Imports

```
# Imports
import wikipedia
import pandas as pd
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
import time
from tqdm import tqdm
import re
import requests
from bs4 import BeautifulSoup
import seaborn as sns
```

Functions

```
# From https://stackoverflow.com/questions/18916616/get-first-link-of-wikipedia-article-us
def isValid(ref, paragraph):
   if not ref or "#" in ref or "//" in ref or ":" in ref:
      return False
   if "/wiki/" not in ref:
      return False
   if ref not in paragraph:
      return False
   prefix = paragraph.split(ref,1)[0]
   if prefix.count("(")!=prefix.count(")"):
      return False
   return True
```

```
help_link_regex = re.compile("^/wiki/Help:")


# Gets the "first link" from a wikipedia page using the full URL
# Help from https://stackoverflow.com/questions/54170998/crawl-the-first-paragraph-link-in
def get_first_link(page_url, help_link_regex=help_link_regex):
    # Get page
    response = requests.get(page_url)
    html = response.text
```

```python
soup = BeautifulSoup(html, "html.parser")

# Many pages lack a first link or have a strange structure requiring a lot of try stat
try:
    # Paragraphs of the page
    paragraphs = (
        soup.find(id="mw-content-text").find(class_="mw-parser-output").find_all("p",
    )

    # Find all links in the first paragraph
    if len(paragraphs) > 5:
        filtered_paragaphs = [p for p in paragraphs[:4] if len(p) > 1]
    else:
        filtered_paragaphs = [p for p in paragraphs if len(p) > 1]

    link_list = []

    for p in filtered_paragaphs:
        p_links = p.find_all("a")

        # Leave them out if they match the previous RegEx
        for link in p_links:
            if not help_link_regex.match(link.get("href")):
                href = str(link.get("href"))
                link_class = str(link.get("class"))
                # print(link_class)

                # Pages that start with # wrongly redirect to the homepage
                if href.startswith('#'):
                    href = href.lstrip("#")

                # Filters out bad links in wikipedia pages
                if (
                    href != None
                    and not href.endswith('.ogg')
                    and not href.endswith("Wikipedia:Please_clarify")
                    and "#cite_note" not in href
                    and "upload.wikimedia.org" not in href
                    and link_class != "['extiw']"
                    and not href.startswith("file:")
                    and not href.startswith("https://geohack.toolforge.org/")
```

```python
                            and not href.startswith("special:")
                            and not href.startswith("/w/index.php?")
                            and not href.startswith("cite_note")
                            and isValid(href, str(p))
                        ):
                            link_list.append(href)
                            if len(link_list) > 3:
                                break

        try:
            first_link = link_list[0]

            # Cleaning the name
            if '/wiki/' in first_link:
                first_link_output = first_link.split("/wiki/", 1)[1]
            else:
                first_link_output = first_link
            return first_link_output

        # If it has no links, it is disconnected
        except IndexError:
            # NO LINKS
            original_name = page_url.split("https://en.wikipedia.org/wiki/", 1)[1]
            return original_name

    # These errors get manually checked
    except (AttributeError, TypeError) as error:
        # NO LINKS
        fail_output = "!FAIL!: " + page_url
        return fail_output
```

```python
# Testing the function. It should be 'Private_university'

get_first_link("https://en.wikipedia.org/wiki/georgetown_university")
```

```
'Private_university'
```

```python
help_link_regex = re.compile("^/wiki/Help:")
```

```python
# Gets the "second link" from a wikipedia page using the full URL
# The same as the first link function it just gets the second link
# Help from https://stackoverflow.com/questions/54170998/crawl-the-first-paragraph-link-in
def get_second_link(page_url, help_link_regex=help_link_regex):
    # Get page
    response = requests.get(page_url)
    html = response.text
    soup = BeautifulSoup(html, "html.parser")

    # Many pages lack a first link or have a strange structure requiring a lot of try stat
    try:
        # Paragraphs of the page
        paragraphs = (
            soup.find(id="mw-content-text").find(class_="mw-parser-output").find_all("p",
        )

        # Find all links in the first paragraph
        if len(paragraphs) > 5:
            filtered_paragaphs = [p for p in paragraphs[:4] if len(p) > 1]
        else:
            filtered_paragaphs = [p for p in paragraphs if len(p) > 1]

        link_list = []

        for p in filtered_paragaphs:
            p_links = p.find_all("a")

            # Leave them out if they match the previous RegEx
            for link in p_links:
                if not help_link_regex.match(link.get("href")):
                    href = str(link.get("href"))
                    link_class = str(link.get("class"))
                    # print(link_class)

                    # Pages that start with # wrongly redirect to the homepage
                    if href.startswith('#'):
                        href = href.lstrip("#")

                    # Filters out bad links in wikipedia pages
                    if (
                        href != None
```

```
                        and not href.endswith('.ogg')
                        and not href.endswith("Wikipedia:Please_clarify")
                        and "#cite_note" not in href
                        and "upload.wikimedia.org" not in href\
                        # and link_class != "['mw-redirect']" # Redirects don't work since
                        and link_class != "['extiw']"
                        and not href.startswith("file:")
                        and not href.startswith("https://geohack.toolforge.org/")
                        and not href.startswith("special:")
                        and not href.startswith("/w/index.php?")
                        and not href.startswith("cite_note")
                        and isValid(href, str(p))
                    ):
                        link_list.append(href)

                        if len(link_list) > 3:
                            break
            try:
                first_link = link_list[1]

                # Cleaning the name
                if '/wiki/' in first_link:
                    first_link_output = first_link.split("/wiki/", 1)[1]
                else:
                    first_link_output = first_link
                return first_link_output

            # If it has no links, it is disconnected
            except IndexError:
                # NO LINKS
                original_name = page_url.split("https://en.wikipedia.org/wiki/", 1)[1]
                return original_name

    # These errors get manually checked
    except (AttributeError, TypeError) as error:
        # NO LINKS
        fail_output = "!FAIL!: " + page_url
        return fail_output
```

```
# Testing the function. It should be 'Society_of_Jesus'
get_second_link("https://en.wikipedia.org/wiki/georgetown_university")
```

```
'Society_of_Jesus'
```

```python
# Function that computes how far the average page is from the Philosophy page
import statistics
def avg_dist_from_phil(G):
    distance_from_phil = []

    # Get nodes connected to "philosophy"
    connected_nodes = [node for node in G.nodes if nx.has_path(G, node, "philosophy")]

    for node in connected_nodes:

        shortest_path_length = nx.shortest_path_length(G, node, "philosophy")
        distance_from_phil.append(shortest_path_length)

    # Return the mean
    return statistics.mean(distance_from_phil)
```

```python
# Function that finds a new random page on Wikipedia
def wiki_random_page(seen_pages):

    # Set up while loop that searches
    already_seen_page = True

    while already_seen_page:

        # Use the wikipedia api to find a random page
        random_page = wikipedia.random()

        # If it has already seen the page, go back up and try again
        if random_page in seen_pages:
            already_seen_page = True
            continue

        # Ignoring pages that are just lists of other pages and disambiguation because the
        elif "list of " in random_page.lower() or "list_of" in random_page.lower():
            already_seen_page = True

        elif '(disambiguation)' in random_page.lower():
            already_seen_page = True
```

```python
        # If it is a new page, return the page and its url
        else:
            page_url = "https://en.wikipedia.org/wiki/" + random_page.replace(" ", "_")
            already_seen_page = False

    return random_page, page_url
```

```python
# Testing the function
seen_pages = ['philosophy']
wiki_random_page(seen_pages)
```

```
('Air shuttle', 'https://en.wikipedia.org/wiki/Air_shuttle')
```

Creating the First-Link Network

```python
# Create Graph
G = nx.DiGraph()

# Add Philosophy as the first node
G.add_node("philosophy")

# Starting page, can be anything. I think Georgetown is a fun starting point but wikipedia
seed_page = "Georgetown University"
seed_page = seed_page.replace(" ", "_").lower()
is_root = True

# These start as the same thing
root_page = seed_page

# Create first url
first_page_url = "https://en.wikipedia.org/wiki/" + seed_page
old_page = seed_page

# List of pages already hit. This avoids finding the same paths for pages we already know.
seen_pages = ["philosophy"]

# List of fails
fails = []
```

```
# Not connected to philosophy
disconnects = []

# Create a dataframe that will be used to demonstrate the convergence of important pages
convergence_df = pd.DataFrame(columns=["iteration", "node", "betweeness_centrality", "clos
```

```
def gcc_fraction(G):
    # Get all strongly connected components as subgraphs
    sccs = list(nx.weakly_connected_components(G))

    sorted_scc = sorted(sccs, key=len, reverse=True)
    # Find the largest strongly connected component (GCC)
    gcc_nodes = sorted_scc[0]

    # Create a subgraph containing only nodes from the GCC
    gcc = G.subgraph(gcc_nodes)

    fraction = (len(gcc.nodes()) / len(G.nodes()))

    return fraction
```

```
# Function to expand the network
# New pages is the number of random pages that will be added to the network. All of the pa
def network_expander(
    G, page_url, seen_pages, is_root, fails, disconnects, convergence_df, new_pages=100
):
    # If the DF already exists, this will be used later to ensure the iteration value is a
    start_iteration = len(convergence_df.index)

    # From experience, these are the most notable pages I want to focus on. If their centr
    notable_nodes = [
        "philosophy",
        "awareness",
        "knowledge",
        "science",
        "language",
        "philosophy_of_logic",
        "county_(united_states)"
    ]

    # Remove the link extra to get a cleaner page name
```

```
root_page = page_url.split("https://en.wikipedia.org/wiki/", 1)[1]

# Pretty display of function progress using TQDM
# Help from https://stackoverflow.com/questions/57473107/how-to-set-the-r-bar-part-of-
for i in tqdm(
    range(new_pages),
    desc="Finding Paths",
    bar_format="{l_bar}{bar}| {n_fmt}/{total_fmt}",
    colour="Green",
    ncols=75,
):
    # Get the first link
    # print(page_url)
    full_first_link = get_first_link(page_url=page_url)

    if full_first_link is None:
        if is_root:
            root_page, page_url = wiki_random_page(seen_pages)
            continue

        # If it has nodes connected to it, add it to a list of fails to be manually fi
        else:
            fails.append(page_url)
            # print(full_first_link)
            root_page, page_url = wiki_random_page(seen_pages)
            full_first_link = get_first_link(page_url=page_url)
            is_root = True
            continue

    # Get the cleaner name for the node
    first_link = full_first_link.lower()

    # Accounts for this page redirect
    if first_link == "philosophical":
        first_link = "philosophy"

    # If it fails, find try a new random page
    if "!FAIL!: " in str(full_first_link):
        # We don't need to care if it was a root page
        if is_root:
            root_page, page_url = wiki_random_page(seen_pages)
```

```
            continue

        # If it as nodes connected to it, add it to a list of fails to be manually fix
        else:
            fails.append(first_link)
            # print(full_first_link)
            root_page, page_url = wiki_random_page(seen_pages)
            full_first_link = get_first_link(page_url=page_url)
            is_root = True
            continue

    elif "DEAD END: " in str(full_first_link):
        # print("DISCONNECTED NODE AT:", root_page)
        disconnects.append(first_link)

        # New root
        root_page, page_url = wiki_random_page(seen_pages)
        is_root = True
        continue

    # Add node
    G.add_node(first_link)
    # Add edge if there is one
    if not is_root:
        G.add_edge(old_page, first_link)

    # Every 1/100th of the total length and once all of the notable nodes have been hi
    if (i % (new_pages / 100) == 0 and all(
        [True if node in G.nodes else False for node in notable_nodes]
    )) or i==new_pages:
        # Calculate values
        new_between_cent = {
            node: val
            for node, val in nx.betweenness_centrality(
                G, endpoints="philosophy", normalized=True
            ).items()
            if node in notable_nodes
        }
        new_closeness_cent = {
            node: val
            for node, val in nx.closeness_centrality(G).items()
```

```python
            if node in notable_nodes
        }
        new_in_degree_cent = {
            node: val
            for node, val in nx.in_degree_centrality(G).items()
            if node in notable_nodes
        }

        # Calculate average distance
        new_avg_dist = avg_dist_from_phil(G)

        # Size of GCC
        size_of_gcc = gcc_fraction(G)

        # Function may take existing list, calculated at beginning of function
        iteration = start_iteration + i

        # Each node gets a row, these get filtered by the hue in the plots
        for node in notable_nodes:
            new_row = [
                iteration,
                node,
                new_between_cent[node],
                new_closeness_cent[node],
                new_in_degree_cent[node],
                new_avg_dist,
                size_of_gcc,
            ]
            convergence_df.loc[len(convergence_df.index)] = new_row

    # If it is philosophy, new root page
    if first_link.lower() == "philosophy":
        root_page, page_url = wiki_random_page(seen_pages)
        is_root = True

    # Page did a self loop! Not connected to philosophy
    elif first_link == root_page:
        # print("DISCONNECTED NODE AT:", root_page)
        disconnects.append([root_page, first_link])

        # New root
```

```python
            root_page, page_url = wiki_random_page(seen_pages)
            is_root = True

        # If we have already seen where it goes, new root page
        elif first_link in seen_pages:
            root_page, page_url = wiki_random_page(seen_pages)
            is_root = True

        # Keep going until a known page is hit
        else:
            page_url = "https://en.wikipedia.org/wiki/" + full_first_link
            seen_pages.append(first_link)
            old_page = first_link
            is_root = False

    # Return key values
    return G, seen_pages, fails, disconnects, convergence_df
```

```python
# Running the function
G, seen_pages, fails, disconnects, convergence_df = network_expander(
    G,
    page_url=first_page_url,
    seen_pages=seen_pages,
    is_root=True,
    fails=fails,
    disconnects=disconnects,
    convergence_df=convergence_df,
    new_pages=10000,
)
```

```
Finding Paths: 100%|                       | 10000/10000
```

```python
# Viewing the meaningful fails. These are very rare (about 1 for every 100000 seed pages)
fails = [fail for fail in fails if fail != None]
fails
```

```
['!fail!: https://en.wikipedia.org/wiki/central_district_(hirmand_county)',
 '!fail!: https://en.wikipedia.org/wiki/women_in_film_and_television_international']
```

```python
# Viewing the Convergence DF
convergence_df
```

	iteration	node	betweeness_centrality	closeness_centrality	in_degree_centrality
0	500	philosophy	0.002372	0.101888	0.012658
1	500	awareness	0.003957	0.091424	0.005063
2	500	knowledge	0.004213	0.080678	0.010127
3	500	science	0.002755	0.049210	0.015190
4	500	language	0.001068	0.014097	0.005063
...	...	...	...	...	...
779	9984	knowledge	0.000189	0.052397	0.000987
780	9984	science	0.000130	0.045598	0.002303
781	9984	language	0.000028	0.007655	0.001755
782	9984	philosophy_of_logic	0.000017	0.008454	0.000110
783	9984	county_(united_states)	0.000012	0.007848	0.007128

```python
# Function to clean failed pages
def fail_fixer(G, fail, next_link, seen_pages):
    clean_name = fail.lstrip("!fail!: ")
    G = nx.relabel_nodes(G, {fail:clean_name})
    G.add_node(next_link)
    G.add_edge(clean_name, next_link)

    page_url = "https://en.wikipedia.org/wiki/" + next_link

    if next_link != "philosophy":
        philosophy_page = False
        old_page = next_link
    else: philosophy_page = True

    while not philosophy_page:
        full_first_link = get_first_link(page_url=page_url)
        # Get the cleaner name for the node
        first_link = full_first_link.lstrip('/wiki/').lower()
        # print(first_link)

        # Add node
        G.add_node(first_link)
        G.add_edge(old_page, first_link)
```

```
        # If it is philosophy, new root page
        if first_link.lower() == "philosophy":
            philosophy_page = True

        # If we have already seen where it goes, new root page
        elif first_link in seen_pages:
            philosophy_page = True

        # Keep going until a known page is hit
        else:
            page_url = "https://en.wikipedia.org/" + full_first_link
            seen_pages.append(first_link)
            old_page = first_link

    # Making sure it worked
    try:
        nx.shortest_path(G, fail, "philosophy")

    except nx.NodeNotFound:
        print("FAILED")

    return G, seen_pages
```

```
G, seen_pages = fail_fixer(G, fail='!fail!: crime city', next_link="crime_film", seen_page
```

```
# Saving the network and DataFrame
first_link_path = './data/first-links-10000.gml'
nx.write_gml(G, first_link_path)

convergence_df.to_csv("./data/convergence_data-10000.csv")
```

Second Link Functions

```
# Create Second-Link Graph
G2 = nx.DiGraph()

# Starting page, can be anything. I think Georgetown is a fun starting point but wikipedia
seed_page = "Georgetown University"
seed_page = seed_page.replace(" ", "_").lower()
```

```
is_root = True

# Adding first node
G2.add_node(seed_page)

# These start as the same thing
root_page = seed_page

# Create first url
first_page_url = "https://en.wikipedia.org/wiki/" + seed_page
old_page = seed_page

# List of pages already hit. This avoids finding the same paths for pages we already know.
second_link_seen_pages = [seed_page]

# List of fails
second_link_fails = []

# Dead ends: pages with no second link
second_link_disconnects = []

# Create a dataframe that will be used to demonstrate the convergence of important pages
# This one does not have an average distance from philosophy as there is no guarantee we e
convergence_df2 = pd.DataFrame(columns=["iteration", "node", "betweeness_centrality", "clo
```

Creating the Second Link Network

```
import heapq
# Function to expand the network
# New pages is the number of random pages that will be added to the network. All of the pa
def second_link_network_expander(G, page_url, seen_pages, is_root, fails, disconnects, con

    # If the DF already exists, this will be used later to ensure the iteration value is a
    start_iteration = len(convergence_df.index)

    # Remove the link extra to get a cleaner page name
    root_page = page_url.split("https://en.wikipedia.org/wiki/", 1)[1]

    # Pretty display of function progress
    # Help from https://stackoverflow.com/questions/57473107/how-to-set-the-r-bar-part-of-
```

```
for i in tqdm(range(new_pages), desc="Finding Paths", bar_format="{l_bar}{bar}| {n_fmt
    first_page = page_url.split("https://en.wikipedia.org/wiki/", 1)[1]
    pages_hit = [first_page]

    # Get the second link
    # print(page_url)
    full_second_link = get_second_link(page_url=page_url)

    if full_second_link is None:
        if is_root:
            root_page, page_url = wiki_random_page(seen_pages)
            continue

        # If it has nodes connected to it, add it to a list of fails to be manually fi
        else:
            fails.append(full_second_link)
            # print(full_second_link)
            root_page, page_url = wiki_random_page(seen_pages)
            full_second_link = get_second_link(page_url=page_url)
            is_root = True
            continue

     # Get the cleaner name for the node
    second_link = full_second_link.lower()

    # If it fails, find try a new random page
    if "!FAIL!: " in str(full_second_link):
        # We don't need to care if it was a root page
        if is_root:
            root_page, page_url = wiki_random_page(seen_pages)
            continue

        # If it as nodes connected to it, add it to a list of fails to be manually fix
        else:
            fails.append(second_link)
            # print(full_second_link)
            root_page, page_url = wiki_random_page(seen_pages)
            full_second_link = get_second_link(page_url=page_url)
            is_root = True
            continue
```

```python
        elif "DEAD END: " in str(full_second_link):
            # New root
            root_page, page_url = wiki_random_page(seen_pages)
            is_root = True
            continue

        # Add node
        G.add_node(second_link)
        # Add edge if there is one
        if not is_root:
            G.add_edge(old_page, second_link)

        # Every 100 iterations check the top 5 in key statistics
        if i >=3000 and (i % (new_pages / 100) == 0) or i == new_pages:

            # Calculate values
            new_between_cent = {node: val for node, val in nx.betweenness_centrality(G, en
            new_closeness_cent = {node: val for node, val in nx.closeness_centrality(G).it
            new_in_degree_cent = {node: val for node, val in nx.in_degree_centrality(G).it

            # Get the top 5 nodes for each centrality measure
            top_3_between = heapq.nlargest(3, new_between_cent, key=new_between_cent.get)
            top_3_closeness = heapq.nlargest(3, new_closeness_cent, key=new_closeness_cent
            top_3_in_degree = heapq.nlargest(3, new_in_degree_cent, key=new_in_degree_cent

            # Combine all top 5 nodes into a single list
            notable_nodes = top_3_between + top_3_closeness + top_3_in_degree

            # Remove duplicates by converting to a set and back to a list
            notable_nodes = list(set(notable_nodes))

            # Function may take existing list, calculated at beginning of function
            iteration = start_iteration + i

            # Size of GCC
            size_of_gcc = gcc_fraction(G)

            # Each node gets a row, these get filtered by the hue in the plots
            for node in notable_nodes:
                new_row = [iteration, node, new_between_cent[node], new_closeness_cent[nod
                convergence_df.loc[len(convergence_df.index)] = new_row
```

```
        # Page did a loop! No need to keep repeating it
        if second_link in pages_hit:
            disconnects.append(root_page)

            # New root
            root_page, page_url = wiki_random_page(seen_pages)
            is_root = True

        # If we have already seen where it goes, new root page
        elif second_link in seen_pages:
            root_page, page_url = wiki_random_page(seen_pages)
            is_root = True

        # Keep going until a known page is hit
        else:
            page_url = "https://en.wikipedia.org/wiki/" + full_second_link
            seen_pages.append(second_link)
            pages_hit.append(second_link)
            old_page = second_link
            is_root = False

    # Return key values
    return G, seen_pages, fails, disconnects, convergence_df
```

```
# Running the function
G2, second_link_seen_pages, second_link_fails, second_link_disconnects, convergence_df2 =
    G2,
    page_url=first_page_url,
    seen_pages=second_link_seen_pages,
    is_root=True,
    fails=second_link_fails,
    disconnects=second_link_disconnects,
    convergence_df=convergence_df2,
    new_pages=10000,
)
```

```
Finding Paths: 100%|                         | 10000/10000
```

```
# Saving the network and DataFrame
second_link_path = './data/second-links-10000.gml'
```

```
nx.write_gml(G2, second_link_path)

convergence_df.to_csv("./data/second-link-convergence-data-10000.csv")
```

Analysis

First Link Convergence

```
# Reading the data in
first_link_path = './data/first-links-33000.gml'
G = nx.read_gml(first_link_path)

convergence_df = pd.read_csv("./data/convergence_data-33000.csv")
convergence_df.pop("Unnamed: 0")
convergence_df.tail()
```

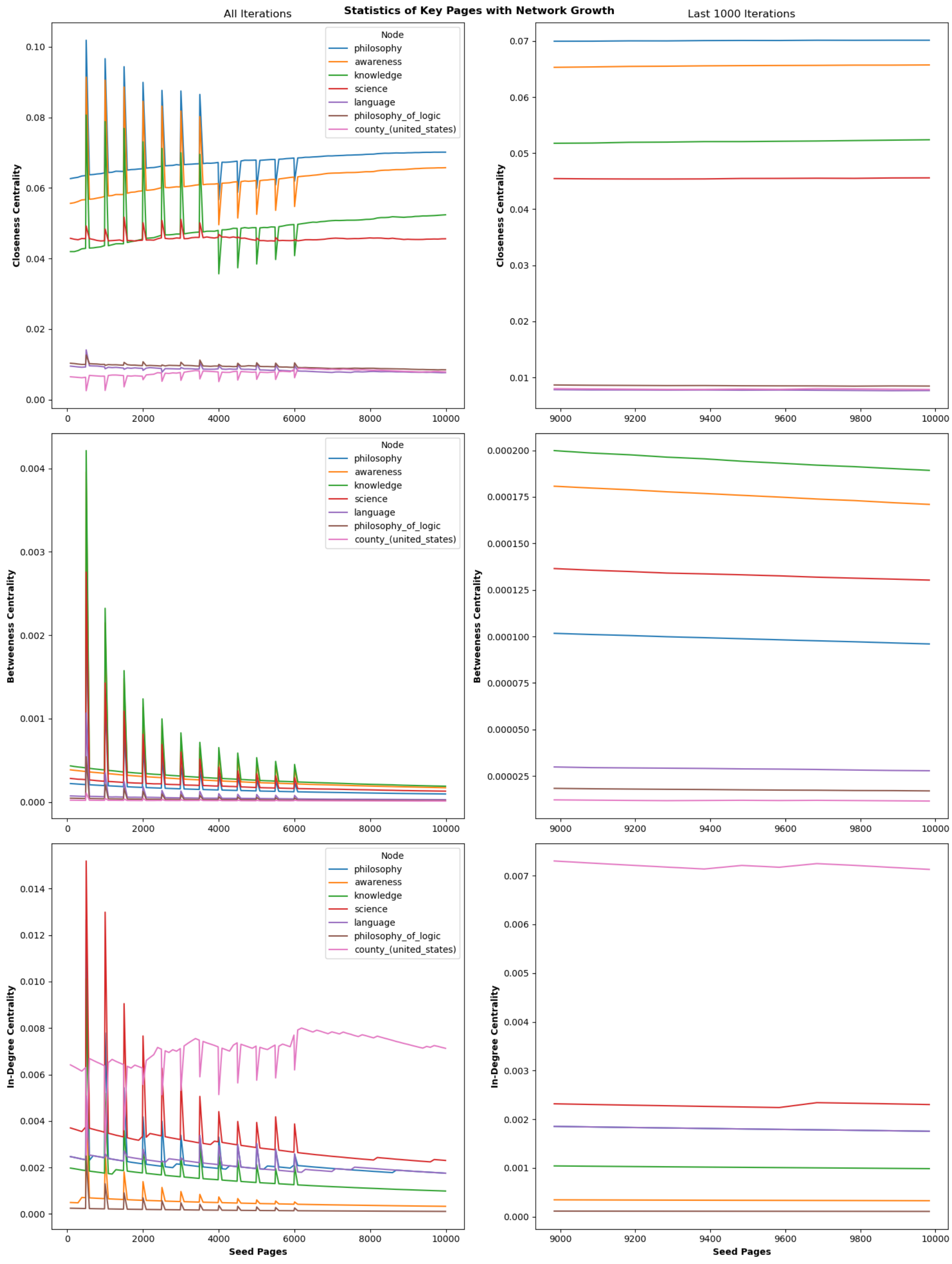
	iteration	node	betweeness_centrality	closeness_centrality	in_degree	avg_dist_
1987	33300	awareness	0.000093	0.076282	0.000120	10.977308
1988	33300	knowledge	0.000103	0.066760	0.000662	10.977308
1989	33300	science	0.000076	0.047266	0.001986	10.977308
1990	33300	language	0.000015	0.007144	0.001264	10.977308
1991	33300	philosophy_of_logic	0.000009	0.007965	0.000060	10.977308

```
total_iterations = max(convergence_df['iteration'])
last_1000_iterations = convergence_df[convergence_df["iteration"] >= total_iterations - 10
last_1000_iterations.reset_index(inplace=True)
last_1000_iterations.head()
```

	index	iteration	node	betweeness_centrality	closeness_centrality	in_degree_centrality	avg_
0	707	8984	philosophy	0.000102	0.069980	0.001854	12.5
1	708	8984	awareness	0.000181	0.065319	0.000348	12.5
2	709	8984	knowledge	0.000200	0.051769	0.001043	12.5
3	710	8984	science	0.000136	0.045473	0.002318	12.5
4	711	8984	language	0.000030	0.007805	0.001854	12.5

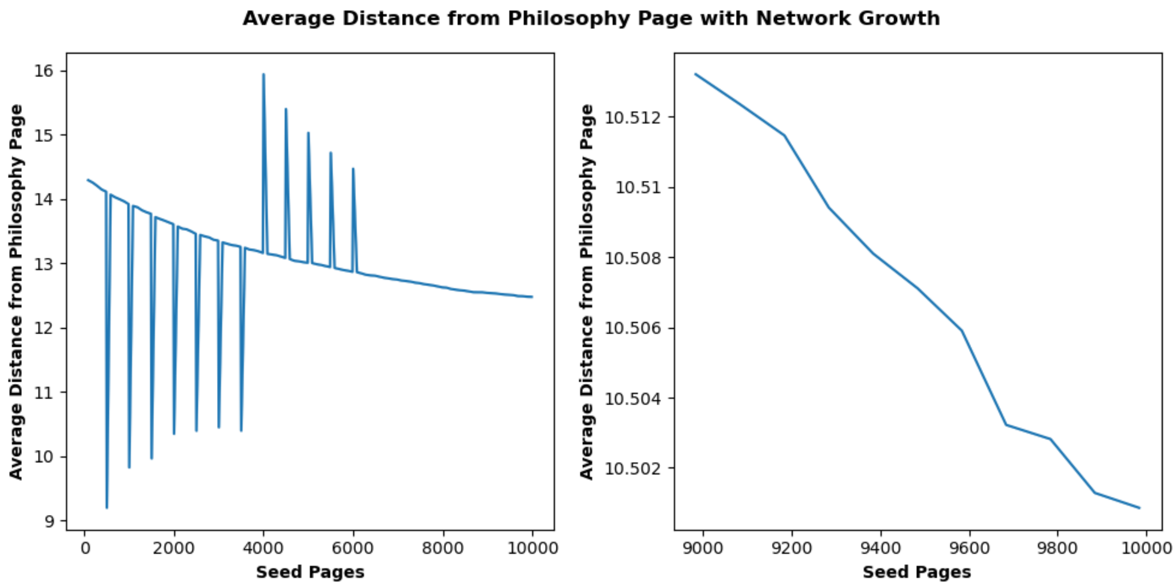
```
fig, axs = plt.subplots(3, 2, figsize=(15, 20), dpi=100)
sns.lineplot(convergence_df, x="iteration", y="closeness_centrality", hue="node", ax=axs[0
axs[0,0].set_ylabel("Closeness Centrality", fontweight='bold')
axs[0,0].set_xlabel("")
axs[0,0].set_title("All Iterations")

legend = axs[0,0].legend()
legend.set_title("Node")

sns.lineplot(last_1000_iterations, x="iteration", y="closeness_centrality", hue="node", ax
axs[0,1].set_ylabel("Closeness Centrality", fontweight='bold')
axs[0,1].set_xlabel("")
axs[0,1].set_title("Last 1000 Iterations")

sns.lineplot(convergence_df, x="iteration", y="betweeness_centrality", hue="node", ax=axs[
axs[1,0].set_ylabel("Betweeness Centrality", fontweight='bold')
axs[1,0].set_xlabel("")

legend = axs[1,0].legend()
legend.set_title("Node")


sns.lineplot(last_1000_iterations, x="iteration", y="betweeness_centrality", hue="node", a
axs[1,1].set_ylabel("Betweeness Centrality", fontweight='bold')
axs[1,1].set_xlabel("")

sns.lineplot(convergence_df, x="iteration", y="in_degree_centrality", hue="node", ax=axs[2
axs[2,0].set_ylabel("In-Degree Centrality", fontweight='bold')
axs[2,0].set_xlabel("Seed Pages", fontweight='bold')

legend = axs[2,0].legend()
legend.set_title("Node")

sns.lineplot(last_1000_iterations, x="iteration", y="in_degree_centrality", hue="node", ax
axs[2,1].set_ylabel("In-Degree Centrality", fontweight='bold')
axs[2,1].set_xlabel("Seed Pages", fontweight='bold')


plt.suptitle("Statistics of Key Pages with Network Growth", fontweight='bold')
plt.tight_layout()
plt.savefig("../images/first-link-centrality-convergence.png")
```

```
plt.show()
```


Statistics of Key Pages with Network Growth
All Iterations
Last 1000 Iterations
Node
philosophy
awareness
knowledge
science
language
philosophy_of_logic
county_(united_states)
Closeness Centrality
Betweeness Centrality
In-Degree Centrality
Seed Pages

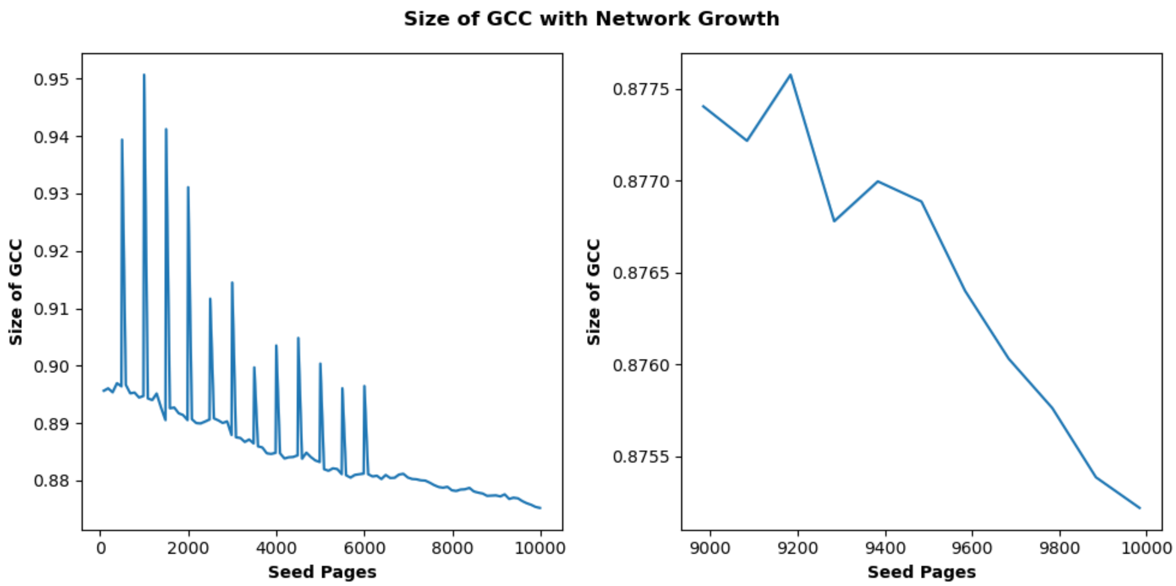
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
sns.lineplot(convergence_df, x="iteration", y="avg_dist_from_phil", ax=ax1)
ax1.set_ylabel("Average Distance from Philosophy Page", fontweight="bold")
ax1.set_xlabel("Seed Pages", fontweight="bold")


sns.lineplot(last_1000_iterations, x="iteration", y="avg_dist_from_phil", ax=ax2, legend=F
ax2.set_ylabel("Average Distance from Philosophy Page", fontweight="bold")
ax2.set_yticklabels([round(n, 3) for n in list(np.arange(start=10.5, stop=11.5, step=0.002
ax2.set_xlabel("Seed Pages", fontweight="bold")

plt.suptitle("Average Distance from Philosophy Page with Network Growth", fontweight="bold
plt.tight_layout()
plt.savefig("../images/first-link-dist-from-phil.png")
plt.show()
```

```
/var/folders/hg/dd3yfd8j7vx8qtmvm42400j80000gn/T/ipykernel_30233/3745581654.py:9: UserWarning
  ax2.set_yticklabels([round(n, 3) for n in list(np.arange(start=10.5, stop=11.5, step=0.002)
```




```
import seaborn as sns
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5), dpi=100)
sns.lineplot(convergence_df, x="iteration", y="size_of_gcc", ax=ax1)
```

```
ax1.set_ylabel("Size of GCC", fontweight="bold")
ax1.set_xlabel("Seed Pages", fontweight="bold")


sns.lineplot(last_1000_iterations, x="iteration", y="size_of_gcc", ax=ax2, legend=False)
ax2.set_ylabel("Size of GCC", fontweight="bold")
ax2.set_xlabel("Seed Pages", fontweight="bold")


plt.suptitle("Size of GCC with Network Growth", fontweight="bold")
plt.tight_layout()
plt.savefig("../images/first-link-gcc-convergence.png")
plt.show()
```



Second Link Convergence

```
# Reading the data in
second_link_path = './data/second-links-10000.gml'
G2 = nx.read_gml(second_link_path)

convergence_df2 = pd.read_csv("./data/second-link-convergence-data-10000.csv")
convergence_df2.pop("Unnamed: 0")
```

```
convergence_df2.head()
```

	iteration	node	betweeness_centrality	closeness_centrality	in_degree	avg_dist_from_phil
0	100	philosophy	0.010870	0.125000	0.021978	7.913043
1	100	awareness	0.015647	0.101372	0.021978	7.913043
2	100	knowledge	0.017439	0.092742	0.032967	7.913043
3	100	science	0.013736	0.060672	0.021978	7.913043
4	100	language	0.007406	0.019536	0.010989	7.913043

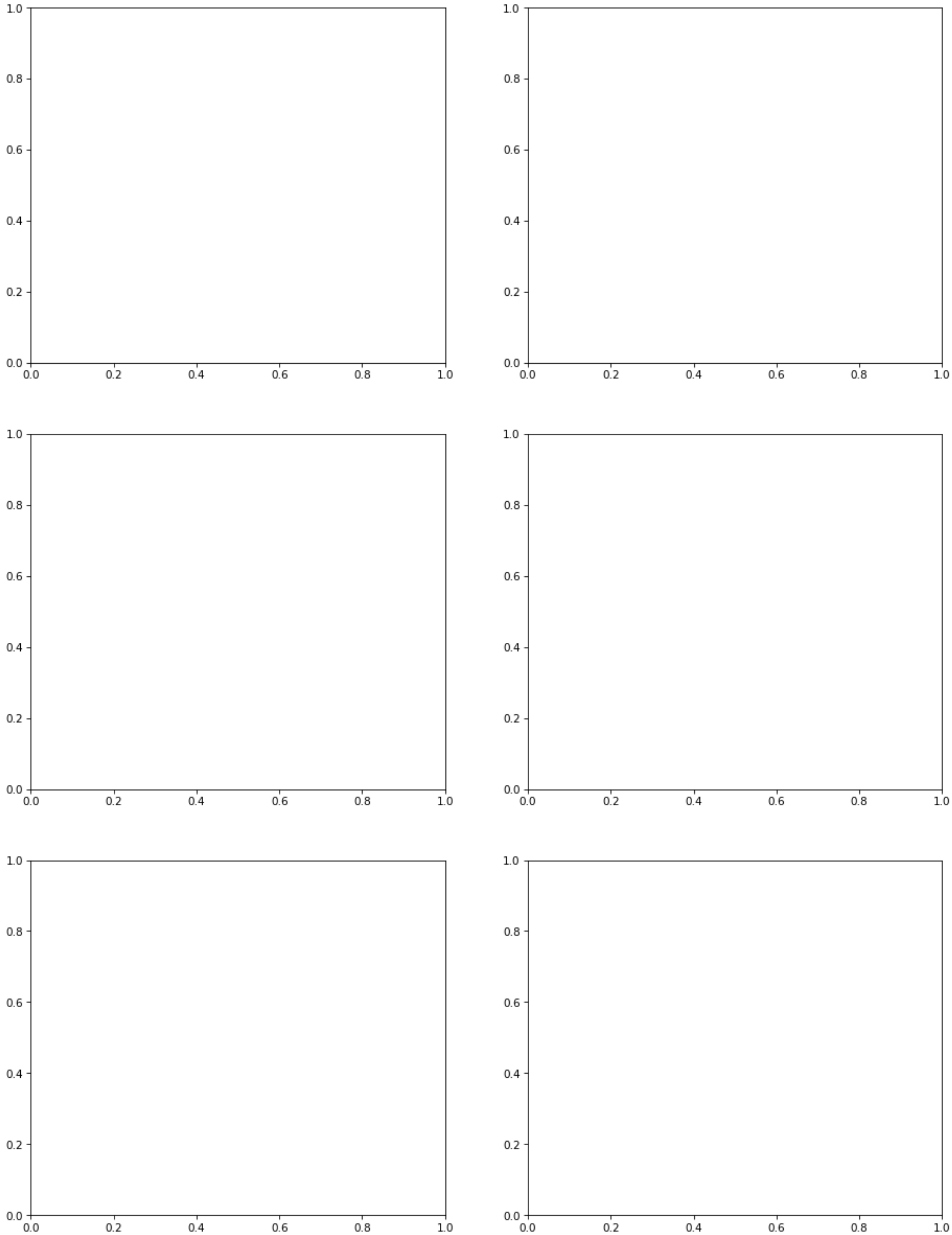
```
second_link_total_iterations = max(convergence_df2['iteration'])
second_link_last_1000_iterations = convergence_df2[convergence_df2["iteration"] >= second_
second_link_last_1000_iterations.reset_index(inplace=True)
second_link_last_1000_iterations.head()
```

	index	iteration	node	betweeness_centrality	closeness_centrality	in_degree	avg_dist_from_
0	528	8900	philosophy	0.000164	0.083839	0.002028	10.625491
1	529	8900	awareness	0.000292	0.080971	0.000553	10.625491
2	530	8900	knowledge	0.000321	0.070403	0.001660	10.625491
3	531	8900	science	0.000232	0.049862	0.003688	10.625491
4	532	8900	language	0.000054	0.008615	0.002766	10.625491

```
fig, axs = plt.subplots(3, 2, figsize=(15, 20), dpi=75)
sns.lineplot(convergence_df2, x="iteration", y="closeness_centrality", hue="node", ax=axs[
axs[0,0].set_ylabel("Closeness Centrality", fontweight='bold')
axs[0,0].set_xlabel("")
axs[0,0].set_title("All Iterations")

legend = axs[0,0].legend()
legend.set_title("Node")

sns.lineplot(second_link_last_1000_iterations, x="iteration", y="closeness_centrality", hu
axs[0,1].set_ylabel("Closeness Centrality", fontweight='bold')
axs[0,1].set_xlabel("")
axs[0,1].set_title("Last 1000 Iterations")

sns.lineplot(convergence_df2, x="iteration", y="betweeness_centrality", hue="node", ax=axs
axs[1,0].set_ylabel("Betweeness Centrality", fontweight='bold')
axs[1,0].set_xlabel("")
```

```
sns.lineplot(second_link_last_1000_iterations, x="iteration", y="betweeness_centrality", h
axs[1,1].set_ylabel("Betweeness Centrality", fontweight='bold')
axs[1,1].set_xlabel("")

sns.lineplot(convergence_df2, x="iteration", y="in_degree_centrality", hue="node", ax=axs[
axs[2,0].set_ylabel("In-Degree Centrality", fontweight='bold')
axs[2,0].set_xlabel("Seed Pages", fontweight='bold')

sns.lineplot(second_link_last_1000_iterations, x="iteration", y="in_degree_centrality", hu
axs[2,1].set_ylabel("In-Degree Centrality", fontweight='bold')
axs[2,1].set_xlabel("Seed Pages", fontweight='bold')


plt.suptitle("Statistics of Key Pages with Network Growth", fontweight='bold')
plt.tight_layout()
plt.savefig("../images/second-link-centrality-convergence.png")
plt.show()
```

```
NameError: name 'convergence_df2' is not defined
```



```
degree_dict = dict(G.in_degree())

# Create a dataframe with nodes and their degrees
df_nodes = pd.DataFrame.from_dict(degree_dict, orient='index', columns=['Degree'])

# Sort the dataframe by degree in descending order
df_nodes_sorted = df_nodes.sort_values(by='Degree', ascending=False)

df_nodes_sorted.head(10)
```

	Degree
county_(united_states)	125
association_football	116
public_university	94
u.s._state	91
family_(biology)	84
capital_city	67
rural_districts_of_iran	56
administrative_division	52
united_states	52
india	52

```
def fail_fixer(G, fail, next_link, seen_pages):
    clean_name = fail.lstrip("!fail!: ")
    G = nx.relabel_nodes(G, {fail:clean_name})
    G.add_node(next_link)
    G.add_edge(clean_name, next_link)

    page_url = "https://en.wikipedia.org/wiki/" + next_link

    if next_link != "philosophy":
        philosophy_page = False
        old_page = next_link
    else: philosophy_page = True

    while not philosophy_page:
        full_first_link = get_first_link(page_url=page_url)
        # Get the cleaner name for the node
        first_link = full_first_link.lstrip('/wiki/').lower()
        # print(first_link)
```

```python
        # Add node
        G.add_node(first_link)
        G.add_edge(old_page, first_link)


        # If it is philosophy, new root page
        if first_link.lower() == "philosophy":
            philosophy_page = True

        # If we have already seen where it goes, new root page
        elif first_link in seen_pages:
            philosophy_page = True

        # Keep going until a known page is hit
        else:
            page_url = "https://en.wikipedia.org/" + full_first_link
            seen_pages.append(first_link)
            old_page = first_link

    # fails = fails[True if f[0]!=fail else False for f in fails]

    return G, seen_pages
```

```python
# Calculate the degrees of each node
degrees = dict(G.degree())

# Sort the nodes by degree (in descending order)
sorted_nodes = sorted(degrees.items(), key=lambda x: x[1], reverse=True)

# Print the top nodes by degree
print("Top nodes by degree:")
for node, degree in sorted_nodes[:10]:
    print(f"{node}: {degree}")
```

```
Top nodes by degree:
county_(united_states): 126
association_football: 117
public_university: 95
u.s._state: 92
family_(biology): 85
capital_city: 68
```

```
rural_districts_of_iran: 57
administrative_division: 53
united_states: 53
india: 53
```

```
# Calculate the degrees of each node
in_degree = dict(nx.in_degree_centrality(G))

# Sort the nodes by degree (in descending order)
sorted_nodes = sorted(in_degree.items(), key=lambda x: x[1], reverse=True)

# Print the top nodes by degree
print("Top nodes by in_degree_centrality:")
for node, degree in sorted_nodes[:10]:
    print(f"{node}: {degree}")
```

```
Top nodes by in_degree:
county_(united_states): 0.007504803073967339
association_football: 0.006964457252641691
public_university: 0.005643611911623439
u.s._state: 0.005463496637848223
family_(biology): 0.005043227665706052
capital_city: 0.004022574447646494
rural_districts_of_iran: 0.0033621517771373678
administrative_division: 0.003121998078770413
united_states: 0.003121998078770413
india: 0.003121998078770413
```

```
neighbors = [n[0] for n in list(G.in_edges("philosophy"))]
neighbor_degrees = [G.in_edges(n) for n in neighbors]
neighbors
```

```
['awareness',
 'philosophy_of_logic',
 'philosophical_tradition',
 'political_philosophy',
 'philosophy_of_culture',
 'aesthetics',
 'metaphysics',
 'specialty_(medicine)',
```

```
 'outline_of_philosophy',
 'modernism',
 'medical_specialty',
 'american_enlightenment',
 'object_(philosophy)',
 'ethics',
 'raphael_woolf',
 'philosophy_of_science',
 'modernist',
 'public_benefit']
```

```
neighbors = [n[0] for n in list(G.in_edges("association_football"))]
neighbor_degrees = [G.in_edges(n) for n in neighbors]
neighbors
```

```
['wrexham_f.c.',
 'saff_u-20_championship',
 'football_federation_tasmania',
 'football_in_france',
 'college_soccer',
 'midfielder',
 'campeonato_carioca_s%c3%a9rie_b1',
 'national_soccer_league',
 'azerbaijan_cup',
 'vancouver_whitecaps_fc',
 'manager_(association_football)',
 'football_club_(association_football)',
 'torneio_rio-s%c3%a3o_paulo',
 'tie_cup',
 'wessex_football_league',
 'list_of_football_clubs_in_spain',
 'eliteserien',
 'marc-vivien_fo%c3%a9',
 'fifa_world_cup',
 'soccer-specific_stadium',
 'russian_professional_football_league',
 'bundesliga',
 'guam_league',
 'football_in_austria',
 'copa_am%c3%a9rica',
 'gr%c3%aamio_de_esportes_maring%c3%a1',
```

```
'rochdale_a.f.c.',
'forward_(association_football)',
'league_of_ireland',
'uefa_european_under-17_championship',
'millwall_f.c.',
'beach_soccer',
'utility_player',
'uefa_champions_league',
'football_in_germany',
'minifootball',
'kategoria_e_par%c3%ab',
'fifa_u-17_world_cup',
'bologna_f.c._1909',
'liberian_premier_league',
'uefa_europa_conference_league',
'football_in_iran',
'kawasaki_frontale',
'luxembourg_national_division',
'vegalta_sendai',
'highland_football_league',
'hungarian_football_association',
'super_cup',
'the_football_league',
'%c3%9arvalsdeild',
'council_of_southern_africa_football_associations',
'russian_second_division',
'blaxnit_cup',
'superta%c3%a7a_c%c3%a2ndido_de_oliveira',
'swindon_town_f.c.',
'scottish_league_cup',
'indonesia_national_football_team',
'british_home_championship',
'list_of_association_football_clubs_in_the_republic_of_ireland',
'inter-cities_fairs_cup',
'c.d._santa_clara',
'football_at_the_pacific_games',
'campeonato_brasileiro_s%c3%a9rie_a',
'aston_villa',
'uefa',
'sunderland_a.f.c.',
'concacaf_nations_league',
'sanfrecce_hiroshima',
'central_coast_mariners_fc',
```

```
'toronto_fc_ii',
'football_in_scotland',
'chinese_super_league',
'northern_ireland_football_league',
'newcastle_jets_fc',
'concacaf_men%27s_olympic_qualifying_championship',
'football_in_sweden',
'fa_trophy',
'fc_ryukyu',
'suruga_bank_championship',
'usl_championship',
'myanmar_national_league',
'celtic_f.c.',
'west_midlands_(regional)_league',
's.l._benfica',
'mestaruussarja',
'league_cup',
'queen%27s_park_f.c.',
'scottish_professional_football_league',
'football_in_italy',
'dfb-pokal',
'italian_football',
'uefa_euro_2016_qualifying',
'2016%e2%80%9317_uefa_champions_league',
'ta%c3%a7a_de_portugal',
'lists_of_association_football_players',
'irish_cup',
'jordan_football_association',
'fleetwood_town_f.c.',
'1924_in_association_football',
'kategoria_superiore',
'dundee_f.c.',
'afc_asian_cup',
'association_football_positions',
'mexican_football_federation',
'tottenham_hotspur_f.c.',
'usl_premier_development_league',
'northern_football_league',
'sc_waterloo_region',
'nk_kr%c5%a1ko',
'j._league_cup',
'yorkshire_football_league',
'french_football_federation',
```

```
 'derby_county_f.c.',
 'football_in_india',
 'football_at_the_2012_summer_olympics',
 'professional_soccer']
```

```python
# Calculate the degrees of each node
betweeness_centrality = dict(nx.betweenness_centrality(G))

# Sort the nodes by degree (in descending order)
sorted_nodes = sorted(betweeness_centrality.items(), key=lambda x: x[1], reverse=True)

# Print the top nodes by degree
print("Top nodes by betweeness centrality:")
for node, betweeness_centrality in sorted_nodes[:10]:
    print(f"{node}: {betweeness_centrality}")
```

```
Top nodes by betweeness centrality:
knowledge: 6.849895791607439e-05
science: 5.713293253404184e-05
state_(polity): 5.228443772312088e-05
physics: 4.894636620273961e-05
politics: 4.7136742117176546e-05
mind: 4.6906033070411536e-05
psychology: 4.655636467140831e-05
awareness: 4.629681699379767e-05
branches_of_science: 4.565155262862678e-05
natural_science: 4.286141509431242e-05
```

```python
# Calculate the degrees of each node
closeness_centrality = dict(nx.closeness_centrality(G))

# Sort the nodes by degree (in descending order)
sorted_nodes = sorted(closeness_centrality.items(), key=lambda x: x[1], reverse=True)

# Print the top nodes by degree
print("Top nodes by closeness centrality:")
for node, closeness_centrality in sorted_nodes[:10]:
    print(f"{node}: {closeness_centrality}")
```

```
Top nodes by closeness centrality:
```

```
philosophy: 0.07873499527885429
awareness: 0.07622244898013505
knowledge: 0.06670369616253845
science: 0.04724344565043322
geography: 0.02658103153692291
continent: 0.024049358981052222
mind: 0.02218398510337653
branches_of_science: 0.021724793727557496
psychology: 0.020631970363685345
state_(polity): 0.02052677766193913
```

```
distance_from_phil = []
path_to_phil = []
nodes = []
for node in G.nodes:
    try:
        shortest_path = nx.shortest_path(G, node, "philosophy")
        shortest_path_length = nx.shortest_path_length(G, node, "philosophy")
        nodes.append(node)
        distance_from_phil.append(shortest_path_length)
        path_to_phil.append(shortest_path)
    except nx.NetworkXNoPath:
        shortest_path = np.NAN
        shortest_path_length = np.NAN
        continue


# nodes  = [n for n in G.nodes]
df = pd.DataFrame(
    {"node": nodes, "distance": distance_from_phil, "shortest_path": path_to_phil}
)
print("AVERAGE DISTANCE TO PHILOSOPHY: ", np.mean(distance_from_phil))
df.sort_values("distance", inplace=True)
df.reset_index(inplace=True)
df.pop("index")
nodes = df["node"]
distances = df["distance"]
paths = df["shortest_path"]
print(
    "FURTHEST NODE: ",
    nodes[len(nodes) - 1],
```

```
        " is ",
        distances[len(distances) - 1],
        " pages away with a path of: ",
        paths[len(distances) - 1],
    )
    print(
        "THERE IS/ARE "
        + str(
            sum([True if distance == max(distances) else False for distance in distances])
        )
        + " PATH(S) WITH A DISTANCE OF "
        + str(max(distances))
    )
```

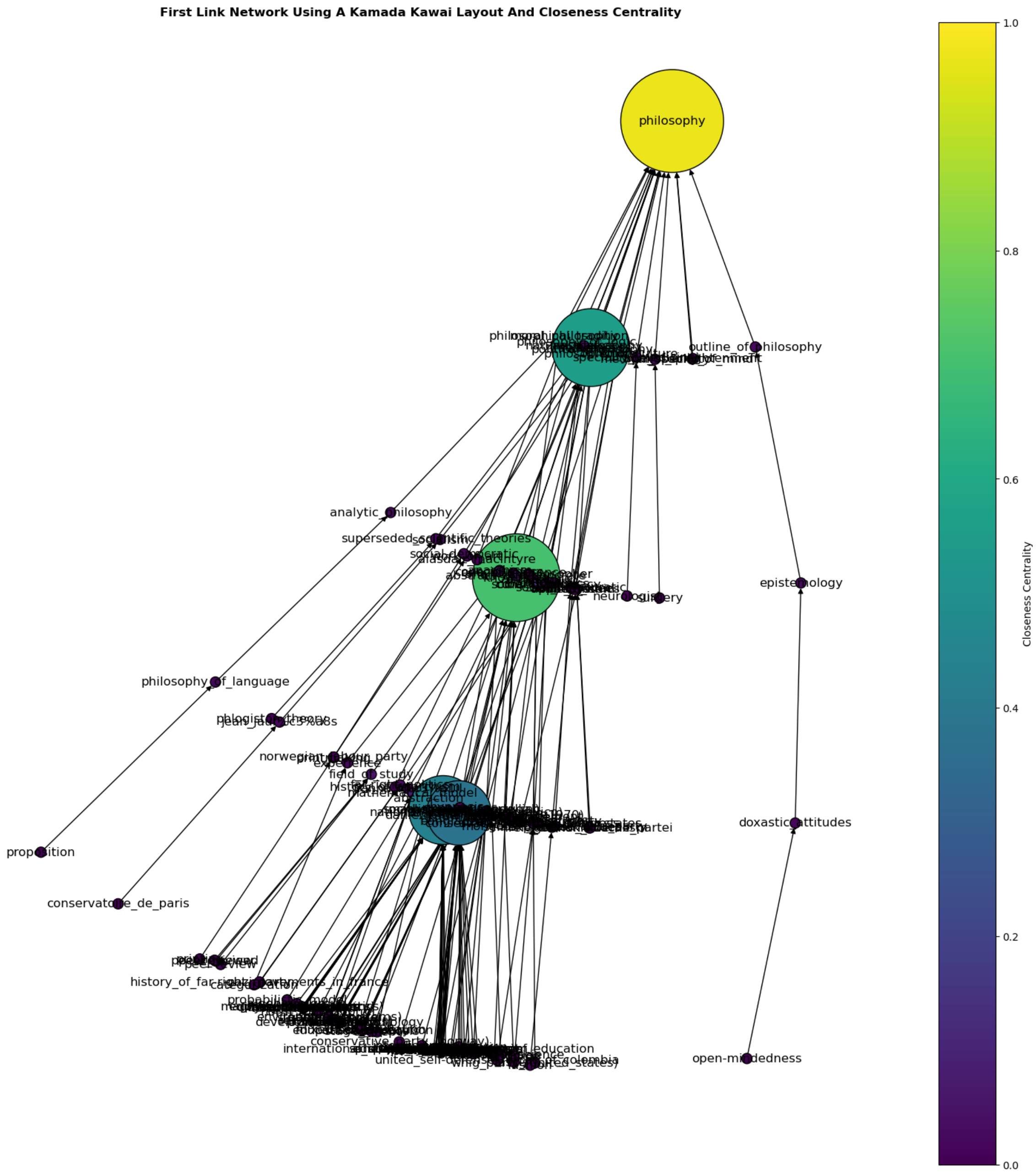
```
AVERAGE DISTANCE TO PHILOSOPHY:  12.462381664175386
FURTHEST NODE:  meshir_13  is  175  pages away with a path of:  ['meshir_13', 'meshir_12', 'r
THERE IS/ARE 1 PATH(S) WITH A DISTANCE OF 175
```

```
# ------------------------------
# NETWORK PLOTTING FUNCTION
# ------------------------------
def plot_network(G, node_color="degree", layout="random", link_number="first", output=""):
    # INITALIZE PLOT
    fig, ax = plt.subplots()
    fig.set_size_inches(20, 20)

    # NODE COLORS
    cmap = plt.cm.get_cmap("viridis")

    # DEGREE
    if node_color == "degree":
        centrality = list(dict(nx.degree(G)).values())

    # BETWENNESS
    if node_color == "betweeness":
        centrality = list(dict(nx.betweenness_centrality(G, endpoints="philosophy")).value

    # CLOSENESS
    if node_color == "closeness":
        centrality = list(dict(nx.closeness_centrality(G)).values())
```

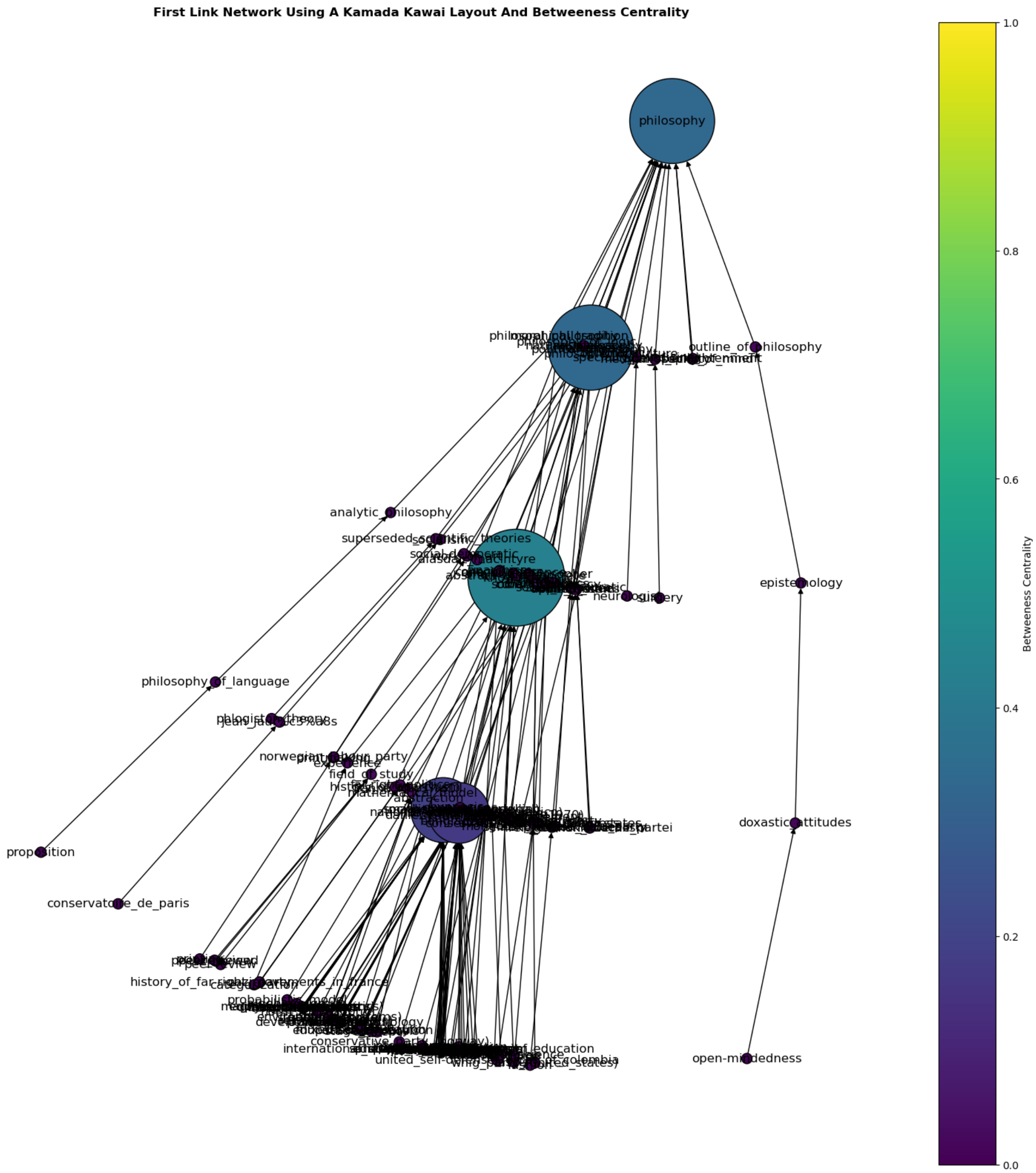
```
# NODE SIZE CAN COLOR
node_colors = [cmap(u / (0.01 + max(centrality))) for u in centrality]
node_sizes = [10000 * u / (0.001 + max(centrality)) for u in centrality]

scaled_node_sizes = [size if size > 3000 else 100 for size in node_sizes]

scale = [size / max(node_sizes) for size in scaled_node_sizes]

# POSITIONS LAYOUT
if layout == "spring":
    # pos=nx.spring_layout(G,k=50*1./np.sqrt(N),iterations=100)
    pos = nx.spring_layout(G, scale=scale)

if layout == "random":
    pos = nx.random_layout(G)

if layout == "spiral":
    pos = nx.spiral_layout(G, scale=scale)

if layout == "spectral":
    pos = nx.spectral_layout(G, scale=scale)

if layout == "kamada_kawai":
    pos = nx.kamada_kawai_layout(G)


# Creating legend
sm = plt.cm.ScalarMappable(cmap=cmap)
sm.set_array([])
plt.colorbar(sm, ax=ax, label=node_color.capitalize() + " Centrality")

# PLOT NETWORK
nx.draw(
    G,
    edgecolors="black",
    node_color=node_colors,
    node_size=scaled_node_sizes,
    pos=pos,
    with_labels=True
)
```

```
    if output != "":
        plt.savefig(output)

    title = link_number + " Link Network Using a " + layout.replace("_", " ") + " Layout a
    title = title.title()
    plt.title(title, fontweight="bold")
    plt.show()
```

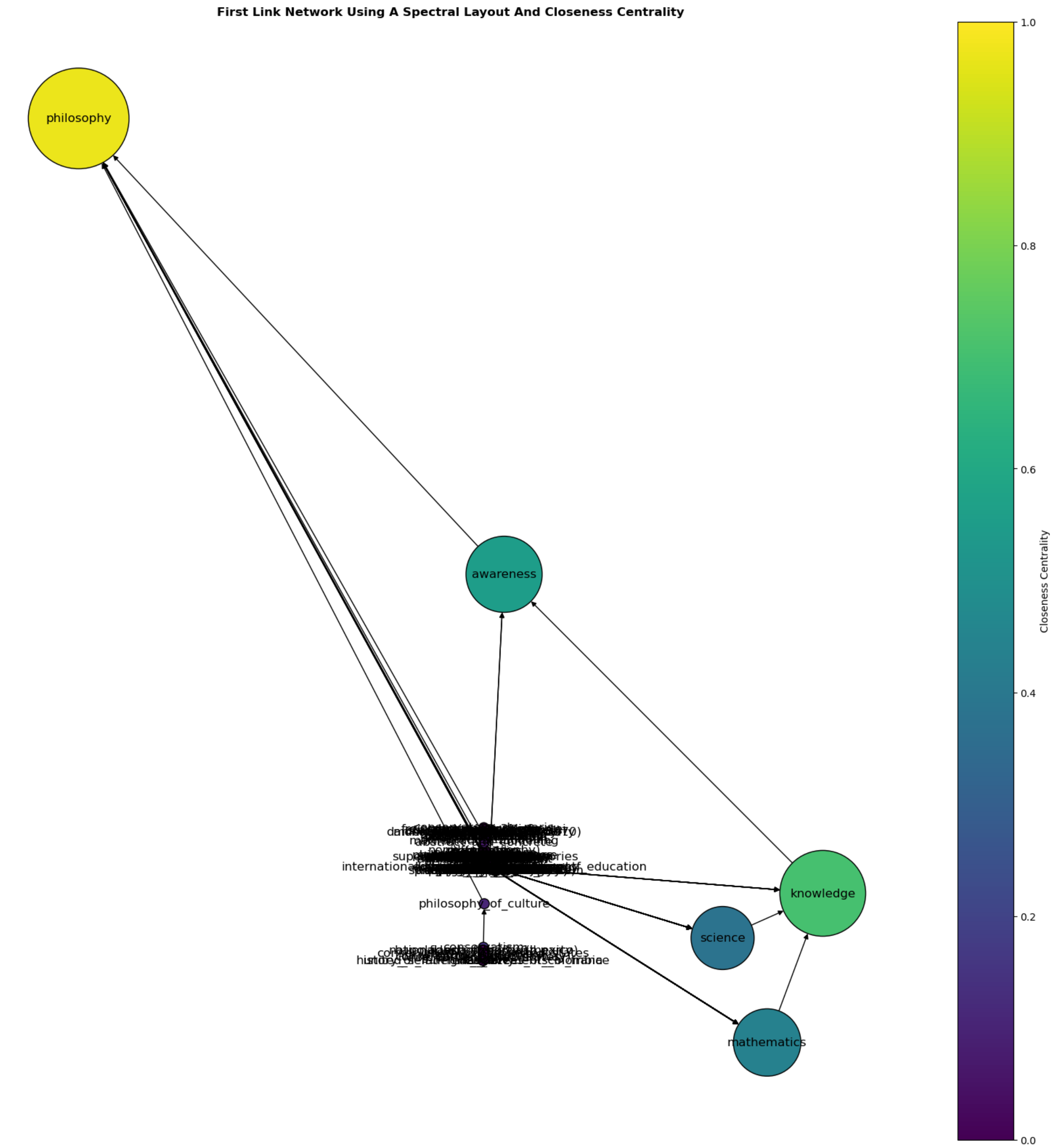
```
five_away = df[df['distance'] < 5]

subgraph = G.subgraph(five_away['node'])

plot_network(subgraph, node_color="closeness", layout="kamada_kawai", output="../images/fi
```



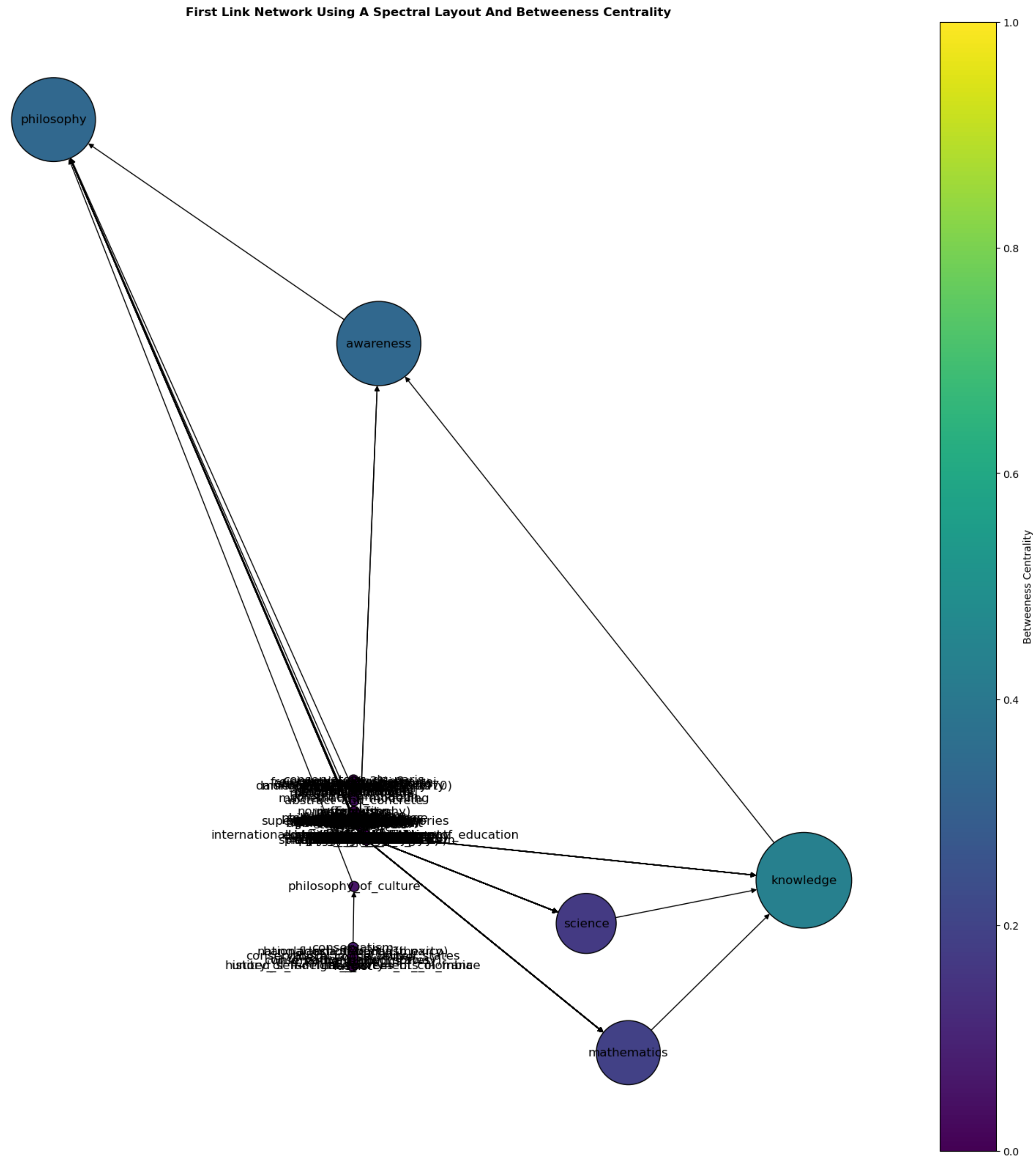

```
plot_network(subgraph, node_color="betweeness", layout="kamada_kawai", output="../images/f
```




```python
plot_network(subgraph, node_color="closeness", layout="spectral", output="../images/first-
```




```
plot_network(subgraph, node_color="betweeness", layout="spectral", output="../images/first
```




```
path_appearances = []
for node in G.nodes:
    count = 0
    for path in paths:
```

```python
        if node in path:
            count +=1

    path_appearances.append(count)

path_count_df = pd.DataFrame({"node":G.nodes, "appearances":path_appearances})
path_count_df = path_count_df.sort_values("appearances", ascending=False)
path_count_df.reset_index(inplace=True)
path_count_df.pop("index")

# Remove philosophy
path_count_df = path_count_df[1:]
path_count_df.head(80)
```

	node	appearances
1	philosophy_of_logic	4304
2	rule_of_inference	4303
3	abstraction	4302
4	information	4253
5	communication	4182
...	...	...
76	variety_(linguistics)	184
77	intention	173
78	psychology	170
79	trade	165
80	business	164

```python
def isolate_gcc(G):
    # Get all strongly connected components as subgraphs
    sccs = list(nx.weakly_connected_components(G))

    sorted_scc = sorted(sccs, key=len, reverse=True)
    # Find the largest strongly connected component (GCC)
    gcc_nodes = sorted_scc[0]

    # Create a subgraph containing only nodes from the GCC
    gcc = G.subgraph(gcc_nodes)

    return gcc
```

```
weak_gcc = isolate_gcc(G)
fraction = round((len(weak_gcc.nodes()) / len(G.nodes())) * 100, 2)
print("GCC PERCENTAGE OF NETWORK: ", fraction, "%")
```

```
GCC PERCENTAGE OF NETWORK:  87.56 %
```

```
def isolate_gcc(G):
    # Get all strongly connected components as subgraphs
    sccs = list(nx.weakly_connected_components(G))

    sorted_scc = sorted(sccs, key=len, reverse=True)
    # Find the largest strongly connected component (GCC)
    gcc_nodes = sorted_scc[6]

    # Create a subgraph containing only nodes from the GCC
    gcc = G.subgraph(gcc_nodes)

    return gcc

weak_gcc = isolate_gcc(G)
# Calculate the degrees of each node
degrees = dict(weak_gcc.degree())

# Sort the nodes by degree (in descending order)
sorted_nodes = sorted(degrees.items(), key=lambda x: x[1], reverse=True)

# Print the top nodes by degree
print("Top nodes by degree:")
for node, degree in sorted_nodes[:20]:
    print(f"{node}: {degree}")
```

```
Top nodes by degree:
central_bank: 5
money: 5
payment: 3
currency: 3
arcade_game: 3
reserve_bank_of_india: 2
numismatics: 2
banknote: 2
```

```
negotiable_instrument: 2
coin-operated: 2
private_sector_banks_in_india: 2
reserve_bank_of_india_act,_1934: 2
scheduled_banks_(india): 2
central_bank_of_iran: 1
south_african_reserve_bank: 1
professional_golfer: 1
stern_(game_company): 1
business_expense: 1
national_bank_of_angola: 1
numismatist: 1
```

Notable Disconnected nodes:

- Name
- Marketplace
- Entity
- Accounting
- Candidate
- Money

Second Link Plots

```
G2.remove_edges_from(nx.selfloop_edges(G2))
subgraph2 = nx.k_core(G2, 2)
plot_network(subgraph2, node_color="closeness", layout="spiral", output="../images/second-
```


```
path_appearances = []
for node in G2.nodes:
    count = 0
    for path in paths:
        if node in path:
            count +=1
```

```python
    path_appearances.append(count)

path_count_df = pd.DataFrame({"node":G.nodes, "appearances":path_appearances})
path_count_df = path_count_df.sort_values("appearances", ascending=False)
path_count_df.reset_index(inplace=True)
path_count_df.pop("index")

# Remove philosophy
path_count_df = path_count_df[1:]
path_count_df.head(80)
```

```python
# Calculate the degrees of each node
degrees = dict(G2.degree())

# Sort the nodes by degree (in descending order)
sorted_nodes = sorted(degrees.items(), key=lambda x: x[1], reverse=True)

# Print the top nodes by degree
print("Top nodes by degree:")
for node, degree in sorted_nodes[:10]:
    print(f"{node}: {degree}")
```

```
Top nodes by degree:
u.s._state: 65
association_football: 32
powiat: 29
american_english: 27
united_states: 22
german_language: 19
romanization_of_arabic: 18
pinyin: 16
research_university: 16
french_language: 15
```

```python
# Calculate the degrees of each node
in_degree = dict(nx.in_degree_centrality(G2))

# Sort the nodes by degree (in descending order)
sorted_nodes = sorted(in_degree.items(), key=lambda x: x[1], reverse=True)
```

```python
# Print the top nodes by degree
print("Top nodes by in_degree_centrality:")
for node, degree in sorted_nodes[:10]:
    print(f"{node}: {degree}")
```

```
Top nodes by in_degree:
u.s._state: 0.010018785222291797
association_football: 0.004852849092047589
powiat: 0.004383218534752661
american_english: 0.004070131496556042
united_states: 0.003287413901064496
german_language: 0.002817783343769568
romanization_of_arabic: 0.0026612398246712585
pinyin: 0.00234815278647464
research_university: 0.00234815278647464
french_language: 0.0021916092673763305
```

```python
# Calculate the degrees of each node
betweeness_centrality = dict(nx.betweenness_centrality(G2))

# Sort the nodes by degree (in descending order)
sorted_nodes = sorted(betweeness_centrality.items(), key=lambda x: x[1], reverse=True)

# Print the top nodes by degree
print("Top nodes by betweeness centrality:")
for node, betweeness_centrality in sorted_nodes[:10]:
    print(f"{node}: {betweeness_centrality}")
```

```
Top nodes by betweeness centrality:
ancient_greece: 0.00016919052956562225
romanization_of_greek: 0.0001566660676493492
alphabet: 0.00013914162485065008
ancient_greek_language: 0.00013406811483760014
organism: 0.0001318377312086748
letter_(alphabet): 0.00012044071596196839
animal: 0.00011884758479845029
state_(polity): 0.00011838190030449884
population: 0.00011529367681829452
eukaryotic: 0.00011264662811583368
```

```python
# Calculate the degrees of each node
closeness_centrality = dict(nx.closeness_centrality(G2))

# Sort the nodes by degree (in descending order)
sorted_nodes = sorted(closeness_centrality.items(), key=lambda x: x[1], reverse=True)

# Print the top nodes by degree
print("Top nodes by closeness centrality:")
for node, closeness_centrality in sorted_nodes[:10]:
    print(f"{node}: {closeness_centrality}")
```

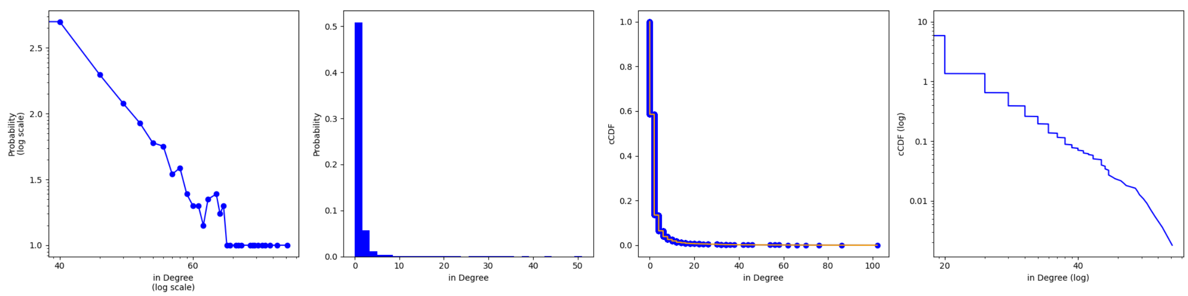
```
Top nodes by closeness centrality:
ancient_greece: 0.012006887914840326
romanization_of_greek: 0.011707771688833152
language: 0.011671205885298276
u.s._state: 0.011010013652647895
alphabet: 0.010995510457588542
grammar: 0.010852667652139918
linguistics: 0.010761638436764389
letter_(alphabet): 0.010300499059633223
natural_language: 0.010240507849732912
language_studies: 0.010114373742803153
```

Network Structures

```python
from importlib import reload
import nx_tools as nxt
reload(nxt)

nxt.plot_degree_distribution(G)
nxt.ave_degree(G)
```

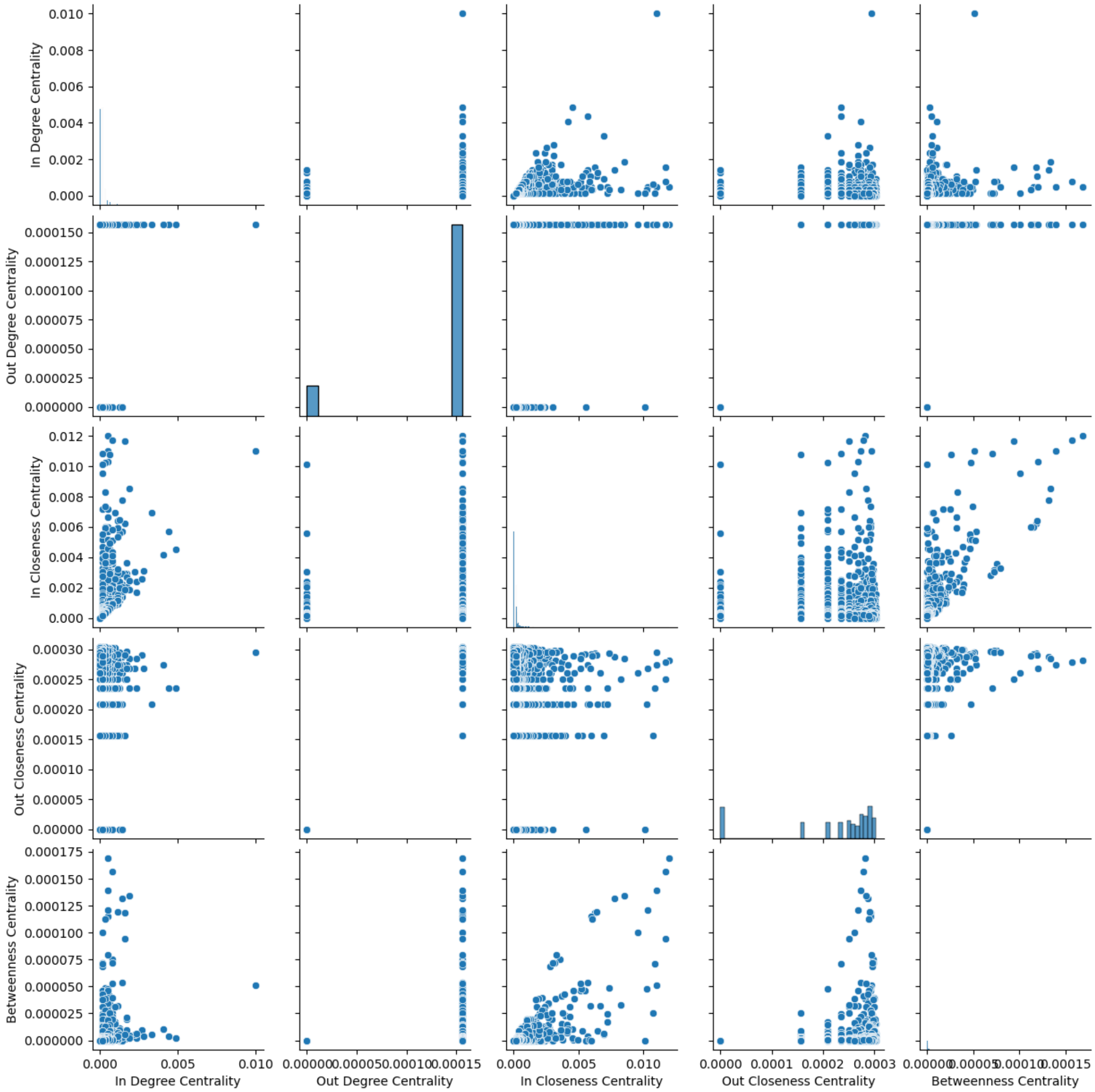
```
/Users/Austin/Desktop/Data Science/645-final-project/code/nx_tools.py:155: UserWarning: Fixe
  # AVERAGE DEGREE
/Users/Austin/Desktop/Data Science/645-final-project/code/nx_tools.py:159: UserWarning: Fixe
  directed = nx.is_directed(G)
/Users/Austin/Desktop/Data Science/645-final-project/code/nx_tools.py:179: UserWarning: Fixe
  "AVEREAGE DEGREE CONNECTIVITY: "
/Users/Austin/Desktop/Data Science/645-final-project/code/nx_tools.py:211: UserWarning: Fixe
  degree_sequence = sorted([d for _, d in data], reverse=True)
```

```
/Users/Austin/Desktop/Data Science/645-final-project/code/nx_tools.py:217: UserWarning: Fixed
  axs[0].set_xlabel(xlabel_start + "Degree\n(log scale)")
```



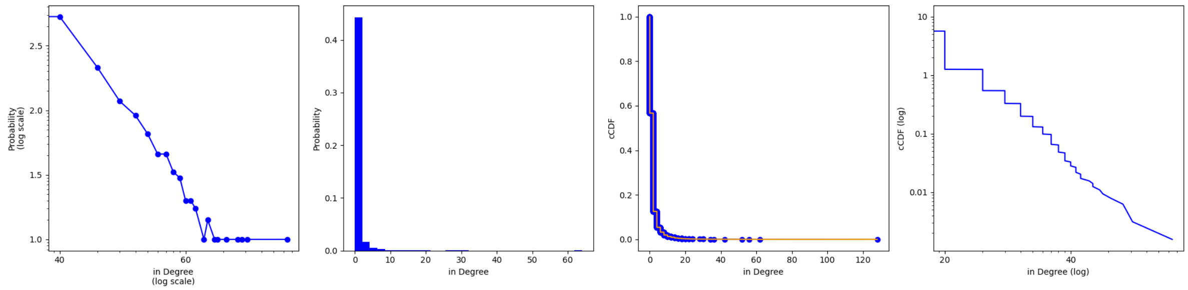

```
AVEREAGE IN DEGREE CONNECTIVITY: 2.0773339998767786
AVEREAGE OUT DEGREE CONNECTIVITY: 3.217170785687938
```

```
nxt.plot_centrality_correlation(G2)
nxt.plot_degree_distribution(G2)
nxt.ave_degree(G2)
```




```
/Users/Austin/Desktop/Data Science/645-final-project/code/nx_tools.py:155: UserWarning: Fixe
  # AVERAGE DEGREE
/Users/Austin/Desktop/Data Science/645-final-project/code/nx_tools.py:159: UserWarning: Fixe
  directed = nx.is_directed(G)
/Users/Austin/Desktop/Data Science/645-final-project/code/nx_tools.py:179: UserWarning: Fixe
  "AVEREAGE DEGREE CONNECTIVITY: "
/Users/Austin/Desktop/Data Science/645-final-project/code/nx_tools.py:211: UserWarning: Fixe
  degree_sequence = sorted([d for _, d in data], reverse=True)
```

```
/Users/Austin/Desktop/Data Science/645-final-project/code/nx_tools.py:217: UserWarning: Fixed
  axs[0].set_xlabel(xlabel_start + "Degree\n(log scale)")
```




```
AVEREAGE IN DEGREE CONNECTIVITY: 1.8268825028152322
AVEREAGE OUT DEGREE CONNECTIVITY: 2.389060512447756
```