently (red and green nodes). Perhaps the takeaway here is that you never really know somebody.

References

- NetworkX edge_betweenness_centrality, https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.edge_betweenness_centrality.html?highlight=edge_betweenness_centrality#networkx.algorithms.centrality.edge_betweenness_centrality
- NetworkX karate_club_graph, https://networkx.org/documentation/stable/reference/generated/networkx.generators.social.karate_club_graph.html?highlight=karate_club_graph#networkx.generators.social.karate_club_graph
- Python Dictionaries, https://www.w3schools.com/python/python_dictionaries.asp
- How to Sort Python Dictionaries by Key or Value, https://www.pythoncentral.io/how-to-sort-python-dictionaries-by-key-or-value/#:~:text=Sorting%20Python%20dictionaries%20by%20Keys.%20If%20we%20want,has%20been%20sorted%20%28in%20ascending%20order%20by%20default%29.
- NetworkX remove_edge, https://networkx.org/documentation/stable/reference/classes/generated/networkx.Graph.remove_edge.html?highlight=remove_edge#networkx.Graph.remove_edge
- NetworkX is_connected, https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.components.is_connected.html?highlight=is_connected#networkx.algorithms.components.is_connected
- NetowrkX number_connected_components, https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.components.number_connected_components.html?highlight=number_connected_components#networkx.algorithms.components.number_connected_components
- Week-07 Slides, https://docs.google.com/presentation/d/1FzrzxRzslE20nWOjb_uM8jz2xvT1IOOZ9uIoYVjco7s/edit#slide=id.g30f264e945_1_6
- Week-09 Slides, https://docs.google.com/presentation/d/1POtPTmBw6MSBI7qIT85uedit#slide=id.g311d60d551_0_6
- An Information Flow Model for Conflict and Fission in Small Groups, http://aris.ss.uci.edu/~lin/76.pdf