Assignment 3

Last updated: June 23, 2022

Name: Adam Bakopolus

Uniqname: abakop

Instructions

Please turn in:

1. A Jupyter Notebook file. This file should show all of the required work, including code, results, visualizations (if any), and necessary comments to your code. Irrelevant code and results should be deleted prior to submission. This file is submitted automatically when you submit your notebook to be autograded. This is done in Assignment 3 -- Create.
2. An HTML file of the Notebook. Submit this file in Assignment 3 - Submit.
3. A PDF file of the Notebook. Submit this file in Assignment 3 - Submit.

Before submitting, please select Kernel -> Restart & Run All.

Please do not remove any code outside of the Not Implemented Error sections. The autograder may need it.

In [1]:
```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
from networkx.drawing.nx_pydot import graphviz_layout
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep
import operator
import random
```

German highway system network

The data source used in this assignment is adopted and modified from the original Matlab file on the website (http://www.biological-networks.org/?page_id=25). You can read more about the data source in its paper (https://www.dynamic-connectome.org/pubs/Kaiser2004b.pdf).

> Kaiser M., and Hilgetag C.-C. (2004) Spatial growth of real-world networks. Physical Review E 69:036103.

Q1. (1 point, Autograded) Load the graph from the dataset. How many nodes in the graph have a degree of one?

In [2]:
```
G=nx.read_edgelist("assets/german.txt", delimiter=' ')

node_list = [val for (node, val) in G.degree()]
degree_is_one = node_list.count(1) # assign it with the number of nodes of degree one. This should be an int.
```

In [3]:
```
#hidden tests for Question 1 are within this cell
```

Part 1. Diffusion models

In this part, you will practice simulating diffusion process on the graph with several difussion models we have learned.

Q2. (7 points, Autograded) Threshold model

Use the threshold model (https://ndlib.readthedocs.io/en/latest/reference/models/epidemics/Threshold.html) provided by the NDlib library:

```
ndlib.models.epidemics.ThresholdModel.ThresholdModel(graph, seed=None)
```

Complete the following function with the given signature, so that it simulates a diffusion process with a threshold model and returns a list of infected number in each iteration.

The function has two options for determining seed nodes:

- Passing a function as input (`importance_measure`), which takes in G and outputs a dictionary with nodes as the keys and scores as the values. Our function should use the top `n` nodes based on these scores as the seed nodes. For example, `importance_measure` could output a dictionary with the degree of each node.
- If `importance_measure` is not passed, then our function should pick a random `float(n)/len(G.nodes)` fraction of nodes to be infected.

Hint

The `ModelConfig` object supports two configuration parameters:

- To specify a list of initially infected nodes `infected_nodes` use:

```
config.add_model_initial_configuration("Infected", infected_nodes)
```

- To choose a random fraction of infected nodes use:

```
config.add_model_parameter('fraction_infected', fraction)
```

In [4]:
```
def simulate_threshold(G, importance_measure=None, iterate=50, n=1, threshold=0.25):
    if importance_measure:
        # select seed nodes
        sorted_node = sorted(importance_measure(G).items(), key=operator.itemgetter(1))[::-1]
        highest_nodes = [n for n, _ in sorted_node[:n]]

    # Model selection
    model = ep.ThresholdModel(G, seed = 42)

    # Model Configuration
    config = mc.Configuration()

    if importance_measure:
        config.add_model_initial_configuration("Infected", highest_nodes)

    else:
        config.add_model_parameter('fraction_infected', float(n)/len(G.nodes))

    for i in G.nodes():
        config.add_node_configuration("threshold", i, threshold)

    model.set_initial_status(config)

    # Simulation execution
    iterations = model.iteration_bunch(iterate)
    return [it['node_count'][1] for it in iterations]
```

In [5]:
```
#hidden tests for Question 2 are within this cell
```

Q3. (3 points, Autograded) Selection of seed nodes

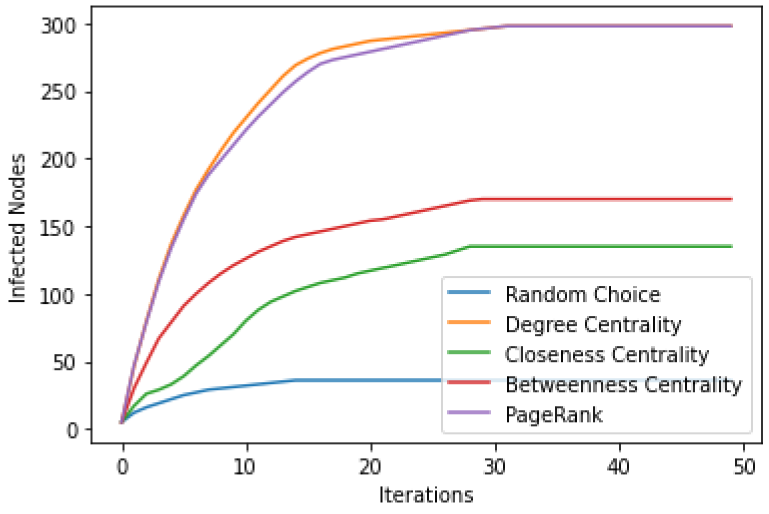
Compare 5 different settings of seed nodes:

- randomly select `N` nodes
- select `N` nodes with the highest degree
- select `N` nodes with the highest closeness centrality
- select `N` nodes with the highest betweenness centrality
- select `N` nodes with the highest PageRank value

Use the function in question 2 to simulate the diffusion for each seed node set, with `iterate=50` , `N=5` , and `threshold=0.3` . Create a single plot with 5 curves (one for each set of seed nodes) with iteration number on the x-axis and the number of infected nodes on the y-axis.

Note: please do not modify the random seed. This is to ensure that your simulation will be consistent with the autograder.

In [6]:
```
I=50
N=5
T=0.3
node_count_random = simulate_threshold(G, importance_measure=None, iterate= I, n = N, threshold = T)
node_count_deg = simulate_threshold(G, importance_measure= nx.degree_centrality, iterate= I, n = N, threshold =
node_count_closeness= simulate_threshold(G, importance_measure= nx.closeness_centrality, iterate= I, n = N, thre
node_count_betweenness = simulate_threshold(G, importance_measure= nx.betweenness_centrality, iterate= I, n = N,
node_count_pagerank = simulate_threshold(G, importance_measure= nx.pagerank, iterate= I, n = N, threshold = T)

x = range(I)

plt.plot(x, node_count_random, label = "Random Choice")
plt.plot(x, node_count_deg, label = "Degree Centrality")
plt.plot(x, node_count_closeness, label = "Closeness Centrality")
plt.plot(x, node_count_betweenness, label = "Betweenness Centrality")
plt.plot(x, node_count_pagerank, label = "PageRank")

plt.legend()
plt.xlabel('Iterations')
plt.ylabel('Infected Nodes')
plt.show()
```



In [7]:
```
#hidden tests for Question 3 are within this cell
```

Q4. (1 point, Autograded) Which seed nodes generation methods converges first?

Informally, we say that a method converges when the number of infected nodes stops changing significantly as the number of iterations increases.

Accepted strings:

```
"random" | "degree" | "closeness" | "betweenness" | "pagerank"
```

Select all that apply to the list in the following cell. For example, if "closeness" and "degree" are the equally fast and converge first, you should input

```
method = ["closeness", "degree"]
```

In [8]:
```
method = ['random'] # This should be a list of strings.
```

In [9]:
```
#hidden tests for Question 4 are within this cell
```

Q5. (3 points, Autograded) Randomized threshold model

We can add randomness to the threshold model by randomly assigning thresholds to the nodes.

Modify the function you wrote in Q2. Instead of setting the same threshold value for each node, use the `random.uniform(lower, upper)` function to randomly generate a threshold between $[lower, upper)$.

In [10]:
```
def simulate_rand_threshold(G, importance_measure=None, iterate=100, n=1, lower=0, upper=1):
    if importance_measure:
        # select seed nodes
        sorted_node = sorted(importance_measure(G).items(), key=operator.itemgetter(1))[::-1]
        highest_nodes = [n for n, _ in sorted_node[:n]]

    # Model selection
    model = ep.ThresholdModel(G, seed = 42)
    random.seed(42)

    # Model Configuration
    config = mc.Configuration()

    if importance_measure:
        config.add_model_initial_configuration("Infected", highest_nodes)

    else:
        config.add_model_parameter('fraction_infected', float(n)/len(G.nodes))

    for i in G.nodes():
        config.add_node_configuration("threshold", i, random.uniform(lower, upper))

    model.set_initial_status(config)

    # Simulation execution
    iterations = model.iteration_bunch(iterate)
    return [it['node_count'][1] for it in iterations]
```

In [11]:
```
#hidden tests for Question 5 are within this cell
```

Q6. (4 points, Autograded) Threshold range selection

Find a range $[a, b)$ such that the seed nodes selected using a random approach **converges** faster than at least one other method. You should run at least 50 iterations.

Hint: You can solve through trial and error, but think about what range of values would lead to slower convergence of the random strategy. There is no exact answer, but you can reason about the range of values that could work. What happens if the thresholds are all very large? Very small? Mid-range?

In [12]:
```
a = .4  # seleted lower bound. This should be a float in [0,1].
b = .5  # seleted upper bound. This should be a float in [0,1].
```

In [13]:
```
x = simulate_rand_threshold(G, n=N, iterate=I, lower=a, upper=b)
r1 = simulate_rand_threshold(G, nx.degree_centrality, n=N, iterate=I, lower=a, upper=b)
r2 = simulate_rand_threshold(G, nx.closeness_centrality, n=N, iterate=I, lower=a, upper=b)
r3 = simulate_rand_threshold(G, nx.betweenness_centrality, n=N, iterate=I, lower=a, upper=b)
r4 = simulate_rand_threshold(G, nx.pagerank, n=N, iterate=I, lower=a, upper=b)

#hidden tests for Question 6 are within this cell
```

Q7. (5 points, Autograded) Independent cascade model

The independent cascade model (https://ndlib.readthedocs.io/en/latest/reference/models/epidemics/IndependentCascades.html) also has a threshold parameter. Note that this threshold is not the same as the threshold of the threshold model. Here it is specified for edges and represents the probability of diffusion along each edge.

Complete the following function with the given signature, so that it simulates a diffusion process with an independent cascade model and returns a list of the number infected in each iteration. The seed nodes should be selected in the same way as in Q2.

```
In [14]: def simulate_IC(G, importance_measure=None, iterate=100, n=1, threshold=0.3):
             if importance_measure:
                 # select seed nodes
                 sorted_node = sorted(importance_measure(G).items(), key=operator.itemgetter(1))[::-1]
                 highest_nodes = [n for n, _ in sorted_node[:n]]

             # Model selection
             model = ep.IndependentCascadesModel(G, seed = 42)
             random.seed(42)

             # Model Configuration
             config = mc.Configuration()

             if importance_measure:
                 config.add_model_initial_configuration("Infected", highest_nodes)

             else:
                 config.add_model_parameter('fraction_infected', float(n)/len(G.nodes))

             for i in G.edges():
                 config.add_edge_configuration("threshold", i, threshold)

             model.set_initial_status(config)

             # Simulation execution
             iterations = model.iteration_bunch(iterate)
             return [it['node_count'][1] + it['node_count'][2] for it in iterations]  # number of people who already know
```

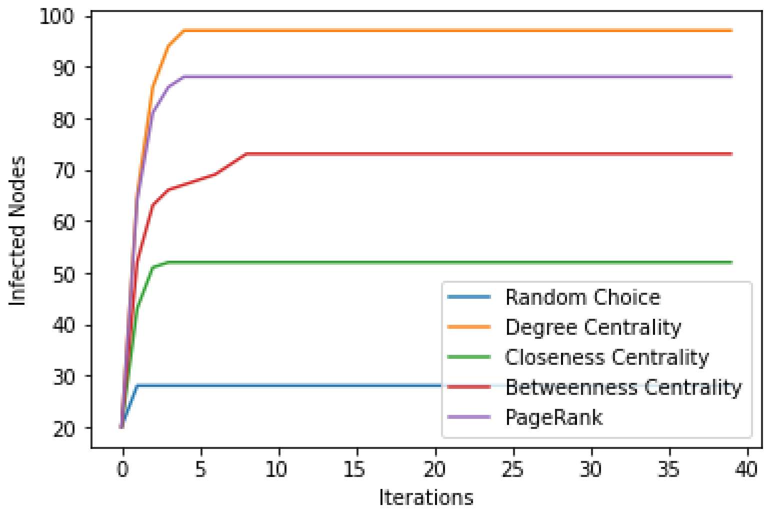
```
In [15]: #hidden tests for Question 7 are within this cell
```

Q8. (5 points, Manually graded) Seed node comparison

Apply the same comparison on the 5 seed node sets for the independent cascade model with the function you implemented in Q7. Set the parameters as `N=20` , `iterate=40` and `threshold=0.3` .

Create a single plot with 5 curves (one for each set of seed nodes) with iteration number on the x-axis and the number of infected nodes on the y-axis.

In [16]:
```
I=40
N=20
T=0.3
node_count_random = simulate_IC(G, importance_measure=None, iterate= I, n = N, threshold = T)
node_count_deg = simulate_IC(G, importance_measure= nx.degree_centrality, iterate= I, n = N, threshold = T)
node_count_closeness= simulate_IC(G, importance_measure= nx.closeness_centrality, iterate= I, n = N, threshold =
node_count_betweenness = simulate_IC(G, importance_measure= nx.betweenness_centrality, iterate= I, n = N, thresh
node_count_pagerank = simulate_IC(G, importance_measure= nx.pagerank, iterate= I, n = N, threshold = T)

x = range(I)

plt.plot(x, node_count_random, label = "Random Choice")
plt.plot(x, node_count_deg, label = "Degree Centrality")
plt.plot(x, node_count_closeness, label = "Closeness Centrality")
plt.plot(x, node_count_betweenness, label = "Betweenness Centrality")
plt.plot(x, node_count_pagerank, label = "PageRank")

plt.legend()
plt.xlabel('Iterations')
plt.ylabel('Infected Nodes')
plt.show()
```



Q9. (2 points, Autograded) Which method results in the largest cascade size?

Accepted strings:

```
"random" | "degree" | "closeness" | "betweenness" | "pagerank"
```

Select all that apply to the list in the following cell. For example, if "random" and "degree" both result in the largest cascade, you should input

```
method = ["random", "degree"]
```

In [17]:
```
method = ['degree'] # This should be a list of strings.
```

In [18]:
```
#hidden tests for Question 9 are within this cell
```

Q10. (5 points, Autograded) SI model

The SI model (https://ndlib.readthedocs.io/en/latest/reference/models/epidemics/SIm.html) considers two states: "susceptible" and "infected". The statuses are interpreted as:

Name	Code
Susceptible	0
Infected	1

The infection probability is given by the parameter `beta` , which should be between 0 and 1.

Complete the following function with the given signature, so that it simulates a diffusion process with an SI model and returns a list of currently infected number in each iteration.

```
In [19]: def simulate_SI(G, importance_measure=None, iterate=100, n=1, beta=0.1):
             if importance_measure:
                 # select seed nodes
                 sorted_node = sorted(importance_measure(G).items(), key=operator.itemgetter(1))[::-1]
                 highest_nodes = [n for n, _ in sorted_node[:n]]

             # Model selection
             model = ep.SIModel(G, seed = 42)

             # Model Configuration
             config = mc.Configuration()

             if importance_measure:
                 config.add_model_initial_configuration("Infected", highest_nodes)

             else:
                 config.add_model_parameter('fraction_infected', float(n)/len(G.nodes))

             config.add_model_parameter('beta', beta)

             model.set_initial_status(config)

             # Simulation execution
             iterations = model.iteration_bunch(iterate)
             return [it['node_count'][1] for it in iterations]
```

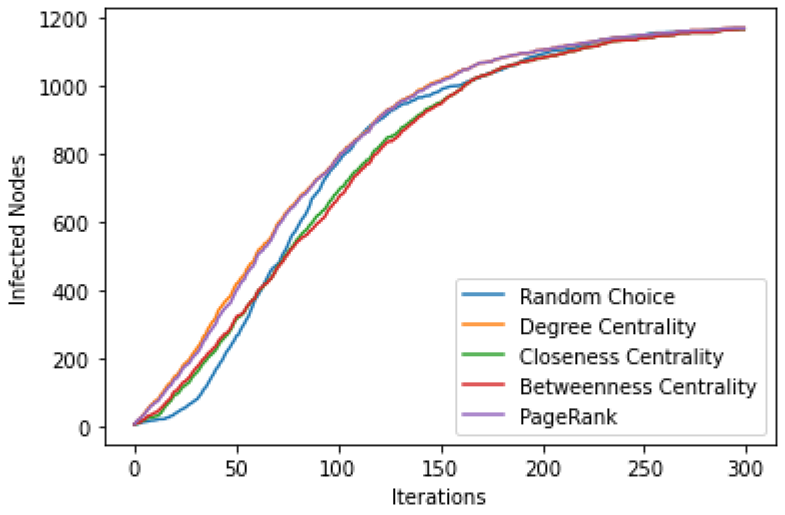
```
In [20]: #hidden tests for Question 7 are within this cell
```

Q11. (3 points, Manually graded) Seed node comparison

Apply the same comparison on the 5 seed node sets for the SI model with the function you implemented in question 10. Set the parameters `N=5` , `iterate=300` , `beta=0.1` .

Create a single plot with 5 curves (one for each set of seed nodes) with iteration number on the x-axis and the number of currently infected nodes on the y-axis.

```
In [21]: #random.seed(42)
         N=5
         I=300
         b = 0.1
         node_count_random = simulate_SI(G, importance_measure=None, iterate= I, n = N, beta = b)
         node_count_deg = simulate_SI(G, importance_measure= nx.degree_centrality, iterate= I, n = N, beta = b)
         node_count_closeness= simulate_SI(G, importance_measure= nx.closeness_centrality, iterate= I, n = N, beta = b)
         node_count_betweenness = simulate_SI(G, importance_measure= nx.betweenness_centrality, iterate= I, n = N, beta =
         node_count_pagerank = simulate_SI(G, importance_measure= nx.pagerank, iterate= I, n = N, beta = b)

         x = range(I)

         plt.plot(x, node_count_random, label = "Random Choice")
         plt.plot(x, node_count_deg, label = "Degree Centrality")
         plt.plot(x, node_count_closeness, label = "Closeness Centrality")
         plt.plot(x, node_count_betweenness, label = "Betweenness Centrality")
         plt.plot(x, node_count_pagerank, label = "PageRank")

         plt.legend()
         plt.xlabel('Iterations')
         plt.ylabel('Infected Nodes')
         plt.show()
```



Q12. (5 points, Autograded) SIR model

The SIR model (https://ndlib.readthedocs.io/en/latest/reference/models/epidemics/SIR.html) has one more state than the SI model:

Name	Code
Susceptible	0
Infected	1
Removed	2

And in addition to `beta`, it has an additional `gamma` parameter which indicates removal probability. Complete the following function with the given signature, so that it simulates a diffusion process with a SIR model and returns a list of the number of currently infected nodes and the cumulative number of infected nodes at each iteration.

Notice

For the SIR model, since it has an additional recovery stage, the numer of infected nodes can decrease. Therefore, the function `simulate_SIR` will return two lists: one is the same as before --- the number of currently infected nodes. The other one (`total_infected_nodes`) is the total number of nodes that have been infected at least once across all iterations, whether or not they have recovered.

In [22]:
```
def simulate_SIR(G, importance_measure=None, iterate=100, n=1, beta=0.1, gamma=0.05):
    if importance_measure:
        # select seed nodes
        sorted_node = sorted(importance_measure(G).items(), key=operator.itemgetter(1))[::-1]
        highest_nodes = [n for n, _ in sorted_node[:n]]

    # Model selection
    model = ep.SIRModel(G, seed = 42)

    # Model Configuration
    config = mc.Configuration()

    if importance_measure:
        config.add_model_initial_configuration("Infected", highest_nodes)

    else:
        config.add_model_parameter('fraction_infected', float(n)/len(G.nodes))

    config.add_model_parameter('beta', beta)
    config.add_model_parameter('gamma', gamma)

    model.set_initial_status(config)

    # Simulation execution
    iterations = model.iteration_bunch(iterate)

    total_infected_nodes = [it['node_count'][1] + it['node_count'][2] for it in iterations]
    return [it['node_count'][1] for it in iterations], total_infected_nodes
```

In [23]:
```
#hidden tests for Question 12 are within this cell
```

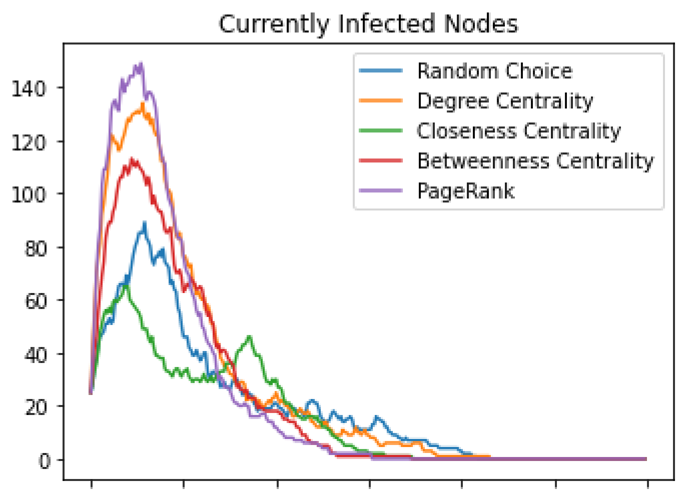
Q13. (6 points, Manually graded) Seed node comparison

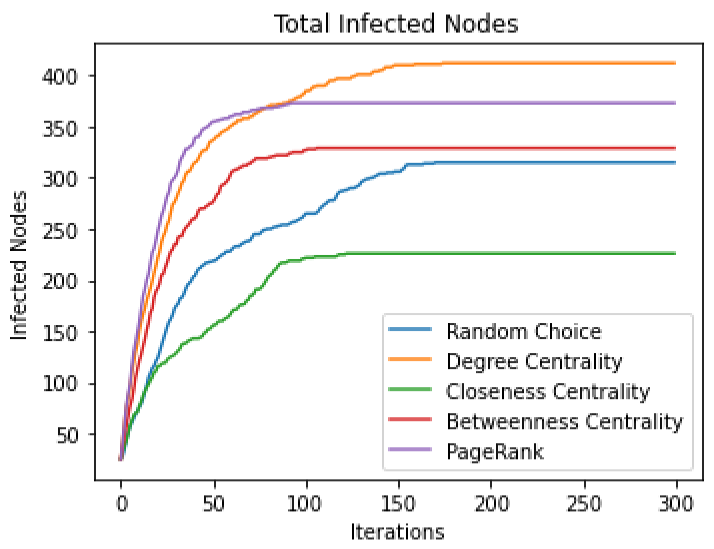
Apply the same comparison on the 5 seed node sets for the SIR model with the function you implemented in question 12. Set the parameters `N=25` , `iterate=300` , `beta=0.1` , `gamma=0.05` .

(a) Create a single plot with 5 curves (one for each set of seed nodes) with iteration number on the x-axis and the number of currently infected nodes on the y-axis.

(b) Create a single plot with 5 curves (one for each set of seed nodes) with iteration number on the x-axis and the total number of nodes that have ever been infected up to the current iteration on the y-axis.

In [24]:
```python
N=25
I=300
b = 0.1
g = 0.05

node_count_random = simulate_SIR(G, importance_measure=None, iterate= I, n = N, beta = b, gamma = g)[0]
node_count_deg = simulate_SIR(G, importance_measure= nx.degree_centrality, iterate= I, n = N, beta = b, gamma =
node_count_closeness= simulate_SIR(G, importance_measure= nx.closeness_centrality, iterate= I, n = N, beta = b,
node_count_betweenness = simulate_SIR(G, importance_measure= nx.betweenness_centrality, iterate= I, n = N, beta
node_count_pagerank = simulate_SIR(G, importance_measure= nx.pagerank, iterate= I, n = N, beta = b, gamma = g)[

total_random = simulate_SIR(G, importance_measure=None, iterate= I, n = N, beta = b, gamma = g)[1]
total_deg = simulate_SIR(G, importance_measure= nx.degree_centrality, iterate= I, n = N, beta = b, gamma = g)[1]
total_closeness= simulate_SIR(G, importance_measure= nx.closeness_centrality, iterate= I, n = N, beta = b, gamma
total_betweenness = simulate_SIR(G, importance_measure= nx.betweenness_centrality, iterate= I, n = N, beta = b,
total_pagerank = simulate_SIR(G, importance_measure= nx.pagerank, iterate= I, n = N, beta = b, gamma = g)[1]

fig, axes = plt.subplots(1,2, figsize = (12, 4))
axes[0].title.set_text("Currently Infected Nodes")
axes[1].title.set_text("Total Infected Nodes")

axes[0].plot(x, node_count_random, label = "Random Choice")
axes[0].plot(x, node_count_deg, label = "Degree Centrality")
axes[0].plot(x, node_count_closeness, label = "Closeness Centrality")
axes[0].plot(x, node_count_betweenness, label = "Betweenness Centrality")
axes[0].plot(x, node_count_pagerank, label = "PageRank")
axes[0].legend()


axes[1].plot(x, total_random, label = "Random Choice")
axes[1].plot(x, total_deg, label = "Degree Centrality")
axes[1].plot(x, total_closeness, label = "Closeness Centrality")
axes[1].plot(x, total_betweenness, label = "Betweenness Centrality")
axes[1].plot(x, total_pagerank, label = "PageRank")
axes[1].legend()

plt.xlabel('Iterations')
plt.ylabel('Infected Nodes')
plt.show()
```





Q14. (5 points, Autograded) SIS model

The SIS model (https://ndlib.readthedocs.io/en/latest/reference/models/epidemics/SIS.html) has two states: susceptible and infected:

Name	Code
Susceptible	0
Infected	1

And in addition to `beta` , it has an additional `lambda` parameter which indicates the probability of an infected node transferring back to the susceptible state. Complete the following function with the given signature, so that it simulates a diffusion process with an SIS model and returns a list of infected number in each iteration.

Notice

`lambda` is a reserved keyword in Python, so we use `_lambda` instead in the function argument.

In [25]:
```
def simulate_SIS(G, importance_measure=None, iterate=100, n=1, beta=0.1, _lambda=0.05):
    if importance_measure:
        # select seed nodes
        sorted_node = sorted(importance_measure(G).items(), key=operator.itemgetter(1))[::-1]
        highest_nodes = [n for n, _ in sorted_node[:n]]

    # Model selection
    model = ep.SISModel(G, seed = 42)

    # Model Configuration
    config = mc.Configuration()

    if importance_measure:
        config.add_model_initial_configuration("Infected", highest_nodes)

    else:
        config.add_model_parameter('fraction_infected', float(n)/len(G.nodes))

    config.add_model_parameter('beta', beta)
    config.add_model_parameter('lambda', _lambda)

    model.set_initial_status(config)

    # Simulation execution
    iterations = model.iteration_bunch(iterate)
    return [it['node_count'][1] for it in iterations]
```

In [26]:
```
#hidden tests for Question 14 are within this cell
```

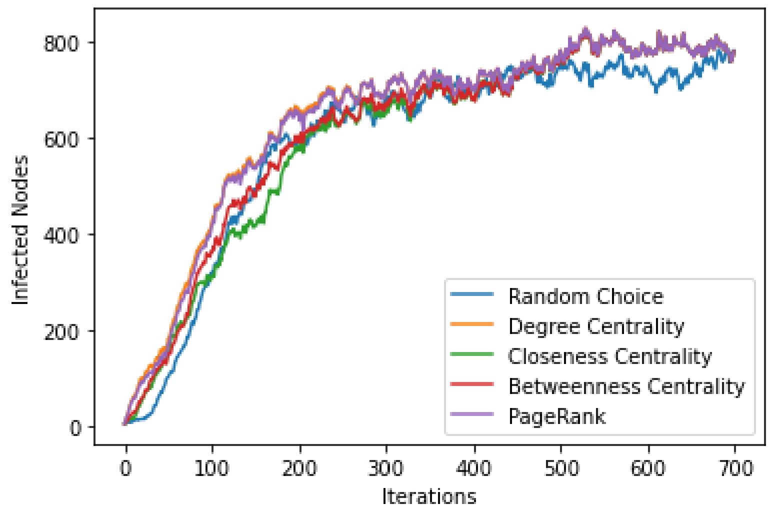
Q15. (3 points, Manually graded) Seed node comparison

Apply the same comparison on the 5 seed node sets for the SIS model with the function you implemented in question 14.

Set parameters `N=5` , `iterate=700` , `beta=0.1` , `lambda=0.05` .

Create a single plot with 5 curves (one for each set of seed nodes) with iteration number on the x-axis and the number of currently infected nodes on the y-axis.

In [27]:
```
N=5
I=700
b = 0.1
l = 0.05

node_count_random = simulate_SIS(G, importance_measure=None, iterate= I, n = N, beta = b, _lambda = l)
node_count_deg = simulate_SIS(G, importance_measure= nx.degree_centrality, iterate= I, n = N, beta = b, _lambda
node_count_closeness= simulate_SIS(G, importance_measure= nx.closeness_centrality, iterate= I, n = N, beta = b,
node_count_betweenness = simulate_SIS(G, importance_measure= nx.betweenness_centrality, iterate= I, n = N, beta
node_count_pagerank = simulate_SIS(G, importance_measure= nx.pagerank, iterate= I, n = N, beta = b, _lambda = l)

x = range(I)

plt.plot(x, node_count_random, label = "Random Choice")
plt.plot(x, node_count_deg, label = "Degree Centrality")
plt.plot(x, node_count_closeness, label = "Closeness Centrality")
plt.plot(x, node_count_betweenness, label = "Betweenness Centrality")
plt.plot(x, node_count_pagerank, label = "PageRank")

plt.legend()
plt.xlabel('Iterations')
plt.ylabel('Infected Nodes')
plt.show()
```



Q16. (5 points, Manually graded) SI, SIR, SIS

Compare the plots of the number of currently infected nodes of the SI, SIR, and SIS models and answer the following questions with a written response.

1. Which model is the first to converge? Why do you think this model converges first?
2. Which model is the last to converge? Why do you think this model converges last?
3. Which model has the largest final number of infected nodes? Why?
4. Which model has the smallest final number of infected nodes? Why?
5. Why does the SIS model fail to completely stabilize in terms of the number of infected nodes?

The SIR model is the first to converge as once nodes are infected and recover, they are removed and no longer susceptible. As a result, the "pool" of susceptible nodes will shrink much faster than the other models, leading to a quicker convergence.

The SIS model is the last to converge as once nodes are infected they have the ability to return to susceptible (and not remain infected or removed like for SI and SIR models, respectively) which results in continued re-infection and a slower convergence.

The SI model has the largest final number of infected nodes as once a node is infected it remains infected unlike being removed or returned to susceptible like in the SIR and SIS models, respectively. As a result, the infected numer will continue to rise until convergence.

The SIR model has the smallest final number of infected nodes due to the points noted above as nodes, once infected, have the ability to recover and be removed from susceptibility. As a result, the model will converge to 0 infected nodes as the model will eventually result in nodes being either susceptible or removed.

The SIS model fails to completely stabilize in terms of the number of infected nodes due to its allowance for nodes to move from susceptible to infected AND back to susceptible as opposed to remaining infected or being removed. This will therefore require many iterations to converge (if at all) as nodes will continue to be re-infected without an option to be removed.

Part 2. Alternate threshold model

In this part, you are going to make some modifications to the threshold model to create a new model and compare it with the original version.

Q17. (10 points, Manually graded) The Volatile Threshold model class

In the below cell, an incomplete class `VolatileThreshold` is provided. The diffusion rule for the volatile threshold model is the following:

> In each iteration, only a random subset of `n` neighbors for each node `v` is evaluated. If the ratio of infection is ***at or above*** the threshold of `v` *within the subset*, then node `v` will become infected. A parameter `sample` determines the number of neighbors to sample.

Based on this specification of the model, complete the `iteration` method. Please do not make any changes to the rest of the class.

Notes

1. If the sample size is ***larger*** than the number of neighbors a node has, just take all its neighbors as a sample.
2. You can use the function `random.sample(original_data, sample_size)` to get a sample of size `sample_size` from collection `original_data` .
3. `class` object: in this part, we introduce a class object to construct our `VolatileThreshold` model. In Python, each class maintains its member attributes and methods with the `self` keyword. For example, `self.G` , `self.config` are simply `G` and `config` and belong to a specific instance of this class. You can see more in this tutorial (https://www.w3schools.com/python/python_classes.asp).

```
In [28]: from tqdm import tqdm
         class VolatileThreshold:
             def __init__(self, graph):
                 self.G = graph
                 self.config = None
                 self.status = {n: 0 for n in graph.nodes}
                 self.threshold = {n: 0 for n in graph.nodes}
                 self.num_sample = 0  # since the graph is connected
                 self.N = len(graph.nodes)

             def set_initial_status(self, config):
                 self.config = config
                 # set threshold
                 thred = config.__dict__['config']['nodes']['threshold']
                 for n in self.G.nodes:
                     self.threshold[n] = thred[n]
                 # set number of samples
                 self.num_sample = config.__dict__['config']['model']['num_sample']
                 # set seed nodes
                 if 'fraction_infected' in config.__dict__['config']['model']:
                         seed_nodes = random.sample(
                             self.G.nodes(), int(config.__dict__['config']['model']['fraction_infected'] * len(self.G.noc
                 else:
                     seed_nodes = config.__dict__['config']['status']['Infected']
                 for n in seed_nodes:
                     self.status[n] = 1

             def iteration(self):
                 num_infected_nodes_total = 0  # number of infected node (with status 1)
                 tmp = {n: self.status[n] for n in self.G.nodes} #current status of all nodes
                 for n in self.G.nodes:
                     if self.status[n] == 0:

                         num_infected_nodes_in_sample = 0 # initiate a counter for n's infected neighbors

                         neighbors = list(self.G.neighbors(n))
                         if len(neighbors) > self.num_sample:
                             # get a sample of n's neighbors of size self.num_sample
                             neighbors = list(random.sample(neighbors, self.num_sample))

                         # count the number of infected neighbors in the sampled neighbors using the statuses stored in t
                         for neighbor in neighbors:
```

```
                    if tmp[neighbor] == 1:
                        num_infected_nodes_in_sample += 1

                # compute the ratio of infected nodes in the sampled neighbors
                neighbor_ratio = num_infected_nodes_in_sample/len(neighbors)

                # if the ratio is AT OR ABOVE n's threshold (self.threshold[n]), the node becomes infected.
                # in this case, update self.status[n] to 1 and increase the count of total infected nodes by 1
                if neighbor_ratio >= self.threshold[n]:
                    self.status[n] = 1
                    num_infected_nodes_total += 1
            else:
                num_infected_nodes_total += 1
        return num_infected_nodes_total

    def iteration_bunch(self, bunch_size):
        results = []
        num_infected_nodes_total = 0
        for n in self.G.nodes:
            num_infected_nodes_total += self.status[n]
        results.append({0: self.N - num_infected_nodes_total, 1: num_infected_nodes_total})
        for i in tqdm(range(bunch_size-1)):
            num_infected_nodes_total = self.iteration()
            results.append({0: self.N - num_infected_nodes_total, 1: num_infected_nodes_total})
        return results
```

Q18. (7 points, Autograded) Seed nodes comparison

Complete the following function with the given signature, so that it simulates a diffusion process with the volatile threshold model and returns a list of infected number in each iteration.

In [29]:
```
def simulate_volatile(G, importance_measure=None, iterate=100, n=1, threshold=0.5, sample=5):
    if importance_measure:
        # select seed nodes
        sorted_node = sorted(importance_measure(G).items(), key=operator.itemgetter(1))[::-1]
        highest_nodes = [n for n, _ in sorted_node[:n]]

    # Model selection
    model = VolatileThreshold(G)
    random.seed(0)

    config = mc.Configuration()

    if importance_measure:
        config.add_model_initial_configuration("Infected", highest_nodes)

    else:
        config.add_model_parameter('fraction_infected', float(n)/len(G.nodes))

    for i in G.nodes():
        config.add_node_configuration("threshold", i, threshold)

    config.add_model_parameter('num_sample', sample)

    model.set_initial_status(config)

    # Simulation execution
    iterations = model.iteration_bunch(iterate)
    return [it[1] for it in iterations]
```

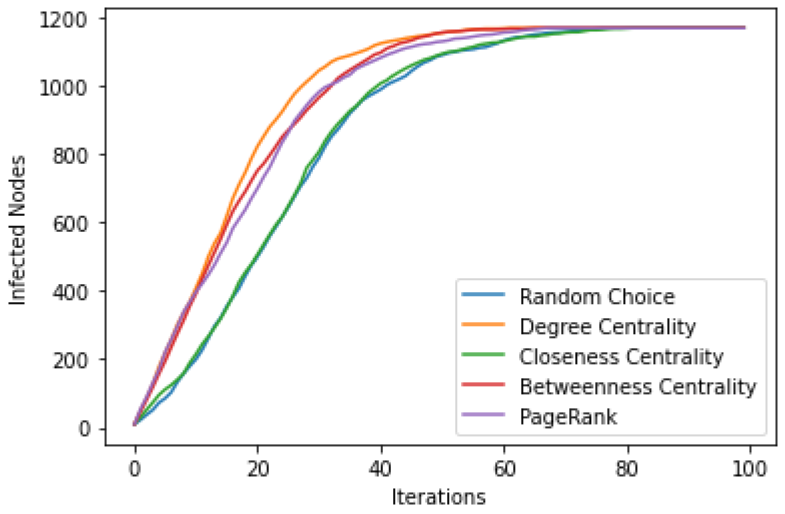
In [30]:
```
#hidden tests for Question 18 are within this cell
```

In [31]:
```
#hidden tests for Question 18 are within this cell
```

Q19. (12 points, Manually graded) Using parameters `I=100`, `N=10`, `T=0.5`, create a plot of the diffusion process (number of currently infected nodes at each iteration) when the sample size $S = [1, 2, 3, 4, 5, 100]$, one in each cell. As we have done in previous questions, pick the seed nodes using the 5 different strategies: random, degree, closeness, betweenness, pagerank.

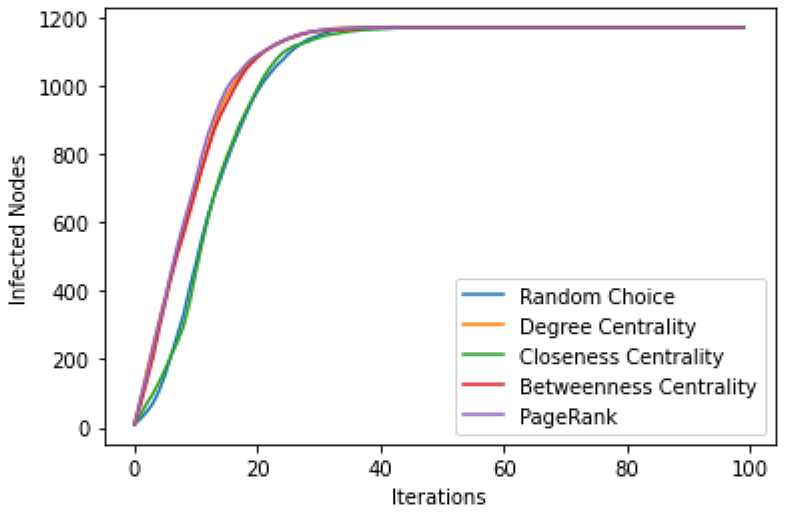
```
In [32]: I=100
N=10
T = 0.5
S = 1

node_count_random = simulate_volatile(G, importance_measure=None, iterate= I, n = N, threshold = T, sample = S)
node_count_deg = simulate_volatile(G, importance_measure=nx.degree_centrality, iterate= I, n = N, threshold = T,
node_count_closeness = simulate_volatile(G, importance_measure=nx.closeness_centrality, iterate= I, n = N, thres
node_count_betweenness = simulate_volatile(G, importance_measure=nx.betweenness_centrality, iterate= I, n = N, t
node_count_pagerank = simulate_volatile(G, importance_measure=nx.pagerank, iterate= I, n = N, threshold = T, sam



x = range(I)

plt.plot(x, node_count_random, label = "Random Choice")
plt.plot(x, node_count_deg, label = "Degree Centrality")
plt.plot(x, node_count_closeness, label = "Closeness Centrality")
plt.plot(x, node_count_betweenness, label = "Betweenness Centrality")
plt.plot(x, node_count_pagerank, label = "PageRank")

plt.legend()
plt.xlabel('Iterations')
plt.ylabel('Infected Nodes')
plt.show()
```

```
100%|██████████| 99/99 [00:00<00:00, 1096.03it/s]
100%|██████████| 99/99 [00:00<00:00, 1514.61it/s]
100%|██████████| 99/99 [00:00<00:00, 1141.70it/s]
100%|██████████| 99/99 [00:00<00:00, 1441.44it/s]
100%|██████████| 99/99 [00:00<00:00, 1407.94it/s]
```


Infected Nodes
Iterations
Random Choice
Degree Centrality
Closeness Centrality
Betweenness Centrality
PageRank
0
200
400
600
800
1000
1200
0
20
40
60
80
100

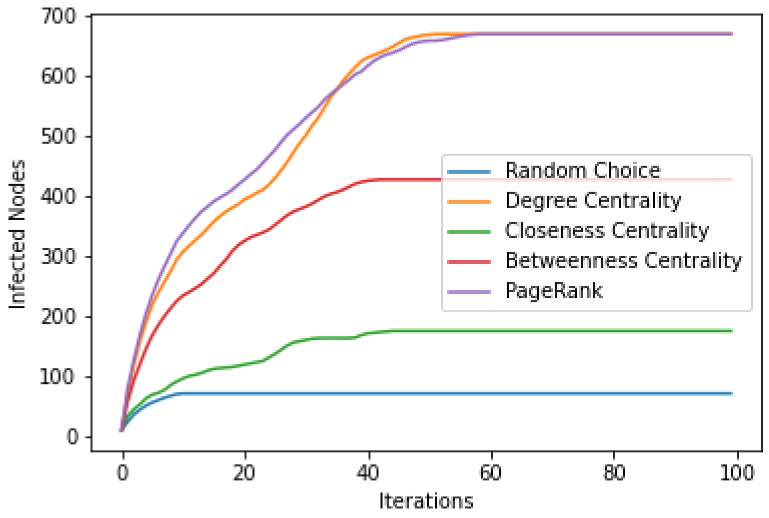
```
In [33]: I=100
         N=10
         T = 0.5
         S = 2

         node_count_random = simulate_volatile(G, importance_measure=None, iterate= I, n = N, threshold = T, sample = S)
         node_count_deg = simulate_volatile(G, importance_measure=nx.degree_centrality, iterate= I, n = N, threshold = T,
         node_count_closeness = simulate_volatile(G, importance_measure=nx.closeness_centrality, iterate= I, n = N, thres
         node_count_betweenness = simulate_volatile(G, importance_measure=nx.betweenness_centrality, iterate= I, n = N, t
         node_count_pagerank = simulate_volatile(G, importance_measure=nx.pagerank, iterate= I, n = N, threshold = T, sam



         x = range(I)

         plt.plot(x, node_count_random, label = "Random Choice")
         plt.plot(x, node_count_deg, label = "Degree Centrality")
         plt.plot(x, node_count_closeness, label = "Closeness Centrality")
         plt.plot(x, node_count_betweenness, label = "Betweenness Centrality")
         plt.plot(x, node_count_pagerank, label = "PageRank")

         plt.legend()
         plt.xlabel('Iterations')
         plt.ylabel('Infected Nodes')
         plt.show()
```

```
100%|██████████| 99/99 [00:00<00:00, 2811.37it/s]
100%|██████████| 99/99 [00:00<00:00, 3158.10it/s]
100%|██████████| 99/99 [00:00<00:00, 2793.38it/s]
100%|██████████| 99/99 [00:00<00:00, 3183.89it/s]
100%|██████████| 99/99 [00:00<00:00, 3243.80it/s]
```


1200
1000
800
600
400
200
0
Infected Nodes
0
20
40
60
80
100
Iterations
Random Choice
Degree Centrality
Closeness Centrality
Betweenness Centrality
PageRank

In [34]:
```
I=100
N=10
T = 0.5
S = 3

node_count_random = simulate_volatile(G, importance_measure=None, iterate= I, n = N, threshold = T, sample = S)
node_count_deg = simulate_volatile(G, importance_measure=nx.degree_centrality, iterate= I, n = N, threshold = T,
node_count_closeness = simulate_volatile(G, importance_measure=nx.closeness_centrality, iterate= I, n = N, thres
node_count_betweenness = simulate_volatile(G, importance_measure=nx.betweenness_centrality, iterate= I, n = N, t
node_count_pagerank = simulate_volatile(G, importance_measure=nx.pagerank, iterate= I, n = N, threshold = T, sam



x = range(I)

plt.plot(x, node_count_random, label = "Random Choice")
plt.plot(x, node_count_deg, label = "Degree Centrality")
plt.plot(x, node_count_closeness, label = "Closeness Centrality")
plt.plot(x, node_count_betweenness, label = "Betweenness Centrality")
plt.plot(x, node_count_pagerank, label = "PageRank")

plt.legend()
plt.xlabel('Iterations')
plt.ylabel('Infected Nodes')
plt.show()
```

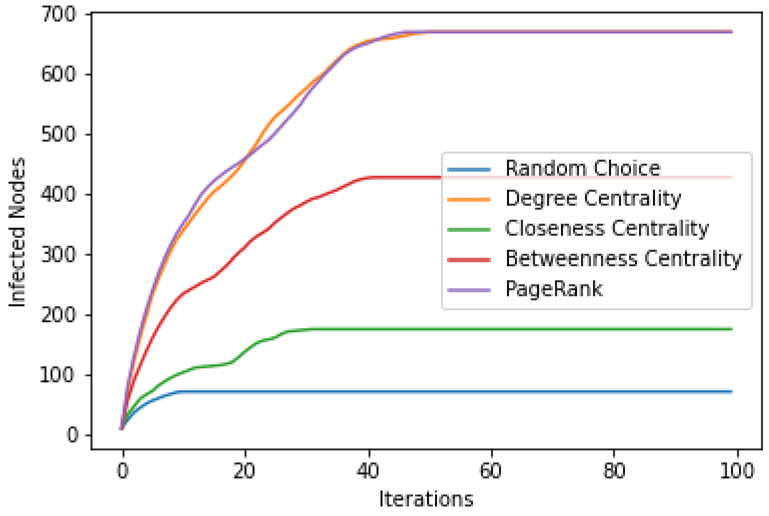
```
100%|██████████| 99/99 [00:00<00:00, 810.90it/s]
100%|██████████| 99/99 [00:00<00:00, 1285.46it/s]
100%|██████████| 99/99 [00:00<00:00, 884.12it/s]
100%|██████████| 99/99 [00:00<00:00, 1085.31it/s]
100%|██████████| 99/99 [00:00<00:00, 1313.28it/s]
```



700
600
500
400
300
200
100
0
Infected Nodes
0
20
40
60
80
100
Iterations
Random Choice
Degree Centrality
Closeness Centrality
Betweenness Centrality
PageRank

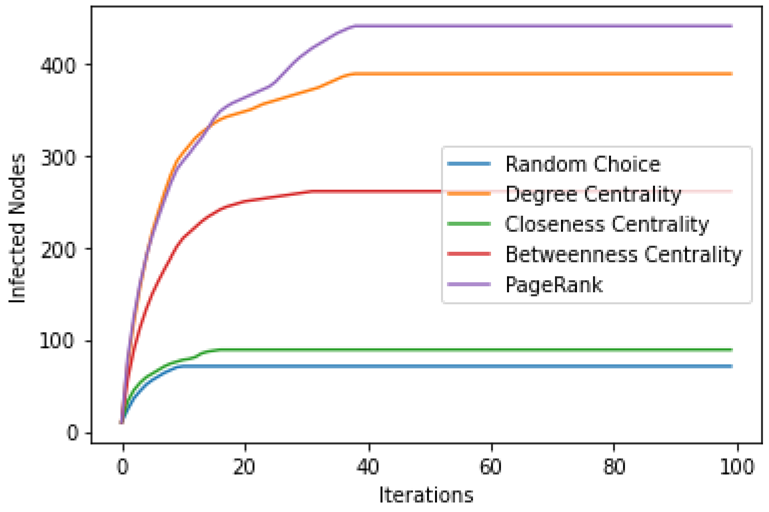
```
In [35]: I=100
         N=10
         T = 0.5
         S = 4

         node_count_random = simulate_volatile(G, importance_measure=None, iterate= I, n = N, threshold = T, sample = S)
         node_count_deg = simulate_volatile(G, importance_measure=nx.degree_centrality, iterate= I, n = N, threshold = T,
         node_count_closeness = simulate_volatile(G, importance_measure=nx.closeness_centrality, iterate= I, n = N, thres
         node_count_betweenness = simulate_volatile(G, importance_measure=nx.betweenness_centrality, iterate= I, n = N, t
         node_count_pagerank = simulate_volatile(G, importance_measure=nx.pagerank, iterate= I, n = N, threshold = T, sam



         x = range(I)

         plt.plot(x, node_count_random, label = "Random Choice")
         plt.plot(x, node_count_deg, label = "Degree Centrality")
         plt.plot(x, node_count_closeness, label = "Closeness Centrality")
         plt.plot(x, node_count_betweenness, label = "Betweenness Centrality")
         plt.plot(x, node_count_pagerank, label = "PageRank")

         plt.legend()
         plt.xlabel('Iterations')
         plt.ylabel('Infected Nodes')
         plt.show()
```

```
100%|██████████| 99/99 [00:00<00:00, 905.33it/s]
100%|██████████| 99/99 [00:00<00:00, 1458.22it/s]
100%|██████████| 99/99 [00:00<00:00, 1021.64it/s]
100%|██████████| 99/99 [00:00<00:00, 1221.80it/s]
100%|██████████| 99/99 [00:00<00:00, 1471.59it/s]
```


Infected Nodes
Iterations
Random Choice
Degree Centrality
Closeness Centrality
Betweenness Centrality
PageRank

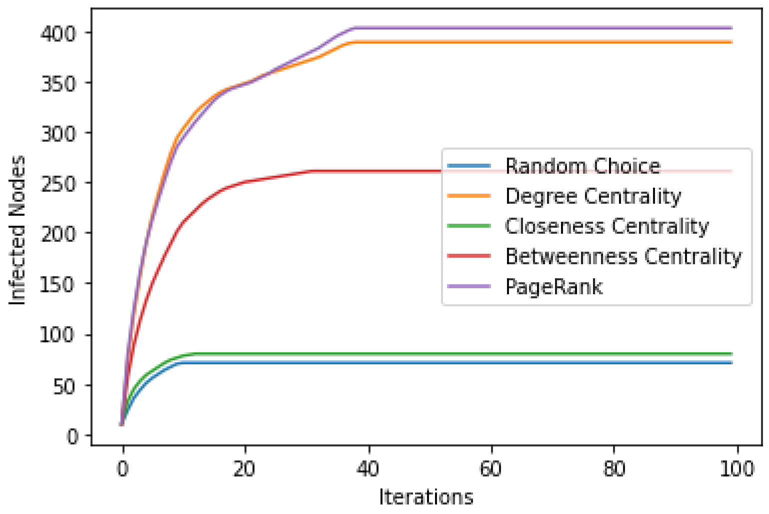
In [36]:
```python
I=100
N=10
T = 0.5
S = 5

node_count_random = simulate_volatile(G, importance_measure=None, iterate= I, n = N, threshold = T, sample = S)
node_count_deg = simulate_volatile(G, importance_measure=nx.degree_centrality, iterate= I, n = N, threshold = T,
node_count_closeness = simulate_volatile(G, importance_measure=nx.closeness_centrality, iterate= I, n = N, thres
node_count_betweenness = simulate_volatile(G, importance_measure=nx.betweenness_centrality, iterate= I, n = N, t
node_count_pagerank = simulate_volatile(G, importance_measure=nx.pagerank, iterate= I, n = N, threshold = T, sam



x = range(I)

plt.plot(x, node_count_random, label = "Random Choice")
plt.plot(x, node_count_deg, label = "Degree Centrality")
plt.plot(x, node_count_closeness, label = "Closeness Centrality")
plt.plot(x, node_count_betweenness, label = "Betweenness Centrality")
plt.plot(x, node_count_pagerank, label = "PageRank")

plt.legend()
plt.xlabel('Iterations')
plt.ylabel('Infected Nodes')
plt.show()
```

```
100%|██████████| 99/99 [00:00<00:00, 926.37it/s]
100%|██████████| 99/99 [00:00<00:00, 1197.68it/s]
100%|██████████| 99/99 [00:00<00:00, 964.66it/s]
100%|██████████| 99/99 [00:00<00:00, 1067.76it/s]
100%|██████████| 99/99 [00:00<00:00, 1232.49it/s]
```


Random Choice
Degree Centrality
Closeness Centrality
Betweenness Centrality
PageRank
Infected Nodes
Iterations
0
100
200
300
400
0
20
40
60
80
100

```
In [37]: I=100
         N=10
         T = 0.5
         S = 100

         node_count_random = simulate_volatile(G, importance_measure=None, iterate= I, n = N, threshold = T, sample = S)
         node_count_deg = simulate_volatile(G, importance_measure=nx.degree_centrality, iterate= I, n = N, threshold = T,
         node_count_closeness = simulate_volatile(G, importance_measure=nx.closeness_centrality, iterate= I, n = N, thres
         node_count_betweenness = simulate_volatile(G, importance_measure=nx.betweenness_centrality, iterate= I, n = N, t
         node_count_pagerank = simulate_volatile(G, importance_measure=nx.pagerank, iterate= I, n = N, threshold = T, sam



         x = range(I)

         plt.plot(x, node_count_random, label = "Random Choice")
         plt.plot(x, node_count_deg, label = "Degree Centrality")
         plt.plot(x, node_count_closeness, label = "Closeness Centrality")
         plt.plot(x, node_count_betweenness, label = "Betweenness Centrality")
         plt.plot(x, node_count_pagerank, label = "PageRank")

         plt.legend()
         plt.xlabel('Iterations')
         plt.ylabel('Infected Nodes')
         plt.show()
```

```
100%|██████████| 99/99 [00:00<00:00, 1013.47it/s]
100%|██████████| 99/99 [00:00<00:00, 1259.47it/s]
100%|██████████| 99/99 [00:00<00:00, 1039.04it/s]
100%|██████████| 99/99 [00:00<00:00, 1150.81it/s]
100%|██████████| 99/99 [00:00<00:00, 1262.20it/s]
```



Q20. (8 points, Manually graded) Based on your results in the Q19, given the same set of seed nodes and a sample size within $[0, 5]$, how does the total number of infected nodes change as you increase the number of sampled neighbors? Why?

Hint: Note that we using a relatively high T of 0.5. Think about what this implies about the chances that a node becomes infected when using a large vs. a small sample size.

As the number of sampled neighors increases, the total number of infected nodes decreases. With a relatively high threshold of 0.5, it is much "easier" for a node to become infected when only 1 neighbor is sampled relative to 5. When 1 or 2 neighbors are sampled, only 1 needs to be infected to infect the node being evaluated. Whereas if 5 neighbors are sampled, 3 neighbors must be infected to turn the evaluated node. Therefore, as sample size increases it becomes more difficult to meet the required threshold, leading to the trend seen in Q19.

End

```
In [ ]:
```