project-mitigating-malware

April 10, 2024

1 Modelling Methods of Selective Anti-Malware Training Based on Network Centralities

1.1 Casual Motivation

The most common methods of mitigating malware spread across a network involve a "Security Awareness" course during company onboarding. This training is then repeated when necessary, usually every few years. Frequently, but not too often to be suspicious, a security team will test employees using various techniques, such as "phishing" emails (creating an email with an inauthentic link to accept a user's secrets) or supplying files containing malware disguised as non-threatening content. When employees fall victim to these tests, they usually repeat specific training covering the section they failed and potentially more. This method allows companies to detect weak points for social engineering in their cyber network and correct them as soon as they become apparent.

While this method includes testing all employees equally, members with more connections, who are more likely to be targets and have more people to spread malware to if infected, are not given special consideration. Additionally, those members who are more trusted within the network should be more vigilant of attempts to infect their accounts, as malicious attacks would hijack the trust of the victim to spread the data further across the network. Therefore, this project aims to look at methods of using proactive anti-malware training to further secure a network against hackers, based on the connectivity and centrality of a companies members.

1.2 Data-set

This project uses internal email communications from the defunct company Enron. The email dataset contains more than 1,000,000 emails from approximately 1998 to 2003, including messages from high-ranking members. It was created as part of the Federal Energy Regulatory Commission, and released in 2011, after the company's collapse in 2001.

This dataset allows for the production of a network containing employee contacts and frequent email relations, allowing us to produce a graph to simulate communication and trust between employees across the companies. Therefore, this dataset is a good choice for this project, as it is both a natural (formed in an actual due course of business, as opposed to an artificial network) and representative of a large company. While, given the circumstances, "trust" being modelled through Enron's communications may seem dubious (given its well-known and documented large-scale fraud), as we presume that internal communications are representative of trust between employees who were not direct victims of the fraud, "trust" should be a reasonable metric to examine.

A copy of the dataset can be found here: https://www.cs.cmu.edu/~enron/

It can be explored online here: http://www.enron-mail.com/

Example sample contents of a file:

```
maildir/allen-p/inbox/1.

...

 -----Original Message-----
From:   Dunton, Heather
Sent:   Wednesday, December 05, 2001 1:43 PM
To: Allen, Phillip K.; Belden, Tim
Subject:    FW: West Position

Attached is the Delta position for 1/16, 1/30, 6/19, 7/13, 9/21


 -----Original Message-----
From:   Allen, Phillip K.
Sent:   Wednesday, December 05, 2001 6:41 AM
To: Dunton, Heather
Subject:    RE: West Position

Heather,

This is exactly what we need.  Would it possible to add the prior day for each of the dates bel

Thank you,

Phillip Allen
...
```

However, as we do not need the content of the emails, we can use a smaller database that already contains the information regarding which addresses sent messages to others. This removes the pre-screening required to remove all of the context and attribute names to pseudonymised nodes. This database can be found here: http://konect.cc/networks/enron/

Content sample:

```
/enron/out.enron
...

1000 966 1 1004411463
1000 966 1 1004411913
100 100 1 1010781654
100 10756 1 1010687491
100 10756 1 1010781654
100 10756 1 1011379321
1001 1002 1 1006901795
1001 1077 1 1002320021
```

```
100 111 1 1010515702
100 111 1 1011379321
1001 129 1 1002320021
...
```

Note the data is in the structure sendNode<Space>recieveNode<Space>Weight<Space>unixTimeStamp

2 Code

This section includes the main code body for the project.

2.1 Importing packages

This section import the packages used for the project so you do not have to run other code segments just to import the package into the python kernel

[1]:
```
import tarfile # Used to uncompress data file...
import pandas as pd # Reading files to databases/tables
import networkx as nx # Used for network functions and visualisations
import matplotlib.pyplot as plt # Visualisation exporting library
import numpy as np # Various useful mathematical functions...
import sys # Replacing lines and output stuff
import json

def inpri(string:str):
    sys.stdout.write(f"\r{string}")
    sys.stdout.flush()

inpri("Import Complete! You have run this block!")
```

```
Import Complete! You have run this block!
```

2.2 Extracting the TarGz archive

Due to the large data size, it may be unfeasible to include the raw, unarchive format within this project. Therefore, if you do have the compressed TarBz2 archive, run the following code to decompress the data.

[2]:
```
def untar_enron():
    with tarfile.open("./enron-network.tar.bz2") as enron_tar:
        inpri("Extracting enron files:")
        enron_tar.extractall(filter='data')
        enron_tar.close()
    inpri("Files extracted!         ")
untar_enron()
```

```
Files extracted!
```

The data should appear in the `./enron/out.enron` file

2.3 Cleaning the data file

While the data file contains all the relevent information to complete this project, the format of the data is not optimal for the later stages of the program. Therefore, instead of storing the file in memory and adjusting it for each step, this process will extract all unused data for the project. It will additionally change the structure of the data from an unweighted graph with repeated edges, to a weighted graph without repeated edges.

The format of the data should be `senderNode,receiverNode,weight`.

[3]:
```python
def clean_data_file(fileName : str = "./enron/out.enron"):
    inpri("Cleaning data")
    with open(fileName, "r", encoding='utf-8') as enron:

        count_dict = {}
        send_dict = {} # Keeps track of how many emails sent by node
        receive_dict = {} # Keeps track of how many emails received
        for line in enron.readlines():
            fields = line.split(" ")
            try:
                sender = int(fields[0])
                receiver = int(fields[1])
            except ValueError:
                inpri(f"Line not in correct format: {line}")
                continue

            if sender == receiver:
                # Reject loops, not required for this project
                continue

            keyString = f"{sender},{receiver}"

            try:
                count_dict[keyString] += 1
            except KeyError:
                count_dict[keyString] = 1

            try:
                send_dict[sender] += 1
            except KeyError:
                send_dict[sender] = 1

            try:
                receive_dict[receiver] += 1
            except KeyError:
                receive_dict[receiver] = 1

        enron.close()
```

```
inpri("Information converted")

# Remove inactive / less frequent email addresses.
# and renumber nodes

inpri("Removing infrequent addresses and reordering IDs")
remap_ids = {}
next_id = 1
out_dict = {}
for key in count_dict.keys():
    s,r = key.split(",")

    try:
        if send_dict[int(s)] < 200 or send_dict[int(r)] < 200:
            continue
    except KeyError:
        continue
    new_key = ""
    if s in remap_ids.keys():
        new_key += str(remap_ids[s]) + ","
    else:
        remap_ids[s] = next_id
        new_key += str(next_id) + ","
        next_id += 1

    if r in remap_ids.keys():
        new_key += str(remap_ids[r])
    else:
        remap_ids[r] = next_id
        new_key += str(next_id)
        next_id += 1

    out_dict[new_key] = count_dict[key]

# out_string = "sender,receiver,quantity\n"
out_string = ""
def sortKey(x):
    send,receive = x[0].split(",")
    return [int(send), int(receive)]

inpri("Formatting data to string...")
for key, value in sorted(out_dict.items(), key=sortKey):
    out_string += f"{key},{value}\n"

inpri("Writing to './project-networks/cleaned_data.csv' ...")
```

```
    with open("./project-networks/cleaned_data.csv", "w", encoding="utf-8") as␣
 ↪out:
        out.write(out_string)

    inpri("\nCompleted")

clean_data_file()
```

```
Line not in correct format: % asym positive
Writing to './project-networks/cleaned_data.csv' …
Completed
```

2.3.1 Let's have a brief look over the data

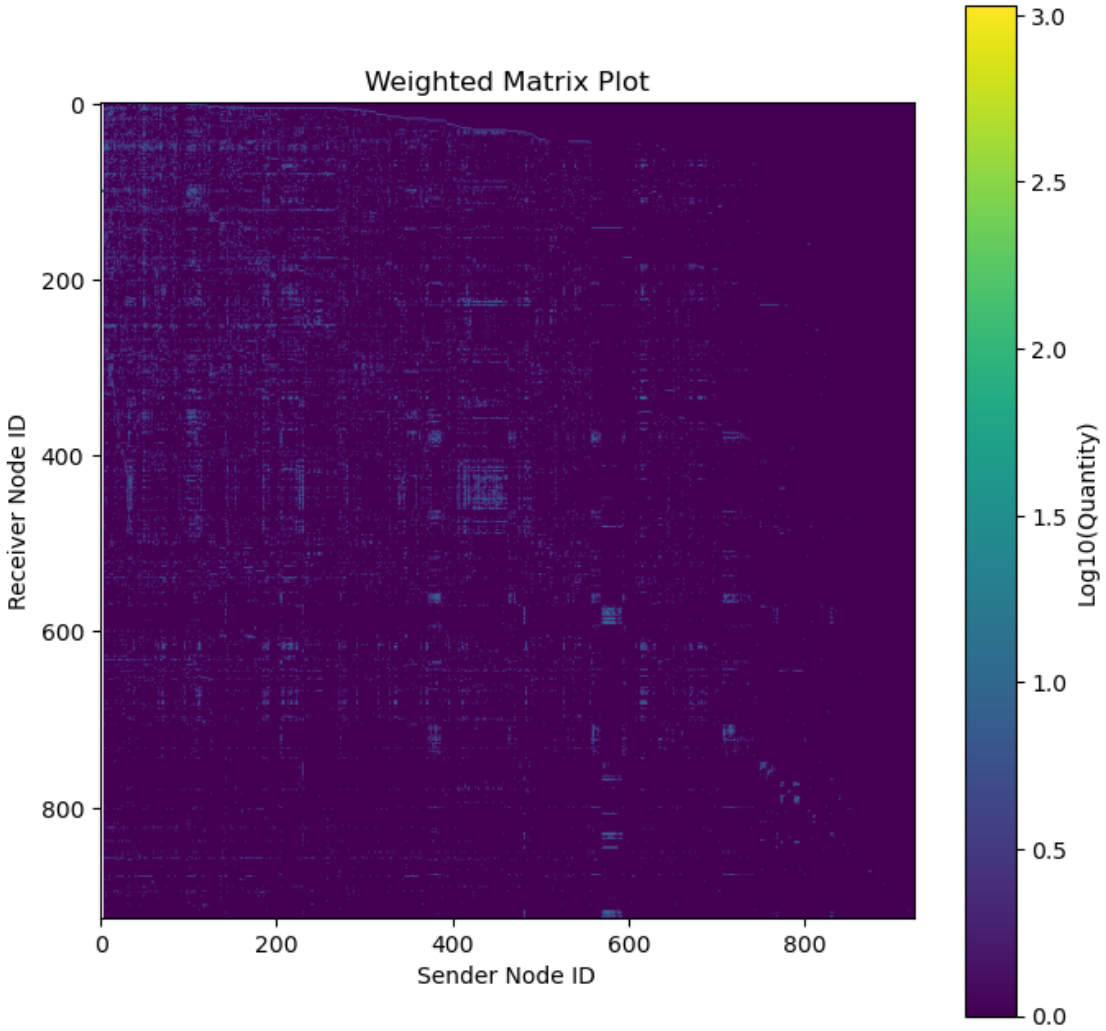
Success! The data is now cleaned. All loops are removed (as it doesn't make sense to model sending malware to oneself), and the edges refer to the amount one person sends emails to another. Let's see what insight we can gain into the network now.

[4]:
```
def log_plot_of_quants(fileName:str='./project-networks/cleaned_data.csv'):
    df = pd.read_csv(fileName, header=None, names=['sender', 'receiver',␣
 ↪'quantity'])

    # We're going to use NetworkX for a reasonable amount of visualisation, so
    # best to keep things consistent here...
    G = nx.from_pandas_edgelist(df, source='sender', target='receiver',␣
 ↪edge_attr='quantity', create_using=nx.DiGraph)
    adj_matrix = nx.to_numpy_array(G)

    # Converting to G using NetworkX automatically normalises the Weights
    # Let's undo that here
    max_quant = df["quantity"].max()
    inpri(f"Max value: {max_quant}")
    for i in range(len(adj_matrix)):
        adj_matrix[i][2] *= max_quant

    # Log10 the data...
    # This throws a Div0 error, but as only when weights are 0 it doesn't
    # matter
    adj_matrix[:,2] = np.log10(adj_matrix[:,2])


    # Plotting the matrix...
    plt.figure(figsize=(8, 8))
    plt.imshow(adj_matrix, cmap='viridis', origin='upper')
    plt.colorbar(label='Log10(Quantity)')
    plt.title('Weighted Matrix Plot')
    plt.xlabel('Sender Node ID')
    plt.ylabel('Receiver Node ID')
```

```
    plt.show()

log_plot_of_quants()
```

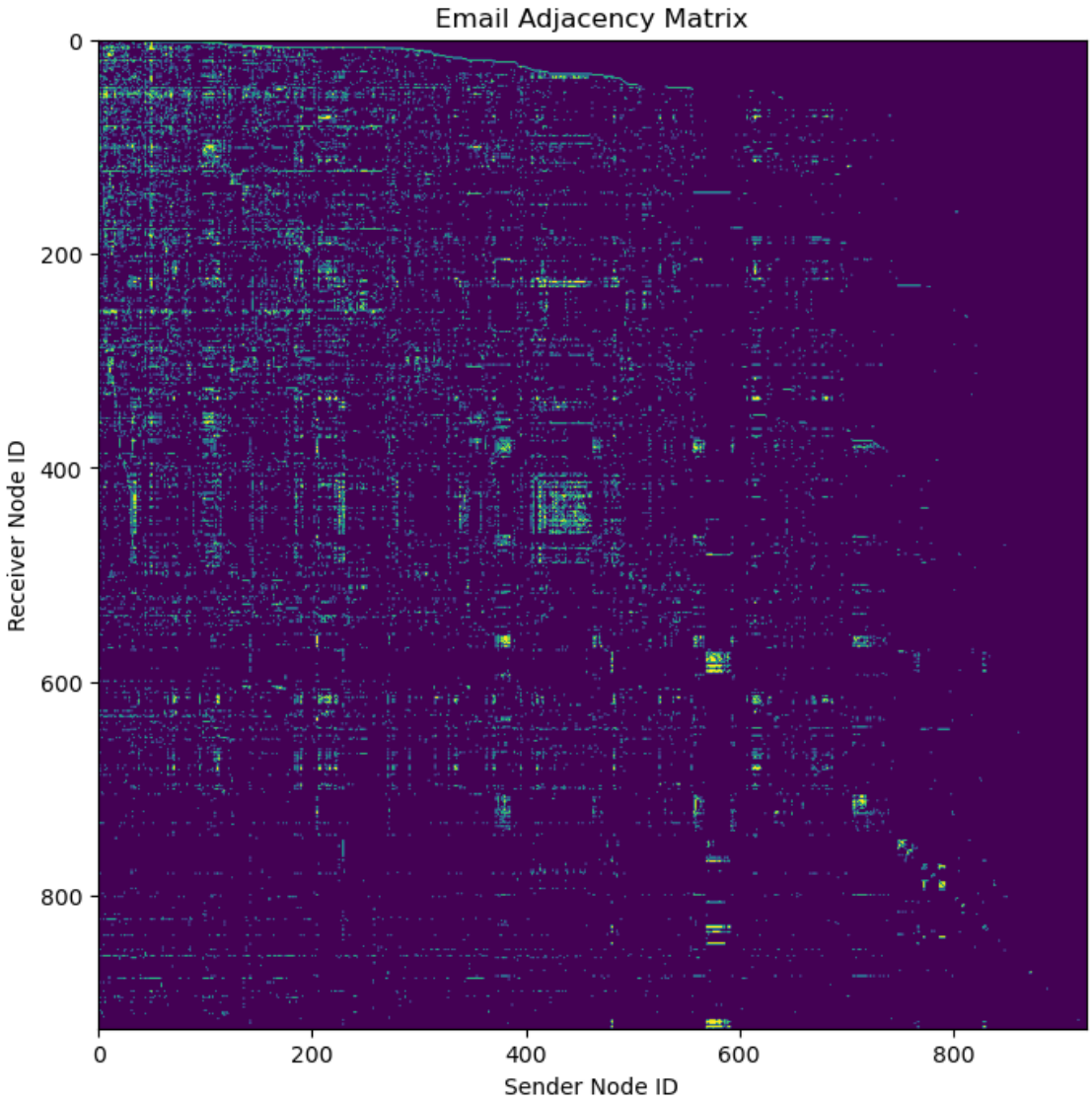
```
Max value: 1074
```

```
/tmp/ipykernel_3256/4192940284.py:19: RuntimeWarning: divide by zero encountered
in log10
  adj_matrix[:,2] = np.log10(adj_matrix[:,2])
```



Log10 Data shows most of the communication occurs within the diagonal. This could represent that the majority of communications happen between users of the same standing within the company. Additionally, there are multiple clear horizontal and vertical lines, indicating that these people tend to send or receive information from everyone in the company. These indicate potential members of HR, potential interviewers, or management that require constant knowledge.

[5]:
```
def adj_plot_of_quants(fileName:str='./project-networks/cleaned_data.csv'):
    df = pd.read_csv(fileName, header=None, names=['sender', 'receiver',␣
 ↪'quantity'])

    # We're going to use NetworkX for a reasonable amount of visualisation, so
    # best to keep things consistent here...
    G = nx.from_pandas_edgelist(df, source='sender', target='receiver',␣
 ↪create_using=nx.DiGraph)
    adj_matrix = nx.to_numpy_array(G)

    # Plotting the matrix...
    plt.figure(figsize=(8, 8))
    plt.imshow(adj_matrix, cmap='viridis', origin='upper')
    plt.title('Email Adjacency Matrix')
    plt.xlabel('Sender Node ID')
    plt.ylabel('Receiver Node ID')
    plt.show()

adj_plot_of_quants()
```



This is something to expect! Not much communication outside the divisions (the empty space away from the diagonal), but there are some straight lines across both the sender and receiver nodes. This is a small indication that there are area leads that receive and send more emails than others, and all across the board, not just from one division.

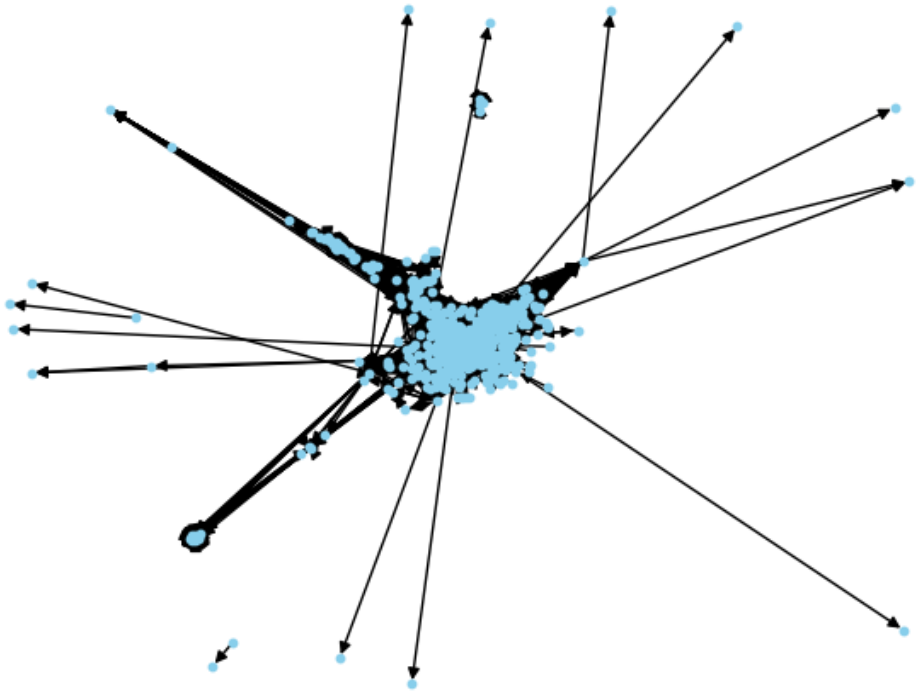
The low-id receiver nodes have a distinct tail-off, where they go black quickly and absolutely. This represents the high-level members of the company, potentially sending emails to all members, or at least high-level ones, but not receiving emails from low-level employees.

There is a big sender node spree from the ~500 IDs. Although the nodes' IDs are very difficult to trace back, this could be marketing, HR, or other fields that require constant "in the know".

Additionally, there is a lot of empty space from the 1400 sender node onwards. This could be groups of lower-level employees who only message their supervisor or small team but do so frequently

enough to have over 200 emails sent.

[6]:
```
def circular_plot_of_adj(fileName:str='./project-networks/cleaned_data.csv'):
    df = pd.read_csv(fileName, header=None, names=['sender', 'receiver',␣
 ↪'quantity'])

    # We're going to use NetworkX for a reasonable amount of visualisation, so
    # best to keep things consistent here...
    G = nx.from_pandas_edgelist(df, source='sender', target='receiver',␣
 ↪create_using=nx.DiGraph)

    # Plotting the matrix...
    pos = nx.spring_layout(G) # Define the layout for node positioning
    nx.draw(G, pos, with_labels=False, node_size=10, node_color='skyblue')
    # Display the graph
    plt.show()

circular_plot_of_adj()
```



2.4 Limiting network to the largest strong network

This visualisation shows a few sinks in the data (nodes which do not send emails to any other). While beneficial for visualising the given network, this wouldn't necessarily help the malware simulation. Since these sinks would be infected and then not send any malware anywhere, it doesn't make much sense even to consider them when studying how malware spreads. Therefore, we shall remove them for this project. This also applies to the unconnected segments of the network. Let's do some more cleaning of the data and see what the weakly or strongly connected networks look like...

[7]:
```
def get_strong_network(fileName : str = "./project-networks/cleaned_data.csv",␣
↪v=False):
    df = pd.read_csv(fileName, header=None, names=['sender', 'receiver',␣
↪'quantity'])
    G = nx.from_pandas_edgelist(df, source='sender', target='receiver',␣
↪edge_attr='quantity', create_using=nx.DiGraph)

    s_connected = list(nx.strongly_connected_components(G))
    max_size = 0
    for i, network in enumerate(s_connected):
        if len(network) > max_size:
            max_size = len(network)

    if v:
        print(f"The size of the biggest strongly connected network is␣
↪{max_size}")
    return s_connected

def get_weak_network(fileName : str = "./project-networks/cleaned_data.csv",␣
↪v=False):
    df = pd.read_csv(fileName, header=None, names=['sender', 'receiver',␣
↪'quantity'])
    G = nx.from_pandas_edgelist(df, source='sender', target='receiver',␣
↪edge_attr='quantity', create_using=nx.DiGraph)

    w_connected = list(nx.weakly_connected_components(G))

    max_size = 0
    for i, network in enumerate(w_connected):
        if len(network) > max_size:
            max_size = len(network)

    if v:
        print(f"The size biggest weakly connected network is {max_size}")
    return w_connected

_ = get_strong_network(v=True)
```

```
_ = get_weak_network(v=True)
```

```
The size of the biggest strongly connected network is 802
The size biggest weakly connected network is 917
```

Both the strongly and weakly connected networks are large and contain most nodes. This is brilliant as I expected certain significant areas of nodes to be weakly connected but not strongly connected, acting as larger sinks. This has not been as impactful as I initially thought it might. I initially considered calculating the weakly connected network as a fallback to run simulations on if the strongly connected network was not extensive enough or allowed for a substantial representation of the email network. However, since the strongly connected network is large enough and has a significant number of all nodes, we shall use that for the project.

Let's recompute the clean data only to contain the strongly connected nodes...

[8]:
```
def export_strong_network(fileName:str='./project-networks/cleaned_data.csv'):
    networkNodes = list(get_strong_network(fileName))

    maxSize = 0
    connectNodes = []
    for nodeList in networkNodes:
        if len(nodeList) > maxSize:
            connectNodes = list(nodeList)
            maxSize = len(nodeList)

    outFile = open("./project-networks/strong_data.csv", "w", encoding="utf-8")
    with open(fileName, "r", encoding="utf-8") as cData:
        for line in cData.readlines():
            if len(line)<3: continue;
            src,dest,_ = line.split(",")
            if int(src) in connectNodes and int(dest) in connectNodes:
                outFile.write(line)

    inpri("Strong network export completed!")

export_strong_network()
```

```
Strong network export completed!
```

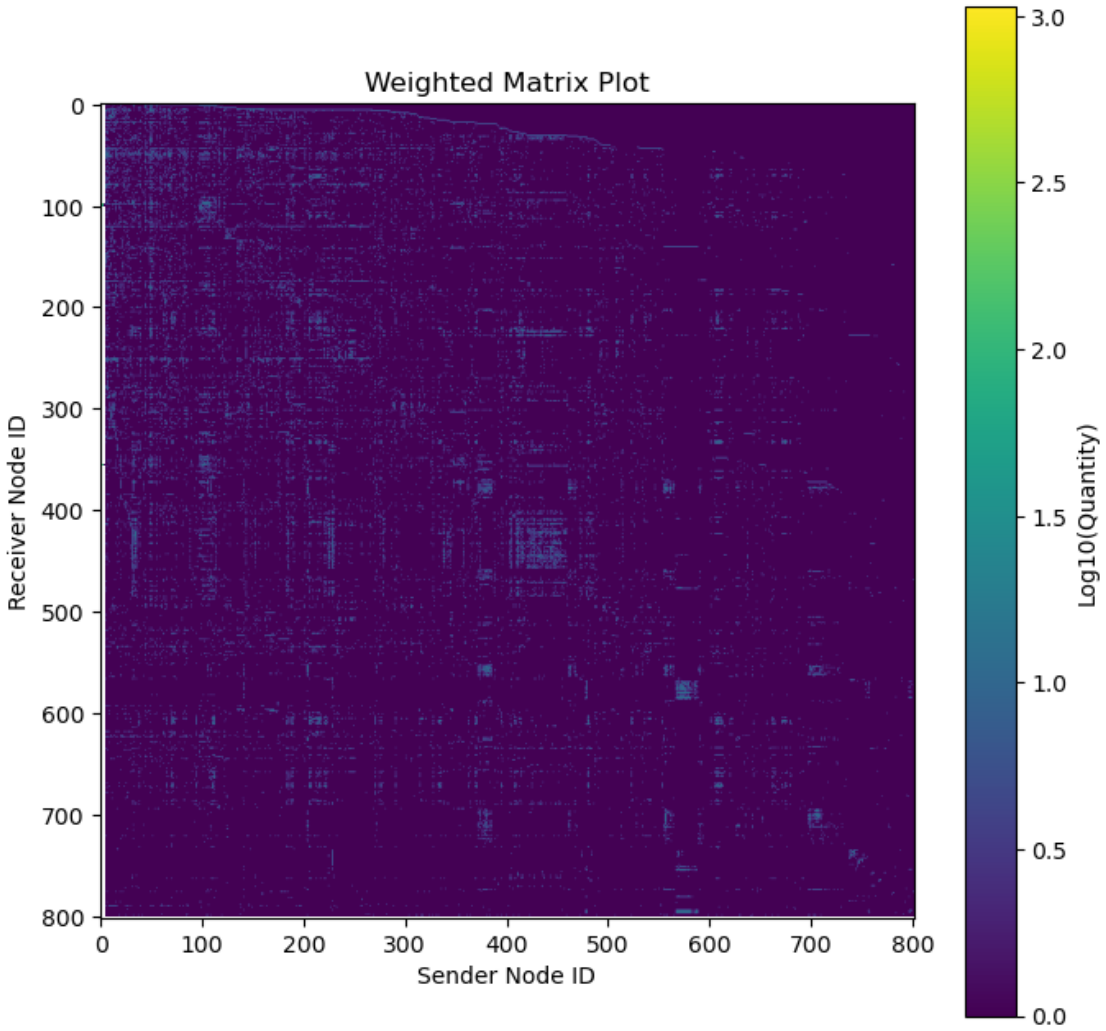
2.4.1 Visulasing Strong network

Let's run those visualisation commands on the new strong network and see what we get.

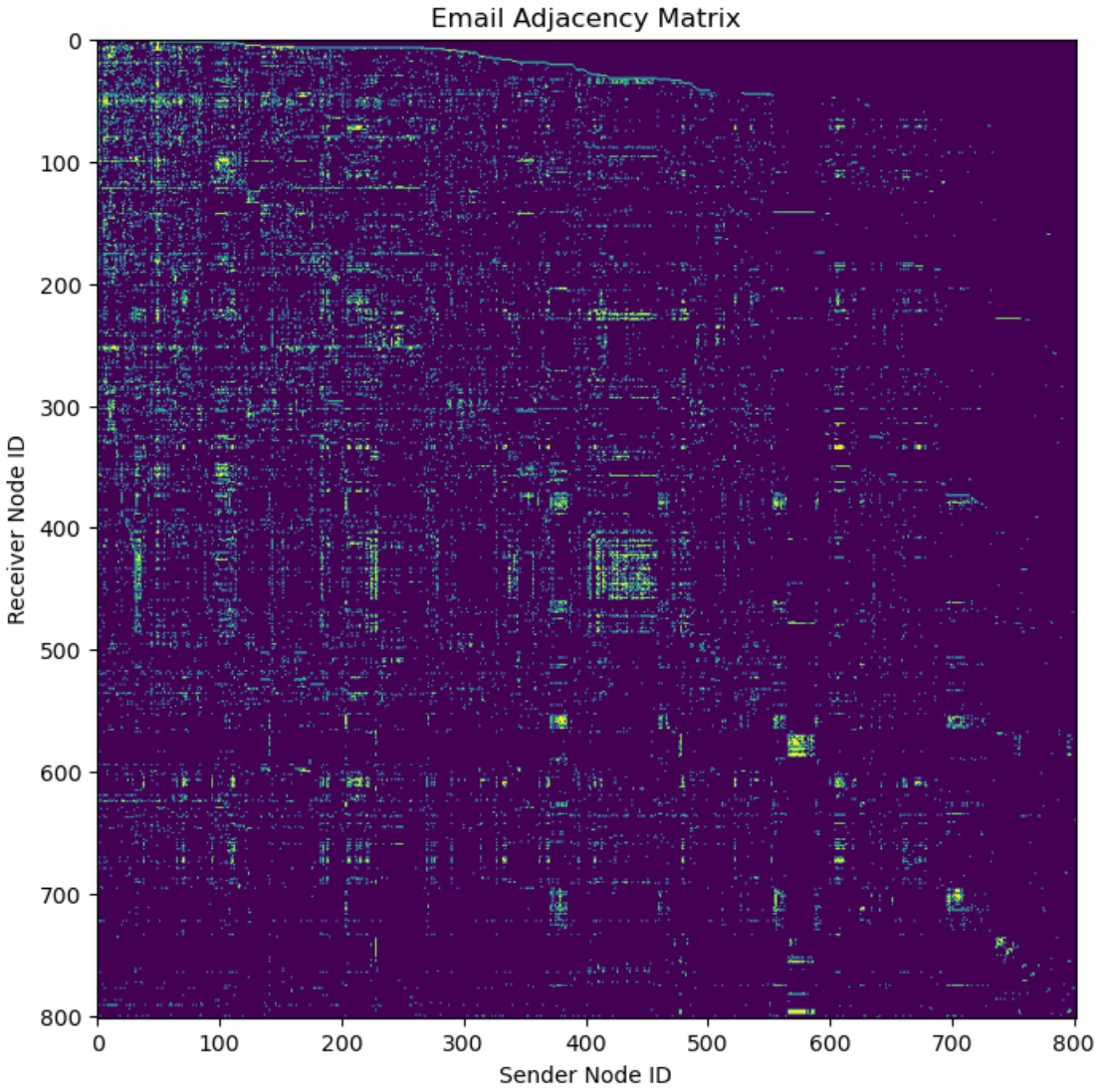
[9]:
```
log_plot_of_quants('./project-networks/strong_data.csv')
```

```
Max value: 1074
```

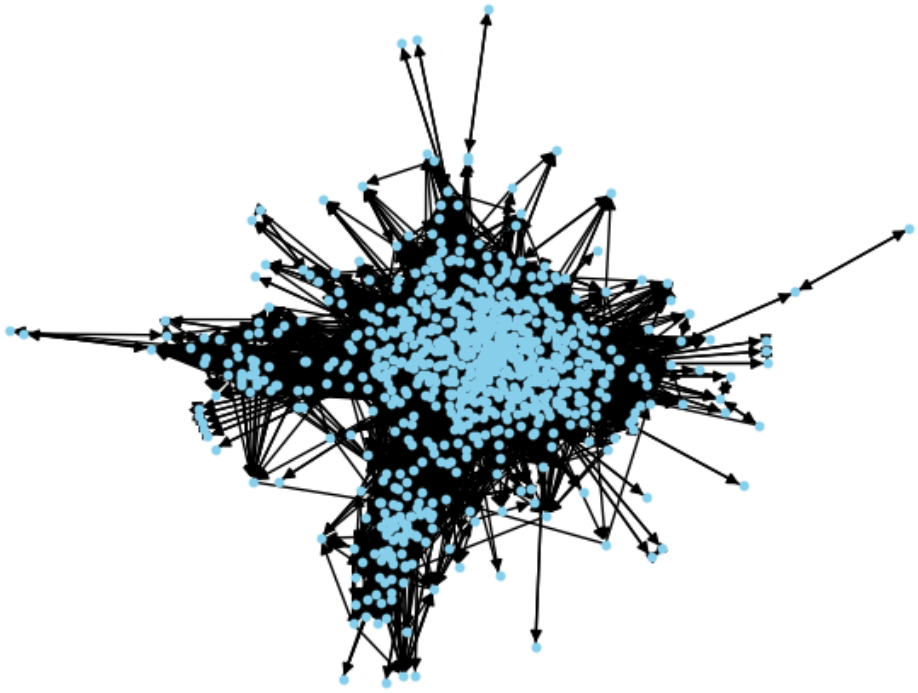
```
/tmp/ipykernel_3256/4192940284.py:19: RuntimeWarning: divide by zero encountered
in log10
  adj_matrix[:,2] = np.log10(adj_matrix[:,2])
```



[10]:
```
adj_plot_of_quants('./project-networks/strong_data.csv')
```



[11]:
```
circular_plot_of_adj('./project-networks/strong_data.csv')
```



2.5 Converting email network to a trust network.

The following section relies heavily on work completed previously on modelling and creating email trust networks (EMT). More specifically, this section utilised work from Dijiang Huang, and Vetri Asaran in Email-based Social Network Trust further expanded on by Dijiang Huang and others in Establishing Email-based Social Network Trust for Vehicular Networks.

2.5.1 Tier-1 Trust Checking

Tier one trust checking is tried when a user knows and has prior communication with the email account they have recieved communication from. This trust checking is based solely on the amount of outgoing and incoming emails between the two accounts. This is the primary reason why we require a directed graph rather than just an undirected graph of email communications. To model this trust network, we use the following equations:

$$\gamma_{ij} = \frac{2}{\frac{Lij}{Lji} + \frac{Lji}{Lij}} \cdot \frac{Lij}{Lij + Lji}\beta.$$

$$T_{ij} = [\log_2 (N_{ij} + 2)]^{\gamma_{ij}}$$

Where L_{ij} is the number of emails sent from i to j, $\beta = 1/0.553$, and N_{ij} is the total number of emails sent between i and j (that is $N_{ij} = L_{ij} + L_{ji}$).

2.5.2 Remodelling the network using T-trust checking

[12]:
```
def build_trust_network(fileName:str = './project-networks/strong_data.csv',␣
↪verb=False):
    df = pd.read_csv(fileName, header=None, names=['sender', 'receiver',␣
↪'quantity'])
    G = nx.from_pandas_edgelist(df, source='sender', target='receiver',␣
↪edge_attr='quantity', create_using=nx.DiGraph)
    nodeList = list(G)
    rowsList = []

    for i, iNodeID in enumerate(nodeList):
        for jNodeID in nodeList[i+1:]:
            fail = 0
            oTrust = -1.0
            iTrust = -1.0
            try:
                outEdges = G.edges[iNodeID, jNodeID]
            except KeyError:
                fail += 1
                oTrust = 0.0
            try:
                inEdges = G.edges[jNodeID, iNodeID]
            except KeyError:
                fail += 1
                iTrust = 0.0

            if fail == 2:
                continue
            if fail == 1:
                if oTrust < 0.0:
                    oTrust = find_trust(outEdges['quantity'], 1)
                    iTrust = find_trust(1, outEdges['quantity'])
                else:
                    oTrust = find_trust(1, inEdges['quantity'])
                    iTrust = find_trust(inEdges['quantity'], 1)
            else:
                oTrust = find_trust(outEdges['quantity'], inEdges['quantity'])
                iTrust = find_trust(inEdges['quantity'], outEdges['quantity'])

            dict0 = {}
            dict0['sender']=int(iNodeID)
            dict0['receiver']=int(jNodeID)
            dict0['trust']=oTrust-1
```

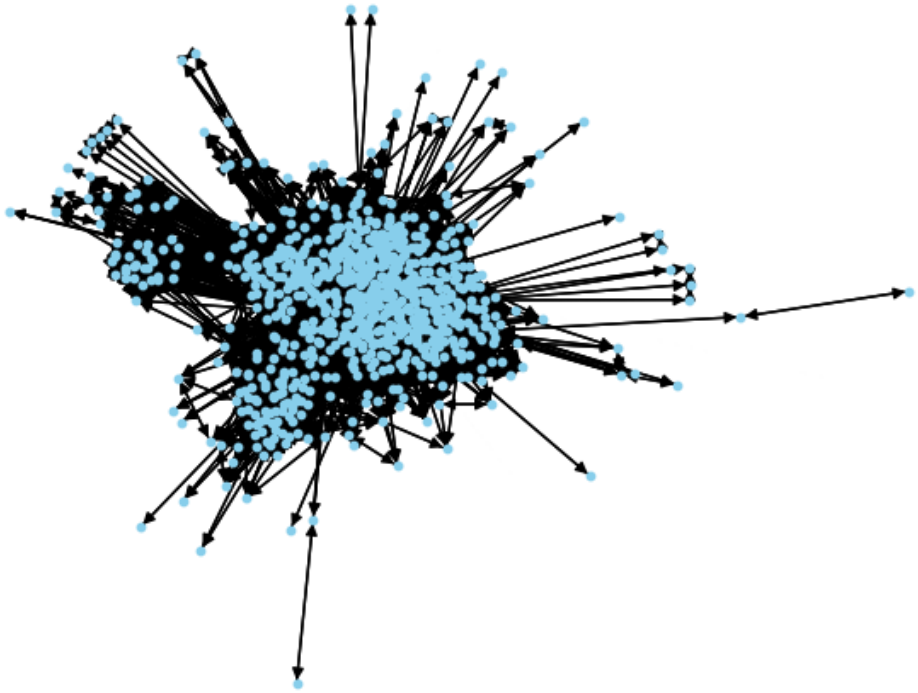
```
            dictI = {}
            dictI['sender']=int(jNodeID)
            dictI['receiver']=int(iNodeID)
            dictI['trust']=iTrust-1

            rowsList.append(dictO.copy())
            rowsList.append(dictI.copy())

    rowsList.sort(reverse=False, key=lambda x: x['sender'])
    trustDF = pd.DataFrame(rowsList)
    trustDF.to_csv("./project-networks/trust_network.csv", encoding="utf-8",␣
 ↪index=False)
    if verb:
        inpri("Trust Network created")

def find_trust(quantIn : int, quantOut : int):
    n = quantIn + quantOut
    gamma = 2/((quantIn/quantOut) + (quantOut/quantIn)) * quantOut/n * 1/0.553
    trust = np.float_power([np.log2([n+2])], [gamma])[0]
    return trust[0]

build_trust_network()
```

2.5.3 Converting to likelihood of infection accross connections

[13]:
```
def find_likelihood_infection(trust:float,defSRate:float,trustCoef:float):
    # Infection-Rate = 1 - Non-infection rate = 1 -␣
 ↪(survivalRate-trustCoef*trust)
    # *Trust, or rather, someone hijacking trust, makes it more likely to␣
 ↪transfer a virus*

    # This allows both the survival rate (general training) and trust factor␣
 ↪(social exploitability)
    # to be individually controlled. Producing more realistic simulations

    # Bind to range 0-1
    return max(0, min(1, 1-(defSRate-trustCoef*trust)))

def create_infection_network(fileName:str = './project-networks/trust_network.
 ↪csv',
                             outPath:str='./project-networks/
 ↪transmission_network.csv',
                             nodeDF:pd.DataFrame=None,
                             stdSRate:float=0.42,
                             stdTC:float=0.1,
                             verb=False):
```

```
    df = pd.read_csv(fileName)

    if nodeDF is None:
        for i, row in df.iterrows():
            infectionRate =␣
 ↪find_likelihood_infection(row['trust'],stdSRate,stdTC)
            df.at[i, 'trust'] = infectionRate

    else:
        for i, row in df.iterrows():
            ndetails = nodeDF.loc[int(row['sender'])]

            infectionRate = find_likelihood_infection(row['trust'],
                                                      ndetails['SurvivalRate'],
                                                      ndetails['TrustCoef'])
            df.at[i, 'trust'] = infectionRate

    df.rename(columns = {'trust':"transmissionRate"}, inplace=True)
    df.to_csv(outPath, header=None, index=False)

    if nodeDF is None:
        create_node_df(stdSRate=stdSRate, stdTC=stdTC)
    if verb:
        inpri(f"Infection Network created at {outPath}                                    ␣
 ↪ ")

def create_node_df(fileName:str="./project-networks/transmission_network.csv",␣
 ↪stdSRate=0.42, stdTC=0.1):
    df = pd.read_csv(fileName, header=None, names=("sender", "receiver", "tr"))
    G = nx.from_pandas_edgelist(df, source='sender', target='receiver',␣
 ↪create_using=nx.DiGraph)

    nodeList = []
    for n in list(G):
        dict0 = {}
        dict0["NodeID"] = n
        dict0["SurvivalRate"]=stdSRate
        dict0["TrustCoef"]=stdTC
        nodeList.append(dict0.copy())

    nodeList.sort(reverse=False, key=lambda x: x['NodeID'])
    trustDF = pd.DataFrame(nodeList)
    trustDF.to_csv("./node-database/node_status.csv", encoding="utf-8",␣
 ↪index=False)

create_infection_network()
```

[14]:
```
circular_plot_of_adj("./project-networks/transmission_network.csv")
```



2.5.4 Determining Training Simulations

A 2012 Study showed that training occurring over 2 months reduced the infection rate to 24.5%. However, this data may be out of date, as the 2023 Gone Phishing Tournement found an overall click-rate of 10.4%. This data may be biased as companies have to opt-in to the tournament, meaning the most susceptible companies are not evaluated. If this is the case, the 10.4% value is a good representation of a continually tested population of users, and so, when training is completed, the node's survival rate should be at this value.

A more impactful measure would be the **decay** of a user's survival rate. The NoPhish study found that users can retain anti-phishing training for up to 5 months after each session. For this program, we shall presume this is the case for all participants equally. Persistent training Studies found that an untrained user has an average click rate of 58%, meaning in this case, they would be infected 58% of the time from a non-trustworthy user. This is exceptionally high and further incentivises the need for further anti-phishing / malware awareness courses.

With the regularity of training measures, we shall presume that there is an opportunity to train users every two months. This number is arbitrary; however, it complies with the suggestions from NoPhish and other knowledge retention studies included in this comprehensive literature review published in 2020.

[15]:
```python
def find_train_node(sr:float):
    # No need to update edges just yet as that can be completed in the final␣
 ↪step
    # When we create the final network

    return max(0.9, 1-(1-sr)*0.40)

def find_decay_node(sr:float, timeMonths:int=1):
    # after 5 months, the knowledge is lost, and the rate becomes ~0.42 again
    # 0.895 -> 0.42. 0.895 * (x)^5 = 0.42, x = 0.860
    # 14% decrease per month.
    return max(0.420, sr*(0.860**timeMonths))
```

[16]:
```python
def standard_training(nwPath:str, nodeDBPath:str, months:int=12, recurrence:
 ↪int=1, seed=0, verb=False):
    nwDF = pd.read_csv(nwPath, header=None, names=("sender", "receiver", "tr"))
    nodeDB = pd.read_csv(nodeDBPath)
    nodeDB.set_index('NodeID', inplace=True)

    if seed == 0:
        rng = np.random.default_rng()
    else:
        rng = np.random.default_rng(seed)

    for session in range(int(months/recurrence)):
        if verb:
            inpri(f"Starting standard training session {session+1} of␣
 ↪{int(months/recurrence)}")
        for i, row in nodeDB.iterrows():
            if rng.random() > row['SurvivalRate']:
                nodeDB.loc[i, 'SurvivalRate'] =␣
 ↪find_train_node(row['SurvivalRate'])
            else:
                nodeDB.loc[i, 'SurvivalRate'] =␣
 ↪find_decay_node(row['SurvivalRate'], recurrence)

    if verb:
        inpri("Simulation complete")
        inpri("Exporting trained node DB to ./node-database/std_trained.csv     ␣
 ↪     ")
    nodeDB.to_csv("./node-database/std_trained.csv", encoding="utf-8",␣
 ↪index=False)

    if verb:
        inpri("Export complete!")
        inpri("Generating new trained network")
```

```
    create_infection_network(nwPath, './project-networks/std.csv', nodeDB)
    return './project-networks/std.csv'
```

[17]:
```
standard_training("./project-networks/trust_network.csv",
                  "./node-database/node_status.csv")
```

[17]:
```
'./project-networks/std.csv'
```

[18]:
```
def special_training(nwPath:str, nodeDBPath:str,func:tuple, months:int=12,␣
↪recurrence:int=1,
                     generalTestRec:int=4, seed=0, verb=False, rev=True):
    nwDF = pd.read_csv(nwPath)
    for i, row in nwDF.iterrows():
        # Inverse the trust as higher trust means easier infection rate / lower␣
↪distance between nodes
        nwDF.loc[i, 'trust'] = 10-float(row['trust'])

    trainFunc = func[0]
    funcName = func[1]

    G = nx.from_pandas_edgelist(nwDF, source='sender', target='receiver',␣
↪edge_attr='trust', create_using=nx.DiGraph)
    centralityList = [k for k, _ in sorted(trainFunc(G).items(), key=lambda x:␣
↪x[1], reverse=(True != rev))]

    nodeDB = pd.read_csv(nodeDBPath)
    nodeDB.set_index('NodeID', inplace=True)

    typeFunc = func[0]
    funcName = func[1]
    if seed == 0:
        rng = np.random.default_rng()
    else:
        rng = np.random.default_rng(seed)
    for session in range(int(months/recurrence)):
        if session % generalTestRec == 0:
            if verb:
                inpri(f"Starting standard training session {session+1} of␣
↪{int(months/recurrence)}")
            for i, row in nodeDB.iterrows():
                if rng.random() > row['SurvivalRate']:
                    nodeDB.loc[i, 'SurvivalRate'] =␣
↪find_train_node(row['SurvivalRate'])
                else:
                    nodeDB.loc[i, 'SurvivalRate'] =␣
↪find_decay_node(row['SurvivalRate'], recurrence)
```

```
        if verb:
            inpri(f"Starting {funcName} training session {session+1} of␣
↪{int(months/recurrence)}")
        nodesToTrain = centralityList[(session%generalTestRec-1)*200:
↪(session%generalTestRec)*200]
        for i, row in nodeDB.iterrows():
            if i in nodesToTrain:
                nodeDB.loc[i, 'SurvivalRate'] =␣
↪find_train_node(row['SurvivalRate'])
            else:
                nodeDB.loc[i, 'SurvivalRate'] =␣
↪find_decay_node(row['SurvivalRate'], recurrence)

    if verb:
        sys.stdout.write("\rDoing thing %i" % i)
        sys.stdout.flush()

        inpri("Simulation complete")
        inpri(f"Exporting trained node DB to ./node-database/{funcName}.csv")

    nodeDB.to_csv(f"./node-database/{funcName}.csv", encoding="utf-8",␣
↪index=False)

    if verb:
        inpri("Export complete!")
        inpri("Generating new trained network")

    create_infection_network(nwPath, f'./project-networks/{funcName}.csv',␣
↪nodeDB)

    return f'./project-networks/{funcName}.csv'

special_training("./project-networks/trust_network.csv",
                 "./node-database/node_status.csv",
                [nx.closeness_centrality, "close"])
```

[18]:
```
'./project-networks/close.csv'
```

2.6 Simulation

Now we have generated all of the infection networks, we can import these into the simulation in order to test their selection methods.

[19]:
```
def run_sim(network:str, randSeed:int=None, preInfected=None, startInfected:
↪int=10,
            recoverTime:int = 4, maxTime = 50):
    if randSeed is not None:
```

```
        rng = np.random.default_rng(randSeed)
    else:
        rng = np.random.default_rng()

    df = pd.read_csv(network, header=None, names=["s", "r", "inf"])
    G = nx.from_pandas_edgelist(df, source='s', target='r', edge_attr='inf',␣
↪create_using=nx.DiGraph)

    df.set_index(["s", "r"], inplace=True)
    safeNodes = list(G) # Nodes not exposed to the malware and not infected
    exposedNodes = [] # Nodes able to be infected by the malware
    infectedNodes = [] # Nodes infected by the malware
    infectedTimes = []
    recoveredNodes = [] # Nodes recovered [immune] from the malware

    transferEdges = [] # Edges where the malware can be transfered.
    if preInfected is None:
        for _ in range(startInfected):
            node = rng.choice(safeNodes, 1).tolist()[0]
            safeNodes.remove(node)
            infectedNodes.append(node)
            infectedTimes.append(1)
    else:
        for node in preInfected:
            safeNodes.remove(node)
            infectedNodes.append(node)
            infectedTimes.append(1)

    infectedEdges = []
    for e in G.edges(infectedNodes):
        if e[1] in safeNodes or e[1] in exposedNodes:
            infectedEdges.append(e)
            if e[1] not in exposedNodes:
                exposedNodes.append(e[1])
                safeNodes.remove(e[1])

    history = {'safe':[],'exp':[],'inf':[],'rec':[]}
    cycle = 0
    while cycle < maxTime:
        newInfNodes = []
        newInfTimes = []

        history['safe'].append(len(safeNodes))
        history['exp'].append(len(exposedNodes))
        history['inf'].append(len(infectedNodes))
        history['rec'].append(len(recoveredNodes))
        # Spread Across Network
```

```
        for e in infectedEdges:
            if e[1] in infectedNodes:
                continue
            if rng.random() < 0.02 and rng.random() > df.loc[(e[0], e[1]),␣
↪'inf']:
                infectedNodes.append(e[1])
                infectedTimes.append(0)
                exposedNodes.remove(e[1])

        # Remove recovered nodes
        for node, time in zip(infectedNodes, infectedTimes):
            if time >= recoverTime:
                recoveredNodes.append(node)
                continue
            newInfNodes.append(node)
            newInfTimes.append(time+1)

        infectedNodes = newInfNodes
        infectedTimes = newInfTimes

        # Update infected Edges
        infectedEdges = []
        for e in G.edges(infectedNodes):
            if e[1] in safeNodes or e[1] in exposedNodes:
                infectedEdges.append(e)
                if e[1] not in exposedNodes:
                    exposedNodes.append(e[1])
                    safeNodes.remove(e[1])

        cycle+=1
    return history
```

2.7 Now our sim is up and running, let's do some tests

Due to the randomness of the spread throughout the data, we need to complete multiple runs over the same data to determine which methodology was the most successful.

[20]:
```
def test_methods(tests:int=10, resetTraining:int=5):
    # As the first step of the rng to select the pre infected nodes is␣
↪identical,
    # we don't need to assign them here.

    MAXTIME = 30

    # Where results will be stored
    stdRes = initResDict(MAXTIME)
    cloRes = initResDict(MAXTIME)
```

```
    degRes = initResDict(MAXTIME)
    eigRes = initResDict(MAXTIME)
    stdResI = initResDict(MAXTIME)
    cloResI = initResDict(MAXTIME)
    degResI = initResDict(MAXTIME)
    eigResI = initResDict(MAXTIME)

    # Values for statistical Tests
    markerLayout = [10, 25, 50, "peak", "final"]
    stdMarker = [[],[],[],[],[]]
    degMarker = [[],[],[],[],[]]
    cloMarker = [[],[],[],[],[]]
    eigMarker = [[],[],[],[],[]]
    degIMarker = [[],[],[],[],[]]
    cloIMarker = [[],[],[],[],[]]
    eigIMarker = [[],[],[],[],[]]

    tn = "./project-networks/trust_network.csv"
    ndb = "./node-database/node_status.csv"

    deg = [nx.degree_centrality, "deg"]
    clo = [nx.closeness_centrality, "clo"]
    eig = [nx.eigenvector_centrality, "eig"]
    degI = [nx.degree_centrality, "degI"]
    cloI = [nx.closeness_centrality, "cloI"]
    eigI = [nx.eigenvector_centrality, "eigI"]

    for i in range(tests):
        if i % resetTraining == 0:
            inpri(f"[{i+1}/{tests}] Resetting training networks... { ' ' * 50␣
↪}")
            stdNet = standard_training(tn, ndb, seed=i)
            degNet = special_training(tn,ndb,deg,seed=i,rev=False)
            cloNet = special_training(tn,ndb,clo,seed=i,rev=False)
            eigNet = special_training(tn,ndb,eig,seed=i,rev=False)
            degNetI = special_training(tn,ndb,degI,seed=i,rev=True)
            cloNetI = special_training(tn,ndb,cloI,seed=i,rev=True)
            eigNetI = special_training(tn,ndb,eigI,seed=i,rev=True)

        inpri(f'[{i+1}/{tests}] Running simulations... { " " * 50 }')

        stdResSingle = run_sim(stdNet, i, maxTime=MAXTIME)
        degResSingle = run_sim(degNet, i, maxTime=MAXTIME)
        cloResSingle = run_sim(cloNet, i, maxTime=MAXTIME)
        eigResSingle = run_sim(eigNet, i, maxTime=MAXTIME)
        degResISingle = run_sim(degNetI, i, maxTime=MAXTIME)
        cloResISingle = run_sim(cloNetI, i, maxTime=MAXTIME)
```

```
        eigResISingle = run_sim(eigNetI, i, maxTime=MAXTIME)

        inpri(f'[{i+1}/{tests}] Collating results... { " " * 50 }')

        stdMarker = getMarkers(stdResSingle, stdMarker)
        degMarker = getMarkers(degResSingle, degMarker)
        cloMarker = getMarkers(cloResSingle, cloMarker)
        eigMarker = getMarkers(eigResSingle, eigMarker)
        degIMarker = getMarkers(degResISingle, degIMarker)
        cloIMarker = getMarkers(cloResISingle, cloIMarker)
        eigIMarker = getMarkers(eigResISingle, eigIMarker)

        stdRes=combineResultDict(stdResSingle, stdRes)
        degRes=combineResultDict(degResSingle, degRes)
        cloRes=combineResultDict(cloResSingle, cloRes)
        eigRes=combineResultDict(eigResSingle, eigRes)
        degResI=combineResultDict(degResISingle, degResI)
        cloResI=combineResultDict(cloResISingle, cloResI)
        eigResI=combineResultDict(eigResISingle, eigResI)

    inpri(f"Results recorded and completed. Exporting Now...{ ' '*20 }\n")
    results = { "Averages" : {
            "Standard" : normResultDict(stdRes, tests),
            "Degree" : normResultDict(degRes, tests),
            "Closeness" : normResultDict(cloRes, tests),
            "Eigenvector": normResultDict(eigRes, tests),
            "Degree Inverse" : normResultDict(degResI, tests),
            "Closeness Inverse" : normResultDict(cloResI, tests),
            "Eigenvector Inverse": normResultDict(eigResI, tests)
            },
        "10%" : {
            "Standard" : stdMarker[0],
            "Degree" : degMarker[0],
            "Closeness" : cloMarker[0],
            "Eigenvector": eigMarker[0],
            "Degree Inverse" : degIMarker[0],
            "Closeness Inverse" : cloIMarker[0],
            "Eigenvector Inverse": eigIMarker[0]
        },
        "25%" : {
            "Standard" : stdMarker[1],
            "Degree" : degMarker[1],
            "Closeness" : cloMarker[1],
            "Eigenvector": eigMarker[1],
            "Degree Inverse" : degIMarker[1],
            "Closeness Inverse" : cloIMarker[1],
            "Eigenvector Inverse": eigIMarker[1]
```

```
        },
        "50%" : {
            "Standard" : stdMarker[2],
            "Degree" : degMarker[2],
            "Closeness" : cloMarker[2],
            "Eigenvector": eigMarker[2],
            "Degree Inverse" : degIMarker[2],
            "Closeness Inverse" : cloIMarker[2],
            "Eigenvector Inverse": eigIMarker[2]
        },
        "Peak" : {
            "Standard" : stdMarker[3],
            "Degree" : degMarker[3],
            "Closeness" : cloMarker[3],
            "Eigenvector": eigMarker[3],
            "Degree Inverse" : degIMarker[3],
            "Closeness Inverse" : cloIMarker[3],
            "Eigenvector Inverse": eigIMarker[3]
        },
        "Final" : {
            "Standard" : stdMarker[4],
            "Degree" : degMarker[4],
            "Closeness" : cloMarker[4],
            "Eigenvector": eigMarker[4],
            "Degree Inverse" : degIMarker[4],
            "Closeness Inverse" : cloIMarker[4],
            "Eigenvector Inverse": eigIMarker[4]
        }
    }
    f = open(f"./results/results_{tests}.json", "w", encoding="utf-8")

    f.write(json.dumps(results, indent=4))
    f.close()

    inpri(f"Simulations Completed! { ' ' * 50 }")

def initResDict(maxTime):
    res = {}
    res['safe'] = [0 for _ in range(maxTime)]
    res['exp'] = [0 for _ in range(maxTime)]
    res['inf'] = [0 for _ in range(maxTime)]
    res['rec'] = [0 for _ in range(maxTime)]

    return res

def combineResultDict(resDict:dict, mainDict:dict):
    for i in range(len(mainDict['safe'])):
```

```
        mainDict['safe'][i] += resDict['safe'][i]
        mainDict['exp'][i] += resDict['exp'][i]
        mainDict['inf'][i] += resDict['inf'][i]
        mainDict['rec'][i] += resDict['rec'][i]
    return mainDict

def normResultDict(mainDict:dict, runs:int):
    for i in range(len(mainDict['safe'])):
        mainDict['safe'][i] = int(mainDict['safe'][i]/runs)
        mainDict['exp'][i] = int(mainDict['exp'][i]/runs)
        mainDict['inf'][i] = int(mainDict['inf'][i]/runs)
        mainDict['rec'][i] = int(mainDict['rec'][i]/runs)
    return mainDict

def getMarkers(resultsDict, outputArray):
    tenth = False
    quarter = False
    half = False

    for time, result in enumerate(resultsDict['rec']):
        if not tenth and result > 802/10:
            outputArray[0].append(time-4)
            tenth = True
        if not quarter and result > 802/4:
            outputArray[1].append(time-4)
            quarter = True
        if not half and result > 802/2:
            half = True
            outputArray[2].append(time-4)

    outputArray[3].append(resultsDict['inf'].index(max(resultsDict['inf'])))
    outputArray[4].append(resultsDict['rec'][-1])

    return outputArray
```

[21]:
```
test_methods(500, 25)
```

```
Results recorded and completed. Exporting Now…
Simulations Completed!
```

[68]:
```
# Just changing font settings
import matplotlib as mpl
rc_fonts = {
    "font.family": "serif",
    "font.size": 20,
    'figure.figsize': (10, 5),
    "text.usetex": True,
    'text.latex.preamble':
```

```
        r"""
        \usepackage{libertine}
        \usepackage[libertine]{newtxmath}
        """,
}
mpl.rcParams.update(rc_fonts)
```

[113]:
```
def graph_data(resultsPath : str):
    # Load results data
    file = open(resultsPath, "r", encoding="utf-8")
    results = json.load(file)

    graph_population_avgs(results)
    graph_notable_times(results)
    graph_finals(results)

    inpri("Done!" + " " * 70)

def graph_finals(results):
    values = results['Final']
    methodNames=list(values.keys())
    pairings=[[0,1,2,3],
              [0,4,5,6],
              [0,1,4],
              [0,2,5],
              [0,3,6]]

    for group in pairings:
        for i in range(len(group)):
            group[i]=methodNames[group[i]]

    for pair in pairings:
        graph_final(values, pair)

def graph_final(values, pair):
    # Gets the total infected for each sim...
    inpri(f"Plotting results for final infected. { ' ' * 22}")

    #Get minimum:
    xMin = 9999

    for method in pair:
        if min(values[method]) < xMin:
            xMin = min(values[method])
    #Get Maximum
    xMax = 0
    for method in pair:
```

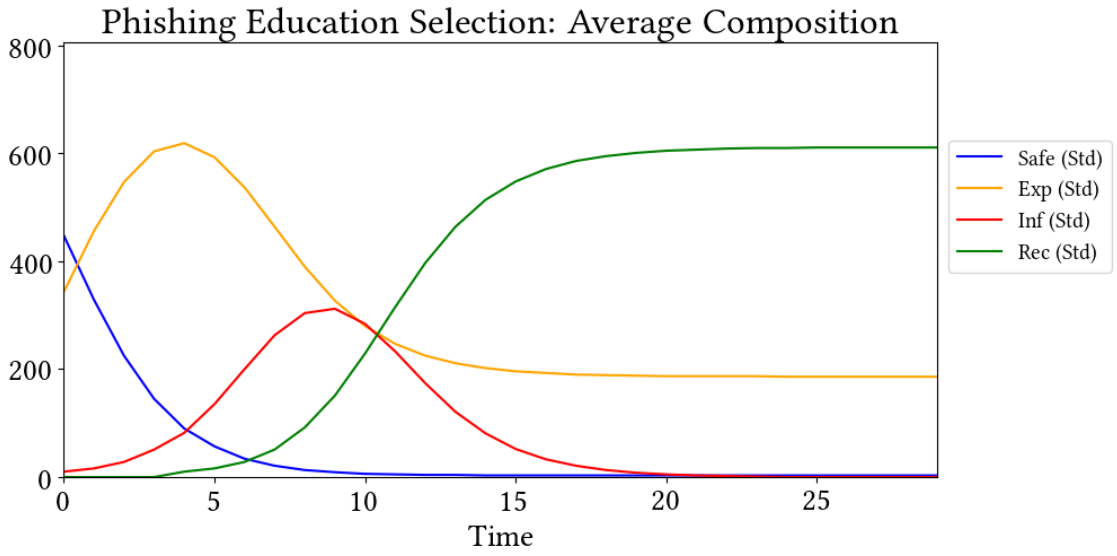
```
        if max(values[method]) > xMax:
            xMax = max(values[method])

    xAxis = list(range(xMax+1))
    plottingResults= [[0 for _ in xAxis] for _ in pair]

    fig = plt.figure()
    ax = plt.subplot(111)

    for i, method in enumerate(pair):
        for val in values[method]:
            plottingResults[i][val] += 1

    styleList = ["--", "-.", "-", ":"]
    styleI = 0
    fileName = ""
    for tag, data in zip(pair, plottingResults):
        if tag == "Standard":
            key = "Std"
        elif " " in tag:
            key = tag[0:3].capitalize() + " Inv"
        else:
            key = tag[0:3].capitalize()
        ax.plot(xAxis, data, label=f"{key}", linestyle=styleList[styleI])
        styleI += 1
        styleI = styleI % 4
        fileName += key

    fileName = fileName.replace(" ", "-")
    plt.xlim(int(xMin/50)*50,(int(xMax/50)+1)*50)
    plt.ylim(0,40)
    plt.title(f"Phishing Education Selection: Total infected")
    plt.xlabel("Nodes Infected")
    ax.legend(bbox_to_anchor=(1, 0.8), fontsize=14)
    plt.savefig(f'./graphs/Final/{fileName}.svg', bbox_inches="tight")
    plt.show()
    plt.close()

def graph_notable_times(results:dict):
    maxTime = len(results['Averages']['Standard']['safe'])

    for label, values in results.items():
        # Reject these as they are not the correct data
        # for these plots
        if label == "Averages" or label == "Final":
            continue
```

```
        methodNames=list(values.keys())
        pairings=[[0,1,2,3],
                  [0,4,5,6],
                  [0,1,4],
                  [0,2,5],
                  [0,3,6]]

        for group in pairings:
            for i in range(len(group)):
                group[i]=methodNames[group[i]]

        for pair in pairings:
            graph_times(values, label, pair, maxTime)

def graph_times(values, label, pairing, maxTime):
    # Gets the total length of time each sim was running for...
    inpri(f"Plotting results for time to reach {label}. { ' ' * 22}")
    xAxis = range(maxTime)
    plottingResults= [[0 for _ in xAxis] for _ in pairing]

    fig = plt.figure()
    ax = plt.subplot(111)

    for i, method in enumerate(pairing):
        for val in values[method]:
            plottingResults[i][val] += 1

    styleList = ["--", "-.", "-", ":"]
    styleI = 0
    fileName = ""
    for tag, data in zip(pairing, plottingResults):
        if tag == "Standard":
            key = "Std"
        elif " " in tag:
            key = tag[0:3].capitalize() + " Inv"
        else:
            key = tag[0:3].capitalize()
        ax.plot(xAxis, data, label=f"{key}", linestyle=styleList[styleI])
        styleI += 1
        styleI = styleI % 4
        fileName += key

    fileName = fileName.replace(" ", "-")
    plt.xlim(xAxis[0],15)
    plt.ylim(0,400)
    tmp = label.replace('%', '\%')
    plt.title(f"Phishing Education Selection: Time to Infect {tmp}")
```

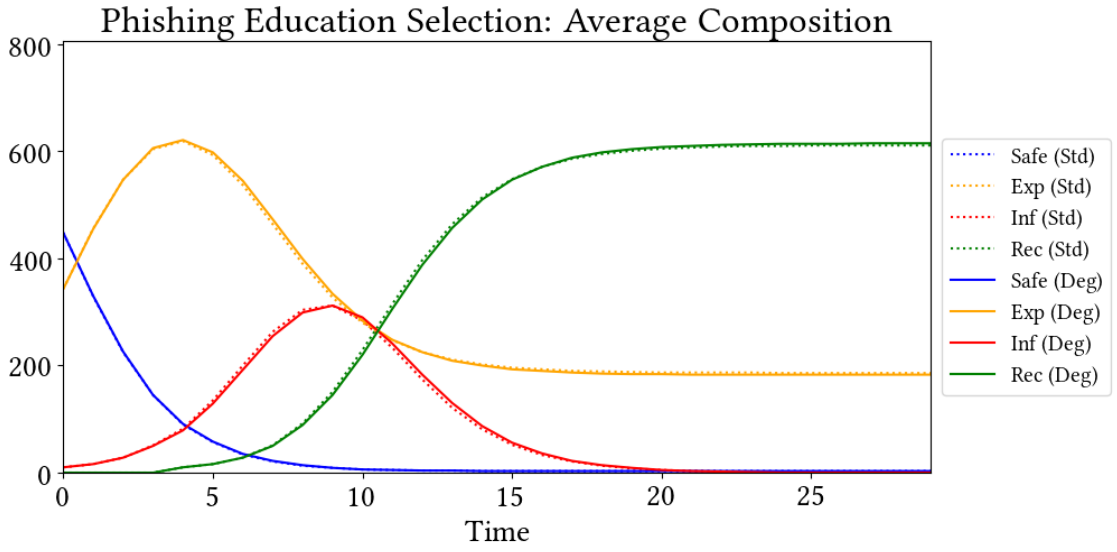
```
    plt.xlabel("Time")
    ax.legend(bbox_to_anchor=(1, 0.8), fontsize=14)
    plt.savefig(f'./graphs/{label}/{fileName}.svg', bbox_inches="tight")
    plt.show()
    plt.close()

def graph_population_avgs(results:dict):
    for selection, values in results['Averages'].items():
        inpri(f"Graphing value received for {selection}...")
        fig = plt.figure()
        ax = plt.subplot(111)

        colors = ["blue", "orange", "red", "green"]
        xAxis = list(range(len(values['safe'])))
        # Plot standard selection method for comparison

        colorIndex = 0
        if selection != "Standard":
            for status, data in results['Averages']['Standard'].items():
                ax.plot(xAxis, data, label=f"{status.capitalize()} (Std)",␣
↪linestyle=":", color=colors[colorIndex])
                colorIndex += 1
                colorIndex = colorIndex % 4

        for status, data in values.items():
            if selection == "Standard":
                key = "Std"
            elif " " in selection:
                key = selection[0:3].capitalize() + " Inv"
            else:
                key = selection[0:3].capitalize()

            ax.plot(xAxis, data, label=f"{status.capitalize()} ({key})",␣
↪color=colors[colorIndex])
            colorIndex += 1
            colorIndex = colorIndex % 4

        plt.xlim(xAxis[0],xAxis[-1])
        plt.ylim(0,806)
        plt.title(f"Phishing Education Selection: Average Composition")
        plt.xlabel("Time")
        ax.legend(bbox_to_anchor=(1, 0.8), fontsize=14)
        plt.savefig(f'./graphs/Averages/{selection.replace(" ", "-")}.svg',␣
↪bbox_inches="tight")
        plt.show()
        plt.close()
```

```
graph_data("./results/results_500.json")
```

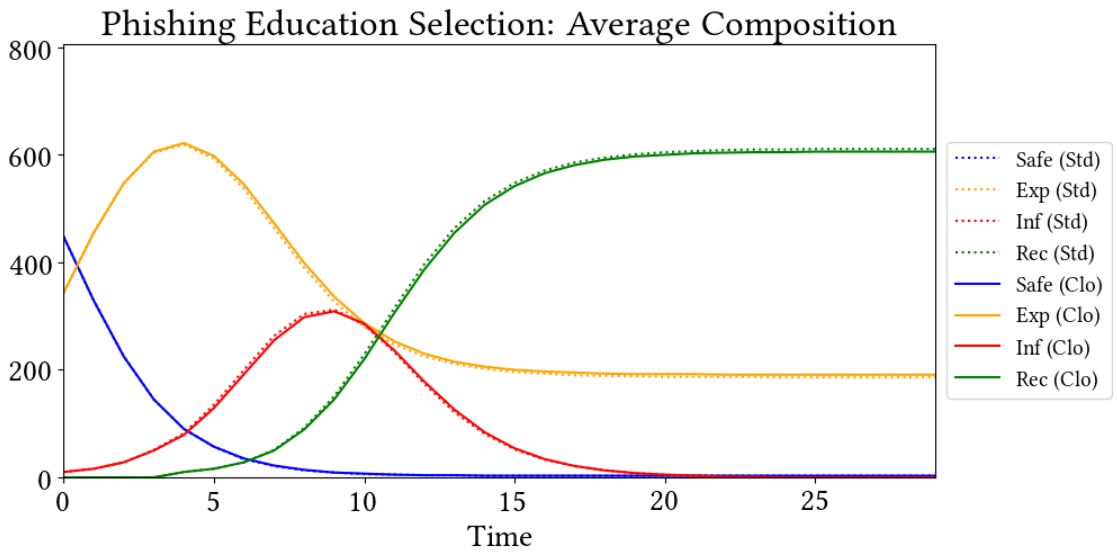
```
Graphing value received for Standard…
```



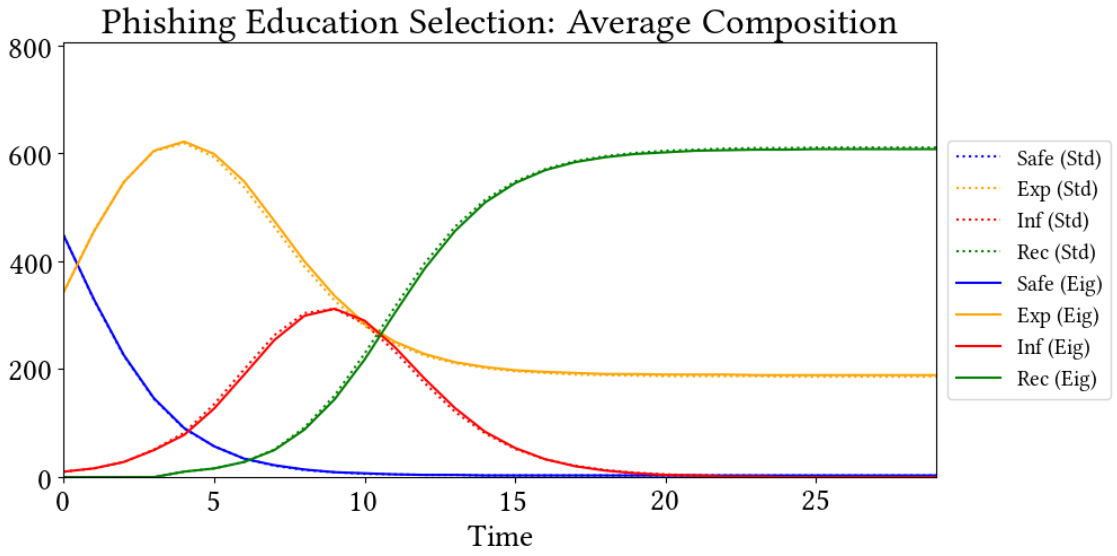

```
Graphing value received for Degree…
```



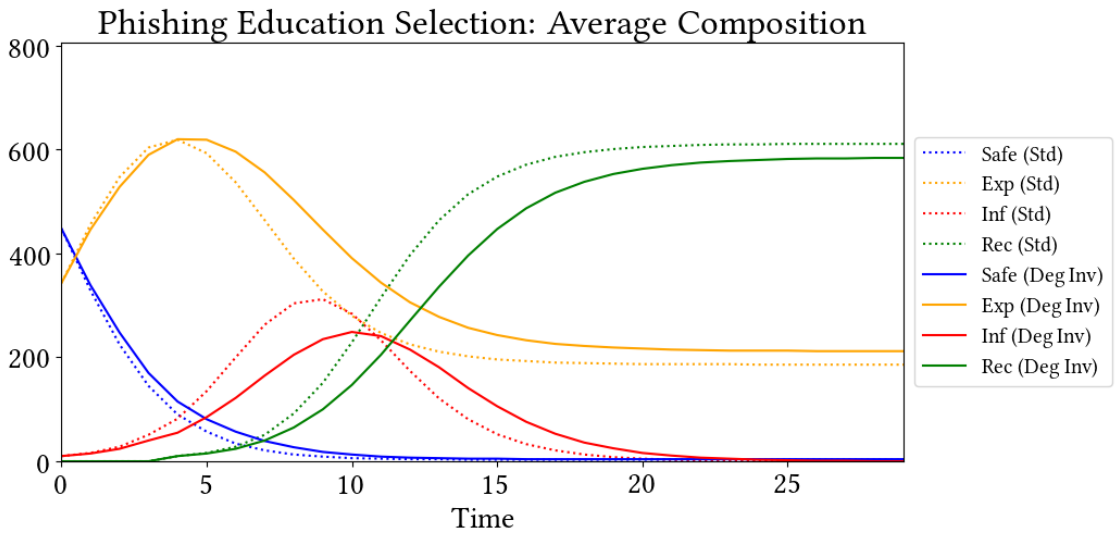

```
Graphing value received for Closeness…
```



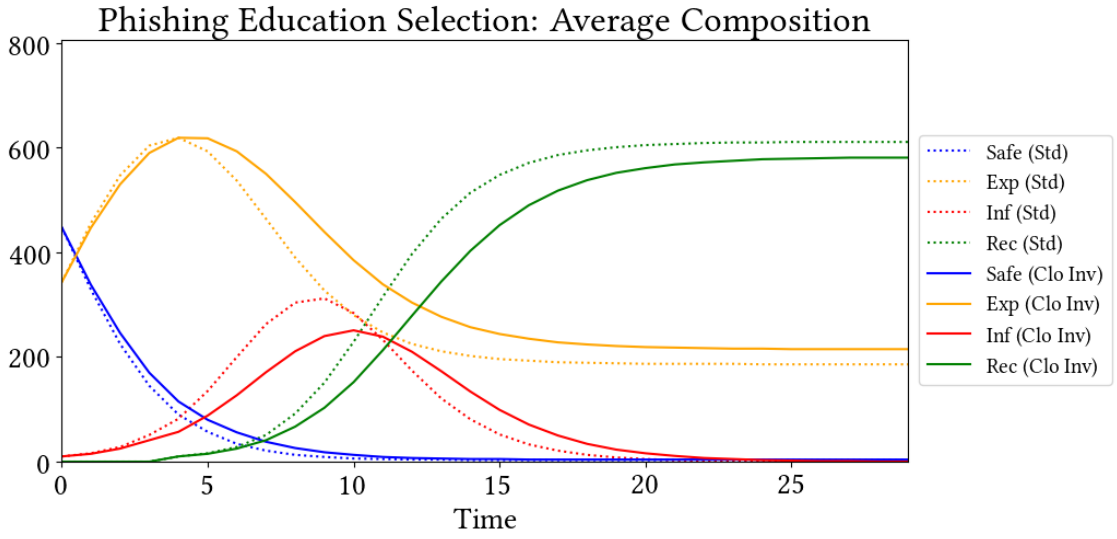

```
Graphing value received for Eigenvector…
```



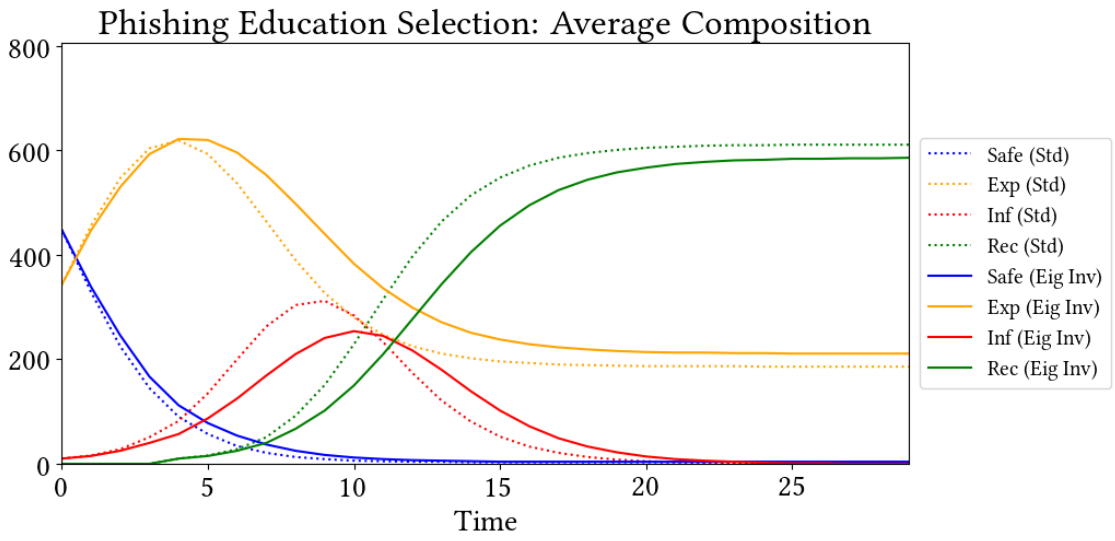

```
Graphing value received for Degree Inverse…
```



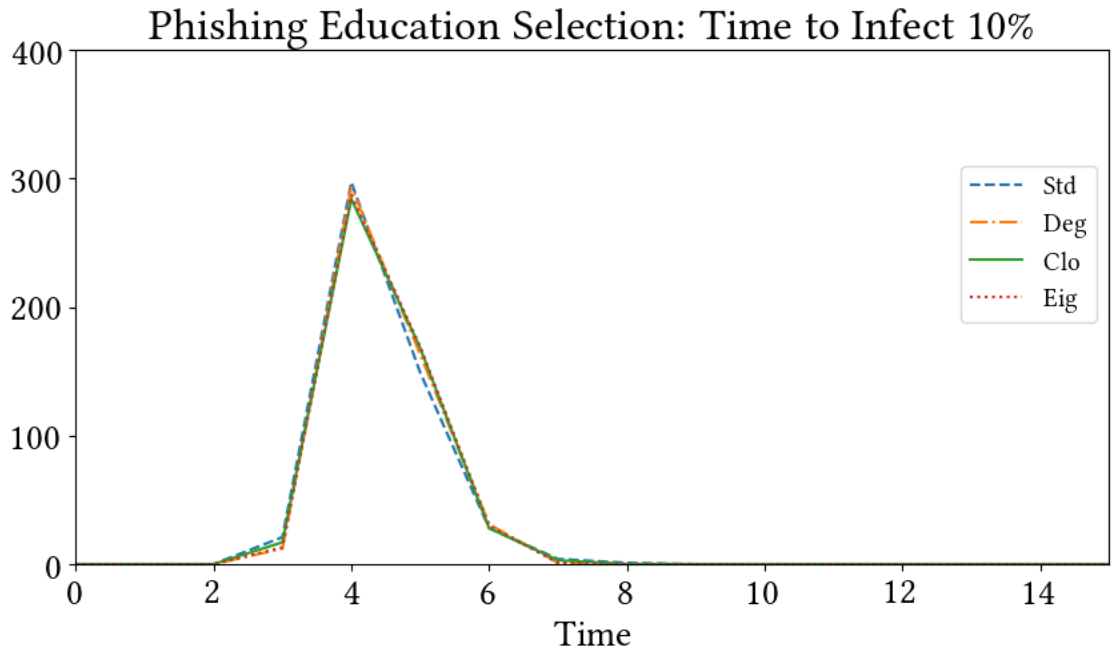

```
Graphing value received for Closeness Inverse…
```



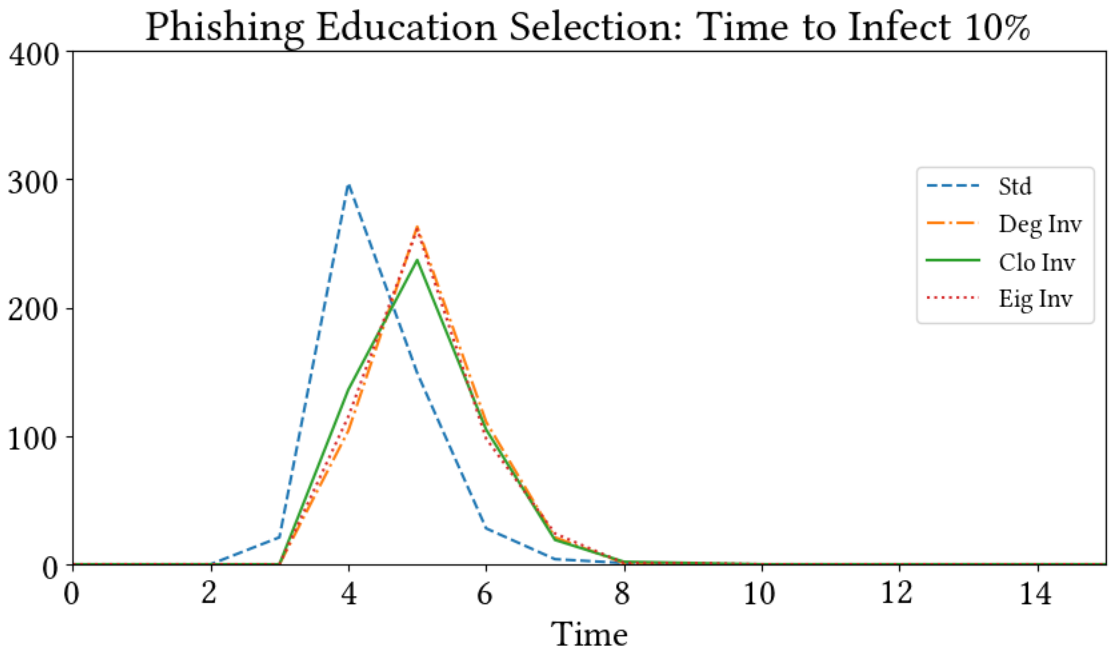

```
Graphing value received for Eigenvector Inverse…
```



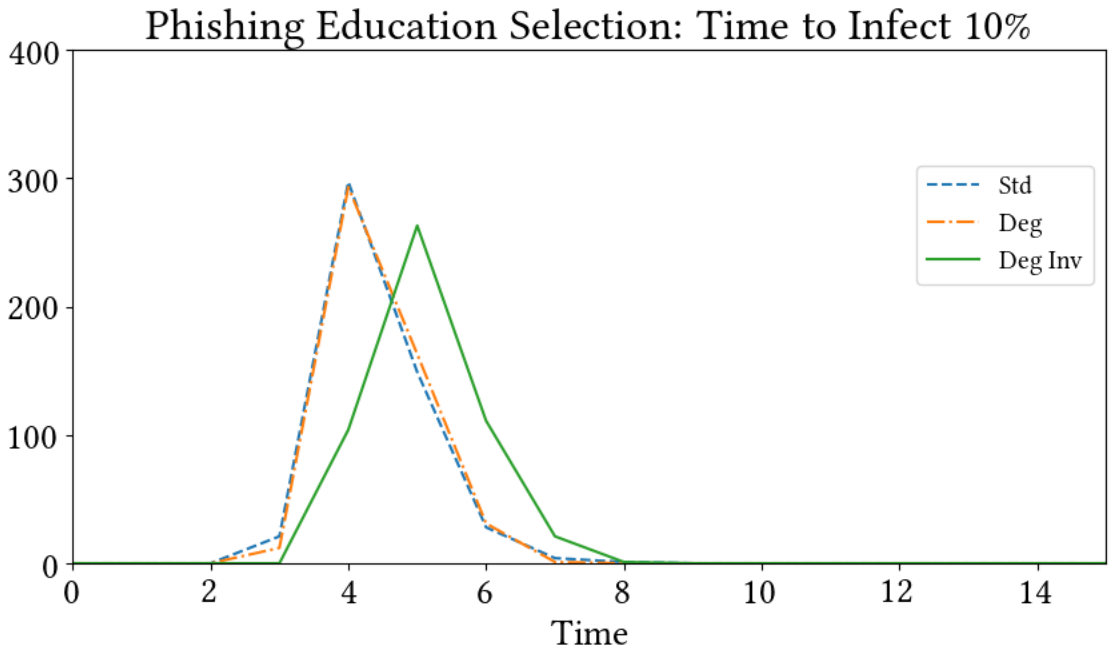

```
Plotting results for time to reach 10%.
```



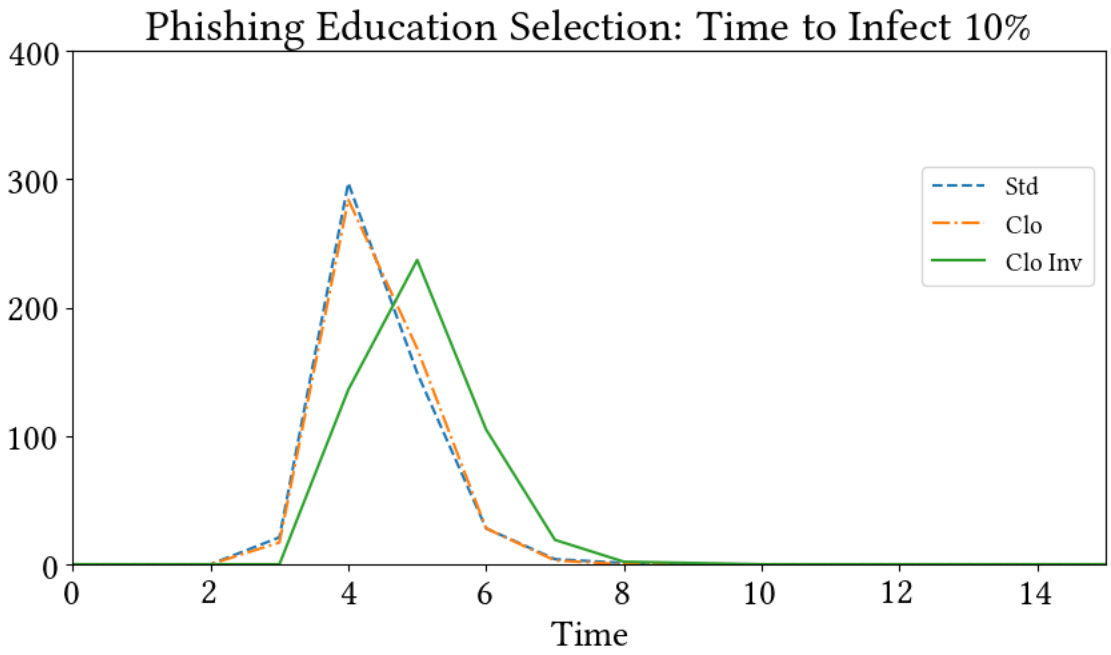

```
Plotting results for time to reach 10%.
```



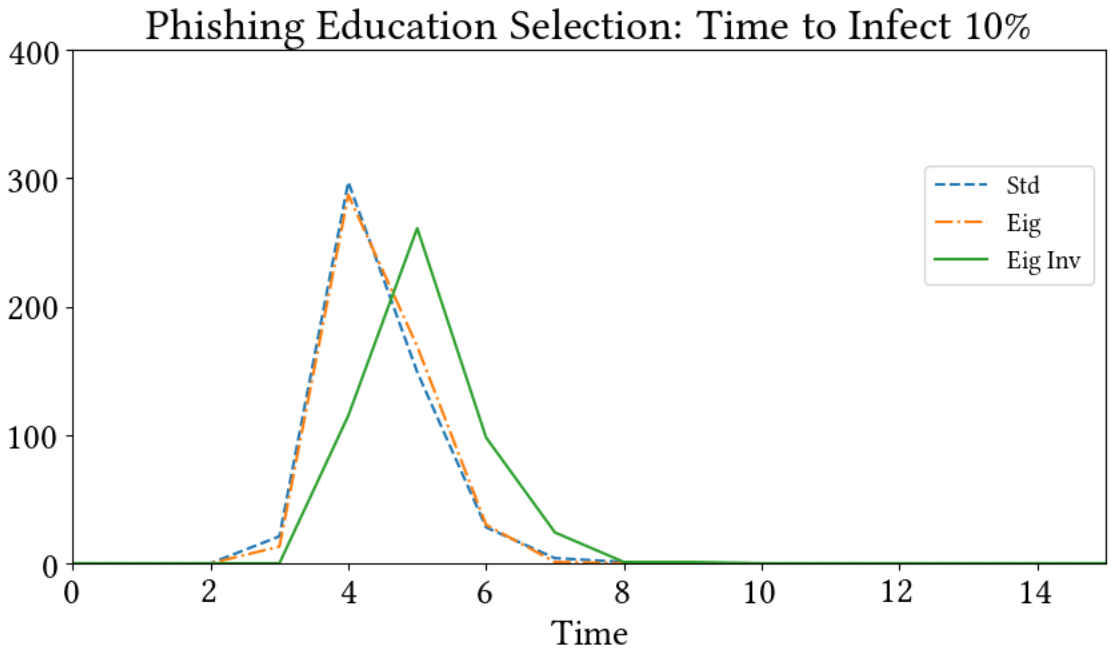

```
Plotting results for time to reach 10%.
```



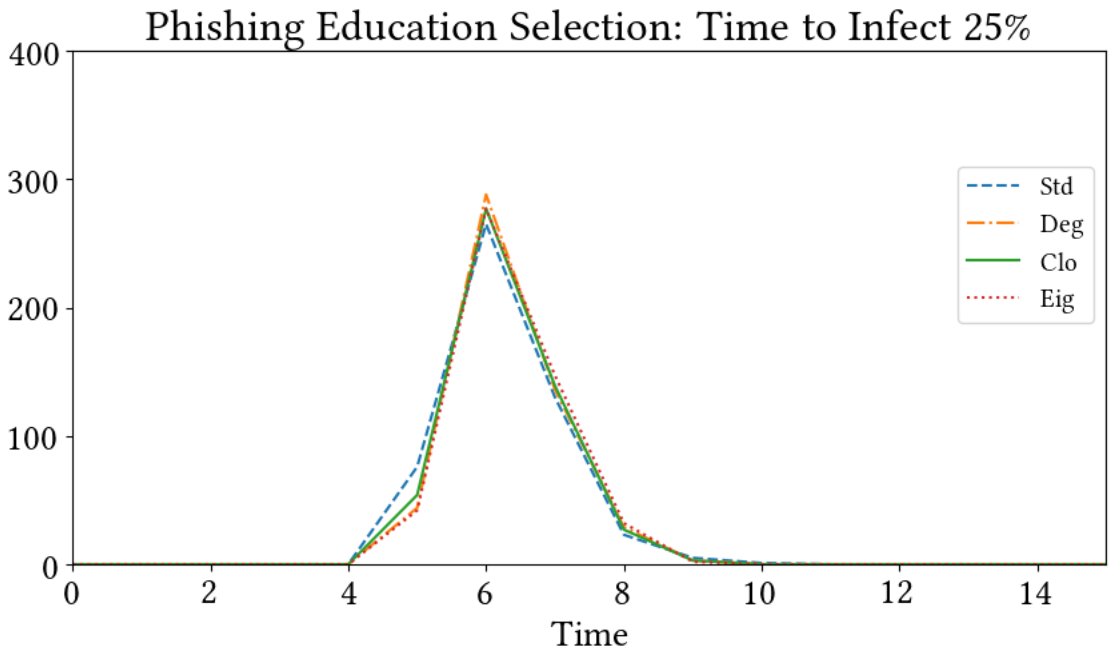

```
Plotting results for time to reach 10%.
```



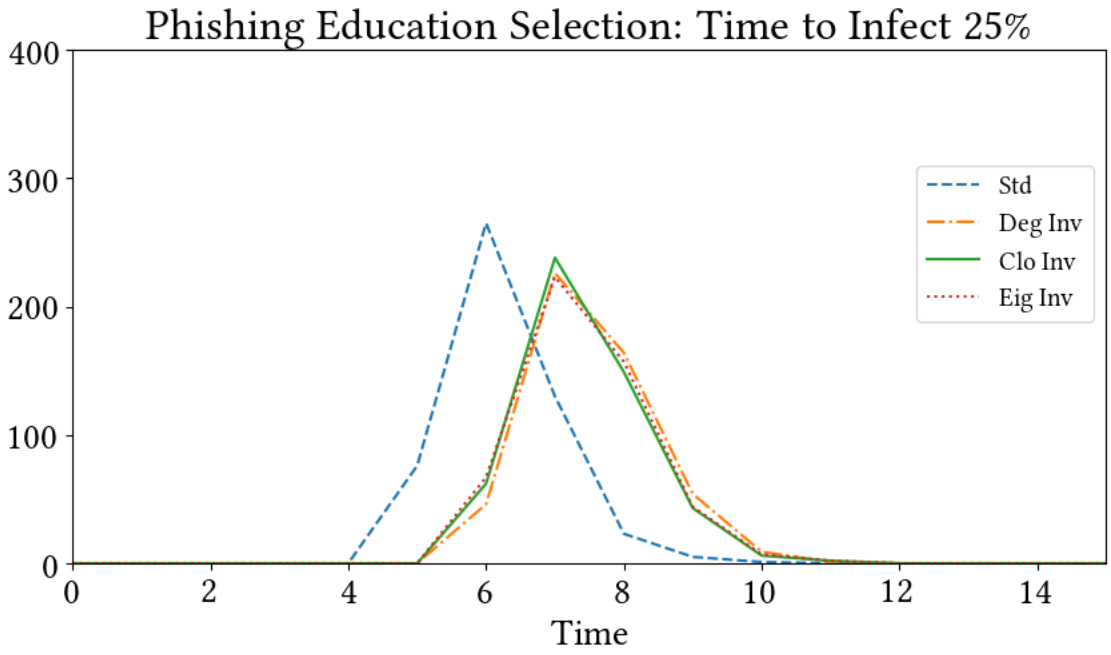

```
Plotting results for time to reach 10%.
```



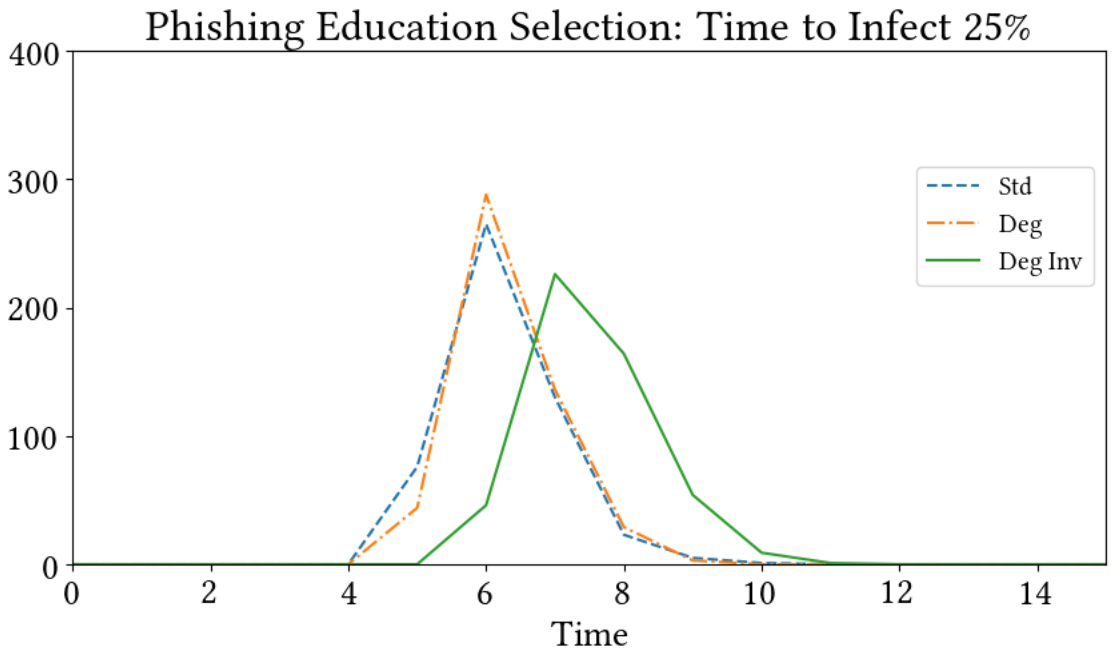

```
Plotting results for time to reach 25%.
```



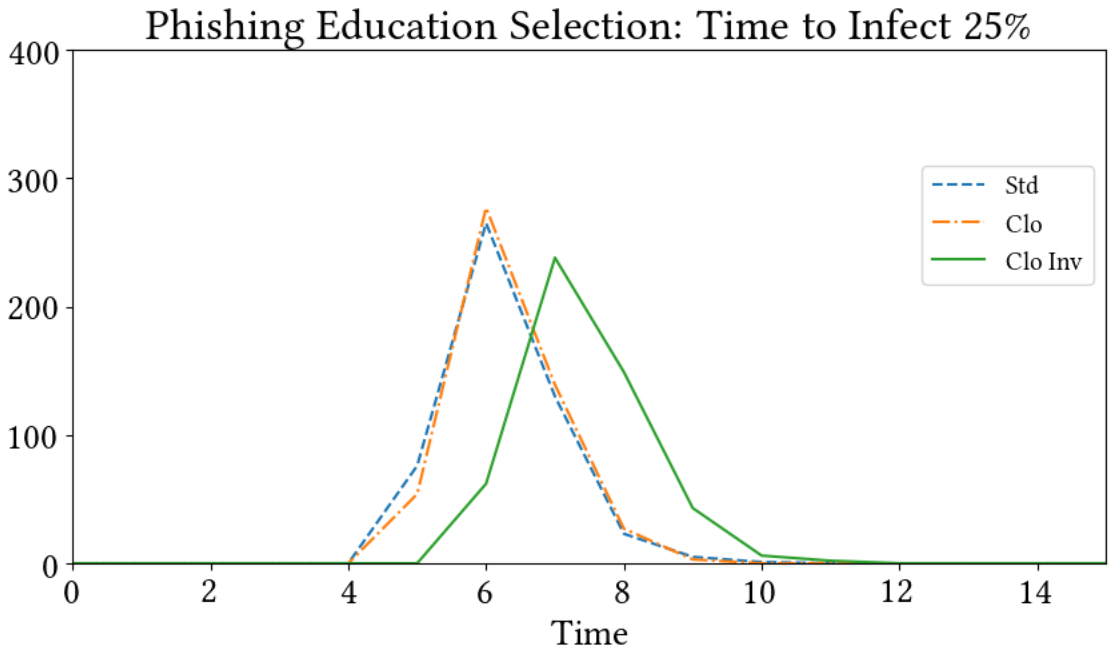

```
Plotting results for time to reach 25%.
```



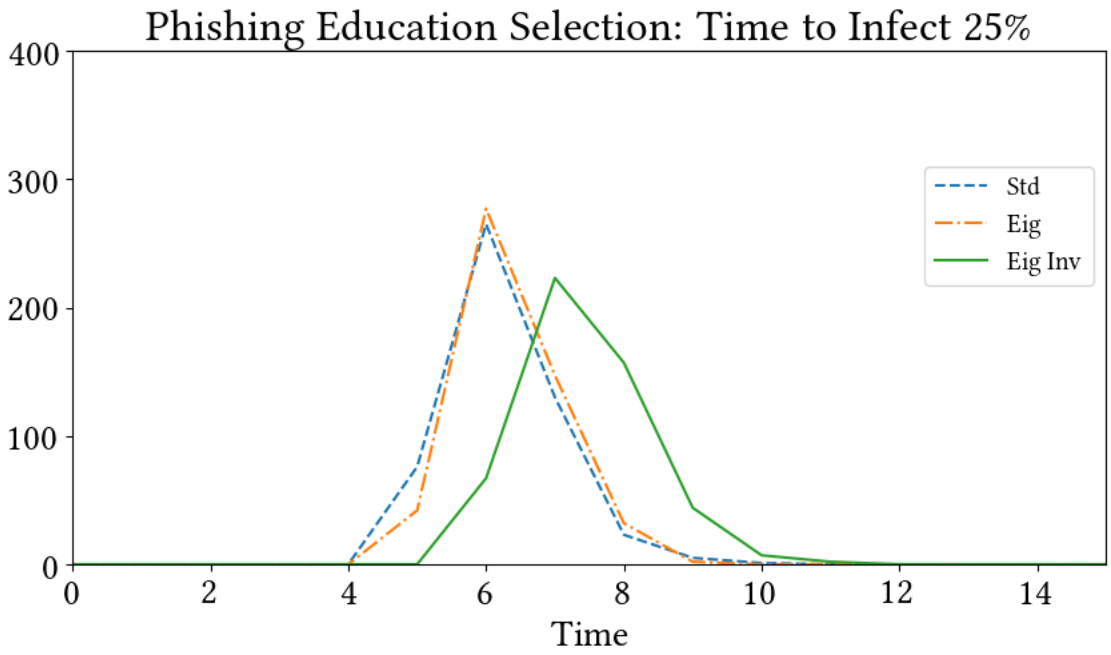

```
Plotting results for time to reach 25%.
```



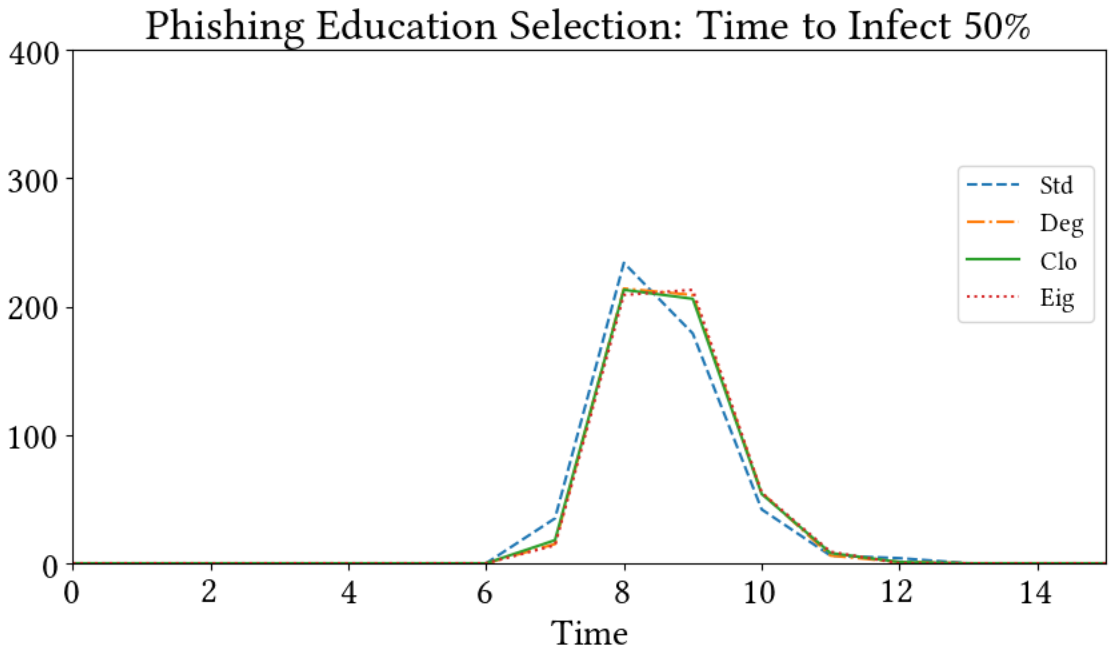

```
Plotting results for time to reach 25%.
```



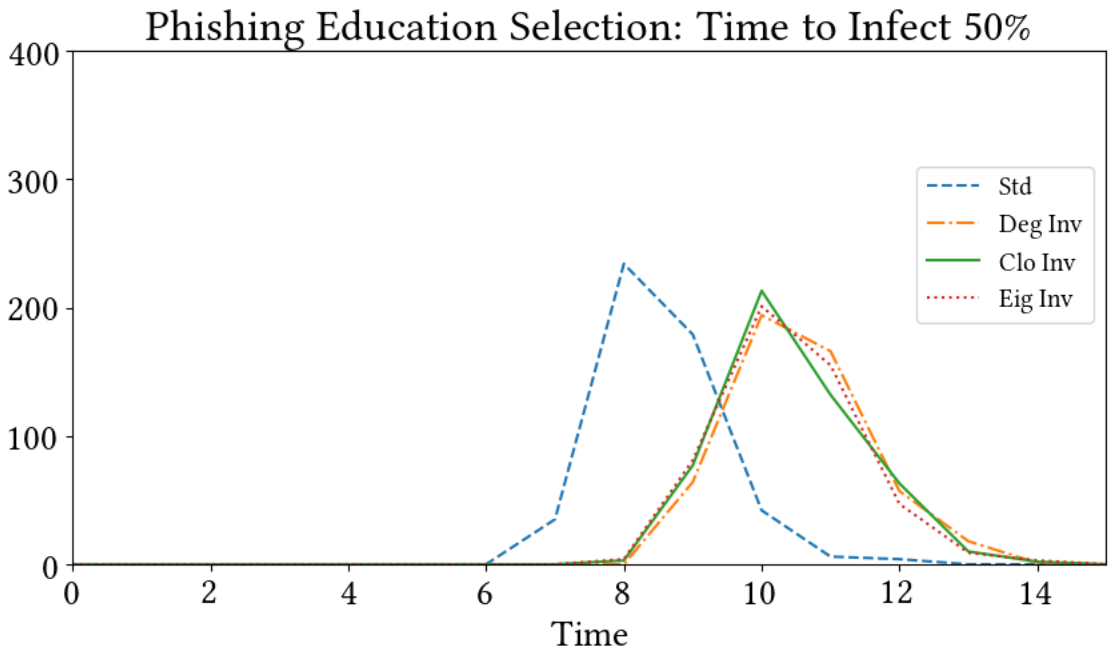

```
Plotting results for time to reach 25%.
```



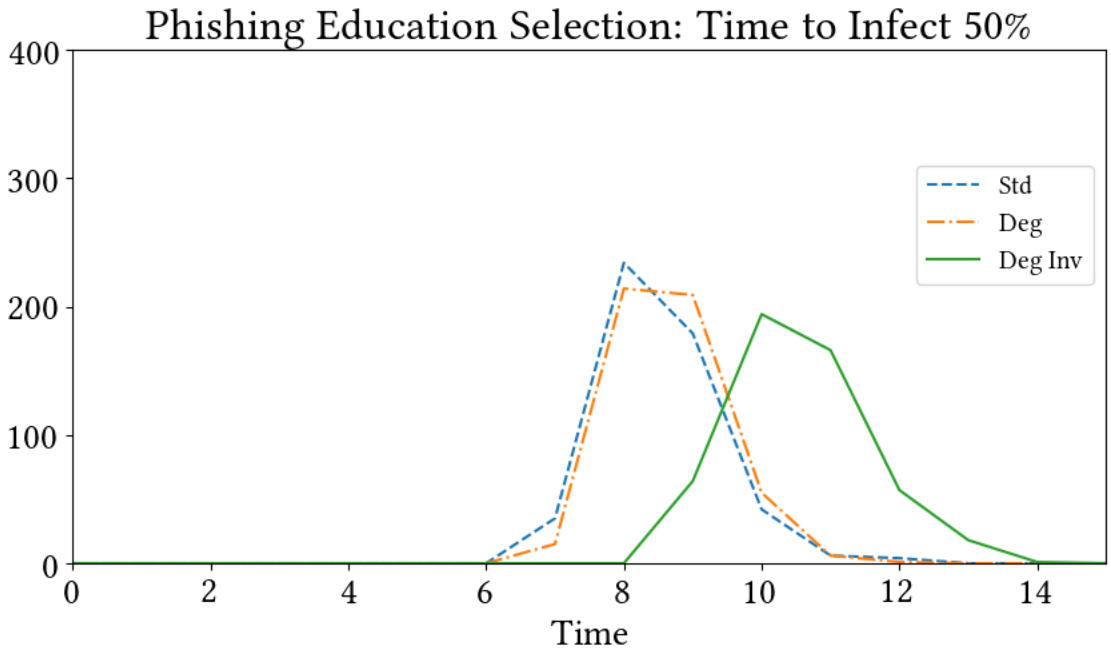

```
Plotting results for time to reach 50%.
```



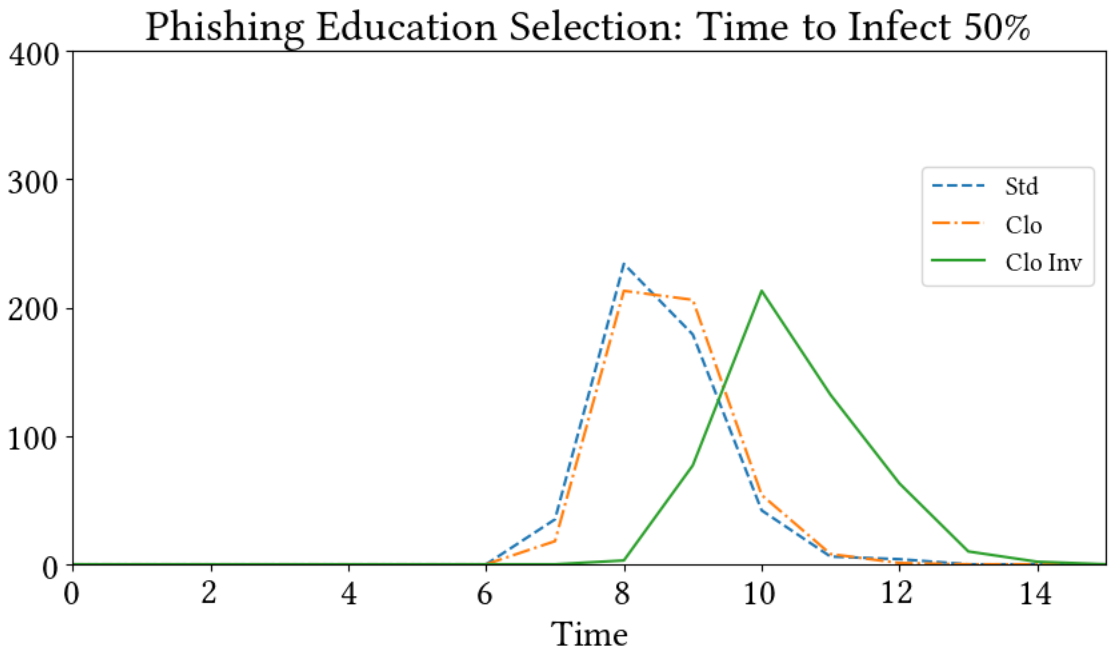

```
Plotting results for time to reach 50%.
```



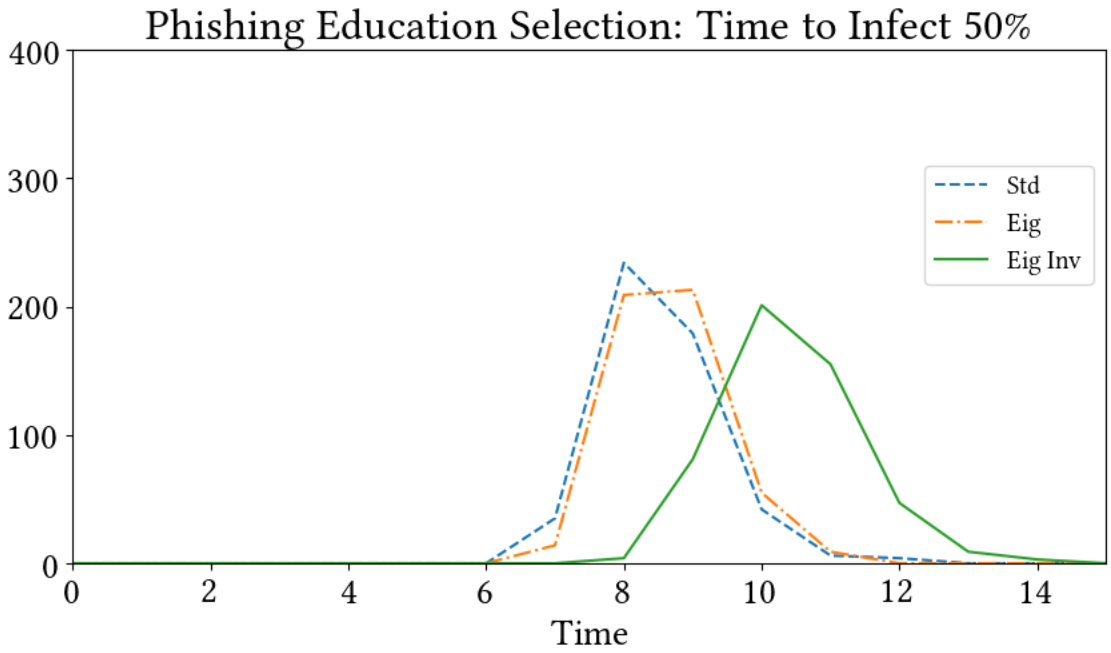

```
Plotting results for time to reach 50%.
```



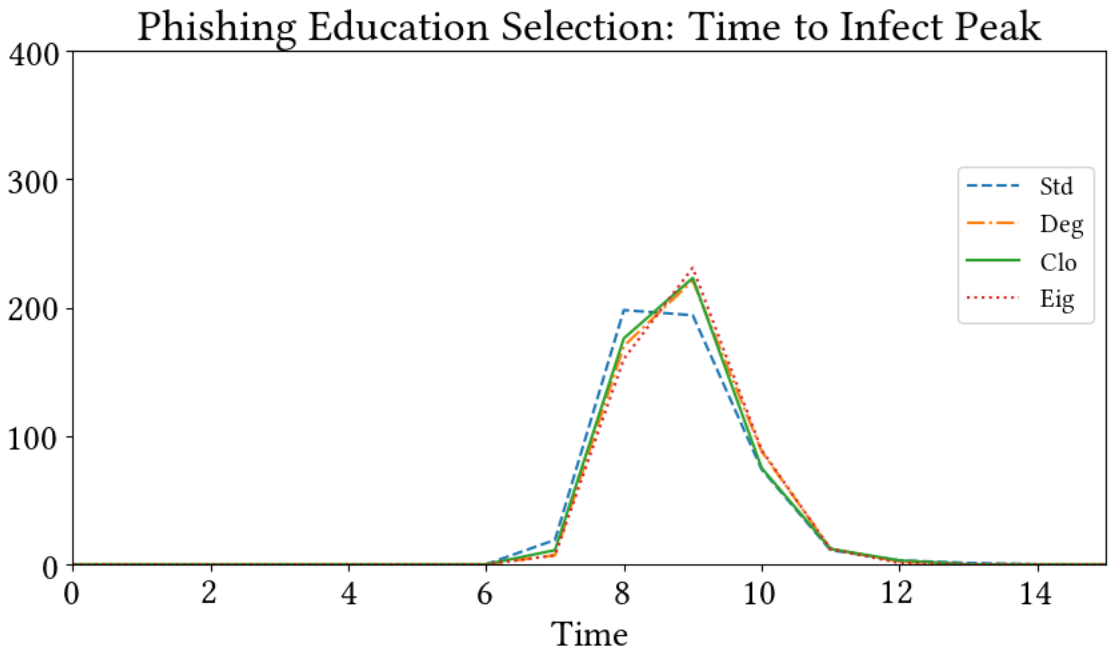

```
Plotting results for time to reach 50%.
```



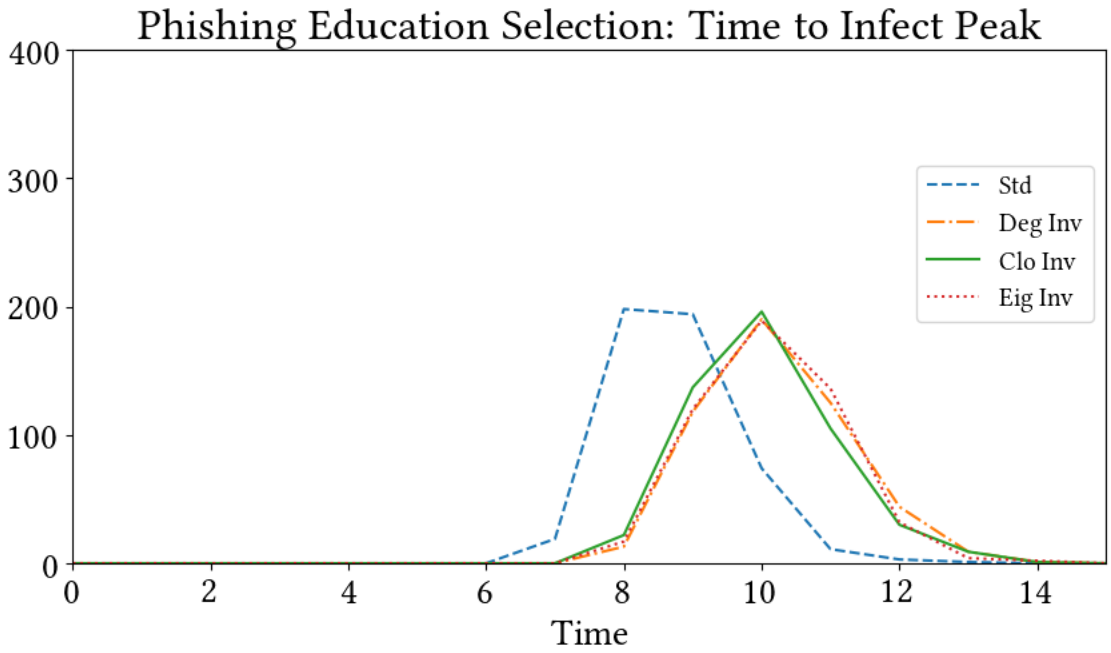
Plotting results for time to reach 50%.



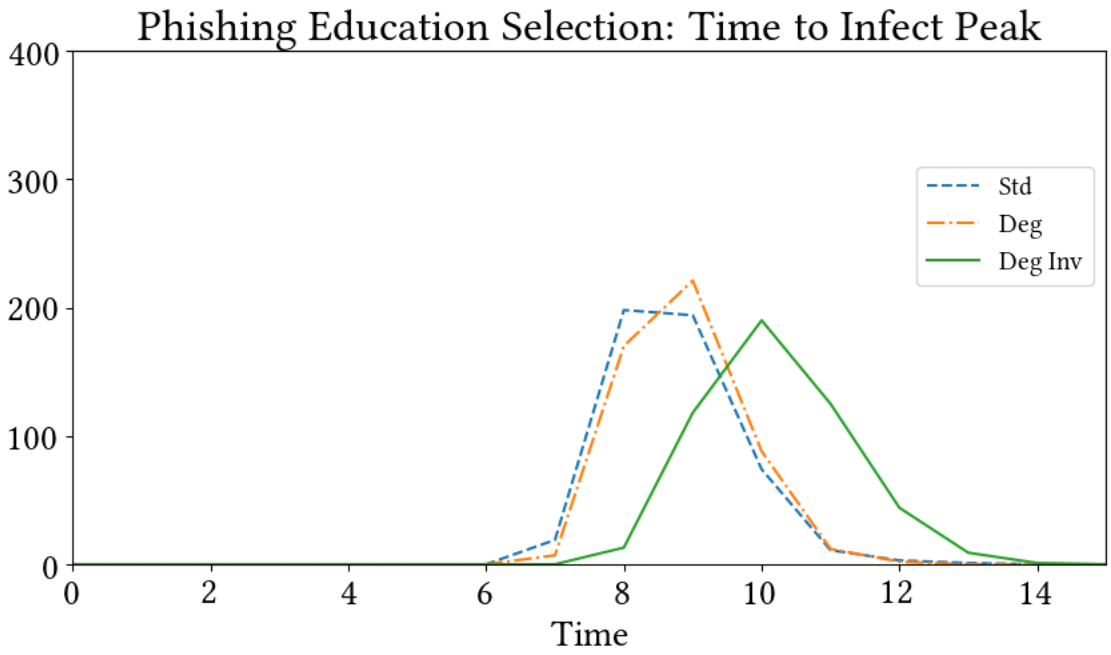
Plotting results for time to reach Peak.



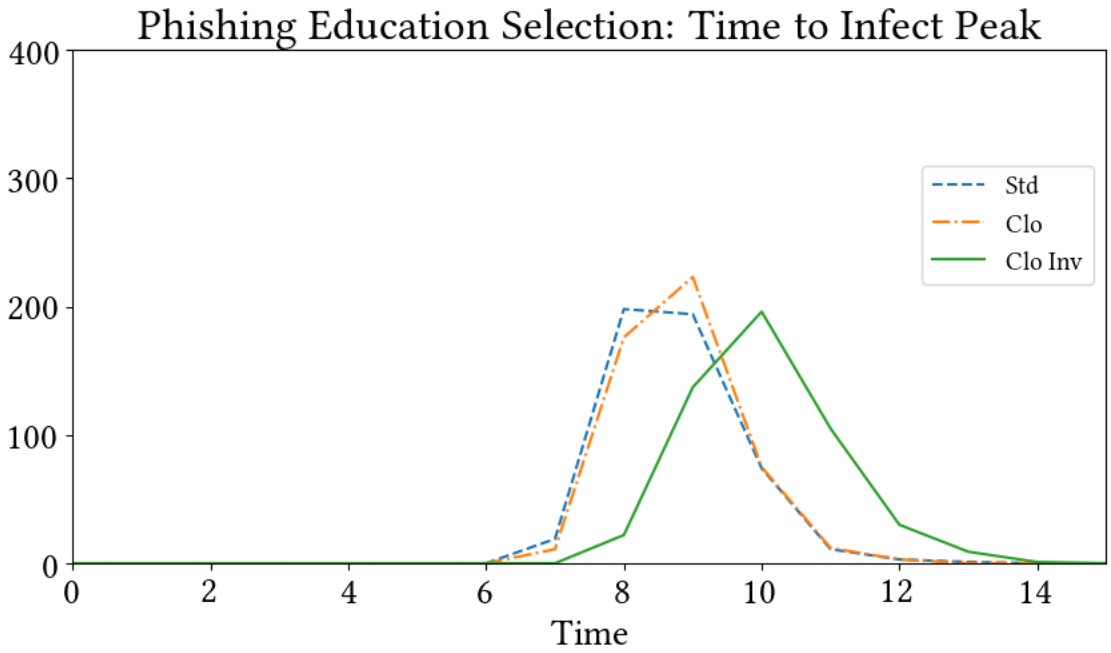

```
Plotting results for time to reach Peak.
```



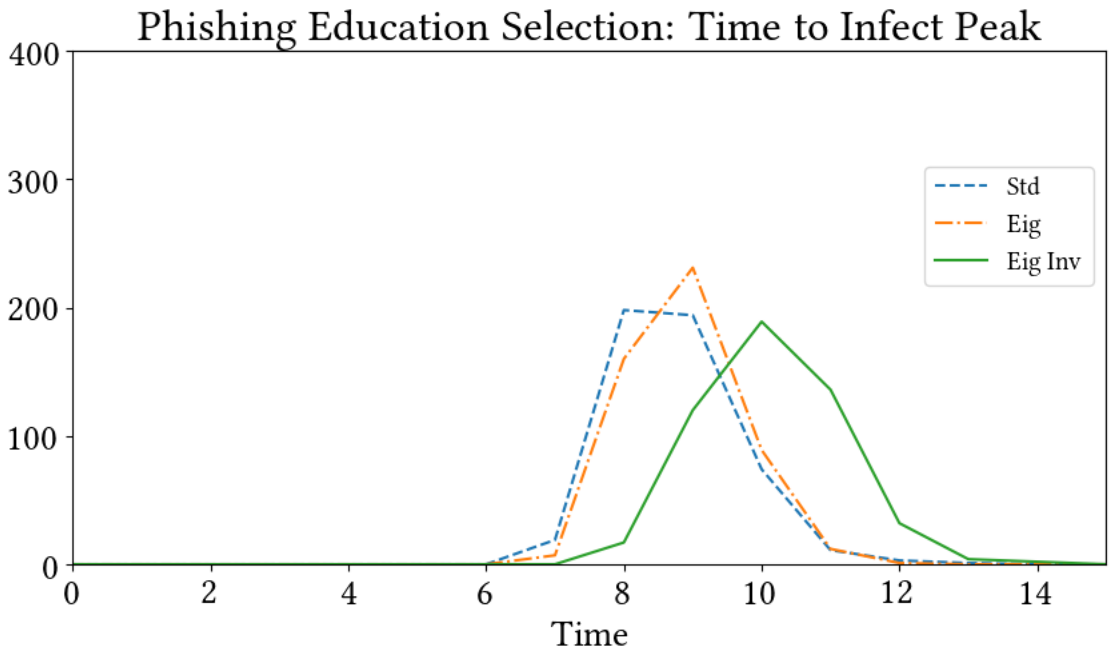

```
Plotting results for time to reach Peak.
```



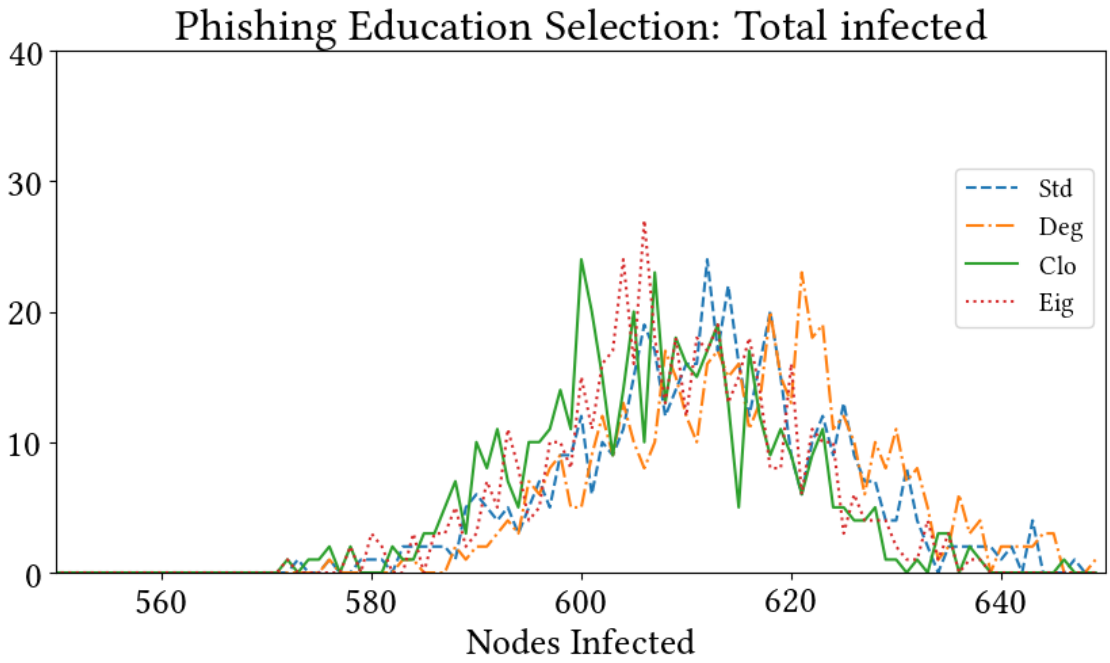

```
Plotting results for time to reach Peak.
```



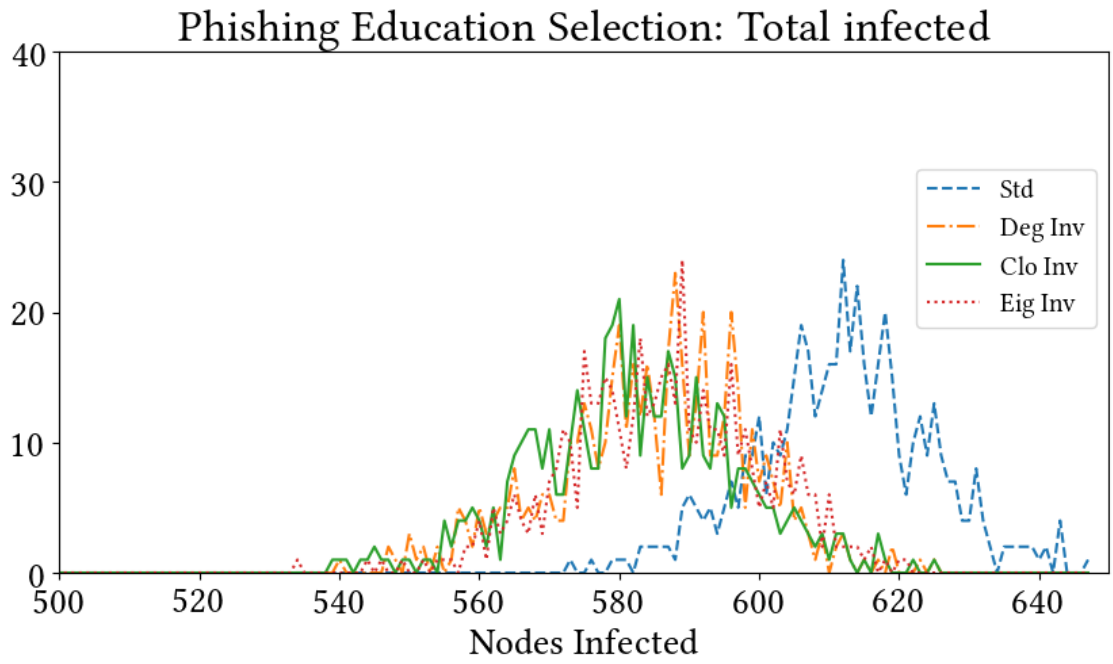

```
Plotting results for time to reach Peak.
```



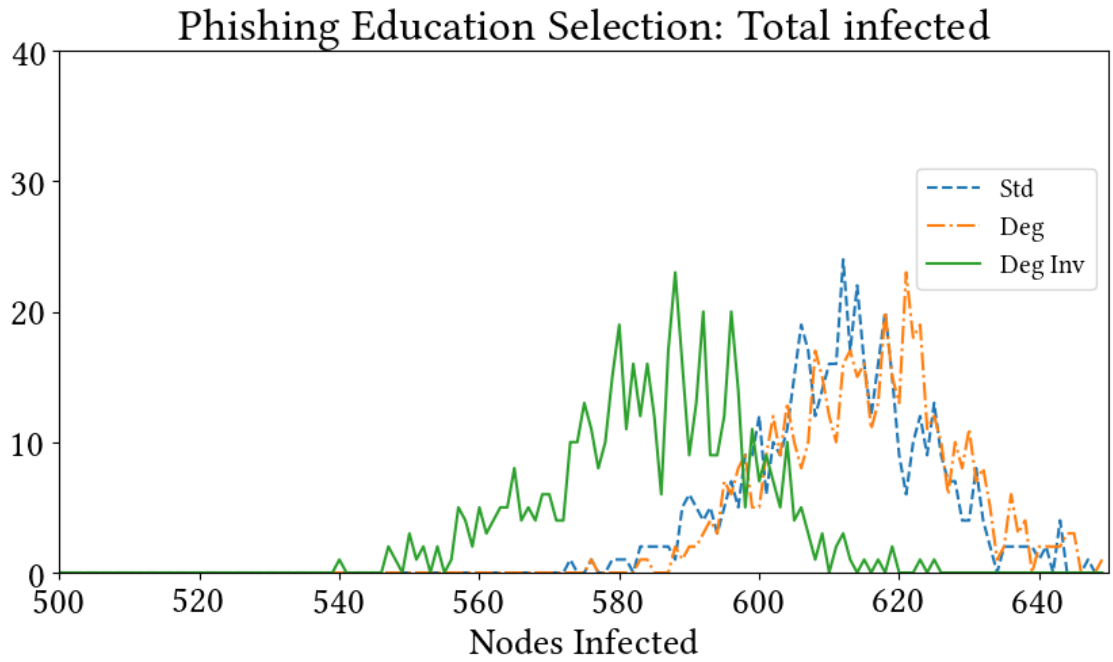

```
Plotting results for final infected.
```



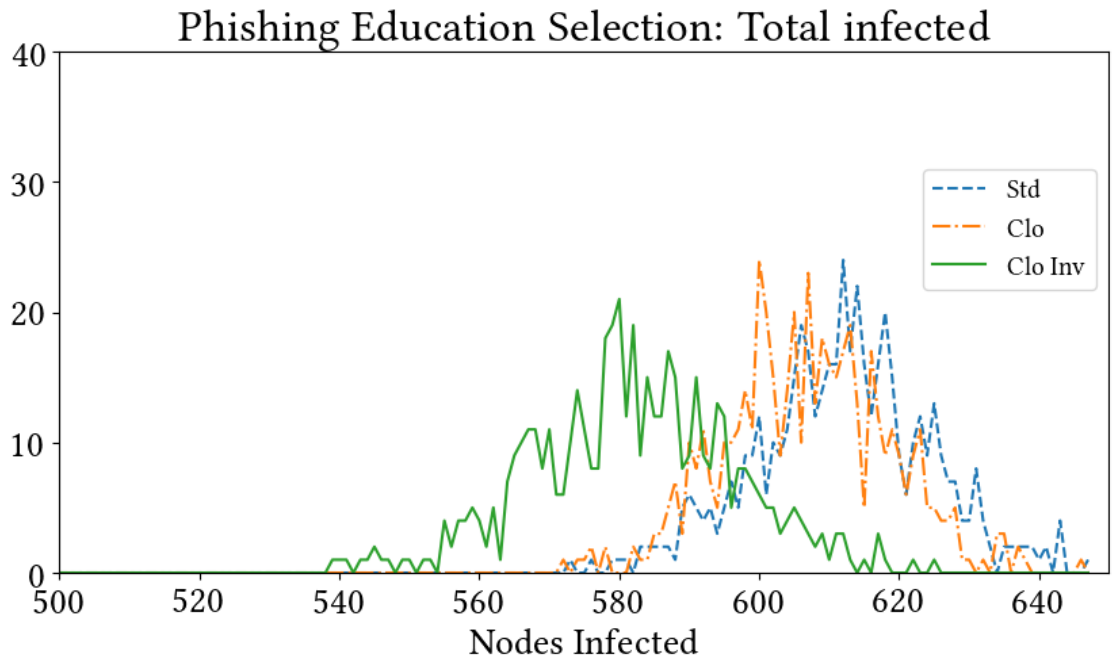

```
Plotting results for final infected.
```



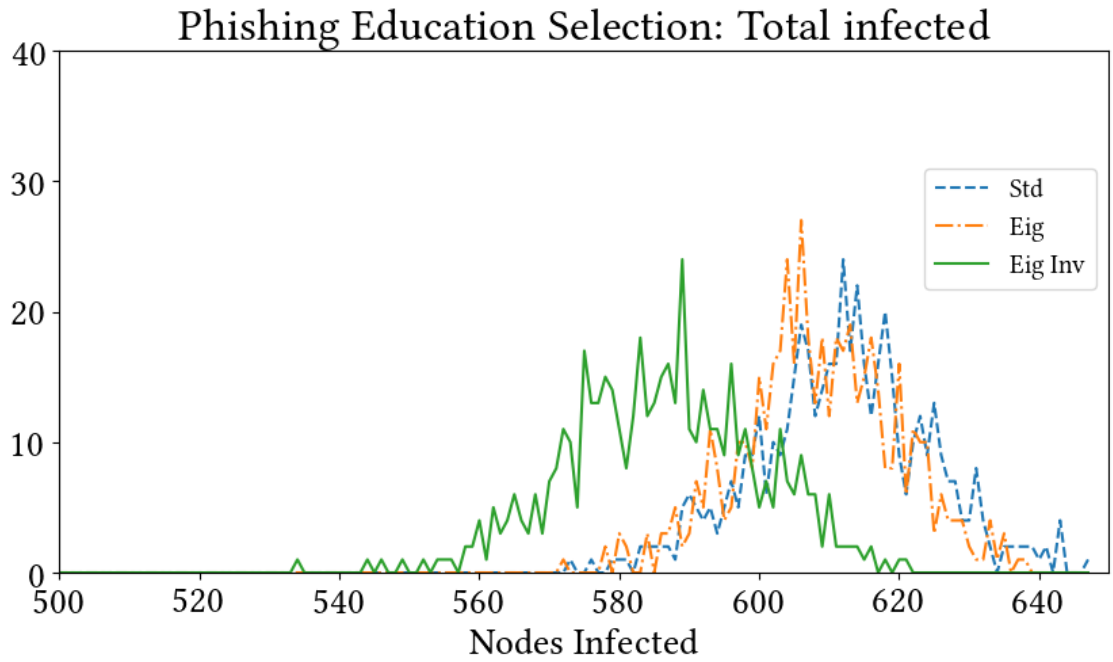

```
Plotting results for final infected.
```




```
Plotting results for final infected.
```




```
Plotting results for final infected.
```




```
Done!
```

[ ]: