Social Network

Week 1: Introduction in Social Network

01 – Introduction & 02 - Answer to the puzzle

What is a Social Network?

A social network is a platform (website or mobile application), where individuals meet online to communicate or share information by making posts, sending messages, or leaving comments. Users can fill in their personal data in account pages to tell the world about themselves.

Types of social networks

The classification of social networks will help you to understand the players of the market and can provide you with some new ideas.

Classification by platform

- **Web-based social networks** can only be accessed through browsers. Most popular networks like Facebook and Twitter started as websites. This can be the case if a social network is a part of your project. For example, our team have built an e-learning platform with social network feature inside, where students can communicate and collaborate with each other and make connections.
- **Hybrid networks** have both web and app versions. This is the most valuable way, as not every user can download an additional application to their phone and would like to access it via a browser. Consider progressive web application as an additional solution.
- **Mobile social networks** are developed to work only on mobile devices and tablets. We think that most of the social networks should have a mobile app, as users use social networks on a daily basis and want to have an application on the home screen.

Classification by audience

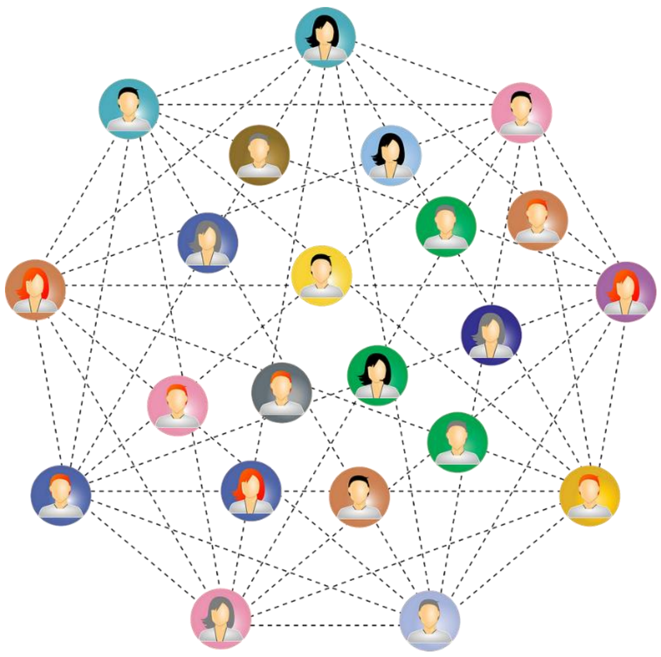
- There are **general social networks** like Facebook that connect all of the people. This place is also kept by Twitter, Instagram, and Pinterest.
- **Niche networks** aim to connect people with the same interest. For example, Soundcloud was developed for musicians to share their music and Goodreads for book lovers.

Classification by purpose

- **Dating social networks** help people to find their future love. Such apps like Badoo and Tinder provide this service.
- **Informational social networks** are developed to provide users with the information they are searching for. This includes review websites like Yelp and Q&A like Quora.
- **Educational social networks** let students communicate, collaborate and share knowledge with each other.
- **Commerce networks** provide users with an ability to shop online, as well as communicate inside a platform. For example on CreativeMarket, independent artists can sell their digital products, like and follow other artists and share the news.
- **Multimedia sharing networks** allow users to share their content and connect with other creators. Take Flickr as an example, where artists share their photos and find inspiration.

- **B2C platforms** let people find information about companies, services, or products and make purchases.
- **Social connections networks** just connect individuals like Facebook does.

The type of your social network will directly depend on the purpose and idea. Read below about the steps of building a social network.



How to build a social network

1. Define Your Target Audience

Who are you building a social network for? Who will use it and why? What problem do you solve? It is critical for you to answer these questions as your strategy, design, and marketing will depend on it.

How to define the target audience?

Cover these points:

Demographics
Demographics will help you start narrowing down your audience, they include:

- Age
- Gender
- Occupation/Industry
- Income
- Marital Status
- Educational Level

Location
Maybe your social network is for local community or work in a specific country only. It's better to define it now as it will be used later in marketing.

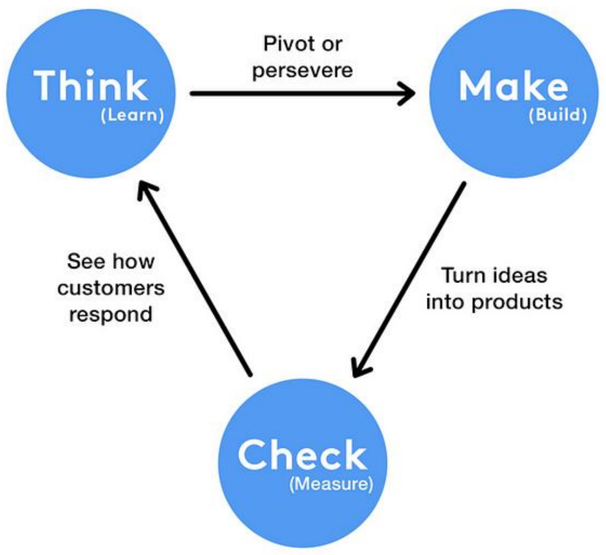
Psychographics
This includes audience behavior, hobbies, interest, and activities. For example:

- Gardeners
- Football lovers
- Housewives
- Paints collectors
- People who are looking for the lowest price
- People who shop online often

2. Develop a Strategy

We are sure you have this great idea in your head, you know what and for whom you want to build. Now it’s time to develop a killer strategy that will help you realize it.

We recommend to start small and build MVP first (minimum valuable product). This approach will help you to see feedback and support from the audience before you build the whole thing up. The agile methodology that we use at Relevant Software helps save time and money by gradually adding features and learning from users feedback.



So the first question to answer here is what your MVP is. What is the main feature users come for? After that think about growth strategy and marketing, you have to be ready to expand. Make at least a first and five years plans.

3. Pre-development work

A social network is a big and complex software that needs a development team to be built. When you have your target audience defined, a clear idea, growth strategy developed and design ideas, you have to consider some things before contacting a software development services vendor:

- **Money**. The development of social network will take from 200 to 1000 hours of work, depending on the set of features. You have to have a clear imagination about the cost of your project and decide whether you will search for a development team inside your country or outsource the development process.
- **The Platform**. Do you want to create a website only, app only or both? You should know it before contacting the development team, so they can give you an accurate estimation.

- **Traffic Estimation**. The amount of traffic your network will hold will define the technology the development team will use. It’s hard to have an accurate estimation but define approximate traffic in 5 years.

4. Design

The design itself includes 4 steps:

- **Sketching**. The very basic and rough image of the application to have a vision of the number of screens and logic between them.
- **Wireframing.** It is a fundamental step that helps a development team to see the social network’s skeleton, the elements layout and how the user will interact with them.
- **Prototyping.** A prototype is working a model of the project that gives a customer and development team to understand how the application will actually work. At this stage, insights are gathered and changes are made before the development process, which saves time and money.
- **UI/UX design**. Here designers convert the prototype into the exact image of the future social network, including animation.

5. Development & Quality Assurance

This is a moment where your idea turns into reality. This process includes back-end work like setting up servers, databases, API, etc and front-end: developing the website or mobile app visual appearance and interactions.

During and after the development quality assurance tests are provided to make sure everything works fine and the code doesn’t have errors, plus we test user experience as well.

6. Publishing & Marketing

When the social network is developed, it’s time to show it to the world by deploying the website to the servers or publishing the app in Google Play Market and AppStore. We recommend to think about marketing before the development process as well as start it before finishing the development stage to gather a base of first users.

7. Support

After the project release, the support and maintenance should be provided, especially if it’s an MVP version. This helps immediately fix issues and push updates based on users feedback. The first month of your network’s life will define its reputation.

Metrics for Social Network Project

We recommend tracking business analytics of your social network that will help you understand how your project is growing and make the right decisions in time.

Customer acquisition cost (CAC)

Customer Acquisition Cost (CAC) is a total sum of all the money spent on advertising to attract the customer. This indicator will define your scalability and profitability of the business model.

User retention and churn rate

User retention rate is a proportion of the number of users to the date they first registered. You should understand if users stay in your network.

Churn rate indicates the number of users who stopped using your website or application. The lower this rate is, the better.

User engagement

This measures how often users take actions in your social network, how long they stay inside and how often they return (every 2 days, or every 2 hours).

Growth

How quickly your userbase is growing from month to month?

Burn rate

The amount of money needed each month to maintain the project. It includes marketing, server costs, staff, etc. This metric will be necessary on the scale and fundraising stages.

03 - Introduction to Python-1 & 04 - Introduction to Python-2

Introduction To Python

Python is a widely used general-purpose, high level programming language. It was created by Guido van Rossum in 1991 and further developed by the Python Software Foundation. It was designed with an emphasis on code readability, and its syntax allows programmers to express their concepts in fewer lines of code.

There are two major Python versions: Python 2 and Python 3. Both are quite different.

Beginning with Python programming:

1) Finding an Interpreter:

Before we start Python programming, we need to have an interpreter to interpret and run our programs. There are certain online interpreters like https://ide.geeksforgeeks.org/ that can be used to run Python programs without installing an interpreter.

Windows: There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) that comes bundled with the Python software downloaded from http://python.org/.

Linux: Python comes preinstalled with popular Linux distros such as Ubuntu and Fedora. To check which version of Python you’re running, type “python” in the terminal emulator. The interpreter should start and print the version number.

macOS: Generally, Python 2.7 comes bundled with macOS. You’ll have to manually install Python 3 from http://python.org/.

2) Writing our first program:

Just type in the following code after you start the interpreter.

```
# Script Begins

print("GeeksQuiz")

# Scripts Ends
```

Output:

GeeksQuiz

Let's analyze the script line by line.

Python designed by Guido van Rossum at CWI has become a widely used general-purpose, high-level programming language.

LANGUAGE FEATURES

1. **Interpreted**
- There are no separate compilation and execution steps like C and C++.
- Directly run the program from the source code.
- Internally, Python converts the source code into an intermediate form called bytecodes which is then translated into native language of specific computer to run it.
- No need to worry about linking and loading with libraries, etc.

2. **Platform Independent**
- Python programs can be developed and executed on multiple operating system platforms.
- Python can be used on Linux, Windows, Macintosh, Solaris and many more.

3. **Free and Open Source; Redistributable**
4. **High-level Language**
- In Python, no need to take care about low-level details such as managing the memory used by the program.

5. **Simple**
- Closer to English language;Easy to Learn
- More emphasis on the solution to the problem rather than the syntax

6. **Embeddable**
- Python can be used within C/C++ program to give scripting capabilities for the program's users.

7. **Robust**
- Exceptional handling features
- Memory management techniques in built

8. **Rich Library Support**
- The Python Standard Library is very vast.
- Known as the "batteries included" philosophy of Python ;It can help do various things involving regular expressions, documentation generation, unit testing, threading, databases, web browsers, CGI, email, XML, HTML, WAV files, cryptography, GUI and many more.
- Besides the standard library, there are various other high-quality libraries such as the Python Imaging Library which is an amazingly simple image manipulation library.

Software's making use of Python

Python has been successfully embedded in a number of software products as a scripting language.

1. GNU Debugger uses Python as a pretty printer to show complex structures such as C++ containers.
2. Python has also been used in artificial intelligence
3. Python is often used for natural language processing tasks.

Current Applications of Python

1. A number of Linux distributions use installers written in Python example in Ubuntu we have the Ubiquity
2. Python has seen extensive use in the information security industry, including in exploit development.

3. Raspberry Pi– single board computer uses Python as its principal user-programming language.
4. Python is now being used Game Development areas also.

Pros:

- Ease of use
- Multi-paradigm Approach

Cons:

- Slow speed of execution compared to C,C++
- Absence from mobile computing and browsers
- For the C,C++ programmers switching to python can be irritating as the language requires proper indentation of code. Certain variable names commonly used like sum are functions in python. So C, C++ programmers have to look out for these.

Industrial Importance

Python is a high-level, interpreted, and general-purpose dynamic programming language that focuses on code readability. It has fewer steps when compared to Java and C. It was founded in 1991 by developer Guido Van Rossum. Python ranks among the most popular and fastest-growing languages in the world. Python is a powerful, flexible, and easy-to-use language. In addition, the community is very active there. It is used in many organizations as it supports multiple programming paradigms. It also performs automatic memory management.

Advantages :

1. Extensive support libraries(NumPy for numerical calculations, Pandas for data analytics etc)
2. Open source and community development
3. User-friendly data structures
4. High-level language
5. Dynamically typed language(No need to mention data type based on the value assigned, it takes data type)
6. Object-oriented language
7. Ideal for prototypes – provide more functionality with less coding

Applications :

1. Graphic design, image processing applications, Games, and Scientific/ computational Applications
2. Web frameworks and applications
3. Operating Systems
4. Database Access
5. Language Development
6. Software Development

Organizations using Python:

1. Google(Components of Google spider and Search Engine)
2. Yahoo(Maps)
3. YouTube
4. Mozilla
5. Dropbox

Introduction and Setup

If you are on Windows OS download Python by Clicking here and now install from the setup and in the start menu type IDLE.IDLE, you can think it as an Python's IDE to run the Python Scripts.

It will look somehow this :

If you are on Linux/Unix-like just open the terminal and on 99% linux OS Python comes preinstalled with the OS.Just type 'python3' in terminal and you are ready to go.

It will look like this :

The " >>> " represents the python shell and its ready to take python commands and code.

- **Variables and Data Structures**

In other programming languages like C, C++, and Java, you will need to declare the type of variables but in Python you don't need to do that. Just type in the variable and when values will be given to it, then it will automatically know whether the value given would be an int, float, or char or even a String.

See, how simple is it, just create a variable and assign it any value you want and then use the print function to print it.

Python have 4 types of built in **Data Structures** namely **List, Dictionary, Tuple and Set.**

List is the most basic Data Structure in python. List is a mutable data structure i.e items can be added to list later after the list creation. It's like you are going to shop at the local market and made a list of some items and later on you can add more and more items to the list.

append() function is used to add data to the list.

- **Comments**

# is used for single line comment in Python

""" this is a comment """ is used for multi line comments

- **Input and Output**

In this section, we will learn how to take input from the user and hence manipulate it or simply display it. input() function is used to take input from the user.

- **Selection**

Selection in Python is made using the two keywords 'if' and 'elif' and else (elseif)

- **Functions**

You can think of functions like a bunch of code that is intended to do a particular task in the whole Python script. Python used the keyword 'def' to define a function.

Syntax:

```
def function-name(arguments):

        #function body
```

- **Iteration (Looping)**

As the name suggests it calls repeating things again and again. We will use the most popular 'for' loop here.

- **Modules**

Python has a very rich module library that has several functions to do many tasks. You can read more about Python's standard library by Clicking here

'import' keyword is used to import a particular module into your python code. For instance consider the following program.

05 - Introduction to Networkx-1

06 - Introduction to Networkx-2

Introduction

The structure of NetworkX can be seen by the organization of its source code. The package provides classes for graph objects, generators to create standard graphs, IO routines for reading in existing datasets, algorithms to analyze the resulting networks and some basic drawing tools.

Most of the NetworkX API is provided by functions which take a graph object as an argument. Methods of the graph object are limited to basic manipulation and reporting. This provides modularity of code and documentation. It also makes it easier for newcomers to learn about the package in stages. The source code for each module is meant to be easy to read and reading this Python code is actually a good way to learn more about network algorithms, but we have put a lot of effort into making the documentation sufficient and friendly. If you have suggestions or questions please contact us by joining the NetworkX Google group.

Classes are named using CamelCase (capital letters at the start of each word). functions, methods and variable names are lower_case_underscore (lowercase with an underscore representing a space between words).

NetworkX Basics

After starting Python, import the networkx module with (the recommended way)

```
>>> import networkx as nx
```

To save repetition, in the documentation we assume that NetworkX has been imported this way.

If importing networkx fails, it means that Python cannot find the installed module. Check your installation and your PYTHONPATH.

The following basic graph types are provided as Python classes:

Graph

This class implements an undirected graph. It ignores multiple edges between two nodes. It does allow self-loop edges between a node and itself.

DiGraph

Directed graphs, that is, graphs with directed edges. Provides operations common to directed graphs, (a subclass of Graph).

MultiGraph

A flexible graph class that allows multiple undirected edges between pairs of nodes. The additional flexibility leads to some degradation in performance, though usually not significant.

MultiDiGraph

A directed version of a MultiGraph.

Empty graph-like objects are created with

```
>>> G = nx.Graph()
>>> G = nx.DiGraph()
>>> G = nx.MultiGraph()
>>> G = nx.MultiDiGraph()
```

All graph classes allow any hashable object as a node. Hashable objects include strings, tuples, integers, and more. Arbitrary edge attributes such as weights and labels can be associated with an edge.

The graph internal data structures are based on an adjacency list representation and implemented using Python dictionary datastructures. The graph adjacency structure is implemented as a Python dictionary of dictionaries; the outer dictionary is keyed by nodes to values that are themselves dictionaries keyed by neighboring node to the edge attributes associated with that edge. This "dict-of-dicts" structure allows fast addition, deletion, and lookup of nodes and neighbors in large graphs. The underlying datastructure is accessed directly by methods (the programming interface "API") in the class definitions. All functions, on the other hand, manipulate graph-like objects solely via those API methods and not by acting directly on the datastructure. This design allows for possible replacement of the 'dicts-of-dicts'-based datastructure with an alternative datastructure that implements the same methods.

Graphs

The first choice to be made when using NetworkX is what type of graph object to use. A graph (network) is a collection of nodes together with a collection of edges that are pairs of nodes. Attributes are often associated with nodes and/or edges. NetworkX graph objects come in different flavors depending on two main properties of the network:

•Directed: Are the edges **directed**? Does the order of the edge pairs (,) matter? A directed graph is specified by the "Di" prefix in the class name, e.g. **DiGraph()**. We make this distinction because many classical graph properties are defined differently for directed graphs.

•Multi-edges: Are multiple edges allowed between each pair of nodes? As you might imagine, multiple edges requires a different data structure, though clever users could design edge data attributes to support this functionality. We provide a standard data structure and interface for this type of graph using the prefix "Multi", e.g., **MultiGraph()**.

The basic graph classes are named: Graph, DiGraph, MultiGraph, and MultiDiGraph

Nodes and Edges

The next choice you have to make when specifying a graph is what kinds of nodes and edges to use.

If the topology of the network is all you care about then using integers or strings as the nodes makes sense and you need not worry about edge data. If you have a data structure already in place to describe nodes you can simply use that structure as your nodes provided it is hashable. If it is not hashable you can use a unique identifier to represent the node and assign the data as a node attribute.

Edges often have data associated with them. Arbitrary data can be associated with edges as an edge attribute. If the data is numeric and the intent is to represent a *weighted* graph then use the 'weight' keyword for the attribute. Some of the graph algorithms, such as Dijkstra's shortest path algorithm, use this attribute name by default to get the weight for each edge.

Attributes can be assigned to an edge by using keyword/value pairs when adding edges. You can use any keyword to name your attribute and can then query the edge data using that attribute keyword.

Once you've decided how to encode the nodes and edges, and whether you have an undirected/directed graph with or without multiedges you are ready to build your network.

Graph Creation

NetworkX graph objects can be created in one of three ways:

•Graph generators—standard algorithms to create network topologies.

•Importing data from preexisting (usually file) sources.

•Adding edges and nodes explicitly.
Explicit addition and removal of nodes/edges is the easiest to describe. Each graph object supplies methods to manipulate the graph. For example,

```
>>> import networkx as nx
>>> G = nx.Graph()
>>> G.add_edge(1, 2)  # default edge data=1
>>> G.add_edge(2, 3, weight=0.9)  # specify edge data
```

Edge attributes can be anything:

```
>>> import math
>>> G.add_edge('y', 'x', function=math.cos)
>>> G.add_node(math.cos)  # any hashable can be a node
```

You can add many edges at one time:

```
>>> elist = [(1, 2), (2, 3), (1, 4), (4, 2)]
>>> G.add_edges_from(elist)
>>> elist = [('a', 'b', 5.0), ('b', 'c', 3.0), ('a', 'c', 1.0), ('c', 'd', 7.3)]
>>> G.add_weighted_edges_from(elist)
```

See the Tutorial for more examples.

Some basic graph operations such as union and intersection are described in the operators module documentation.

Graph generators such as binomial_graph() and erdos_renyi_graph() are provided in the graph generators subpackage.

For importing network data from formats such as GML, GraphML, edge list text files see the reading and writing graphs subpackage.

Graph Reporting

Class views provide basic reporting of nodes, neighbors, edges, and degree. These views provide iteration over the properties as well as membership queries and data attribute lookup. The views refer to the graph data structure so changes to the graph are reflected in the views. This is analogous to dictionary views in Python 3.

If you want to change the graph while iterating you will need to use e.g. for e in list(G.edges):. The views provide set-like operations, e.g. union and intersection, as well as dict-like lookup and iteration of the data attributes using G.edges[u, v]['color'] and for e, datadict in G.edges.items(): Methods G.edges.items() and G.edges.values() are familiar from python dicts. In addition G.edges.data() provides specific attribute iteration e.g. for e, e_color in G.edges.data('color'):

The basic graph relationship of an edge can be obtained in two ways. One can look for neighbors of a node or one can look for edges. We jokingly refer to people who focus on nodes/neighbors as node-centric and people who focus on edges as edge-centric. The designers of NetworkX tend to be node-centric and view edges as a relationship between nodes. You can see this by our choice of lookup notation like G[u] providing neighbors (adjacency) while edge lookup is G.edges[u, v]. Most data structures for sparse graphs are essentially adjacency lists and so fit this perspective. In the end, of course, it doesn't really matter which way you examine the graph. G.edges removes duplicate representations of undirected edges while neighbor reporting across all nodes will naturally report both directions.

Any properties that are more complicated than edges, neighbors, and degree are provided by functions. For example nx.triangles(G, n) gives the number of triangles which include node n as a vertex. These functions are grouped in the code and documentation under the term algorithms.

Algorithms

A number of graph algorithms are provided with NetworkX. These include shortest path, and breadth first search (see traversal), clustering and isomorphism algorithms and others. There are many that we have not developed yet too. If you implement a graph algorithm that might be useful for others please let us know through the NetworkX Google group or the GitHub Developer Zone.

As an example here is code to use Dijkstra's algorithm to find the shortest weighted path:

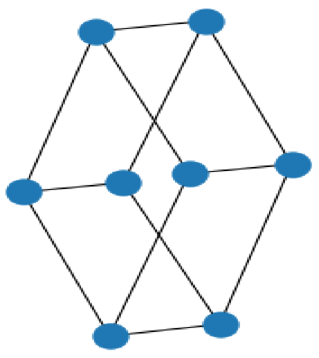
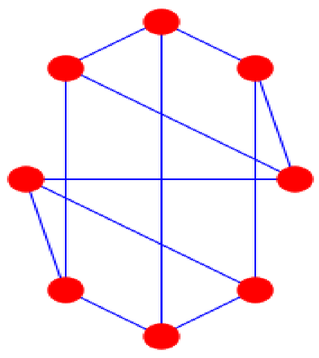
```
>>> G = nx.Graph()
>>> e = [('a', 'b', 0.3), ('b', 'c', 0.9), ('a', 'c', 0.5), ('c', 'd', 1.2)]
>>> G.add_weighted_edges_from(e)
>>> print(nx.dijkstra_path(G, 'a', 'd'))
['a', 'c', 'd']
```

Drawing

While NetworkX is not designed as a network drawing tool, we provide a simple interface to drawing packages and some simple layout algorithms. We interface to the excellent Graphviz layout tools like dot and neato with the (suggested) pygraphviz package or the pydot interface. Drawing can be done using external programs or the Matplotlib Python package. Interactive GUI interfaces are possible, though not provided. The drawing tools are provided in the module drawing.

The basic drawing functions essentially place the nodes on a scatterplot using the positions you provide via a dictionary or the positions are computed with a layout function. The edges are lines between those dots.

```
>>> import matplotlib.pyplot as plt
>>> G = nx.cubical_graph()
>>> subax1 = plt.subplot(121)
>>> nx.draw(G)   # default spring_layout
>>> subax2 = plt.subplot(122)
>>> nx.draw(G, pos=nx.circular_layout(G), node_color='r', edge_color='b')
```

Data Structure

NetworkX uses a "dictionary of dictionaries of dictionaries" as the basic network data structure. This allows fast lookup with reasonable storage for large sparse networks. The keys are nodes so G[u] returns an adjacency dictionary keyed by neighbor to the edge attribute dictionary. A view of the adjacency data structure is provided by the dict-like object G.adj as e.g. for node, nbrsdict in G.adj.items():. The expression G[u][v] returns the edge attribute dictionary itself. A dictionary of lists would have also been possible, but not allow fast edge detection nor convenient storage of edge data.

Advantages of dict-of-dicts-of-dicts data structure:

•Find edges and remove edges with two dictionary look-ups.

•Prefer to "lists" because of fast lookup with sparse storage.

•Prefer to "sets" since data can be attached to edge.

•G[u][v] returns the edge attribute dictionary.

•n in G tests if node n is in graph G.

•for n in G: iterates through the graph.

•for nbr in G[n]: iterates through neighbors.

As an example, here is a representation of an undirected graph with the edges (,) and (,).

```
>>> G = nx.Graph()
>>> G.add_edge('A', 'B')
>>> G.add_edge('B', 'C')
>>> print(G.adj)
{'A': {'B': {}}, 'B': {'A': {}, 'C': {}}, 'C': {'B': {}}}
```

The data structure gets morphed slightly for each base graph class. For DiGraph two dict-of-dicts-of-dicts structures are provided, one for successors (G.succ) and one for predecessors (G.pred). For MultiGraph/MultiDiGraph we use a dict-of-dicts-of-dicts-of-dicts [1] where the third dictionary is keyed by an edge key identifier to the fourth dictionary which contains the edge attributes for that edge between the two nodes.

Graphs provide two interfaces to the edge data attributes: adjacency and edges. So G[u][v]['width'] is the same as G.edges[u, v]['width'].

```
>>> G = nx.Graph()
>>> G.add_edge(1, 2, color='red', weight=0.84, size=300)
>>> print(G[1][2]['size'])
300
>>> print(G.edges[1, 2]['color'])
```

red

07 - Social Networks: The Challenge

Challenges of social network development

The process of building a social network can be challenging, especially if you develop something new and revolutionary. We gathered some main challenges you will face during the development of a social network.

Performance

Nowadays, your project wouldn't survive on the market if its performance is poor. The user becomes more and more demandable. The website should load under 3 seconds, another way 53% of users will leave according to Google's research. Mobile application should also load and work fast without any slowdowns. This can be achieved by using powerful technologies and the right architecture.

Security

Users expect that your platform is secure and they can control their privacy. People have to have the ability to hide their personal data, posts visibility, etc. Moreover, messaging between people must be private and encrypted.

Design and user experience

When a user gets into your web or mobile application, he has to instantly understand how to use it and get what he was looking for. It is called an intuitive design and is necessary in today's world because as mentioned before, the user doesn't want to wait or think about how to use your application. In case you've built a completely unique project with a big number of features, provide the user with a pleasant and quick step by step guide.

Moreover, doesn't forget about beautiful UI and smooth user experience with the help of animations and well thought out navigation to make a user's experience as pleasant as possible.

Marketing

Building a great product is half of the way. One of the biggest challenges is to market it right and attract enough users to survive during the first stage. If you have defined your target audience right and built the product that satisfies their needs, it should go smoothly.

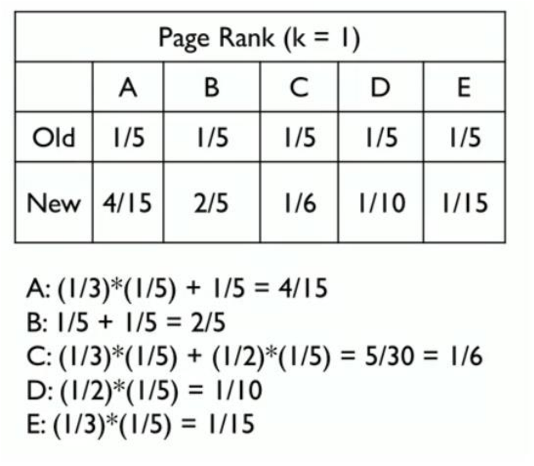
08 - Google Page Rank

Page Rank in Network Analysis

The Social Network Analysis is simply a set of objects, which we call nodes, that have some relationships between each other, which we call edges. The first reason to study networks, is because networks are everywhere. In social networks the nodes are people, and the connections between the nodes represent some type relationship between the people in the network. There are many metrics to estimate the importance or better, the centrality od a node in a network such as **Closeness centrality** and **Betweenness centrality**. The page rank was developed by the Google founders when they were thinking about how to measure the importance of webpages using the hyperlink network structure of the web. And the basic idea, is that PageRank will assign a score of importance to every single node. And the assumption that it makes, is that important nodes are those that have many in-links from important pages or important other nodes.

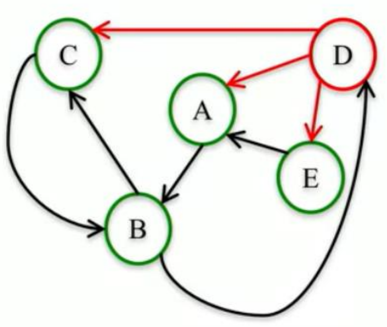
The Page Rank can be also used on any type of network, for example, the web or social networks, but it really works better for networks that have **directed edges**. In fact, the important pages are those that have many **in-links** from more important pages.

How it works
At first, we start with every node having a PageRank value of 1/n. And then what we give all of its PageRank to all the nodes that it points to, and then we do this over and over again performing these **Basic PageRank Updating Rules** k times. And then, the new value of PageRank for every node is going to be simply the sum of all the PageRank that are received from all the nodes that point to it.



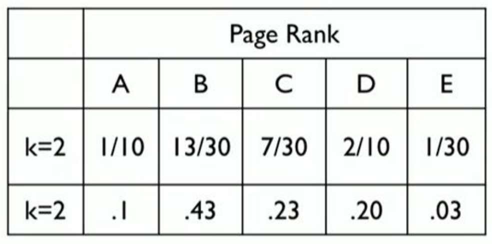
Page Rank (k = 1)					
	A	B	C	D	E
Old	1/5	1/5	1/5	1/5	1/5
New	4/15	2/5	1/6	1/10	1/15

A: (1/3)*(1/5) + 1/5 = 4/15
B: 1/5 + 1/5 = 2/5
C: (1/3)*(1/5) + (1/2)*(1/5) = 5/30 = 1/6
D: (1/2)*(1/5) = 1/10
E: (1/3)*(1/5) = 1/15

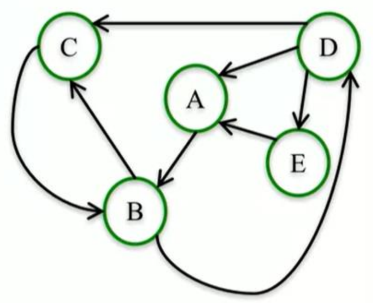


At this point, we assign every node a PageRank of 1/n. Looking at the figure above, every node will have a PageRank value of 1/5. And now we're going to start applying the Update Rule. So we're going to have to keep track of the old values of PageRank, and what the new value of PageRank of each node is. Let's start with node A. Nodes D and E point to node A, so A is going to get PageRank from D and E. Now let's think about how much **PageRank A** is going to receive from each one of those two nodes. So if we look at **D**, D has three edges, that points to three different nodes, C, A, and E. So A is going to receive 1/3 of the current page rank that D has. D currently has 1/ 5 PageRank, and so A is going to get 1/3 of that 1/5 PageRank that D has. Now A is also going to get PageRank from node **E**, and because E only points to A, then it's going to give all of its PageRank to node A. And so A is going to get 1/5 PageRank from node E. And so, in total, A is going to get 4/15 PageRank from those two nodes. And so the new value PageRank of node A is 4/15. Now, let's think of node B. We use the same approach for each nodes on the network. The figure above represents the **PageRank at Step 1**.

Crucially, we do the exact same thing again to get the second step of PageRank, k equals 2 in order to obtain the **PageRank at Step 2** as shown in the figure below.



	Page Rank				
	A	B	C	D	E
k=2	1/10	13/30	7/30	2/10	1/30
k=2	.1	.43	.23	.20	.03



From the figure above, after two steps we find that node B has the highest PageRank (0.43), followed by node C, then node D, node A, and E. This point, this is suggesting that B is the most important node in this network.

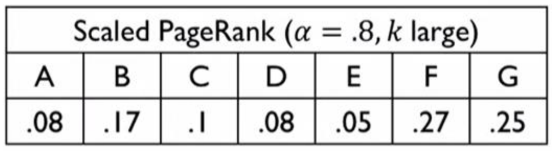
What happens if we continue for another step?

If we continue with more steps we might notice that the values change a little bit, but they still have the same order and B is sitll the highest PageRank node.

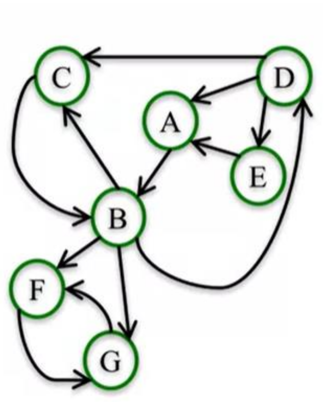
Scaled PageRank

The PageRank value of a node after k steps can be interpreted as the probability that a random walker lands on that node after taking k random steps. Random walk means we would start on a random node, and then we choose outgoing edges at random, and follow those edges to the next node. And then you're going to repeat this k times. We simply randomly choose edges and walk along in the network and the value of PageRank of each node is the probability that you would land on that node after k steps. If you repeat this for a lot of steps, say k equals infinity. These are the values that you can eventually approach, these are the values that you converges to.

Lets make a small change to this network adding two nodes, F and G, where B points to both of those nodes, and then they point to each other as depicted in the figure below.



Scaled PageRank ($\alpha = .8, k$ large)						
A	B	C	D	E	F	G
.08	.17	.1	.08	.05	.27	.25



We want you to look at this network and try and figure out what the PageRank of each node is after taking a lot of PageRank **steps** for example **4k**. For large enough k, F and G are going to have a PageRank value of about one half. And all the other nodes are going to have a PageRank value of 0. That is due to the fact that whenever the random walk lands on F or G, then they're going to stock on F and G because there are no edges to go to. There is no way to get back from G and F to any of the other notes.

The way we fix this problem is by introducing a new parameter to the PageRank computation called this **damping** parameter **alpha** and we make the random woalk with the with probability **1- alpha** and by doing this we get unstuck whenever we actually choose a random node. Crucially, for most networks as k gets larger, the **Scaled PageRank** converges to a unique value. Using Scaled PageRank now we have F and G still with a very high PageRank compared to the other notes. But the other nodes don't have a PageRank value of zero. It is important to note that this damping parameter works better for **large network**s like the **web** or very large **social networks**.

09 - Searching in a Network

A. Searches in Social Networks

Essentially a social network is a set of people and their connections to each other. Social networks evolve organically from the tendency of people make their friendships, and these choices put individuals in specific locations in the networks.

Searching in a social network, whether it's a traditional social media platform like Facebook or Twitter, or a professional network like LinkedIn, involves using various methods and tools to find specific people, content, or topics of interest. Here's a breakdown of how searching typically works in a social network:

1. **Basic Search Bar**: Most social networks have a basic search bar prominently displayed on their homepage or within their app interface. Users can type in keywords, phrases, usernames, or hashtags to find relevant content or profiles.
2. **Advanced Search Filters**: Many social networks offer advanced search filters to help users refine their search results. These filters may include options to narrow down results by location, date, language, or other criteria depending on the nature of the platform.
3. **Keyword Search**: Users can search for specific keywords or phrases to find posts, articles, or discussions related to particular topics. Some platforms also support Boolean operators (AND, OR, NOT) to make searches more precise.
4. **Hashtag Search**: Hashtags are used to categorize content and make it more discoverable. Users can click on hashtags within posts or use the search bar to find all content associated with a particular hashtag.
5. **User Search**: Users can search for specific individuals by typing their names or usernames into the search bar. Social networks may also offer suggestions or autocomplete options as users type.
6. **Content Recommendations**: Some social networks use algorithms to recommend content to users based on their interests, past interactions, or connections. These recommendations may appear in a dedicated section of the platform or within users' feeds.
7. **Trending Topics**: Many social networks display trending topics or hashtags to highlight popular discussions or events. Users can click on these topics to view related content and join the conversation.
8. **Saved Searches**: Some platforms allow users to save their search queries for quick access later. This feature is especially useful for users who regularly search for specific topics or hashtags.
9. **Search History**: Users can often view their search history to revisit previous queries or refine their searches based on past results.
10. **External Search Engines**: In addition to built-in search functionality, users can also use external search engines like Google to find content on social networks. This is particularly helpful for discovering publicly available content that may not be easily accessible through the platform's native search.

The specific algorithm used for searching within a social network can vary depending on the platform and its technical architecture. However, many social networks employ a combination of algorithms to power their search functionality. Here are some common algorithms that may be utilized:

- **Keyword Matching:** This algorithm matches user-entered keywords or phrases with relevant content in the network's database. It often involves techniques like string matching or indexing to efficiently retrieve matching results.
- **Relevance Ranking:** To prioritize search results, social networks typically employ relevance ranking algorithms. These algorithms consider various factors such as the recency of the content, user

engagement metrics (e.g., likes, shares, comments), and relevance to the user's interests or connections.
- **Graph-Based Algorithms:** Social networks are essentially graphs of connections between users, content, and topics. Graph-based algorithms, such as graph traversal algorithms or graph clustering algorithms, can be used to identify related content or recommend connections based on the network's structure.
- **Machine Learning and Natural Language Processing (NLP):** Machine learning techniques, including NLP, are often used to improve search relevance and understand user intent. For example, sentiment analysis may be applied to user queries or content to better match search results with user preferences.
- **Collaborative Filtering:** Collaborative filtering algorithms analyze user behavior and preferences to recommend content or connections that similar users have found relevant. This approach can help personalize search results based on the collective wisdom of the network's user base.
- **Geospatial Algorithms:** In social networks that incorporate location-based features, geospatial algorithms may be used to retrieve content or suggest connections based on geographic proximity.
- **Content-Based Filtering:** Content-based filtering algorithms analyze the characteristics of content (e.g., text, images, videos) to recommend similar or related items. This approach is particularly useful for content discovery and recommendation within social networks.
- **Hybrid Approaches:** Many social networks employ hybrid approaches that combine multiple algorithms to deliver more accurate and personalized search results. For example, a platform might use a combination of keyword matching, collaborative filtering, and NLP to enhance search functionality.

10 - Link Prediction

Link prediction in social networks is a significant area of study in network science and machine learning. It involves predicting the likelihood of a future connection (link) between two nodes (entities) in a social network based on the existing network structure and various node attributes. Here's an overview of the topic:

1. **Importance**: Link prediction is essential for various applications such as recommendation systems, friend suggestion algorithms in social media platforms, identifying potential collaborations in academic or professional networks, and understanding the evolution of social networks over time.
2. **Methods**: There are various methods and algorithms used for link prediction in social networks. Some of the common ones include:
    - **Graph-based Algorithms**: These algorithms analyze the network structure directly, looking at metrics like common neighbors, Jaccard coefficient, Adamic/Adar index, and preferential attachment.
    - **Machine Learning Approaches**: These methods utilize machine learning models to predict links based on features extracted from the network structure, node attributes, and other metadata. Common ML models used include logistic regression, decision trees, random forests, and neural networks.
    - **Deep Learning Models**: Recent advancements in deep learning have led to the development of graph neural networks (GNNs) and deep learning-based approaches specifically tailored

for link prediction tasks in social networks. GNNs can capture complex structural patterns and node features simultaneously.

3. **Features**: Features used for link prediction can be categorized into three types:
    - **Structural Features**: Derived from the network structure, including common neighbors, shortest path length, and neighborhood overlap.
    - **Node Features**: Characteristics of individual nodes such as age, gender, location, interests, and activity level.
    - **Temporal Features**: Information related to the temporal dynamics of the network, such as the frequency and timing of interactions between nodes.
4. **Evaluation**: Evaluating link prediction algorithms involves comparing the predicted links with the ground truth (actual observed links) using metrics such as precision, recall, F1-score, area under the receiver operating characteristic curve (AUC-ROC), and area under the precision-recall curve (AUC-PR).
5. **Applications**: Link prediction has applications in various domains, including social media analysis, recommendation systems, fraud detection, cybersecurity, and biological network analysis.
6. **Challenges**: Challenges in link prediction include dealing with sparsity and noise in network data, handling large-scale networks efficiently, capturing evolving network dynamics, and integrating heterogeneous data sources.

11 - The Contagions

Social Contagion Theory

Social Contagion Theory is a theoretical framework that has received considerable attention in the field of social psychology. The theory suggests that people's thoughts, emotions, and behaviors can be influenced by the people around them, leading to the spread of ideas, attitudes, and behaviors throughout social networks. This review provides an overview of the theory, its key components, and its application in different contexts.

One of the core principles of Social Contagion Theory is the idea of "social influence." This refers to the process by which people conform to the beliefs, values, and behaviors of those around them. This conformity is driven by a variety of factors, including the desire to fit in, the need for social approval, and the perception that others possess more accurate information or better judgment. Social influence can occur through a variety of mechanisms, including persuasion, social comparison, and normative influence.

Another key component of Social Contagion Theory is the idea of "social contagion." This refers to the process by which attitudes, emotions, and behaviors spread through social networks. Social contagion can occur through a variety of channels, including face-to-face interactions, mass media, and social media. The theory suggests that social contagion can be facilitated by a variety of factors, including the strength of social ties, the frequency of contact, and the similarity of attitudes and behaviors.

Social Contagion Theory has been applied in a variety of contexts, including health behavior, political behavior, and consumer behavior. In the context of health behavior, the theory has been used to understand the spread of infectious diseases, the adoption of healthy behaviors, and the diffusion of health information. In the context of political behavior, the theory has been used to understand the spread of political ideologies, the formation of political coalitions, and the impact of political campaigns. In the context of consumer behavior, the theory has been used to understand the spread of product information, the adoption of new products, and the formation of consumer communities.

While Social Contagion Theory has received considerable attention in the field of social psychology, there are some limitations to the theory that should be acknowledged. One limitation is the difficulty in distinguishing between social influence and other factors that may influence behavior, such as individual differences and situational factors. Another limitation is the challenge of identifying the specific mechanisms through which social contagion occurs, as social networks are complex and dynamic.

12 - Importance of Acquaintances

Definition:

Acquaintances in a social network within engineering refer to individuals with whom one has a basic familiarity or connection within the professional context. These connections may range from casual acquaintances to individuals with whom one has collaborated on projects or shared professional experiences.

IMPORTANCE

1. **Diverse Perspectives**: Acquaintances come from various backgrounds, cultures, and professions, offering different viewpoints and experiences. Interacting with acquaintances exposes individuals to diverse perspectives, broadening their understanding of the world and enhancing their critical thinking skills.
2. **Networking Opportunities**: Acquaintances expand one's social circle, providing access to a broader network of contacts. These connections can be valuable for professional networking, career opportunities, and personal growth.
3. **Information Exchange**: Acquaintances often possess unique knowledge and insights that individuals may not have access to otherwise. Engaging with acquaintances facilitates the exchange of information, ideas, and expertise, enriching learning experiences and fostering intellectual curiosity.
4. **Social Support**: While not as close as friends or family members, acquaintances still offer a level of social support. They can provide encouragement, advice, and assistance during challenging times, serving as a source of emotional and practical support.
5. **Opportunities for Collaboration**: Collaborative ventures often emerge from interactions with acquaintances. Whether it's joining forces on a project, starting a business together, or pursuing shared interests, acquaintanceship can lead to fruitful collaborations and synergistic relationships.
6. **Cultural Exposure**: Acquaintances from different cultural backgrounds expose individuals to new customs, traditions, and perspectives. This cultural exchange promotes tolerance, empathy, and appreciation for diversity, fostering a more inclusive and interconnected society.
7. **Personal Development**: Interacting with acquaintances challenges individuals to step out of their comfort zones, develop interpersonal skills, and adapt to different social dynamics. These experiences contribute to personal growth, self-awareness, and emotional intelligence.
8. **Opportunities for Serendipity**: Chance encounters with acquaintances can lead to unexpected opportunities, discoveries, or connections. Serendipitous moments often arise when individuals are open to engaging with others and exploring new possibilities.

Applications:

1. **Professional Networking**: Acquaintances facilitate the expansion of one's professional network, which can lead to career opportunities, collaborations, and knowledge sharing.
2. **Knowledge Exchange**: They serve as valuable sources of information, insights, and expertise, aiding in problem-solving, skill development, and staying updated with industry trends.
3. **Career Development**: Acquaintances can provide referrals, recommendations, and guidance for career advancement, job opportunities, and skill enhancement.
4. **Collaborative Projects**: They offer opportunities for collaboration on engineering projects, leveraging diverse skill sets and perspectives to achieve common goals.
5. **Industry Insights**: Acquaintances working in different sectors of engineering provide valuable insights into market dynamics, emerging technologies, and industry best practices.

Advantages:

1. **Access to Resources**: Acquaintances provide access to a wide range of resources, including job openings, knowledge repositories, and professional development opportunities.
2. **Diverse Perspectives**: They offer diverse perspectives and experiences, enriching problem-solving approaches and fostering innovation.
3. **Support System**: Acquaintances can offer emotional support, advice, and encouragement during challenging times, contributing to overall well-being and resilience.
4. **Professional Growth**: Interactions with acquaintances facilitate continuous learning, skill development, and career advancement.

Disadvantages:

1. **Limited Depth of Relationship**: Compared to close colleagues or mentors, acquaintances may have limited depth in their relationships, which could hinder the level of support and understanding available.
2. **Time Investment**: Building and maintaining relationships with acquaintances require time and effort, which may detract from other priorities or activities.
3. **Potential for Miscommunication**: Lack of familiarity or rapport with acquaintances may lead to miscommunication or misunderstandings, particularly in collaborative projects.
4. **Dependency on Network**: Over-reliance on acquaintances for career opportunities or resources may limit one's ability to explore alternative pathways or opportunities.

13 - Marketing on Social Networks

What Is Social Network Marketing?

Given the popularity of social media, social network marketing is a broad category of marketing. Anytime a business uses social media, it could be considered a form of social network marketing. This form of outreach could take place on any social media platform, whether it's Facebook, Twitter, YouTube, or anything else.

Social network marketing is also varied due to the varied nature of marketing. Marketing outreach can take many different shapes—from outright advertisements to more subtle ways of building a relationship with potential customers. Social network marketing can similarly be straightforward ads for products or services, or it can be a means for a business to develop relationships.

How Does Social Network Marketing Work?

When successful, social network marketing helps a business:

- Increase referrals or sales leads
- Build word-of-mouth reputation
- Increase sales of products or services
- Provide a means of feedback
- Develop a reputation as an expert or thought-leader
- Drive traffic to a business website or blog
- Develop new products or services
- Keep people informed about special events, sales, and other newsworthy events
- Provide customer service

The Most Popular Social Media Platforms

Social media is a relatively new technology, so platforms and popularity are constantly in flux. However, it is possible to gauge the popularity of platforms as a point-in-time measurement.

According to a Pew Research survey in 2019, the most popular platform was YouTube, which was used by 73% of U.S. adults. These are the adult usage stats for the other eight platforms included in the survey:

- Facebook: 69%
- Instagram: 37%
- Pinterest: 28%
- LinkedIn: 27%
- Snapchat: 24%
- Twitter: 22%
- WhatsApp: 20%
- Reddit: 11%

Not All Social Media Platforms May Be Suitable for Your Business

The best social media platform for a business to use will depend on its strategy. Certain strategies work better on certain platforms. There are demographic differences between the platforms, as well.

Pros and Cons of Social Network Marketing

Pros Explained

- **Cheap**: Compared to other forms of marketing, social media is very inexpensive. It is free to create accounts and post on most (if not all) platforms. Those free posts have the potential to reach an audience that will (hopefully) share the message with other members of the social media community. There are also various paid advertising options to ensure that your message will reach an audience. Facebook, for example, offers comprehensive options for ad targeting, such as the ability to target members that reside within a specific geographic radius.
- **Direct engagement with customers**: Social networks allow the business to engage the target audience and develop interactive relationships with customers. Instead of putting out a message and hoping an audience sees it, you can engage directly with people—commenting on their posts and starting conversations.
- **Able to learn more about customers**: As you communicate and engage with customers, you'll learn more about your customer base. Many people put personal details on their social media profiles, including information like tastes and preferences that are relevant to businesses.

Cons Explained

- **Time-intensive**: The main problem with social network marketing from a business perspective is that it can be incredibly time-consuming. Social media marketing campaigns are not one-shot affairs; they need to be nurtured over time. While big businesses have been using this type of marketing effectively, they have the resources to task large numbers of staff with managing social media marketing campaigns. Many small businesses lack these resources.
- **Direct advertising is less effective**: Businesses using social networks obviously want to sell their products or services. However, blatant advertising is fairly easy to spot on social media. Overt attempts to make a sale won't be as effective as more subtle forms of promotion on social media. To use social network marketing effectively, businesses have to be perceived as members of the social media community. Original content and audience engagement could help to this end.
- **Increases risk of public criticism and hacks**: Given the community nature of social media, businesses must be aware of the PR risks. A public social media page can face an onslaught of negative postings by customers, ex-employees, or competitors (whether true or false). Hackers could target your page to spread misinformation or sow chaos. Either situation can make it difficult to manage your company's reputation. Even a harmless posting can turn into a public relations disaster. For example, in 2012, McDonald's tried to engage with the social media community by posting a tweet under the hashtag "#McDStories"; customers responded by posting horror stories of poor McDonald's experiences with the same hashtag, drowning out the original marketing content with negative publicity.

Week 2: Handling Real-world Network Datasets in Social Network

14 - Introduction to Datasets

A dataset is a set of numbers or values that pertain to a specific topic. A dataset is, for example, each student’s test scores in a certain class. Datasets can be written as a list of integers in a random order, a table, or with curly brackets around them. The data sets are normally labelled so you understand what the data represents, however, while dealing with data sets, you don’t always know what the data stands for, and you don’t necessarily need to realize what the data represents to accomplish the problem.

Social networks generate vast amounts of data every day, encompassing interactions, behaviors, and connections between users. These datasets provide valuable insights into human behavior, social dynamics, and network structures. Understanding and analyzing social network datasets have become integral in various fields such as sociology, psychology, marketing, and computer science. Here's an introduction to datasets in social networks:

1. **Characteristics of Social Network Datasets**:
   - **Large Scale**: Social network datasets can be massive, comprising millions or even billions of nodes and edges.
   - **Dynamic**: Social networks evolve over time due to new connections, interactions, and content creation. Datasets need to capture this temporal aspect.
   - **Sparse**: Not all users are connected to each other directly. Social networks often exhibit sparse connectivity.
   - **Biased and Noisy**: Data may contain biases due to factors like user self-selection and platform algorithms. Noise can arise from spam, bots, or inaccuracies in data collection.
2. **Sources of Social Network Datasets**:
   - **Public APIs**: Many social platforms offer APIs allowing researchers to access data within their terms of service.
   - **Academic Repositories**: Datasets collected for research purposes are often shared in academic repositories.
   - **Data Scraping**: Researchers may collect data by scraping publicly available information from social platforms, although this must be done ethically and in accordance with platform policies.
   - **Surveys and Experiments**: Researchers may conduct surveys or experiments to gather social network data directly from participants.
3. **Applications of Social Network Datasets**:
   - **Social Network Analysis (SNA)**: Studying network structure, centrality, clustering, and community detection.
   - **Recommendation Systems**: Utilizing social connections for personalized recommendations.
   - **Marketing and Advertising**: Targeting specific demographics or influencers for campaigns.
   - **Sentiment Analysis**: Understanding public opinion and mood through social media content.
   - **Epidemiology**: Tracking the spread of diseases or misinformation through social networks.
   - **Behavioral Studies**: Investigating human behavior, social influence, and cultural trends.
4. **Challenges in Working with Social Network Datasets**:
   - **Privacy Concerns**: Social network data often contain sensitive information, raising privacy and ethical considerations.
   - **Data Quality**: Ensuring the reliability, completeness, and accuracy of social network datasets can be challenging.

- **Scalability**: Analyzing large-scale social network datasets requires scalable algorithms and computational resources.
- **Ethical Considerations**: Respecting user privacy, consent, and platform policies is essential when collecting and analyzing social network data.

15 - Ingredients Network

A network dataset, in the context of computer science and data analysis, refers to a collection of nodes (or vertices) and edges (or links) that connect these nodes. This structure is used to represent relationships or interactions between entities. When we talk about an "Ingredients Network," we're referring to a network dataset specifically designed to represent relationships between ingredients, typically in the context of recipes, food science, or culinary analysis.

Here's how an Ingredients Network dataset might look:

1. **Nodes**: In an Ingredients Network, nodes represent individual ingredients. Each node would typically have attributes associated with it, such as the name of the ingredient, its category (e.g., protein, vegetable, spice), nutritional information, flavor profile, etc.
2. **Edges**: Edges in the network represent relationships between ingredients. These relationships could signify various things, such as:
    - **Co-occurrence**: If two ingredients frequently appear together in recipes, they might be connected by an edge.
    - **Substitution**: Ingredients that can be substituted for one another in recipes might be linked by an edge.
    - **Complementarity**: Ingredients that complement each other in flavor or function might be connected.
    - **Hierarchy**: Ingredients that belong to the same category (e.g., different types of cheese) might be connected to a parent node representing that category.
    - **Dependency**: Ingredients that depend on each other in terms of preparation or cooking process might be linked.
3. **Attributes**: Besides the basic attributes associated with each ingredient node, additional attributes might be associated with edges to provide more information about the relationships. For example, the strength of the relationship, frequency of co-occurrence, similarity in flavor profile, etc.
4. **Metadata**: Metadata could include information about the source of the data, such as the recipes or culinary databases from which the relationships were extracted.
5. **Analysis and Applications**: Once constructed, an Ingredients Network dataset can be analyzed using network analysis techniques to uncover patterns, clusters, or other insights about ingredient relationships. This can be useful in various applications such as recipe recommendation systems, food pairing suggestions, understanding culinary traditions, ingredient substitution suggestions for dietary restrictions or availability, and more.

16 - Synonymy Network

Synonymy Network in a social network refers to a conceptual framework or system designed to enhance communication and information retrieval by incorporating synonyms and related terms into the platform's functionality. This network aims to improve user experience, search accuracy, and content relevance by recognizing and incorporating synonymous and semantically related terms into various aspects of the social network's interface and algorithms. Here's a detailed introduction to the concept:

Overview:
A synonymy network, in the context of linguistics and natural language processing, refers to a structured representation of synonymous relationships among words or phrases. In such a network, nodes represent words or phrases, and edges represent synonymy relationships between them. Synonymy networks are used in various NLP tasks such as information retrieval, text classification, and word sense disambiguation.

These networks can be constructed using various methods, including lexical resources like thesauri, lexical databases, or by analyzing large corpora of text to identify co-occurring terms. They are valuable for tasks such as expanding search queries, improving the performance of information retrieval systems, and enhancing the accuracy of word sense disambiguation algorithms.

Synonymy networks are essential for understanding the semantic relationships between words and can be utilized in numerous applications to improve the performance of natural language understanding systems.

Key Components:

1. **Semantic Analysis Engine:**
    - The backbone of the Synonymy Network is a sophisticated semantic analysis engine powered by advanced NLP algorithms. This engine parses text data to identify synonyms, antonyms, hypernyms, hyponyms, and other semantic relationships between words and phrases.
2. **Synonym Database:**
    - A comprehensive database containing synonyms and related terms for a wide range of words and phrases is essential for the functioning of the network. This database is continuously updated and refined to reflect changes in language usage and evolving semantic relationships.
3. **Integration with Search Functionality:**
    - Synonyms and related terms are seamlessly integrated into the social network's search functionality. When users enter a query, the Synonymy Network not only matches exact keywords but also considers synonymous and semantically related terms to broaden the scope of search results.
4. **Content Recommendation System:**
    - By understanding the underlying meanings of words and phrases, the Synonymy Network enhances the accuracy of content recommendation systems. It can suggest relevant posts, articles, groups, or users based not only on exact keyword matches but also on related concepts and themes.
5. **Semantic Tagging and Categorization:**
    - Synonyms and related terms are utilized to enrich the semantic tagging and categorization of content within the social network. This allows for more precise content classification and organization, making it easier for users to discover relevant information.
6. **Contextual Understanding:**
    - The Synonymy Network goes beyond mere word matching by considering the context in which words and phrases are used. This contextual understanding enables more accurate

interpretation of user queries and content, leading to improved search relevance and user engagement.

Benefits:

1. **Enhanced Search Accuracy:**
    - Users can find the information they need more efficiently thanks to the Synonymy Network's ability to incorporate synonymous and related terms into search results.
2. **Improved Content Relevance:**
    - Content recommendation algorithms become more effective at suggesting relevant posts, discussions, and users based on a deeper understanding of semantic relationships.
3. **Streamlined Communication:**
    - Synonymous terms facilitate clearer communication by offering alternative expressions and reducing ambiguity in discussions and interactions among users.
4. **Personalized User Experience:**
    - By recognizing synonyms and related terms, the social network can tailor content and recommendations to better align with individual users' interests and preferences.
5. **Better Semantic Understanding:**
    - The Synonymy Network fosters a deeper understanding of language semantics within the social network's ecosystem, contributing to more intelligent and context-aware interactions.

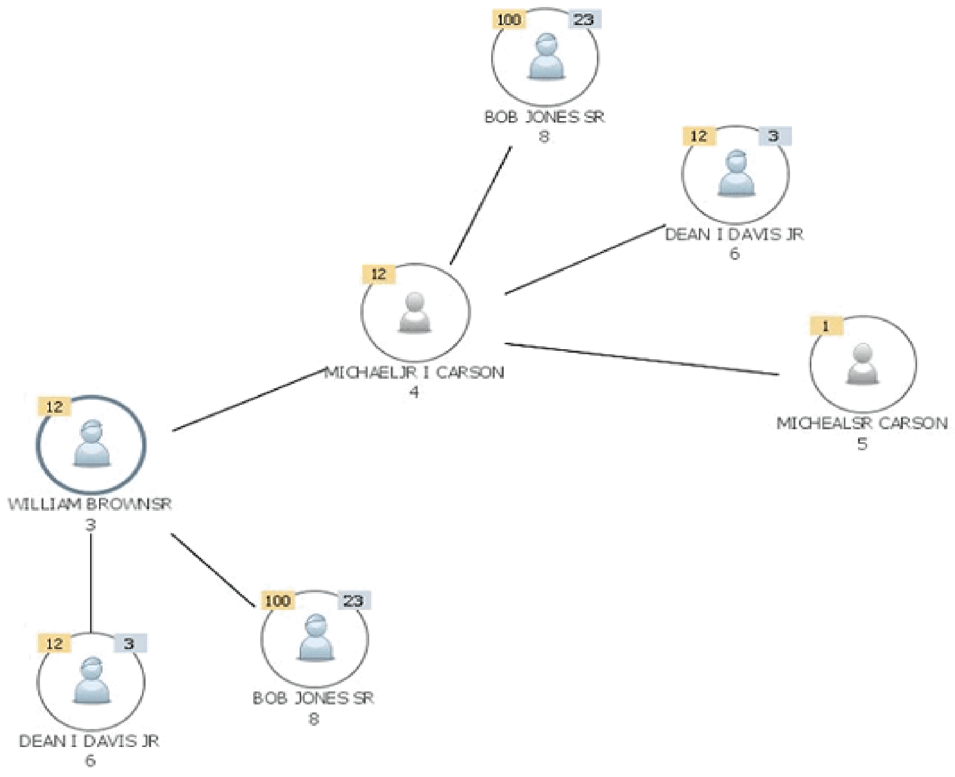
17 - Web Graph

A web graph is a visual representation of the relationships between different web pages, analysed through graph theory and algorithms.

Analysing a web graph involves the use of graph theory, a branch of mathematics that studies graphs and their properties. Graph theory provides a set of concepts and algorithms that can be used to measure various aspects of a web graph, such as its size, density, diameter, and degree distribution. For example, the size of a web graph is the number of its nodes and edges, the density is the proportion of possible edges that actually exist, the diameter is the longest shortest path between any two nodes, and the degree distribution is the probability distribution of the degrees over the entire network.

Web graph analysis can provide valuable insights into the structure and dynamics of the web. It can help us understand how information flows on the web, how web communities form and evolve, and how web pages can be optimised for search engines. Furthermore, the techniques and tools developed for web graph analysis can also be applied to other types of networks, such as social networks, biological networks, and transportation networks.

The Social Network graph helps you to visualize the relationships between the selected entity and all entities that the selected entity is linked to. Using this unique graph, you get another way to see "who knows who".



The Social Network graph shows:

- Entity-to-entity links: You see all the entities related to the main (hub) entity. However, the attributes that link the entities do not display on the graph but are accessible by using the Attribute Explorer in combination with the graph.
- Relationship clusters: The Social Network graph is unique in that it displays the related entities in groups or clusters. This graph can help you see all the relationship clusters a particular entity belongs to and look for patterns in among the clusters and relationships.

You can expand the graph to show all the related entities for any entity. Each time you show all entities related to a particular entity, that entity node becomes the hub entity in a new relationship cluster.

To maintain the integrity of each relationship cluster, an entity can be displayed on the graph multiple times in multiple relationship clusters. But each entity displays in each relationship cluster only once. To see every relationship cluster the entity is part of, select the entity by clicking on that node. The interior of the selected entity node changes to blue in each relationship cluster that the entity is part of.

When an entity is the hub entity, the Related Entities indicator does not display, because all entities related to the hub entity already display in the relationship cluster. When the entity is one of the related entities in the relationship cluster and has other relationships that are not displayed in the that cluster, a Related Entities indicator displays.

18 - Social Network Datasets

Social network datasets refer to collections of data that represent relationships or interactions between entities in a social network. These entities can be individuals, organizations, or any other units with connections between them. Social network datasets are used in various fields such as sociology, computer science, economics, and marketing to analyze patterns, behaviors, and dynamics within social networks.

These datasets typically consist of nodes and edges. Nodes represent the entities, while edges represent the relationships or interactions between them. Each edge usually contains information about the nature of the relationship, such as friendship, communication, collaboration, or any other type of interaction.

Social network datasets can vary widely in size, scope, and structure. They may come from online social networks like Facebook, Twitter, or LinkedIn, where connections are formed through friend requests, follows, or other interactions. Alternatively, they can represent offline social networks, such as interactions between individuals in a community, organization, or academic collaboration network.

Researchers and analysts use social network datasets for a variety of purposes, including:

1. Studying social structures and dynamics: Social network analysis techniques can reveal patterns of connections, centrality, clustering, and community structure within networks, providing insights into social phenomena such as influence, information diffusion, and the formation of cliques or subgroups.
2. Predictive modeling and recommendation systems: By analyzing past interactions and relationships in social networks, algorithms can be developed to predict future connections, interests, or behaviors of individuals, enabling personalized recommendations, targeted advertising, or fraud detection.
3. Understanding information flow and influence propagation: Social network datasets help researchers understand how information, opinions, and behaviors spread through networks, allowing them to study phenomena like viral marketing, rumor propagation, and the dynamics of collective decision-making.
4. Social media monitoring and sentiment analysis: By analyzing interactions and content shared on social media platforms, organizations can monitor public opinion, track trends, and measure sentiment towards products, brands, or political issues.

Popular social network datasets include the Facebook Social Circles dataset, Twitter datasets collected via the Twitter API, the Enron email dataset, and the Stanford Large Network Dataset Collection (SNAP), which contains a variety of social network datasets from different online platforms.

19 - Datasets: Different Formats

5. **Types of formats of datasets:**
   - **CSV(Comma Separated Value):** It has extension either .txt or .csv . CSV format file can have 2 more types it can be either edge list or adjacency list format .
     Example:
     - **EdgeList format:** Basically it can edges and weights if required. Every row contains 2 nodes, first node will be the source node and the second node will be the target node.
     - **Adjacency list format:** Basically every contains 2 or more nodes. The first node is the source node and subsequent nodes in the same row are the nodes connected directly to source node like in first row 1 is directly connected to 2, 5 and 7 nodes.
   - **GML(Graph Modeling Language):** It is the most commonly used format for network datasets because it provides flexibility for assigning attributes to the nodes and edges and it is very simple.

- **Pajek Net:** It has extension .NET or .Paj .It is widely used for network datasets. For every row, you have every node return and all nodes are done you start with information about edges that contain source node and the target node.
- **GraphML:** Here ML stands XML as it is very much similar to XML. As in XML, there are hierarchical structures and their tags. Similarly in graphml also there are tags like XML tag, graphml tag, graph tag, node tag, and edge tag.
- **GEXF(Graph Exchange XML Format):** It was created by Gephi people. Gephi is an opensource software that is used for visualizing and analyzing social networks. This format is also inspired by XML as it has similar tags. Tags are XML tag, GEXF tag, Meta tag, Graph tag, node tag, edge tag.

20 - Datasets: How to Download?

Downloading datasets can vary depending on where the data is hosted and what format it's available in. Here's a general guide:

1. **Identify the Dataset**: First, you need to know what dataset you're interested in. It could be available on various platforms like Kaggle, UCI Machine Learning Repository, government websites, academic repositories, or specific project websites.
2. **Locate the Download Link**: Once you've found the dataset, look for a download link. This might be prominently displayed on the webpage or you might need to navigate through menus or documentation to find it.
3. **Choose the Format**: Some datasets offer multiple formats for download (e.g., CSV, JSON, XML). Choose the format that best suits your needs.
4. **Check for Any Restrictions**: Make sure you're aware of any restrictions on the dataset's use. Some datasets may be freely available for any purpose, while others may have restrictions on commercial use or require attribution.
5. **Download the Dataset**: Click on the download link to initiate the download process. Depending on the size of the dataset and the speed of your internet connection, this may take some time.
6. **Verify the Download**: Once the download is complete, verify that you have successfully downloaded the dataset. Check the file size and, if applicable, compare it to any size information provided on the dataset's webpage to ensure the download was successful.
7. **Extract (if necessary)**: If the dataset is compressed (e.g., in a ZIP file), you'll need to extract it before you can use it. Most operating systems have built-in tools for extracting files, or you can use third-party software like WinZip or 7-Zip.
8. **Explore the Dataset**: After downloading and extracting the dataset, take some time to explore its contents. Understand its structure, the meaning of each column (if applicable), and any accompanying documentation that might provide additional context or instructions.

21 - Datasets: Analysing Using Networkx

Analyzing datasets using NetworkX can be a powerful way to gain insights into the structure and relationships within the data. NetworkX is a Python library for the creation, manipulation, and study of complex networks or graphs.

Here's a basic guide on how to analyze datasets using NetworkX:

1. **Import NetworkX**: First, you need to install NetworkX if you haven't already and import it into your Python environment.

python

```
import networkx as nx
```

2. **Create a Graph**: You can create an empty graph or load data from various formats like edge lists, adjacency matrices, etc.

python

```
# Create an empty graph G = nx.Graph() # Add nodes G.add_node(1) G.add_nodes_from([2, 3]) # Add
edges G.add_edge(1, 2) G.add_edges_from([(1, 3), (2, 3)])
```

3. **Load Data**: If you have data in a specific format like edge lists or adjacency matrices, you can load it into NetworkX directly.

python

```
# Load data from edge list G = nx.read_edgelist('filename.txt') # Load data from adjacency matrix #
Assuming matrix is stored in a list of lists adjacency_matrix = [[0, 1, 1], [1, 0, 1], [1, 1, 0]] G =
nx.from_numpy_matrix(np.array(adjacency_matrix))
```

4. **Analyze the Graph**: Now that you have your graph loaded, you can perform various analyses on it.

python

```
# Basic graph properties print("Number of nodes:", G.number_of_nodes()) print("Number of edges:",
G.number_of_edges()) print("Nodes:", G.nodes()) print("Edges:", G.edges()) # Degree distribution
degree_sequence = sorted([d for n, d in G.degree()], reverse=True) print("Degree sequence:",
degree_sequence) # Clustering coefficient print("Average clustering coefficient:", nx.average_clustering(G))
# Shortest path shortest_path = nx.shortest_path(G, source=1, target=3) print("Shortest path from 1 to 3:",
shortest_path)
```

5. **Visualization**: You can also visualize the graph using NetworkX's built-in functions or export it to other visualization libraries like matplotlib.

python

```
import matplotlib.pyplot as plt # Draw the graph nx.draw(G, with_labels=True) plt.show()
```

These are just some basic examples of what you can do with NetworkX. Depending on your specific dataset and analysis goals, there are many more advanced techniques and algorithms available in the NetworkX library.

22 - Datasets: Analysing Using Gephi

Analyzing datasets using Gephi involves several steps, including data preparation, importing data into

Gephi, and then applying various analysis techniques within Gephi. Here, I'll outline the general process along with some theory and Python code for data preparation.

Step 1: Data Preparation

Before we can analyze data in Gephi, we need to prepare it. This typically involves loading the data into Python, cleaning it, and transforming it into a format that Gephi can understand (such as CSV).

Let's assume we have a dataset in a CSV file with two columns representing edges between nodes (e.g., a social network). We'll use the popular **pandas** library to load and prepare the data.

pythonCopy code

import pandas as pd # Load the dataset data = pd.read_csv('your_dataset.csv') # Data cleaning (if required) # Perform any necessary transformations # Save the cleaned data to a new CSV file data.to_csv('cleaned_data.csv', index=False)

Step 2: Importing Data into Gephi

Once the data is prepared, we can import it into Gephi. Open Gephi and follow these steps:

1. Click on "File" > "Import Spreadsheet".
2. Choose the cleaned CSV file.
3. Specify how the data should be imported (e.g., edges table, nodes table).
4. Click "Next" and follow the prompts to configure import settings if needed.
5. Click "Finish" to import the data.

Step 3: Analysis in Gephi

After importing the data, you can perform various analyses using Gephi's built-in tools. Some common analyses include:

- **Centrality Analysis**: Measures the importance of nodes within a network.
- **Community Detection**: Identifies groups of nodes that are densely connected internally but sparsely connected to other groups.
- **Network Diameter Calculation**: Measures the maximum distance between any pair of nodes in the network.
- **Graph Clustering Coefficient**: Measures the degree to which nodes in a graph tend to cluster together.

Example Code for Analysis (Python)

Let's say we want to calculate the degree centrality of nodes in our network using Python before visualizing it in Gephi. We can use the **NetworkX** library for this purpose.

pythonCopy code

import networkx as nx # Load the cleaned data into a NetworkX graph G = nx.read_edgelist('cleaned_data.csv', delimiter=',') # Calculate degree centrality degree_centrality = nx.degree_centrality(G) # Print degree centrality for each node for node, centrality in degree_centrality.items(): print(f"Node {node}: Degree Centrality = {centrality}")

This Python code calculates the degree centrality for each node in the network and prints the results.

23 - Introduction : Emergence of Connectedness

The emergence of connectedness as a focal point can be attributed to several factors. Firstly, the proliferation of digital technologies has led to an exponential increase in the generation and availability of data. This data deluge has paved the way for the study of complex systems and networks, where understanding the relationships and interactions between components is paramount.

Moreover, the interconnected nature of modern society has highlighted the importance of studying and harnessing connectedness for various purposes, including communication, collaboration, and decision-making. From social media platforms facilitating global communication to transportation networks ensuring efficient movement of people and goods, connectedness underpins the functioning of numerous systems.

Furthermore, advancements in network analysis techniques and tools have provided researchers and practitioners with powerful means to explore and analyze complex networks. Whether it's measuring centrality, identifying communities, or detecting patterns of influence, these analytical methods offer insights into the structure and dynamics of interconnected systems.

24 - Advanced Material : Emergence of Connectedness

In the context of social networks, the emergence of connectedness refers to the way individuals, groups, and communities interconnect and interact within online platforms or real-life social systems. This concept has gained significant attention with the advent of digital technologies and social media platforms, which facilitate unprecedented levels of connectivity and communication among individuals worldwide. Here are some key aspects of the emergence of connectedness in social networks:

1. **Global Interconnectedness**: Social media platforms such as Facebook, Twitter, and Instagram have connected people across the globe, breaking down geographical barriers and enabling instantaneous communication and information sharing. This interconnectedness has facilitated the formation of virtual communities based on shared interests, beliefs, or identities, transcending traditional boundaries of time and space.
2. **Network Structure and Dynamics**: Social networks exhibit complex structures characterized by nodes (individuals or entities) and edges (connections or relationships) between them. The emergence of connectedness in social networks involves the study of network topology, dynamics, and emergent properties arising from the interactions between nodes. Network analysis techniques such as centrality, clustering, and community detection provide insights into the structure and evolution of social networks.
3. **Information Diffusion and Virality**: Connectedness in social networks facilitates the rapid spread of information, ideas, and trends through social contagion processes. Viral phenomena, such as viral videos, memes, or hashtags, often emerge from the interconnectedness of individuals sharing and resharing content within their social circles. Understanding the mechanisms underlying information diffusion in social networks is crucial for predicting trends, influencing public opinion, and controlling the spread of misinformation.
4. **Social Influence and Homophily**: Connectedness within social networks influences individual behavior, attitudes, and preferences through processes of social influence and homophily.

Homophily refers to the tendency of individuals to form connections with others who are similar to them in terms of demographics, interests, or opinions. Social influence mechanisms, such as peer pressure, conformity, and social validation, amplify the effects of connectedness on shaping collective behaviors and cultural norms within social networks.

5. **Network Effects and Platform Dynamics**: Social media platforms leverage network effects, where the value of the network increases with the number of users and their interconnectedness. The emergence of connectedness in social networks drives network effects by fostering user engagement, content creation, and network growth. Platform dynamics, such as algorithmic curation, recommendation systems, and user interactions, further shape the emergent properties of social networks and influence user experiences.
6. **Privacy, Security, and Ethical Considerations**: The emergence of connectedness in social networks raises important privacy, security, and ethical concerns regarding data privacy, information security, and algorithmic biases. As individuals become increasingly interconnected and share personal information online, issues such as data breaches, online harassment, and manipulation of social networks for malicious purposes pose significant challenges for policymakers, regulators, and platform operators.
7. **Social Capital and Collective Action**: Connectedness within social networks can generate social capital, defined as the resources embedded in social relationships and networks, such as trust, reciprocity, and cooperation. Social capital facilitates collective action, collaborative problem-solving, and community resilience in response to social challenges and crises. Understanding how connectedness fosters social capital formation and mobilization is essential for promoting positive social change and addressing collective problems.

25 -Programming Illustration : Emergence of Connectedness

the emergence of connectedness in programming through a simple example using Python, a widely used programming language. We'll create a program that represents a social network where individuals can connect with each other and share information.

pythonCopy code

```
class Person: def __init__(self, name): self.name = name self.connections = set() # Set to store connected persons def connect(self, other_person): self.connections.add(other_person) other_person.connections.add(self) def get_connections(self): return self.connections # Creating some people alice = Person("Alice") bob = Person("Bob") charlie = Person("Charlie") dave = Person("Dave") # Connecting people alice.connect(bob) alice.connect(charlie) bob.connect(charlie) charlie.connect(dave) # Printing connections print("Alice's connections:", [person.name for person in alice.get_connections()]) print("Bob's connections:", [person.name for person in bob.get_connections()]) print("Charlie's connections:", [person.name for person in charlie.get_connections()]) print("Dave's connections:", [person.name for person in dave.get_connections()])
```

Output:

lessCopy code

```
Alice's connections: ['Bob', 'Charlie'] Bob's connections: ['Alice', 'Charlie'] Charlie's connections: ['Alice', 'Bob', 'Dave'] Dave's connections: ['Charlie']
```

Explanation:

- We define a **Person** class with attributes **name** and **connections**, where **connections** is a set to store connected persons.
- The **connect** method establishes a connection between two persons by adding each other to their respective **connections** sets.
- We create four instances of **Person**: Alice, Bob, Charlie, and Dave.
- We establish connections between individuals using the **connect** method.
- Finally, we print out the connections for each person.

26 - Summary to Datasets
Datasets play a crucial role in numerous fields, serving as repositories of structured or unstructured data that researchers, analysts, and practitioners leverage for various purposes. Here's a concise summary highlighting the significance, types, and applications of datasets:
Significance of Datasets: Datasets serve as the foundation for empirical research, analysis, and machine learning model development across diverse domains. They enable researchers and practitioners to derive insights, make informed decisions, and build predictive models based on real-world data.

Week 3: Strength of Weak Ties in Social Network

27 – Introduction

The concept of "strength of weak ties" in social networks was introduced by sociologist Mark Granovetter in his seminal paper titled "The Strength of Weak Ties," published in the American Journal of Sociology in 1973. Granovetter's work challenged the prevailing belief that strong ties (close friends, family) were the most important for individuals in finding jobs or accessing new information.

Granovetter argued that weak ties (acquaintances, distant connections) often provide more novel information and opportunities than strong ties due to their position in different social circles or networks. Weak ties act as bridges between clusters of strong ties, enabling the flow of diverse information. This diversity is crucial because strong ties tend to share similar information, limiting exposure to new ideas or opportunities.

The strength of weak ties lies in their ability to provide access to resources, such as job leads, information about job openings, or new perspectives on problem-solving. Weak ties offer a broader range of contacts and experiences than strong ties, making them valuable for accessing non-redundant information and opportunities.

28 - Granovetter's Strength of weak ties

Granovetter’s Strength of Weak Ties in Social Networks

In Social Networks there are two kinds of relationships or ties :

- **Strong Tie :**
  Strong relationship exists between close members with frequent interactions or meetings. Example – family members and close friends cause strong ties.

- **Weak Tie :**
  Weak relationship is caused by distant social relationships and very infrequent meetings or interactions. Example – Acquaintances and strangers cause weak ties.

In a scenario, there is a person and he has 4 friends out of which 3 are in the same company and 1 friend works in a different company. Now as shown in the figure, there occurs 2 communities, one in which Z and his 3 friends work and the other one friend works.

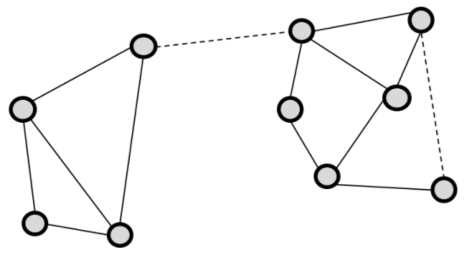


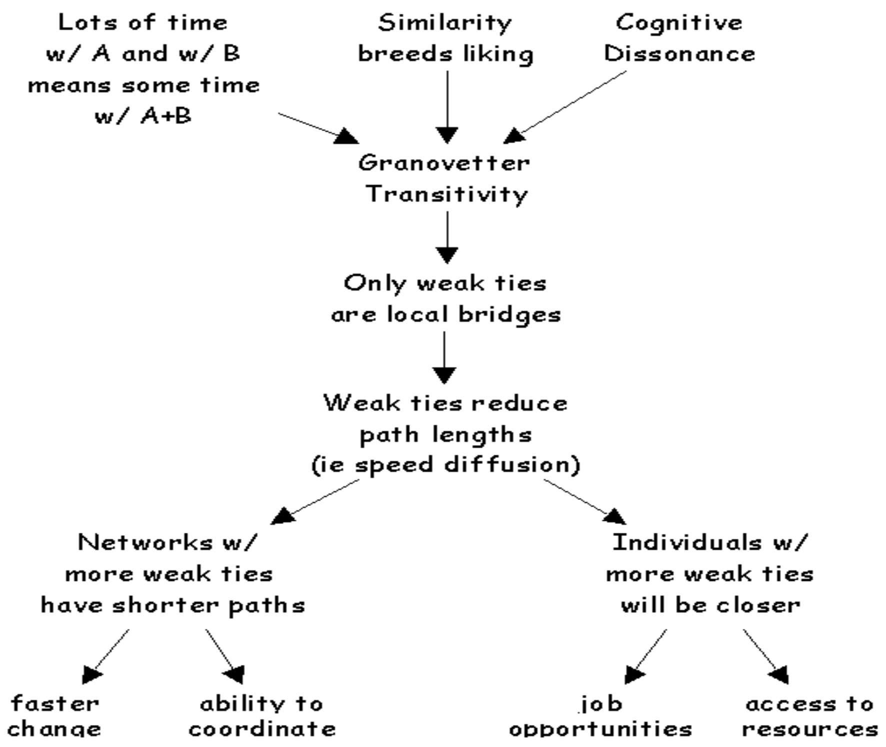
Figure – Strong and Weak Ties

In the above figure, the dotted line represents a weak tie and the plain line represents a strong tie. Now, in 1973 a Stanford professor published a paper called the **Strength of Weak Ties**. According to Granovetter's paper, the strength of weak ties is said that each tie has its different perspective and advantages as well and everyone talks about the advantages of strong ties but there are also some advantages of weak ties as well which is absent in strong ties.

Motivation for Granovetter's Theory

- Careers, job changing. Why is it that people so often get jobs from weak ties?
- How do large groups coordinate to make things happen, for example to meet a threat from outside
- Since then it has developed into a larger perspective known as embeddedness, which holds that all economic action, including that by organizations, is enabled and constrained and shaped by social ties among individuals

Overview of the Theory


Lots of time w/ A and w/ B means some time w/ A+B
Similarity breeds liking
Cognitive Dissonance
Granovetter Transitivity
Only weak ties are local bridges
Weak ties reduce path lengths (ie speed diffusion)
Networks w/ more weak ties have shorter paths
Individuals w/ more weak ties will be closer
faster change
ability to coordinate
job opportunities
access to resources

Implications

- Individuals with more weak ties have greater opportunities for mobility
- Coser's theory of autonomy (built on Simmel): lots of weak ties provide "seedbed of individual autonomy". People with many weak ties [Toennies' Gesellschaft] live up to the expectations of several others in different places and at different times, which makes it possible to preserve an inner core — to withhold inner attitudes while conforming to various expectations. People with strong ties [Gemeinschaft] share norms so thoroughly that little effort is needed to gauge intentions of others
- Relates difference to Basil Bernstein's distinction between restricted and elaborated codes of communication. Elaborated are complex and universal. More reflection is needed in organizing ones communication to very different people. [weak ties]
    - In elaborated speech there is high level of individualism, as it results from the ability to put oneself in imagination in the position of each role partner.
    - Social structure of poor is strong tie based, which does not encourage complex role set that in turn develops intellectual flexibility and self-direction.
    - Weak ties --> complex role sets --> cognitive flexibility --> ability of communities to organize. Complex voluntary orgs may depend on a habit of mind permits one to assess the needs, motives actions of a variety of people simultaneously.
- Adoption of innovation: made difficult by strong ties
- Mobilizing for change in response to environmental jolts:
    - Italian community of the west end in boston in 1962 were unable to fight "urban renewal" process which destroyed it. Gans attributes to working class culture (but other working class neighborhoods have succeeded).

- Divided into kinship and lifelong friendship cliques that were relatively closed. Unable to connect across cliques. People's work was outside the community, so no sources of informal ties.
- More weak ties, more capable of acting in concert. Strong ties breed local cohesion and macro fragmentation

29 - Triads, clustering coefficient and neighborhood overlap

1. **Triads**:

Theory: Triads are fundamental building blocks in social network analysis, consisting of three nodes and the relationships between them. There are various types of triads, classified based on the patterns of ties among the three nodes. These patterns provide insights into the structural properties and dynamics of the network.

Example: Consider a simple social network with individuals A, B, and C. Triads in this network can include situations where:

- A is friends with B, B is friends with C, and A is also friends with C (forming a closed or transitive triad).
- A is friends with B, B is friends with C, but A and C are not directly connected (forming an open or intransitive triad).
- A is friends with B, but B and C are not connected (forming another type of open triad).

Analyzing the prevalence and types of triads in a network provides insights into concepts such as transitivity, balance, and social cohesion.

2. **Clustering Coefficient**:

Theory: The clustering coefficient quantifies the degree to which nodes in a network tend to cluster together. It measures the proportion of triangles (closed triads) among the set of all possible connected triples in the network. A high clustering coefficient indicates a high level of local clustering or cohesion in the network, while a low clustering coefficient suggests a more random or sparse structure.

Example: In a social network, suppose A is friends with B and C, and B is also friends with C. Here, A, B, and C form a closed triad, indicating a high level of clustering. Conversely, if A is friends with B, B is friends with C, but A and C are not directly connected, the clustering coefficient would be lower.

3. **Neighborhood Overlap**:

Theory: Neighborhood overlap measures the extent to which the neighborhoods of two nodes in a network share common neighbors. It quantifies the similarity or overlap in the sets of nodes connected to each of the two nodes. High neighborhood overlap between two nodes indicates a significant overlap in their social connections, while low neighborhood overlap suggests less similarity in their social circles.

Example: Suppose in a social network, A is friends with B, C, and D, while B is friends with C, D, and E. Here, the neighborhood overlap between A and B is high because they share common friends C and D. Conversely, if A is friends with B, but B has entirely different connections, the neighborhood overlap between A and B would be low.

30 - Structure of weak ties, bridges, and local bridges

Weak Ties:

Weak ties refer to connections in a social network that are not as close or strong as those between close friends or family members. These ties typically involve acquaintances or distant connections. Mark Granovetter introduced the concept of weak ties in his seminal work on the strength of weak ties, highlighting their importance in providing access to diverse information and opportunities outside one's immediate social circle.

Bridges and Local Bridges

A has 4 friends:

– C, D, and E connected to a tightly-knit group

– B reaches into a different part of the network

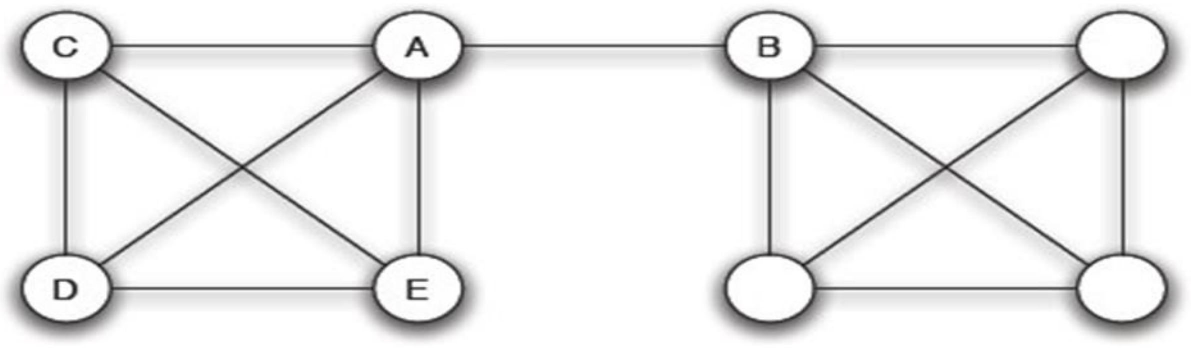
– B offers access to new things

Bridge

Bridges are nodes or edges in a network that connect otherwise disconnected components or clusters. They act as crucial links between different parts of the network, facilitating communication and interaction between otherwise isolated groups. Removing bridges can lead to the fragmentation of the network into smaller disconnected components.

Edge A-B is a bridge if deleting it causes A and B to be in different components

Bridges are extremely rare in real social networks – giant component, many short paths



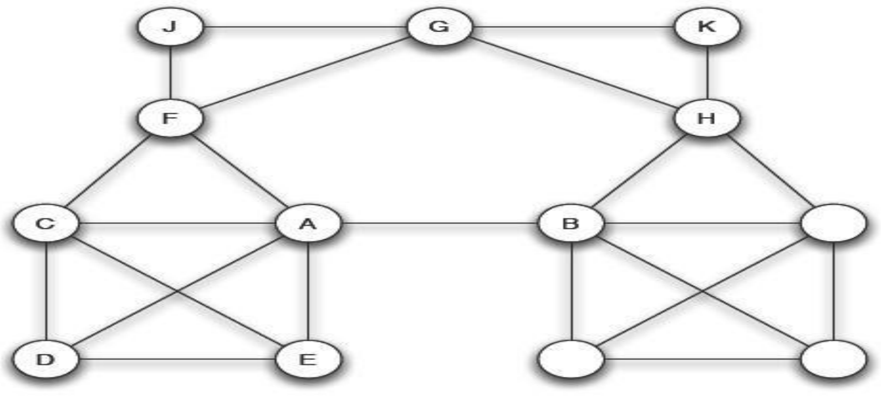
local bridge

Local bridges are edges in a network that connect nodes that are not part of the same neighborhood, but are close enough to be considered local. Unlike bridges, which connect disconnected components of the network, local bridges connect nodes within the same overall connected component but belong to different local neighborhoods. Removing local bridges can disrupt the flow of information between these neighborhoods.

Edge A-B is a local bridge if its endpoints A and B have no friends in common – deleting A-B => d(A,B) increases more than 2

Relation with triadic closure: – a local bridge does not belong to any triangle

Local bridges provide their endpoints with access to parts of the network that they would otherwise be far away from



31 - Validation of Granovetter's experiment using cell phone data

To validate Granovetter's experiment on the "Strength of Weak Ties" using cell phone data, we can follow these steps:

1. Collect cell phone data: Obtain a large-scale dataset that includes anonymized call detail records, with information such as caller ID, callee ID, timestamp, and connected cell tower ID.
2. Preprocess the data: Clean and preprocess the data to remove any inconsistencies or errors. Ensure the dataset has at least a year's worth of data, covers a wide geographical area, and includes a significant portion of the population.
3. Calculate the distance between individuals: Based on the connected cell tower ID, calculate the distance between individuals in each call. This can be done using the Haversine formula to estimate the great-circle distance between two points on a sphere given their longitudes and latitudes.
4. Analyze the frequency and duration of calls: Based on the call records, analyze the frequency and duration of calls between individuals. Define the strength of a tie as a combination of the frequency and duration of calls.
5. Categorize ties as strong or weak: Based on the strength of ties, categorize the relationships between individuals as strong or weak.
6. Measure the geographical distance between individuals: Using the connected cell tower ID, calculate the geographical distance between each pair of individuals in the social network.
7. Analyze the relationship between tie strength and geographical distance: Investigate the relationship between the strength of ties and the geographical distance between individuals. Compare the distribution of distances between strong and weak ties.
8. Validate Granovetter's hypothesis: Based on the analysis, validate Granovetter's hypothesis that weak ties play a crucial role in providing access to new information and opportunities, as they are more likely to facilitate connections between geographically distant individuals.

Here is a Python code snippet to perform the distance calculation:

```
import math
```

```
def haversine_distance(lat1, lon1, lat2, lon2):
   R = 6371  # Radius of the Earth in kilometers
   dlat = math.radians(lat2 - lat1)
   dlon = math.radians(lon2 - lon1)
   a = (math.sin(dlat / 2) * math.sin(dlat / 2) +
       math.cos(math.radians(lat1)) * math.cos(math.radians(lat2)) *
       math.sin(dlon / 2) * math.sin(dlon / 2))
   c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
   return R * c
# Example usage:
tower1_location = (38.7171, -9.1355)  # Lisbon
tower2_location = (40.6413, -8.6674)  # Porto
print(haversine_distance(*tower1_location, *tower2_location))  # ~332 km
```

This code calculates the distance between two cell towers in kilometers using their latitudes and longitudes. This can be used to estimate the geographical distance between individuals in each call.

32 – Emeddedness

Embeddedness in social network analysis refers to the extent to which relationships between individuals are embedded within broader social contexts or structures. It captures the degree to which connections between individuals are reinforced or supported by mutual connections to other individuals or groups within the network. Higher embeddedness implies greater interconnectedness or density within the network.

Here's a code snippet demonstrating how to calculate embeddedness in a social network using Python and the NetworkX library:

pythonCopy code

```
import networkx as nx # Example function to calculate embeddedness between two nodes in a network def calculate_embeddedness(G, node1, node2): # Get neighbors of node1 and node2 neighbors1 = set(G.neighbors(node1)) neighbors2 = set(G.neighbors(node2)) # Calculate embeddedness as the size of the intersection of neighbors embeddedness = len(neighbors1.intersection(neighbors2)) return embeddedness # Example social network graph (replace with your actual network data) G = nx.Graph() G.add_edges_from([('A', 'B'), ('A', 'C'), ('B', 'C'), ('C', 'D')]) # Example calculation of embeddedness between nodes A and B embeddedness_AB = calculate_embeddedness(G, 'A', 'B') print("Embeddedness between nodes A and B:", embeddedness_AB)
```

In this code snippet:

- We define a function **calculate_embeddedness** that takes a network **G** and two nodes **node1** and **node2** as input.
- Inside the function, we obtain the neighbors of each node using the **neighbors** method of the NetworkX graph object.
- We then calculate the embeddedness as the size of the intersection of neighbors between the two nodes.
- Finally, we demonstrate the usage of this function by calculating the embeddedness between nodes A and B in an example social network graph.

33 - Structural Holes

Structural holes refer to the gaps or structural gaps between different individuals or groups within a social network. These gaps represent opportunities for accessing diverse sources of information, control over resources, and brokerage positions. Understanding structural holes is crucial for analyzing social capital, innovation, and competitive advantage within networks.

Here's a code snippet demonstrating how to identify structural holes in a social network using Python and the NetworkX library:

pythonCopy code

```
import networkx as nx # Example function to identify structural holes in a social network def
identify_structural_holes(G): # Calculate the betweenness centrality for each node betweenness =
nx.betweenness_centrality(G) # Find the node with the highest betweenness centrality
max_betweenness_node = max(betweenness, key=betweenness.get) # Find the neighbors of the node with
the highest betweenness centrality neighbors = set(G.neighbors(max_betweenness_node)) # Find the
potential structural holes (nodes not connected to each other) structural_holes = [] for neighbor1 in
neighbors: for neighbor2 in neighbors: if neighbor1 != neighbor2 and not G.has_edge(neighbor1,
neighbor2): structural_holes.append((neighbor1, neighbor2)) return structural_holes # Example social
network graph (replace with your actual network data) G = nx.Graph() G.add_edges_from([('A', 'B'), ('A',
'C'), ('B', 'C'), ('C', 'D'), ('D', 'E')]) # Example identification of structural holes structural_holes =
identify_structural_holes(G) print("Structural Holes:", structural_holes)
```

In this code snippet:

- We define a function **identify_structural_holes** that takes a network **G** as input.
- Inside the function, we calculate the betweenness centrality for each node using the **betweenness_centrality** function from NetworkX.
- We then identify the node with the highest betweenness centrality, which represents a potential broker or connector in the network.
- Next, we find the neighbors of the node with the highest betweenness centrality.
- Finally, we identify potential structural holes by finding pairs of neighbors that are not directly connected to each other.
- We demonstrate the usage of this function by identifying structural holes in an example social network graph.

34 - Social Capital

Social capital refers to the resources, benefits, and advantages that individuals or groups derive from their social networks and social relationships. It encompasses the trust, norms, reciprocity, and social cohesion within a community or network that facilitates cooperation, collaboration, and collective action. Social capital can manifest in various forms, including bonding social capital (within-group connections), bridging social capital (between-group connections), and linking social capital (connections to external institutions or resources).

In the context of social networks, social capital emphasizes the value embedded within the structure and dynamics of relationships between individuals, rather than focusing solely on individual attributes or resources. It recognizes that the connections and interactions within a network can generate benefits such as access to information, social support, opportunities, and resources that contribute to individual and collective well-being.

Here's how social capital manifests in social networks:

1. **Bonding Social Capital**: This form of social capital refers to the strong ties and connections within a tightly-knit group or community. These connections are characterized by high levels of trust, reciprocity, and mutual support. Bonding social capital fosters a sense of belonging, identity, and solidarity among group members, which can provide emotional support, facilitate cooperation, and reinforce group norms and values.
2. **Bridging Social Capital**: Bridging social capital involves connections between individuals or groups from different social, cultural, or economic backgrounds. These connections serve as bridges between diverse communities or social groups, facilitating the exchange of information, resources, and perspectives. Bridging social capital enhances social cohesion, reduces social fragmentation, and promotes understanding, tolerance, and collaboration across social divides.
3. **Linking Social Capital**: Linking social capital refers to connections between individuals or groups and external institutions, organizations, or resources. These connections provide access to formal systems of power, influence, and support beyond the immediate social network. Linking social capital enables individuals or groups to access opportunities, resources, and services provided by external institutions, such as government agencies, businesses, or non-profit organizations.

35 - Tie Strength, Social Media and Passive Engagement

Tie strength refers to the degree of closeness or intensity in a relationship between individuals within a social network. It encompasses various factors such as emotional intensity, frequency of interaction, intimacy, reciprocity, and the breadth of shared activities or experiences. Strong ties typically involve close relationships characterized by trust, emotional support, and mutual dependence, such as family members or close friends. Weak ties, on the other hand, are more casual or distant connections, such as acquaintances or colleagues.

In the context of social media, tie strength plays a crucial role in shaping the dynamics of online interactions and engagement. Here's how tie strength relates to social media and passive engagement:

1. **Tie Strength in Online Relationships**: On social media platforms, individuals can maintain connections with a wide range of acquaintances, friends, and followers, spanning from strong ties to weak ties. The strength of these online relationships influences the nature and depth of interactions. Strong ties may involve more personalized and intimate exchanges, such as private messages, while weak ties may entail more superficial interactions, such as likes, comments, or occasional status updates.
2. **Passive Engagement**: Passive engagement refers to the act of consuming or observing content on social media without actively contributing or interacting with it. Examples of passive engagement

include scrolling through a news feed, watching videos, or reading posts without liking, sharing, or commenting. Passive engagement is common on social media platforms, where users often consume content passively without necessarily engaging in active interactions.

3. **Relationship Between Tie Strength and Passive Engagement**: Tie strength can influence the likelihood and frequency of passive engagement on social media. Individuals are more likely to passively consume content shared by strong ties, such as close friends or family members, as they may be more interested in their activities or updates. In contrast, passive engagement with content shared by weak ties or distant connections may be less frequent, as individuals may have less investment or interest in their updates.
4. **Implications for Social Media Strategy**: Understanding the relationship between tie strength and passive engagement can inform social media strategy and content creation. For example, businesses and content creators may prioritize building strong ties with their audience to encourage more active engagement and interactions. They may also leverage algorithms and targeting strategies to reach specific segments of their audience with content tailored to their interests and preferences, thereby increasing the likelihood of passive engagement.

36 - Betweenness Measures and Graph Partitioning

1. **Betweenness Measures in Social Networks**:
    - In social networks, betweenness centrality can identify individuals who act as bridges or intermediaries between different groups or communities.
    - Individuals with high betweenness centrality often control the flow of information between different parts of the network. They may serve as brokers, connectors, or opinion leaders.
    - Identifying individuals with high betweenness centrality can be valuable for understanding information diffusion, influence propagation, and the overall robustness of the social network.

    This code creates a simple social network graph using the NetworkX library, with nodes representing individuals and edges representing friendships. It then calculates the betweenness centrality for each node in the graph using the **betweenness_centrality()** function provided by NetworkX. Finally, it prints out the betweenness centrality values for each node in the network.

```
import networkx as nx
# Create an example social network graph
G = nx.Graph()
# Add nodes representing individuals
G.add_nodes_from(["Alice", "Bob", "Charlie", "David", "Eve", "Frank"])
# Add edges representing friendships
G.add_edges_from([("Alice", "Bob"), ("Alice", "Charlie"), ("Bob", "David"), ("Charlie", "David"),
("Charlie", "Eve"), ("David", "Eve"), ("David", "Frank"), ("Eve", "Frank")])
# Calculate betweenness centrality for each node
betweenness_centrality = nx.betweenness_centrality(G)
# Print betweenness centrality values for each node
for node, centrality in betweenness_centrality.items():
    print(f"{node}: {centrality}")
```

2. **Graph Partitioning in Social Networks**:
    - Graph partitioning techniques can be applied to social networks to uncover communities or groups of individuals with strong internal connections.
    - Communities in social networks represent groups of individuals who share common interests, affiliations, or interactions.
    - Partitioning a social network into communities can help in targeted marketing, recommendation systems, understanding social dynamics, and identifying influential groups or subcultures.
    - Graph partitioning algorithms aim to maximize the modularity or cohesion within communities while minimizing the connectivity between them.

Graph partitioning in social networks involves dividing the network into disjoint subsets or communities based on the relationships between individuals. The goal is to identify groups of individuals who share strong connections within their group while having fewer connections to individuals outside their group. This process helps uncover underlying structures, communities, or clusters within the social network, which can provide valuable insights for various applications.

Here's an example code demonstrating graph partitioning using the Louvain method, one of the popular community detection algorithms, implemented in Python with the NetworkX library:

Python code

```
import networkx as nx
import community  # Louvain method
# Create an example social network graph
G = nx.Graph()
# Add nodes representing individuals
G.add_nodes_from(["Alice", "Bob", "Charlie", "David", "Eve", "Frank"])
# Add edges representing friendships
G.add_edges_from([("Alice", "Bob"), ("Alice", "Charlie"), ("Bob", "David"), ("Charlie", "David"),
("Charlie", "Eve"), ("David", "Eve"), ("David", "Frank"), ("Eve", "Frank")])
# Apply Louvain method for graph partitioning
partition = community.best_partition(G)
# Print the partition result
print(partition)
```

This code creates a social network graph using NetworkX and then applies the Louvain method for graph partitioning. The best_partition() function from the community module is used to partition the graph into communities. Each node is assigned a community label, and the partition result is printed out.

The Louvain method optimizes a modularity function to identify communities in the graph by iteratively moving nodes between communities to maximize modularity, a measure of the quality of the partitioning. The resulting partition provides insights into the underlying community structure of the social network.

Graph partitioning in social networks can help in various applications such as targeted advertising, recommendation systems, understanding social dynamics, and identifying influential communities or subgroups within the network.

3. **Relationship between Betweenness Measures and Graph Partitioning**:
    - Betweenness measures can inform the partitioning process by identifying pivotal individuals whose presence or absence may significantly impact the structure of communities.
    - Nodes with high betweenness centrality might act as natural boundaries between communities, and their inclusion or exclusion in different partitions can influence the partitioning quality.
    - Integrating betweenness measures into the graph partitioning process can ensure that influential individuals are appropriately distributed across communities, avoiding their isolation or clustering in specific groups.

37 - Finding Communities in a graph (Brute Force Method) – 1

Community detection in social networks using brute-force method

We are going to divide the nodes of the graph into two or more communities using the brute force method. The brute force method means we will try every division of nodes into communities and check whether the communities are correctly divided or not. We will use a brute force method for this task.

Algorithm:

1. Create a graph of N nodes and its edges or take an inbuilt graph like a barbell graph.
2. Now take two lists as FirstCommunity and SecondCommunity.
3. Now start putting nodes into communities like put 1st node in FirstCommunity and rest N-1 nodes to SecondCommunity and check its inter and intra edges.
4. Now we will make combinations using itertools.
5. Repeat steps 3 and 4 for every combination.
6. Now check which division is best by taking the ratio of intra/number of inter-community edges.
7. Now find the value of FirstCommunity and SecondCommunity with maximum ratio and print that value.

Below is the implementation.

- Python3

```
import networkx as nx
import itertools
def communities_using_brute(gfg):
  nodes = gfg.nodes()
  n = gfg.number_of_nodes()
  first_community = []
  for i in range(1, n//2 + 1):
    c = [list(a) for a in itertools.combinations(nodes, i)]
    first_community.extend(c)
  second_community = []
  for i in range(len(first_community)):
    b = list(set(nodes)-set(first_community[i]))
    second_community.append(b)
  # Which division is best...
  intra_edges1 = []
  intra_edges2 = []
  inter_edges = []
  # ratio of number of intra/number of inter
  # community edges
  ratio = []
  for i in range(len(first_community)):
    intra_edges1.append(gfg.subgraph(first_community[i]).number_of_edges())
  for i in range(len(second_community)):
    intra_edges2.append(gfg.subgraph(second_community[i]).number_of_edges())
  e = gfg.number_of_edges()
  for i in range(len(first_community)):
    inter_edges.append(e-intra_edges1[i]-intra_edges2[i])
  # Calculate the Ratio
  for i in range(len(first_community)):
    ratio.append((float(intra_edges1[i]+intra_edges2[i]))/inter_edges[i])
  maxV=max(ratio)
  mindex=ratio.index(maxV)
  print('[ ', first_community[mindex], ' ] , [ ', second_community[mindex], ' ]')
# Example graph
gfg=nx.barbell_graph(5, 0)
communities_using_brute(gfg)
```

Output:

```
[ [0,1,2,3,4] ] , [ [8,9,5,6,7] ]
```

38 - Community Detection Using Girvan Newman Algorithm

What is Community Detection? Community detection methods locate communities based on network structure, such as strongly connected groupings of nodes; however, they often ignore node properties. Nodes of similar types form a community in a network. Intra-community edges are the edges that connect the nodes in a community.

The Girvan-Newman algorithm is a method for detecting communities or clusters within complex networks. It was proposed by Michelle Girvan and Mark Newman in 2002 and has since become a widely used approach for community detection.

Theory:

1. **Background**:
    - **Complex Networks**: Networks representing complex systems where nodes represent entities (e.g., individuals, websites) and edges represent relationships or interactions between them (e.g., friendships, hyperlinks).
    - **Community Structure**: The presence of densely connected groups of nodes within networks, often referred to as communities or clusters.
2. **Objective**:
    - Identify the underlying community structure within a given network.
3. **Key Idea**:
    - The algorithm iteratively removes edges from the network based on their betweenness centrality until communities emerge.
4. **Algorithm Steps**:

a. **Betweenness Centrality Calculation**:

- Compute the betweenness centrality for all edges in the network.
- Betweenness centrality measures the number of shortest paths passing through an edge. Edges with high betweenness centrality are considered important for connecting different parts of the network.

b. **Edge Removal**:

- Remove the edge(s) with the highest betweenness centrality. This step disconnects the network or breaks it into smaller components.

c. **Community Detection**:

- Calculate the number of connected components (communities) in the modified network.
- If the number of components increases after edge removal, update the communities accordingly.

d. **Modularity Maximization**:

- At each step, evaluate the modularity of the partitioned network.
- Modularity measures the quality of the partitioning, indicating how well the network is divided into communities.
- The algorithm aims to maximize modularity, indicating a better community structure.

e. **Iterative Process**:

- Repeat steps (a) to (d) until no edges remain in the network or until a stopping criterion is met.

5. **Output**:
    - The algorithm outputs the detected communities based on the partitioning that maximizes modularity.

Applications:

- **Social Network Analysis**: Identifying communities in social networks such as friendship networks or online social platforms.
- **Biological Networks**: Discovering functional modules in biological networks like protein-protein interaction networks or gene regulatory networks.
- **Web Graph Analysis**: Identifying clusters of interconnected web pages to improve search engine algorithms.

Advantages:

- **Scalability**: The algorithm is applicable to large-scale networks.
- **Flexibility**: It can detect communities in various types of networks without relying on specific assumptions about the community structure.

Limitations:

- **Edge Removal Bias**: The algorithm's performance may depend on the order in which edges are removed, potentially leading to biased results.
- **Resolution Limit**: In some cases, the algorithm may fail to detect small or densely connected communities due to the resolution limit problem.
- **Computational Complexity**: The algorithm's computational complexity can be high, especially for large networks, although optimizations and parallelization techniques can mitigate this issue.

Types of Community Detection

There are primarily two types of methods for detecting communities in graphs:

Agglomerative Methods

In agglomerative methods, we start with an empty graph that consists of nodes of the original graph but no edges. Next, the edges are added one-by-one to the graph, starting from “stronger” to “weaker” edges. This strength of the edge, or the weight of the edge, can be calculated in different ways.

Divisive Methods

In divisive methods, we go the other way round. We start with the complete graph and take off the edges iteratively. The edge with the highest weight is removed first. At every step, the edge-weight calculation is repeated, since the weight of the remaining edges changes after an edge is removed. After a certain number of steps, we get clusters of densely connected nodes.

implement the Girvan-Newman algorithm for community detection in Python! Here's an example of how to do it using the NetworkX library:

```
import matplotlib.pyplot as plt

import networkx as nx

from networkx.algorithms.community.centrality import girvan_newman
```

```
# Create a graph
G = nx.karate_club_graph()
 # Run the Girvan-Newman algorithm
 communities = girvan_newman(G)
# Get the first set of communities
node_groups = []
for com in next(communities):
    node_groups.append(list(com))
# Print the communities
print(node_groups)
# Color the nodes based on their community
color_map = []
for node in G:
    if node in node_groups[0]:
        color_map.append('blue')
    else:
        color_map.append('green')

# Draw the graph
nx.draw(G, node_color=color_map, with_labels=True)
plt.show()
```

This code will output the communities as a list of node groups:

```
[[0, 1, 3, 4, 5, 6, 7, 10, 11, 12, 13, 16, 17, 19, 21],
 [2, 8, 9, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33]]
```

And it will display the graph with the nodes colored according to their community:

39 - Visualising Communities using Gephi

Visualizing communities detected by algorithms like Girvan-Newman using Gephi can provide insights into the network's structure and the relationships between different communities. Below, I'll outline the steps to visualize communities using Gephi:

Step 1: Prepare Data

Ensure you have a network dataset with nodes and edges, along with community assignments for each node. You can export your network data from Python or any other tool into a format compatible with Gephi, such as GEXF or CSV.

Step 2: Import Data into Gephi

1. Open Gephi and create a new project.
2. Import your network data using the "Import Spreadsheet" option if your data is in CSV format or "Import GEXF" if it's in GEXF format.

Step 3: Analyze Network

1. Run algorithms like Girvan-Newman within Gephi or import community assignments if you've already computed them externally.
2. To run Girvan-Newman within Gephi, go to the "Statistics" tab, select "Modularity" from the list of available algorithms, and click "Run." This will partition the network into communities based on modularity optimization.
3. Alternatively, if you have community assignments, you can import them as node attributes.

Step 4: Visualize Communities

1. Go to the "Layout" tab and choose a layout algorithm (e.g., ForceAtlas 2) to arrange the nodes based on their relationships.
2. Adjust the parameters of the layout algorithm to achieve the desired visualization.
3. Go to the "Partition" tab and select the community attribute (either computed by Gephi or imported) from the dropdown menu. This will color the nodes according to their community assignments.
4. You can further customize the visualization by adjusting the colors, sizes, and shapes of nodes and edges to enhance clarity and interpretability.
5. Use Gephi's built-in filters and metrics to explore and analyze the community structure further. For example, you can filter nodes based on degree centrality or modularity class.

Step 5: Export Visualization

Once you're satisfied with the visualization, export it as an image or PDF for presentation or further analysis. You can also export the network data with community assignments for use in other tools or publications.

Other Visualizing Tools

1. **Neo4j:** The downside of this application is to use the data and manipulate it, you need to know Cypher Query Language(CQL) and the tool is less for analysis and more for discovering relationships in your database. For in-depth visualization, you can use the extension Neo4j bloom.

2. **KineViz GraphXR:** GraphXR is a browser-based interactive visual analytics platform. Neo4j has a GraphXR plugin, which takes the visualization to next level, with advanced options like GeoTagging and a VR space to move in. It is worht a try, just follow this tutorial!
3. **Keylines:** KeyLines is a toolkit used by developers to build sophisticated and powerful network visualization applications. These applications run completely in a web browser and can therefore be easily integrated into existing systems and dashboards, or as standalone applications. I haven't used personally, but give it a try!

To visualize communities detected in a network using Gephi, you first need to prepare your network data and community assignments. Then, you can follow the steps outlined above to import the data into Gephi and visualize it. Here's a Python code example to generate a network and export it in GEXF format, which can be imported into Gephi:

Python

```
import networkx as nx
import community
import random
import tkinter as tk
from tkinter import filedialog, messagebox
def generate_network():
    # Generate a random network with 100 nodes, average degree 5, and cluster coefficient 0.2
    G = nx.powerlaw_cluster_graph(100, 5, 0.2, seed=42)
    # Apply Louvain community detection algorithm
    partition = community.best_partition(G)
    for node, comm_id in partition.items():
        G.nodes[node]['community'] = comm_id
    # Export the network to GEXF format with node community information
    network_filename = filedialog.asksaveasfilename(defaultextension=".gexf",
                                                    filetypes=[("GEXF files", "*.gexf")])
    if network_filename:
        nx.write_gexf(G, network_filename)
        messagebox.showinfo("Success", "Network saved successfully!")
def main():
    root = tk.Tk()
    root.title("Network Generator")
    generate_button = tk.Button(root, text="Generate Network", command=generate_network)
    generate_button.pack(pady=10)
    root.mainloop()
if __name__ == "__main__":
    main()
```

Once you have the GEXF file containing your network and community assignments, you can import it into Gephi and follow the steps outlined in the previous response to visualize the communities.

40 - Strong and Weak Relationship – Summary

In social sciences and interpersonal dynamics, the terms "strong" and "weak" relationships refer to the intensity or depth of connections between individuals. Here's a summary of these concepts:

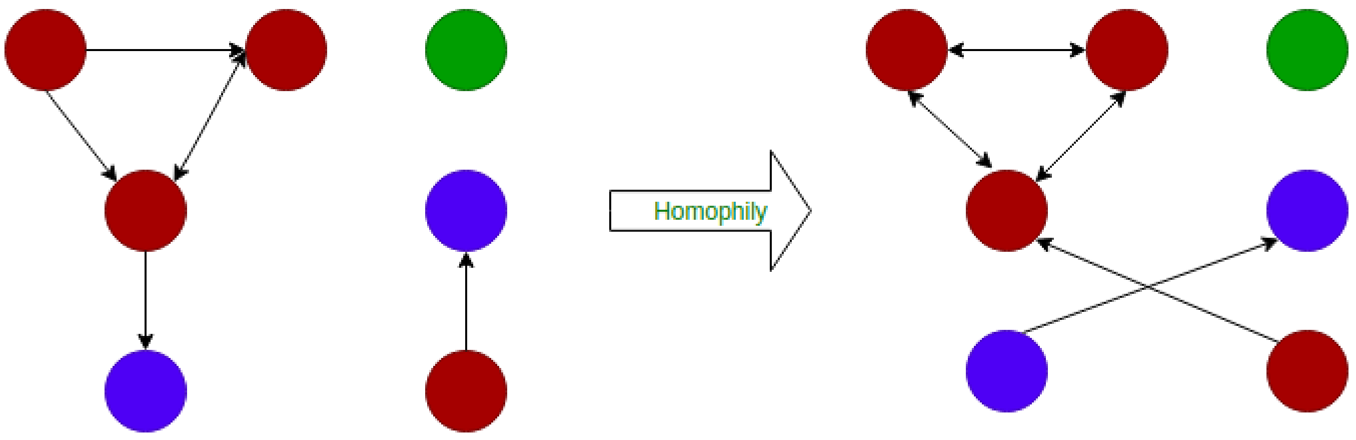
1. **Strong Relationships**:
    - **Definition**: Strong relationships are characterized by deep emotional bonds, trust, mutual respect, and frequent interaction.
    - **Features**:
        - Emotional Support: Individuals in strong relationships provide significant emotional support to each other.
        - Commitment: There's a high level of commitment and loyalty, often enduring through challenging times.
        - Communication: Open and honest communication is common, leading to better understanding and resolution of conflicts.
        - Mutual Understanding: There's a deep understanding of each other's needs, preferences, and boundaries.
    - **Examples**: Close friendships, romantic partnerships, familial bonds, and long-term professional relationships often exhibit characteristics of strong relationships.
2. **Weak Relationships**:
    - **Definition**: Weak relationships involve less emotional investment, lower levels of trust, and limited interaction.
    - **Features**:
        - Limited Emotional Support: Individuals may offer minimal emotional support to each other, especially during difficult times.
        - Lower Commitment: Weak relationships may lack a strong commitment, making them more susceptible to drifting apart.
        - Surface-Level Communication: Interactions are often superficial, lacking depth or meaningful exchange.
        - Limited Understanding: There's a shallower understanding of each other's feelings, leading to potential misunderstandings.
    - **Examples**: Acquaintances, casual friendships, and professional contacts with minimal interaction or shared interests typically fall under weak relationships.

Week 4: Strong and Weak Relationships (Continued) & Homophily in Social Network

41 - Introduction to Homophily - Should you watch your company ?

Homophily is the tendency in social groups of similar people connected together. We often hear similar voices interact with like-minded people. Homophily has a significant impact on social media.

Example –
Birds with feather flock together.



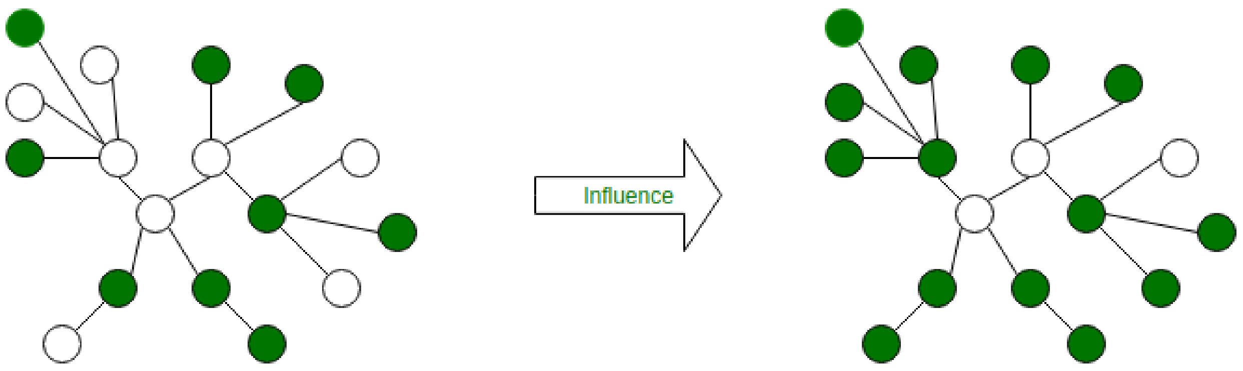
Example of Homophily

Assume there are 1000 people in a party out of which 500 are of age ranges from 18-25 and the other 500 people are from the age group 40-50. So mathematically if we pick any friendship randomly then mostly it will be one teenager and one middle-aged person which is most probable condition. But we all know that teenagers may want to talk to a teenager and middle-aged may want to talk to middle-aged people. This is known as Homophily.

Social Influence :
It is the tendency in which people change their attitude or behavior to meet the social environment by getting influenced by other people is called Social Influence. Social Influence makes connected nodes similar.

Example –

- Smoking.
- Drinking.



Example of Social Influence

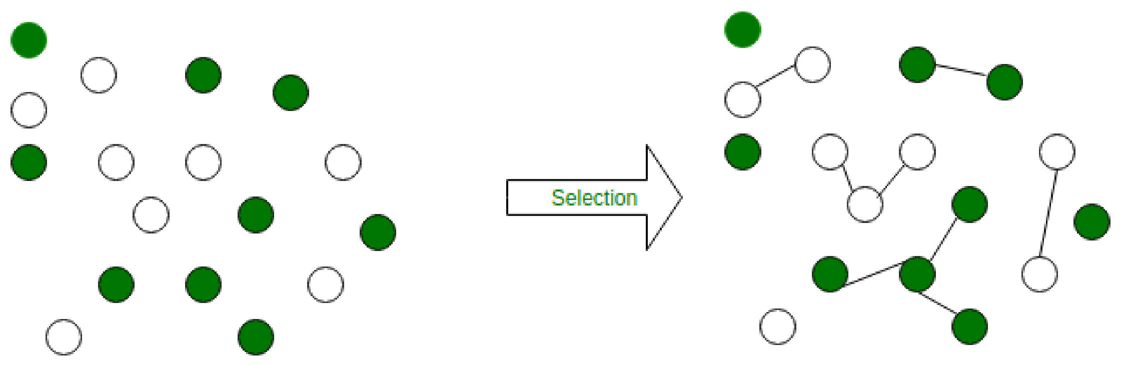
Suppose I have a friend who smokes so he will influence me to smoke. This is **Social Influence**.

42 - Selection and Social Influence

Selection :
It is the tendency in which people make friends with similar interests i.e people select other people having similar habits or interests. In selection, people select similar nodes and connects with them.

Example –

- Two people who speak the same language.



Example of Selection

Suppose there is a person who speaks Spanish and I also know Spanish so I will select that person and talk to him. This is called **Selection**

43 - Interplay between Selection and Social Influence

The interplay between selection and social influence is a fundamental aspect of various fields including sociology, psychology, evolutionary biology, and economics. It refers to how individual behaviors, traits, and preferences are shaped by both genetic factors (selection) and the influence of others within social networks (social influence).

1. **Selection**: Selection refers to the process by which certain traits or behaviors become more or less common in a population over time due to differential reproduction and survival rates. In biological terms, natural selection acts on genetic variation, favoring traits that enhance an individual's reproductive success. In cultural evolution, selection operates on ideas, behaviors, and cultural traits that are passed down through generations.

2. **Social Influence**: Social influence refers to the process by which individuals adapt their thoughts, feelings, and behaviors to conform to social norms, expectations, or pressures. It encompasses various mechanisms such as conformity, peer pressure, social learning, and imitation. Social influence can lead to the spread of ideas, behaviors, and cultural norms within a population.

The interplay between selection and social influence can manifest in several ways:

- **Gene-culture coevolution**: Humans possess a unique capacity for cultural transmission, allowing learned behaviors and norms to influence evolutionary processes. This coevolutionary process can lead to the emergence of cultural traits that shape genetic evolution and vice versa.

- **Cultural evolution**: Cultural traits can spread through populations via social learning and imitation, influencing the fitness of individuals. Traits that confer reproductive or social advantages are more likely to spread, leading to cultural evolution.
- **Gene-environment interactions**: Genetic factors may predispose individuals to certain behaviors or susceptibilities, which can interact with social environments to influence behavior. For example, genetic predispositions towards risk-taking behavior may interact with social norms to influence the likelihood of engaging in risky activities.
- **Social learning and innovation**: Individuals may learn from others within their social networks, leading to the spread of advantageous behaviors or innovations. This social learning process can shape the cultural landscape and influence the trajectory of cultural evolution.
- **Feedback loops**: Selection and social influence can interact in feedback loops, where cultural traits influence genetic evolution, which in turn shapes cultural evolution. These feedback loops can lead to the emergence of complex cultural dynamics and patterns of social behavior.
- **Evolutionary psychology**: Evolutionary psychologists study how human psychological mechanisms have evolved in response to selection pressures over time. These mechanisms may influence how individuals perceive and respond to social influences. For instance, the tendency to conform to social norms or imitate others may have evolved as adaptive strategies to navigate complex social environments and enhance social cohesion.
- **Cultural transmission:** Cultural transmission refers to the process by which cultural information is passed from one individual to another within a population. This transmission can occur through various mechanisms, including teaching, imitation, and language. The success of cultural transmission depends on factors such as the fidelity of information transfer, the credibility of the source, and the relevance of the information to the recipient's needs or goals.
- **Social networks:** Social networks play a crucial role in facilitating the spread of information, behaviors, and cultural norms within populations. Individuals are more likely to adopt behaviors or beliefs that are endorsed by their social connections, leading to the formation of clusters or communities with shared cultural traits. Social network analysis provides insights into how the structure of social networks influences the dynamics of cultural transmission and social influence.
- **Gene-culture coevolution:** Gene-culture coevolution refers to the reciprocal interactions between genetic and cultural evolution. Cultural practices and innovations can influence the selective pressures acting on human populations, leading to genetic adaptations that facilitate the transmission or adoption of cultural traits. Conversely, genetic factors may predispose individuals to certain cultural preferences or behaviors, shaping the trajectory of cultural evolution.
- **Cultural group selection**: Cultural group selection posits that cultural traits that enhance the survival and reproductive success of groups may be favored by selection, leading to the spread of those traits within and across populations. This perspective emphasizes the role of intergroup competition and cooperation in driving cultural evolution and shaping human social behavior.

44 - Homophily - Definition and measurement

What is homophily?

Homophily refers to the tendency of individuals to associate with others who are similar to them in some way, such as in attitudes, beliefs, behaviors, social status, ethnicity, or other characteristics. It's a concept often observed in social networks and communities, where people tend to form connections and relationships with others who share similar traits or interests.

There are various ways to measure homophily, depending on the context and the specific characteristics being examined. Some common methods include:

1. **Observational Measures**: Researchers may directly observe interactions or relationships within a social network and assess the degree to which individuals with similar characteristics tend to associate with each other. For example, they may examine who individuals choose as friends, colleagues, or romantic partners and assess the similarity of their characteristics.
2. **Survey Data**: Surveys can be used to collect self-reported information about individuals' characteristics and the characteristics of their social contacts. Researchers can then analyze the data to determine the extent to which people are more likely to associate with others who share their characteristics.
3. **Statistical Analysis**: Statistical techniques can be used to quantify the level of homophily within a social network. This may involve measures such as correlation coefficients, which assess the degree of association between individuals based on their characteristics.
4. **Simulation Models**: Computational models can be used to simulate the formation of social networks and test hypotheses about the factors that influence homophily. These models can help researchers understand how various factors, such as individual preferences, network structure, and external influences, contribute to patterns of homophily.
5. **Content Analysis**: In online communities or social media platforms, researchers may analyze the content of interactions (e.g., posts, comments) to identify patterns of similarity among users. This can provide insights into the types of individuals who are more likely to engage with each other online.

45 - Foci Closure and Membership Closure

1. **Foci Closure**: In geometry, particularly in the study of conic sections like ellipses and hyperbolas, the foci closure refers to the set of points that can be reached as limits of sequences of points that are obtained by some process. For example, in an ellipse, the foci are the points that define its shape. The foci closure would encompass not just the foci themselves but also any points that can be approached arbitrarily closely by sequences of points on the ellipse. This concept is crucial in understanding the behavior and properties of conic sections.
2. **Membership Closure**: Membership closure refers to the property of a set where applying a certain operation to elements within the set results in another element that remains within the set. In mathematical structures like groups, rings, or vector spaces, closure under an operation is a fundamental property. For instance, if you have a set of real numbers closed under addition, then adding any two numbers in that set will always give you another real number that is also in the set. Similarly, if you have a group under multiplication, closure means that multiplying any two elements from that group will result in another element within the group.

46 - Introduction to Fatman Evolutionary model

The Fatman model is a model that explains how social media grows and changes over time. Social Media has become an integral part of everyone's life. Researchers are constantly trying to understand the social media growth and behavior associated with it. In this article, we will understand the Fatman model and its significance in social media network Analysis.

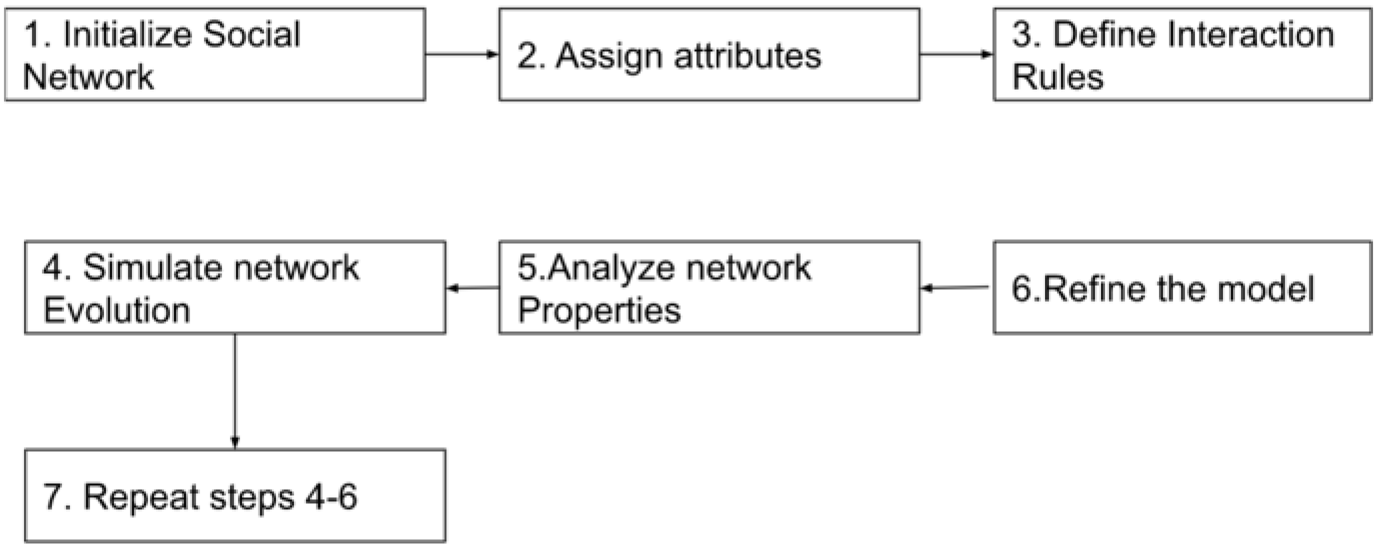
Fatman Model

The Fatman model was first introduced by Alain Barrat, Marc Barthélemy, and Alessandro Vespignani in a paper published in 2004. Fatman's model says that the growth of social networks is influenced by two main factors −

- **Preferential Attachment** −Preferential attachment refers to the tendency of nodes in a network to connect to nodes that already have a large number of connections.
- **Fitness** − Fitness refers to the intrinsic properties of nodes that make them more or less likely to be connected to other nodes in the network.

According to the Fatman model, each node has a fitness value that determines its likeness to receive more or fewer connections. Nodes with higher fitness values receive more connections while the nodes with fewer fitness values receive less number of connections. The fitness value is determined by a set of characteristics that are specific to the nodes such as age, gender, income, education, and so on.

Here is a flowchart to summarize the main steps of the Fatman Evolutionary Model:


1. Initialize Social Network
2. Assign attributes
3. Define Interaction Rules
4. Simulate network Evolution
5.Analyze network Properties
6.Refine the model
7. Repeat steps 4-6

- **Initialize a social network** −Create a network of individuals and connections between them.
- **Assign attributes to individuals** −Each individual in the network is assigned a set of attributes, such as age, gender, occupation, and interests.
- **Define interaction rules** − Define the rules that govern how individuals interact with each other. This can include rules such as how likely two individuals are to form a connection based on their attributes and the strength of their existing connections.
- **Simulate network evolution** −Simulate the evolution of the network over time using the interaction rules defined in step 3. This involves adding new connections, breaking existing connections, and modifying the attributes of individuals.

- **Analyze network properties** – Analyze the resulting network properties, such as the degree distribution, clustering coefficient, and community structure.
- **Refine the model** – Based on the analysis, refine the model by adjusting the interaction rules or adding new attributes to better capture the real-world dynamics of social networks.
- **Repeat steps 4-6** – Iterate through steps 4-6 until the model accurately reflects the properties of real-world social networks.

47 - Fatman Evolutionary Model- The Base Code (Adding people)
To create a basic evolutionary model for simulating the population dynamics of a population of "Fatmen" (assuming you're referring to a fictional population of individuals), you can use a simple Python code. This code will model the population dynamics over multiple generations, including factors like reproduction, mutation, and natural selection.

Here's a basic outline to get you started:

```
import random
import argparse
class Fatman:
    def __init__(self, name, weight, metabolism):
        self.name = name
        self.weight = weight
        self.metabolism = metabolism
    def __repr__(self):
        return f"Fatman(name='{self.name}', weight={self.weight}, metabolism={self.metabolism})"
def initial_population(population_size):
    population = []
    for i in range(population_size):
        name = f"Fatman_{i}"
        weight = random.uniform(80, 200)  # Initial weight between 80 and 200 kg
        metabolism = random.uniform(1, 5)  # Metabolism rate
        population.append(Fatman(name, weight, metabolism))
    return population
def reproduce(parent1, parent2):
    child_weight = (parent1.weight + parent2.weight) / 2  # Average weight
    child_metabolism = (parent1.metabolism + parent2.metabolism) / 2  # Average metabolism
    child_name = f"{parent1.name}_{parent2.name}"
    return Fatman(child_name, child_weight, child_metabolism)
def evolve_population(population, num_generations, mutation_rate):
    for generation in range(num_generations):
        new_generation = []
        for i in range(len(population)):
            parent1 = random.choice(population)
            parent2 = random.choice(population)
            child = reproduce(parent1, parent2)
            if random.random() < mutation_rate:
                child.weight += random.uniform(-10, 10)  # Mutation in weight
                child.metabolism += random.uniform(-0.5, 0.5)  # Mutation in metabolism
            new_generation.append(child)
        population = new_generation
    return population

def main():
    parser = argparse.ArgumentParser(description="Evolutionary simulation of Fatmen population.")
    parser.add_argument("--population_size", type=int, default=100, help="Initial population size.")
    parser.add_argument("--num_generations", type=int, default=10, help="Number of generations.")
    parser.add_argument("--mutation_rate", type=float, default=0.1, help="Mutation rate.")
    args = parser.parse_args()
    population = initial_population(args.population_size)
    final_population = evolve_population(population, args.num_generations, args.mutation_rate)
    # Print final population
    for fatman in final_population:
        print(fatman)
if __name__ == "__main__":
    main()
```

48 - Fatman Evolutionary Model- The Base Code (Adding Social Foci)

1. Define Social Foci:

- Identify key social foci relevant to the model, such as cooperation, competition, altruism, reciprocity, etc.
- Define parameters for each social focus, including strength, frequency of interaction, and impact on individual fitness.

2. Modify Individual Traits:

- Adjust individual traits to include preferences or predispositions towards specific social foci.
- For example, individuals may have varying levels of predisposition towards cooperation or competition.

3. Social Interaction Mechanisms:

- Implement mechanisms for social interactions among individuals based on their traits and the defined social foci.
- Individuals with similar social foci preferences may be more likely to interact positively, while those with conflicting preferences may interact negatively.

4. Fitness Calculation:

- Update the fitness calculation to include the impact of social interactions and adherence to social foci.
- Individuals who effectively engage in social interactions aligned with their preferred foci may have higher fitness.

5. Evolutionary Dynamics:

- Introduce mechanisms for the evolution of social foci preferences over time.
- Implement mutation, selection, and other evolutionary processes to shape the distribution of social foci within the population.

6. Data Collection and Analysis:

- Collect data on the distribution of social foci within the population over multiple generations.
- Analyze the impact of social foci on population dynamics, cooperation, competition, and overall fitness.

Code

```
import numpy as np
class Individual:
  def __init__(self, cooperation_preference, competition_preference):
    self.cooperation_preference = cooperation_preference
    self.competition_preference = competition_preference
    self.fitness = 0
  def interact(self, other):
    # Simulate social interaction
    # Adjust fitness based on interaction and social foci
    if self.cooperation_preference > other.cooperation_preference:
      # Cooperation
      self.fitness += 1
      other.fitness += 1
    elif self.competition_preference > other.competition_preference:
      # Competition
      self.fitness -= 1
      other.fitness -= 1
class Population:
  def __init__(self, size):
    self.size = size
    self.population = [Individual(np.random.rand(), np.random.rand()) for _ in range(size)]
  def evolve(self):
    for i in range(self.size):
      for j in range(i+1, self.size):
        self.population[i].interact(self.population[j])
  def get_average_fitness(self):
    total_fitness = sum([individual.fitness for individual in self.population])
    return total_fitness / self.size
# Example simulation
population_size = 100
generations = 10
population = Population(population_size)
for generation in range(generations):
  population.evolve()
  avg_fitness = population.get_average_fitness()
  print(f"Generation {generation+1}: Average Fitness = {avg_fitness}")
```

In this code:

- **Individual** class represents individuals in the population with traits for cooperation and competition preferences.
- **Population** class represents the population and contains methods for simulating interactions and evolution.
- **interact** method simulates interactions between individuals based on their social foci preferences.
- The **evolve** method drives the evolution process by allowing individuals to interact with each other.
- Fitness is adjusted based on the outcome of interactions and individual preferences for cooperation and competition.
- The simulation runs for a predefined number of generations, and the average fitness of the population is printed at each generation.

49 - Fatman Evolutionary Model- Implementing Homophily

Implementing homophily in an evolutionary model, particularly in the context of the "Fatman" model, can be an intriguing task. The "Fatman" model is a simplified agent-based model used to simulate the evolution of cooperation among selfish individuals. It typically involves a population of agents interacting through repeated Prisoner's Dilemma games, where agents can choose to cooperate or defect.

Homophily, in this context, refers to the tendency of individuals to associate and form connections with others who are similar to them in certain characteristics. In evolutionary models, homophily can be implemented by incorporating preferences or biases in agent interactions based on similarity in traits.

Here's a conceptual outline of how you could implement homophily in the Fatman model:

1. **Define Characteristics**: Start by defining characteristics or traits that agents possess. These traits could be anything relevant to the model, such as strategies, cooperation levels, or even demographic attributes.
2. **Agent Initialization**: Assign initial trait values to agents in the population. You can either randomly assign traits or use some distribution that reflects real-world scenarios.
3. **Interaction Rules**: Modify the interaction rules to incorporate homophily. For instance, when agents select their partners for the Prisoner's Dilemma games, introduce a preference for selecting partners with similar traits. This preference can be quantified using a similarity metric, such as Euclidean distance or cosine similarity, depending on the nature of the traits.
4. **Update Strategy**: Agents can update their strategies not only based on the outcomes of the games but also considering the traits of their interaction partners. For example, if an agent cooperates with another agent who has similar traits and receives a positive outcome, it might reinforce cooperation with similar agents in the future.
5. **Evolutionary Dynamics**: Allow for the evolution of traits over time through mechanisms like mutation, imitation, or selection. Homophily might influence the evolutionary dynamics by promoting the spread of certain traits within the population.
6. **Simulation and Analysis**: Run the simulation over multiple iterations and analyze the effects of homophily on the evolution of cooperation and the distribution of traits in the population. Compare the results with simulations without homophily to understand its impact.

Here's a pseudocode snippet illustrating how you might modify the interaction rules to incorporate homophily:

pythonCopy code

```
for each agent in population:
  # Select interaction partner based on homophily preference
  potential_partners = filter_agents_by_similarity(agent, population)
  partner = select_partner_based_on_preference(agent, potential_partners)
  # Play Prisoner's Dilemma game with the selected partner
  payoff = play_prisoners_dilemma(agent, partner)
  # Update agent's strategy and traits based on the outcome
```

update_strategy_and_traits(agent, partner, payoff) In the above pseudocode, **filter_agents_by_similarity** filters potential interaction partners based on trait similarity, and **select_partner_based_on_preference** selects a partner among the filtered agents according to some preference criterion.

50 - Quantifying the Effect of Triadic Closure

Quantifying the effect of triadic closure in a social network involves understanding how the tendency for individuals to form connections with mutual acquaintances affects network structure and dynamics. Triadic closure refers to the tendency for two individuals who share a mutual connection to form a new connection themselves.

Here's how you can quantify the effect of triadic closure in a social network:

1. **Network Representation**: Represent the social network as a graph, where nodes represent individuals and edges represent connections between them.
2. **Generate Random Network**: Start with an initial network structure, which could be a randomly generated network or a real-world social network dataset.
3. **Simulate Triadic Closure**: Implement a mechanism for simulating triadic closure. This can involve iteratively examining each pair of nodes in the network and checking if they have a mutual connection. If they do, there's a probability or propensity for them to form a new connection.
4. **Measure Network Metrics**: Before and after simulating triadic closure, measure various network metrics to quantify the effect. Some common metrics include:
    - **Clustering Coefficient**: This measures the tendency of nodes to form clusters or tightly knit groups. Triadic closure tends to increase the clustering coefficient as it leads to the formation of triangles in the network.
    - **Average Path Length**: This measures how connected the network is on average. Triadic closure might reduce the average path length by creating shortcuts between distant nodes.
    - **Degree Distribution**: Analyze how the degree distribution of the network changes. Triadic closure can lead to an increase in the number of connections per node, affecting the degree distribution.
5. **Compare with Random Model**: Compare the network metrics observed after simulating triadic closure with those of a randomly generated network with the same number of nodes and edges. This comparison helps to isolate the effect of triadic closure from other network properties.
6. **Statistical Analysis**: Perform statistical analysis to determine the significance of the observed changes in network metrics. This could involve hypothesis testing or calculating confidence intervals.
7. **Parameter Sensitivity Analysis**: Explore how the effect of triadic closure varies with different parameters such as the probability of forming a new connection or the initial network structure.
8. **Visualization**: Visualize the network before and after simulating triadic closure to gain insights into the structural changes induced by the mechanism.

51 - Fatman Evolutionary Model- Implementing Closures

Closure- There are 3 main kinds of closures i.e. triadic closure, membership closure, and foci closure.

- **Triadic closure-** In triadic closure, if A is a friend of B and B is a friend of C then C will eventually become friends of A.
- **Membership closure-** In membership closure if A is a friend of B and A is a member of a club then there is a tendency that B will join the same club.
- **Foci closure-** It is the probability of becoming friends when they have the same foci. In membership closure, if A and B are in the same club then there is a tendency that they will become friends.

```
import tkinter as tk
from tkinter import ttk
import networkx as nx
import random
def initialize_network(num_nodes, num_clubs, num_attributes):
    G = nx.Graph()
    # Add nodes representing individuals
    for i in range(num_nodes):
        G.add_node(i, attributes=set(random.sample(range(num_attributes), random.randint(1, 3))))  # Assign
random attributes to nodes
    # Add edges representing initial connections
    for i in range(num_nodes):
        for j in range(i + 1, num_nodes):
            if random.random() < 0.2:  # Probability of initial connection
                G.add_edge(i, j)
    return G
def triadic_closure(G):
    for A in G.nodes():
        for B in G.nodes():
            for C in G.nodes():
                if A != B and B != C and A != C:
                    if G.has_edge(A, B) and G.has_edge(B, C) and not G.has_edge(A, C):
                        G.add_edge(A, C)
def membership_closure(G, num_clubs):
    clubs = [set(random.sample(range(G.number_of_nodes()), random.randint(2, 5))) for _ in
range(num_clubs)]
    for i, club in enumerate(clubs):
        for A in club:
            for B in club:
                if A != B and not G.has_edge(A, B):
                    G.add_edge(A, B)
def foci_closure(G):
    for A in G.nodes():
        for B in G.nodes():
            if A != B:
                common_attributes = G.node[A]['attributes'].intersection(G.node[B]['attributes'])
                if len(common_attributes) > 0 and not G.has_edge(A, B):
                    G.add_edge(A, B)
def generate_network():
    num_nodes = int(nodes_entry.get())
    num_clubs = int(clubs_entry.get())
    num_attributes = int(attributes_entry.get())
    # Initialize network
    G = initialize_network(num_nodes, num_clubs, num_attributes)
    # Apply closure mechanisms
    triadic_closure(G)
    membership_closure(G, num_clubs)
    foci_closure(G)

    # Update text in result_label
    result_label.config(text=f"Number of nodes: {G.number_of_nodes()}\nNumber of edges: 
{G.number_of_edges()}")
# Create main application window
root = tk.Tk()
root.title("Fatman Evolutionary Model")
# Create input fields and labels
nodes_label = ttk.Label(root, text="Number of nodes:")
nodes_label.grid(row=0, column=0, padx=5, pady=5, sticky="e")
nodes_entry = ttk.Entry(root)
nodes_entry.grid(row=0, column=1, padx=5, pady=5)
clubs_label = ttk.Label(root, text="Number of clubs:")
clubs_label.grid(row=1, column=0, padx=5, pady=5, sticky="e")
clubs_entry = ttk.Entry(root)
clubs_entry.grid(row=1, column=1, padx=5, pady=5)
attributes_label = ttk.Label(root, text="Number of attributes:")
attributes_label.grid(row=2, column=0, padx=5, pady=5, sticky="e")
attributes_entry = ttk.Entry(root)
attributes_entry.grid(row=2, column=1, padx=5, pady=5)
generate_button = ttk.Button(root, text="Generate Network", command=generate_network)
generate_button.grid(row=3, column=0, columnspan=2, padx=5, pady=5)
# Create a label to display the result
result_label = ttk.Label(root, text="")
result_label.grid(row=4, column=0, columnspan=2, padx=5, pady=5)
root.mainloop()
```

52 - Fatman Evolutionary Model- Implementing Social Influence

The Fatman evolutionary model is a theoretical framework used in evolutionary biology to understand the interplay between genetic factors and environmental influences on weight gain and obesity. It proposes that in environments where food is abundant and physical activity is limited, individuals with a genetic predisposition to store fat more efficiently have a reproductive advantage, leading to the propagation of genes associated with obesity within a population.

To implement social influence within the Fatman model, you would consider how social factors affect individuals' behaviors related to eating habits, physical activity, and body image. Here's a basic outline of how you could incorporate social influence into the model:

1. **Social Norms and Cultural Influences**: Integrate cultural and social norms regarding body weight, food consumption, and physical activity into the model. For example, in cultures where thinness is idealized, individuals may be more motivated to control their weight through dieting or exercise.
2. **Peer Influence**: Model how individuals are influenced by their peers in terms of dietary choices, exercise habits, and body image. Peer groups can have a significant impact on behaviors related to food and physical activity, as individuals often conform to group norms to fit in or gain acceptance.
3. **Media Influence**: Consider the influence of media, including advertising, entertainment, and social media, on individuals' perceptions of body image and lifestyle choices. Exposure to images of thin or muscular bodies in the media can contribute to unrealistic body ideals and unhealthy behaviors such as extreme dieting or excessive exercise.
4. **Family Dynamics**: Incorporate the influence of family dynamics, including parental modeling of eating and exercise behaviors, family meals, and socioeconomic factors that affect access to healthy food options and opportunities for physical activity.
5. **Social Support Networks**: Model how social support networks, such as friends, family, and community organizations, can facilitate or hinder individuals' efforts to maintain a healthy weight. Supportive social networks can provide encouragement, accountability, and practical assistance in adopting healthy behaviors.
6. **Stigma and Discrimination**: Consider the impact of weight stigma and discrimination on individuals' self-esteem, mental health, and access to resources for healthy living. Discrimination based on body weight can create barriers to employment, healthcare, and social opportunities, further exacerbating the cycle of obesity.

Code

```
import numpy as np
import matplotlib.pyplot as plt
class Individual:
    def __init__(self, genetic_factor, social_factor):
        self.genetic_factor = genetic_factor  # Genetic predisposition to store fat
        self.social_factor = social_factor    # Social influence on eating habits
        self.weight = 0                       # Initial weight
    def update_weight(self, food_intake):
        self.weight += food_intake * (1 + self.genetic_factor + self.social_factor)
class Population:
    def __init__(self, size, initial_genetic_factor, initial_social_factor):
        self.size = size
        self.individuals = [Individual(initial_genetic_factor, initial_social_factor) for _ in range(size)]
        self.food_intake_mean = 1  # Mean food intake for the population
    def simulate_generation(self):
        for individual in self.individuals:
            individual.update_weight(self.food_intake_mean)
    def get_average_weight(self):
        total_weight = sum(individual.weight for individual in self.individuals)
        return total_weight / self.size
def main():
    population_size = 100
    initial_genetic_factor = 0.1
    initial_social_factor = 0.05
    num_generations = 100
    population = Population(population_size, initial_genetic_factor, initial_social_factor)
    average_weights = []
    for _ in range(num_generations):
        population.simulate_generation()
        average_weight = population.get_average_weight()
        average_weights.append(average_weight)
    plt.plot(average_weights)
    plt.title("Average Weight Evolution")
    plt.xlabel("Generation")
    plt.ylabel("Average Weight")
    plt.show()
if __name__ == "__main__":
    main()
```

53 - Fatman Evolutionary Model- Storing and analyzing longitudnal data

To store and analyze longitudinal data in the context of the Fatman evolutionary model, you can use a variety of approaches and tools depending on the specific needs of your analysis. **Here are some general steps and methods you might consider:**

1. **Data Collection**: The first step in analyzing longitudinal data is to collect data at multiple time points. This might involve conducting surveys, collecting data from social media platforms, or using other methods to gather data on social networks over time.
2. **Data Management**: Once you have collected your data, you will need to manage and organize it in a way that is suitable for analysis. This might involve using a database or data management system to store your data, along with tools for cleaning, transforming, and merging data from different time points.
3. **Data Analysis**: To analyze longitudinal data in the context of the Fatman evolutionary model, you can use a variety of statistical and computational methods. This might include techniques for network analysis, such as social network analysis software like Pajek or Gephi, as well as statistical methods for analyzing change over time, such as regression analysis or time series analysis.
4. **Visualization**: Visualization tools can be particularly useful for analyzing longitudinal data in the context of the Fatman evolutionary model. This might include network visualization tools, as well as tools for visualizing change over time, such as line graphs or heat maps.
5. **Modeling and Simulation**: Finally, you can use the Fatman evolutionary model to simulate the evolution of social networks over time, based on the data you have collected and analyzed. This might involve using computational models and simulations to explore the dynamics of social networks under different conditions, and to make predictions about future changes in social networks.

Some specific tools and approaches you might consider for storing and analyzing longitudinal data in the context of the Fatman evolutionary model include:

- **Relational databases**: Relational databases, such as MySQL or PostgreSQL, can be useful for storing and managing longitudinal data, particularly when working with large datasets.
- **Data analysis software**: Statistical software like R or Python can be used for data analysis and visualization, and can be particularly useful for analyzing longitudinal data using techniques like regression analysis or time series analysis.

- **Network analysis software**: Social network analysis software like Pajek or Gephi can be used to visualize and analyze social networks over time, and to explore the dynamics of social networks using the Fatman evolutionary model.
- **Simulation software**: Computational modeling and simulation software, such as NetLogo or MATLAB, can be used to simulate the evolution of social networks over time, based on the data you have collected and analyzed.

python

```python
import networkx as nx
import pandas as pd
import matplotlib.pyplot as plt

# Load longitudinal data from a CSV file
data = pd.read_csv('longitudinal_data.csv')

# Create a list of networks, one for each time point
networks = []
for i in range(len(data)):
    # Extract the current time point's data
    time_data = data.iloc[0:i+1]

    # Create a new network based on the current time point's data
    G = nx.from_pandas_edgelist(time_data, 'source', 'target', create_using=nx.Graph())

    # Add the network to the list
    networks.append(G)

# Analyze the evolution of the network over time
for i in range(len(networks)):
    # Print the number of nodes and edges at the current time point
    print(f'Time point {i+1}: {networks[i].number_of_nodes()} nodes, {networks[i].number_of_edges()} edges')

    # Visualize the network at the current time point
    nx.draw(networks[i], with_labels=True)
    plt.show()

    # Add the current time point's network to the previous time point's network
    # (assuming the networks are connected by a common set of nodes)
    if i > 0:
        networks[i] = nx.compose(networks[i-1], networks[i])

    # Calculate various network metrics at the current time point
    # (e.g., degree distribution, clustering coefficient, centrality measures)
    # network_metrics = nx.degree_histogram(networks[i])
    # print(network_metrics)
```

This code assumes that you have a CSV file called **longitudinal_data.csv** that contains longitudinal data on a social network, with each row representing a tie between two nodes at a particular time point. The code creates a list of networks, one for each time point, and then analyzes the evolution of the network over time using various network metrics and visualizations.

Week 5: Homophily Continued and +Ve/-Ve Relationships in Social Network

54-: Spatial Segregation: An Introduction

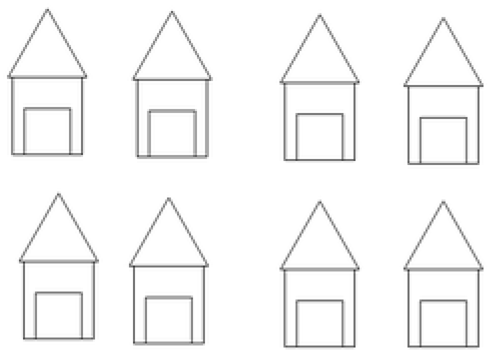
Spatial Segregation in Social Networks

-
-
-

Prerequisite – [Introduction to Social Networks](#)

Segregation refers to the act of separation of people from others or the main group.

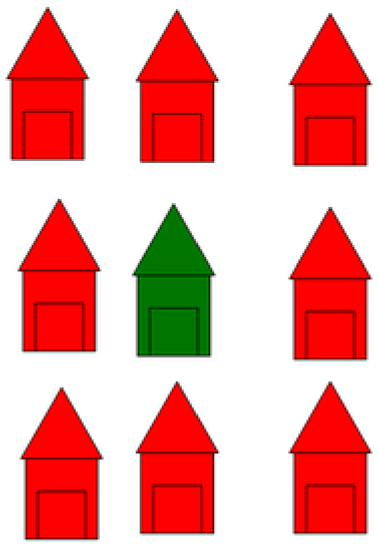
Spatial Segregation refers to the distribution of social groups or any other elements in space. In Spatial Segregation, people tend to migrate to other places where they have more neighbors who are like them.



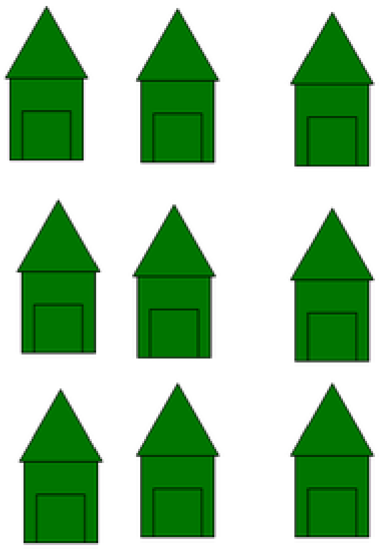
Group of Houses in a locality

According to spatial segregation, it is unlikely that a person can stay at a place where the neighbors are very different from you and you cannot have a good conversation with your neighbor. A person will more likely to migrate to a place where his/her neighbors are similar to them and you can have a good conversation with your neighbor.

Let's take an example – Assume you went to a foreign country for living and you are deciding where to take the house. So you will more likely to choose a place or locality where your neighbors are from your country and speaks the same language as yours and not a place where people are from a different country and speaks a different language. This is spatial segregation.



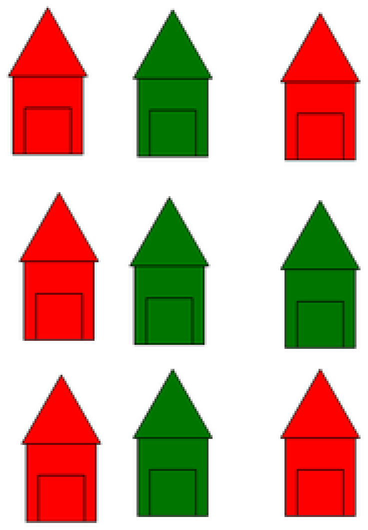
Locality with all different neighbors from you



Locality with all same neighbors as you

But sometimes it is not possible to have all neighbors as same as you. So for that, we can have some threshold value for similar neighbors to stay there.

Let's take an example – Assume there is a total of 9 houses in your locality including yours and let's say there is a threshold of 3 which means that at least 3 houses should be similar to you.



The person will stay here as t =3

55- Spatial Segregation: Simulation of the Schelling Model

The Schelling model is a classic agent-based model used to simulate spatial segregation. It was developed by economist Thomas Schelling to illustrate how individual preferences for similar neighbors can lead to overall segregation in a population. Here's a basic outline of how you can simulate the Schelling model:

1. **Setup**: Create a grid representing the space where agents (individuals) live. Each cell in the grid can be occupied by either an agent or left empty.
2. **Initialization**: Populate the grid with agents of different types (e.g., two types representing different groups or preferences). Initially, agents are randomly distributed across the grid.
3. **Iteration**: Repeat the following steps until a stopping condition is met:
    - For each agent, calculate its dissatisfaction level based on the proportion of neighbors that are different from itself. This dissatisfaction level is typically represented by a threshold value.
    - If an agent's dissatisfaction level exceeds its threshold, it moves to a randomly chosen empty cell. Otherwise, it remains in its current location.
    - Repeat this process for all agents in the grid.
4. **Visualization**: After each iteration or a certain number of iterations, visualize the grid to observe the spatial distribution of agents and any emerging patterns of segregation.
5. **Stopping Condition**: Define a stopping condition, such as a maximum number of iterations or reaching a state where no agents are moving due to satisfaction.

Here's a simple Python code example using the **numpy** library to implement the Schelling model:

```
import numpy as np

import matplotlib.pyplot as plt

class SchellingModel:
    def __init__(self, width, height, empty_ratio, similarity_threshold):
        self.width = width
        self.height = height
        self.empty_ratio = empty_ratio
        self.similarity_threshold = similarity_threshold
        self.grid = np.zeros((width, height))  # Initialize grid
        self.populate_grid()
    def populate_grid(self):
        # Populate grid with agents and empty cells
        for x in range(self.width):
            for y in range(self.height):
                if np.random.random() < self.empty_ratio:
                    self.grid[x, y] = 0  # Empty cell
                else:
                    self.grid[x, y] = np.random.choice([-1, 1])  # Agent of type -1 or 1
    def calculate_similarity(self, x, y):
        agent_type = self.grid[x, y]
        similar_neighbors = 0
        total_neighbors = 0
        for dx in [-1, 0, 1]:
            for dy in [-1, 0, 1]:
                if 0 <= x + dx < self.width and 0 <= y + dy < self.height:
                    total_neighbors += 1
                    if self.grid[x + dx, y + dy] == agent_type:
                        similar_neighbors += 1
        return similar_neighbors / total_neighbors
    def move_agents(self):
        moved = False
        for x in range(self.width):
            for y in range(self.height):
                if self.grid[x, y] != 0:  # Skip empty cells
                    similarity = self.calculate_similarity(x, y)
                    if similarity < self.similarity_threshold:
                        empty_cells = np.argwhere(self.grid == 0)
                        if len(empty_cells) > 0:
                            random_empty_cell = empty_cells[np.random.randint(len(empty_cells))]
                            self.grid[random_empty_cell[0], random_empty_cell[1]] = self.grid[x, y]
                            self.grid[x, y] = 0
                            moved = True
        return moved
    def simulate(self, max_iterations=100):
        for i in range(max_iterations):
            moved = self.move_agents()
            if not moved:
                print(f"Convergence reached after {i} iterations.")
                break
    def visualize(self):
        plt.imshow(self.grid, cmap='bwr', vmin=-1, vmax=1)
        plt.colorbar()
        plt.title("Schelling Model Simulation")
        plt.show()
# Example usage
width, height = 50, 50
empty_ratio = 0.1
similarity_threshold = 0.5
model = SchellingModel(width, height, empty_ratio, similarity_threshold)
model.simulate()
model.visualize()
```

56- Spatial Segregation: Conclusion

In conclusion, the spatial segregation observed in social networks underscores the persistence of social divisions and inequalities within society. While individuals may have the freedom to associate with others of their choosing, the resulting clustering can reinforce existing disparities and limit opportunities for interaction and understanding across diverse groups.

57- Schelling Model Implementation-1(Introduction)

The Schelling model is a computational model used to explore the emergence of segregation patterns in populations. It was introduced by economist Thomas Schelling in 1971 as a way to understand how individual preferences for similarity in neighbors can lead to overall patterns of segregation, even when individuals themselves might not be strongly biased towards segregation.

The model operates on a grid-based environment where agents (representing individuals or households) are placed. Each agent has a preference for the composition of its neighborhood – it wants to be surrounded by a certain proportion of similar agents. If an agent finds itself in a neighborhood where this preference is not met, it will relocate to a new, randomly chosen location.

Implementing the Schelling model provides a hands-on way to explore the dynamics of segregation and understand how simple rules at the individual level can give rise to complex patterns at the macroscopic level. In the following sections, we will delve into the details of how to implement and simulate the Schelling model using Python.

58- Schelling Model Implementation-2 (Base Code)

```
import numpy as np
import matplotlib.pyplot as plt
class SchellingModel:
    def __init__(self, width, height, empty_ratio, similarity_threshold):
        self.width = width
        self.height = height
        self.empty_ratio = empty_ratio
        self.similarity_threshold = similarity_threshold
        self.grid = np.zeros((width, height))  # Initialize grid
        self.populate_grid()
    def populate_grid(self):
        # Populate grid with agents and empty cells
        for x in range(self.width):
            for y in range(self.height):
                if np.random.random() < self.empty_ratio:
                    self.grid[x, y] = 0  # Empty cell
                else:
                    self.grid[x, y] = np.random.choice([-1, 1])  # Agent of type -1 or 1
    def calculate_similarity(self, x, y):
        agent_type = self.grid[x, y]
        similar_neighbors = 0
        total_neighbors = 0
        for dx in [-1, 0, 1]:
            for dy in [-1, 0, 1]:
                if 0 <= x + dx < self.width and 0 <= y + dy < self.height:
                    total_neighbors += 1
                    if self.grid[x + dx, y + dy] == agent_type:
                        similar_neighbors += 1
        return similar_neighbors / total_neighbors
    def move_agents(self):
        moved = False
        for x in range(self.width):
            for y in range(self.height):
                if self.grid[x, y] != 0:  # Skip empty cells
                    similarity = self.calculate_similarity(x, y)
                    if similarity < self.similarity_threshold:
                        empty_cells = np.argwhere(self.grid == 0)
                        if len(empty_cells) > 0:
                            random_empty_cell = empty_cells[np.random.randint(len(empty_cells))]
                            self.grid[random_empty_cell[0], random_empty_cell[1]] = self.grid[x, y]
                            self.grid[x, y] = 0
                            moved = True
        return moved
    def simulate(self, max_iterations=100):
        for i in range(max_iterations):
            moved = self.move_agents()
            if not moved:
                print(f"Convergence reached after {i} iterations.")
                break
    def visualize(self):
        plt.imshow(self.grid, cmap='bwr', vmin=-1, vmax=1)
        plt.colorbar()
        plt.title("Schelling Model Simulation")
        plt.show()
# Example usage
width, height = 50, 50
empty_ratio = 0.1
similarity_threshold = 0.5
model = SchellingModel(width, height, empty_ratio, similarity_threshold)
model.simulate()
model.visualize()
```

59- Schelling Model Implementation- Visualization and Getting a list of boundary and internal nodes

1. **Definition of Boundary and Internal Nodes:**
    - **Boundary Nodes:** These are nodes that lie on the boundary of the computational domain or the object being analyzed. They typically have specific boundary conditions applied to them, such as fixed displacements, prescribed loads, or thermal conditions.
    - **Internal Nodes:** These are nodes that lie within the computational domain and are not part of the boundary. They are often interconnected with other nodes forming elements in the mesh or graph structure.
2. **Identifying Boundary Nodes:**
    - Boundary nodes can be identified based on their connectivity or spatial location.
    - If you have access to information about boundary conditions, nodes associated with these conditions are typically boundary nodes.
    - In mesh-based simulations, nodes belonging to elements with faces or edges lying on the boundary are considered boundary nodes.
3. **Identifying Internal Nodes:**
    - Internal nodes are typically those that are not identified as boundary nodes.
    - In structured grids or meshes, internal nodes can be easily identified as those lying within the boundary nodes.
    - For unstructured meshes or graphs, internal nodes are nodes that are not explicitly defined as boundary nodes.
4. **Obtaining a List of Boundary and Internal Nodes:**
    - Once you have identified the boundary nodes, you can create a list containing their indices or coordinates.
    - Similarly, for internal nodes, you can create another list containing their indices or coordinates.
    - These lists can be generated programmatically using scripting or programming languages commonly used in computational simulations like Python, MATLAB, or C++.
5. **Utilization in Visualization:**
    - Lists of boundary and internal nodes can be utilized in visualization software to highlight or manipulate specific regions of interest.
    - Visualization techniques such as color mapping, contouring, or vector plotting can be applied differently to boundary and internal nodes to better understand the physical phenomena being simulated.
6. **Considerations:**

- Ensure consistency in node numbering or labeling across different components of your simulation to avoid errors in node identification.
- Depending on the complexity of your simulation, you may need to handle special cases such as corner nodes or nodes on curved boundaries differently.

60- Schelling Model Implementation - Getting a list of unsatisfied nodes

The Schelling model is a simple agent-based model used to understand segregation in societies. In the context of this model, "unsatisfied" nodes are those where agents are not happy with their current state or neighborhood and wish to move.

To implement the Schelling model and get a list of unsatisfied nodes, you can follow these steps:

- Initialize the grid: Create a grid representing the space where agents live. Each cell in the grid can be empty or occupied by an agent of a certain type (e.g., red or blue).
- Define agent behavior: Define rules for agent satisfaction based on the proportion of neighbors of the same type. For example, an agent might be satisfied if a certain percentage of its neighbors are of the same type.
- Iterate over the grid: For each cell in the grid, check if the agent occupying that cell is satisfied or not based on the defined rules.
- Identify unsatisfied nodes: Keep track of the unsatisfied nodes (cells) and store their coordinates in a list.

Here's a Python-like pseudocode to illustrate the implementation:

```
def is_satisfied(grid, x, y, threshold):
    # Get the type of the agent at position (x, y)
    agent_type = grid[x][y]
    # Count the number of neighbors of the same type
    same_type_neighbors = 0
    total_neighbors = 0
    for dx in [-1, 0, 1]:
        for dy in [-1, 0, 1]:
            if (dx != 0 or dy != 0) and 0 <= x + dx < len(grid) and 0 <= y + dy < len(grid[0]):
                total_neighbors += 1
                if grid[x + dx][y + dy] == agent_type:
                    same_type_neighbors += 1
    # Calculate the satisfaction ratio
    satisfaction_ratio = same_type_neighbors / total_neighbors
    # Check if the agent is satisfied
    return satisfaction_ratio >= threshold
def get_unsatisfied_nodes(grid, threshold):
    unsatisfied_nodes = []
    # Iterate over the grid
    for x in range(len(grid)):
        for y in range(len(grid[0])):
            # Check if the agent at (x, y) is unsatisfied
            if not is_satisfied(grid, x, y, threshold):
                unsatisfied_nodes.append((x, y))
    return unsatisfied_nodes
# Example usage
grid = [
    ['red', 'blue', 'red'],
    ['blue', 'red', 'blue'],
    ['red', 'blue', 'red']
]

threshold = 0.5  # Satisfaction threshold (50%)
unsatisfied_nodes = get_unsatisfied_nodes(grid, threshold)
print("Unsatisfied nodes:", unsatisfied_nodes)
```

This pseudocode defines a function get_unsatisfied_nodes() that takes a grid representing the society and a satisfaction threshold as input and returns a list of coordinates of unsatisfied nodes based on the defined rules. You can adjust the threshold and other parameters based on your specific requirements and assumptions.

61- Schelling Model Implementation - Shifting the unsatisfied nodes and visualizing the final

Graph

To implement the Schelling model with the shifting of unsatisfied nodes and visualize the final graph, you can follow these steps:

1. **Initialize the grid**: Create a grid representing the space where agents live. Each cell in the grid can be empty or occupied by an agent of a certain type (e.g., red or blue).
2. **Define agent behavior**: Define rules for agent satisfaction based on the proportion of neighbors of the same type. For example, an agent might be satisfied if a certain percentage of its neighbors are of the same type.
3. **Identify unsatisfied nodes**: Determine which nodes (agents) are unsatisfied based on the defined rules.
4. **Shift unsatisfied nodes**: Move unsatisfied nodes to a new location in the grid where they are satisfied. You can choose a random empty cell for each unsatisfied node to move to.
5. **Visualize the final graph**: Use a plotting library to visualize the final state of the grid after the shifting of unsatisfied nodes.

Here's a Python-like pseudocode to illustrate the implementation:

```
import numpy as np
import matplotlib.pyplot as plt
def is_satisfied(grid, x, y, threshold):
    # Get the type of the agent at position (x, y)
    agent_type = grid[x][y]
    # Count the number of neighbors of the same type
    same_type_neighbors = 0
    total_neighbors = 0
    for dx in [-1, 0, 1]:
        for dy in [-1, 0, 1]:
            if (dx != 0 or dy != 0) and 0 <= x + dx < len(grid) and 0 <= y + dy < len(grid[0]):
                total_neighbors += 1
                if grid[x + dx][y + dy] == agent_type:
                    same_type_neighbors += 1

    # Calculate the satisfaction ratio
    satisfaction_ratio = same_type_neighbors / total_neighbors
    # Check if the agent is satisfied
    return satisfaction_ratio >= threshold
def get_unsatisfied_nodes(grid, threshold):
    unsatisfied_nodes = []
    # Iterate over the grid
    for x in range(len(grid)):
        for y in range(len(grid[0])):
            # Check if the agent at (x, y) is unsatisfied
            if not is_satisfied(grid, x, y, threshold):
                unsatisfied_nodes.append((x, y))
    return unsatisfied_nodes
def move_unsatisfied_nodes(grid, unsatisfied_nodes):
    empty_cells = [(x, y) for x in range(len(grid)) for y in range(len(grid[0])) if grid[x][y] == '']
    for x, y in unsatisfied_nodes:
        # Choose a random empty cell
        random_empty_cell = empty_cells[np.random.randint(len(empty_cells))]
        # Move the unsatisfied node to the empty cell
        grid[random_empty_cell[0]][random_empty_cell[1]] = grid[x][y]
        grid[x][y] = ''
        # Remove the chosen empty cell from the list of available empty cells
        empty_cells.remove(random_empty_cell)
def visualize_grid(grid):
    colors = {'red': 'red', 'blue': 'blue', '': 'white'}
    grid_array = np.array([[colors[cell] for cell in row] for row in grid])
    plt.imshow(grid_array, interpolation='nearest')
    plt.axis('off')
    plt.show()
# Example usage
grid_size = 10
grid = [['red' if np.random.random() < 0.5 else 'blue' for _ in range(grid_size)] for _ in range(grid_size)]
threshold = 0.5  # Satisfaction threshold (50%)
iterations = 10
for _ in range(iterations):
    unsatisfied_nodes = get_unsatisfied_nodes(grid, threshold)
    move_unsatisfied_nodes(grid, unsatisfied_nodes)
visualize_grid(grid)
```

This pseudocode defines functions to identify unsatisfied nodes, move them to a new location, and visualize the final state of the grid. You can adjust parameters such as grid size, satisfaction threshold, and the number of iterations as needed.

62- Positive and Negative Relationships – Introduction

Introduction to Positive and Negative Relationships in Social Networks:

In the vast landscape of social networks, relationships play a pivotal role in shaping interactions, behaviors, and overall experiences. These relationships can vary significantly, spanning from positive connections that foster support, collaboration, and mutual growth, to negative ones characterized by conflict, tension, and toxicity. Understanding the dynamics of both positive and negative relationships within social networks is essential for individuals, communities, and platforms alike. Let's delve deeper into each type:

Positive Relationships: Positive relationships in social networks are marked by trust, empathy, and mutual respect. They serve as the foundation for constructive interactions and meaningful connections. Here are some key characteristics:

1. Supportive: Positive relationships entail offering encouragement, assistance, and emotional support to one another. This support can range from simple gestures such as likes and comments to more substantial forms like sharing resources or providing guidance.

2. Collaboration: Individuals in positive relationships often collaborate on various projects, discussions, or initiatives. They leverage each other's strengths, skills, and perspectives to achieve common goals and enhance collective outcomes.

3. Trust and Transparency: Trust is a fundamental element of positive relationships. Participants feel safe and comfortable sharing ideas, opinions, and personal experiences openly. Transparency fosters authenticity and deepens connections.
4. Mutual Growth: Positive relationships contribute to personal and professional development. Through constructive feedback, constructive criticism, and shared learning experiences, individuals in these relationships grow together, enriching their knowledge and skills.
5. Celebration of Diversity: Positive relationships embrace diversity and celebrate differences in perspectives, backgrounds, and experiences. They cultivate an inclusive environment where everyone feels valued and accepted.

Negative Relationships: Negative relationships in social networks can hinder communication, sow discord, and create a hostile atmosphere. Recognizing and addressing these dynamics is crucial for maintaining a healthy online community. Here are some common attributes:

1. Conflict and Hostility: Negative relationships are often characterized by conflict, disagreements, and hostility. Participants may engage in arguments, trolling, or harassment, leading to a toxic environment.
2. Lack of Trust: Trust is eroded in negative relationships due to dishonesty, betrayal, or manipulative behavior. Participants may feel guarded and apprehensive about sharing information or engaging with others authentically.
3. Miscommunication: Negative relationships are prone to miscommunication, misunderstandings, and misinterpretations. Poor communication channels can exacerbate conflicts and escalate tensions.
4. Draining Energy: Interactions in negative relationships can be emotionally draining and exhausting. Participants may feel drained of energy due to constant negativity, criticism, or drama.
5. Disruption of Community: Negative relationships can disrupt the harmony and cohesion of online communities. They create divisions, cliques, or factions, undermining collective efforts and shared objectives.

63- Structural Balance

Structural balance theory, initially proposed by Fritz Heider and later developed by social psychologists such as Leon Festinger, Harold Kelley, and Theodore Newcomb, offers insights into the dynamics of social relationships within a network. It primarily focuses on the balance or imbalance of sentiments among triads of individuals.

Here are the key elements and details of structural balance theory:

1. **Triadic Relationships**: Structural balance theory revolves around the analysis of relationships among three individuals or entities, known as triads. Each triad consists of three nodes connected by positive or negative sentiment links.
2. **Positive and Negative Ties**: In social networks, ties or links between individuals can be positive (indicating friendship, cooperation, liking, etc.) or negative (indicating rivalry, animosity, dislike, etc.). These ties are represented by directed edges connecting the nodes in a triad.

3. **Balance Theory**: The theory postulates that balanced triads have a certain configuration of ties that contribute to stability and harmony within the network. According to Heider's P-O-X model (Positive - Negative - X), there are two balanced configurations:

a. **Transitive Balance (P-P-P)**: In this configuration, all three ties in the triad are positive. For example, A likes B, B likes C, and C likes A.

b. **Cyclic Balance (P-N-P or N-P-N)**: In this configuration, two ties are positive, and one is negative, forming a balanced cycle. For example, A likes B, B dislikes C, and C likes A.

4. **Imbalance and Tension**: Conversely, triads that do not adhere to these balanced configurations are considered imbalanced. Imbalance results in tension within the network, potentially leading to conflict, stress, or the dissolution of relationships.
5. **Structural Balance and Social Stability**: Structural balance theory suggests that networks tend to evolve towards balanced configurations over time as individuals adjust their relationships to alleviate tension and maintain social stability. This process may involve the formation or dissolution of ties to achieve balance.
6. **Applications**: Structural balance theory has found applications in various fields, including social psychology, sociology, anthropology, and computer science. It has been used to understand interpersonal relationships, group dynamics, opinion formation, information diffusion, and network analysis.

64- Enemy's Enemy is a Friend
The concept of "enemy's enemy is a friend" in social networks reflects a phenomenon where individuals or groups form alliances or positive connections based on a shared adversary or mutual opposition. This idea has historical roots but is also relevant in contemporary social dynamics, including online interactions. Here's how it manifests in social networks:

1. **Alliance Formation**: In social networks, individuals or groups may align with others who have a common enemy or adversary. This alliance formation is driven by the belief that cooperating with someone who shares a common opponent can increase one's chances of success or influence.
2. **Shared Goals**: Despite potential differences or conflicts between allies, the shared goal of opposing a common adversary can foster collaboration and cooperation. This shared purpose overrides other differences and motivates individuals to work together.
3. **Strength in Numbers**: The principle of "enemy's enemy is a friend" recognizes the strategic advantage of consolidating forces against a common threat. By banding together, individuals or groups can leverage collective resources, expertise, and influence to confront their mutual enemy effectively.
4. **Temporary Alliances**: In some cases, alliances formed based on shared adversaries may be temporary and opportunistic. Once the common threat is neutralized or the shared goal is achieved, the alliance may dissolve, and former allies may resume their previous relationships or pursue divergent interests.
5. **Online Communities**: In the context of social media and online communities, the concept of "enemy's enemy is a friend" can manifest through the formation of virtual alliances, coalitions, or support networks. Users may come together to combat harassment, misinformation, or other negative aspects of online behavior.

6. **Political and Ideological Movements**: The principle of forming alliances based on shared adversaries is particularly evident in political and ideological movements. Groups with differing beliefs or agendas may unite against a common political opponent, reflecting the pragmatic nature of coalition-building in pursuit of shared objectives.
7. **Ethical Considerations**: While forming alliances based on shared adversaries can be strategic, it also raises ethical considerations. Individuals and groups must assess the alignment of values, goals, and interests beyond the immediate context of opposition to ensure the sustainability and integrity of their alliances.

65- Characterizing the structure of balanced networks

Characterizing the structure of balanced networks involves understanding the patterns and configurations of relationships within the network that adhere to principles of structural balance theory. Balanced networks are those in which triadic relationships tend to be stable, leading to a state of harmony or equilibrium. Here are some key characteristics of balanced networks:

1. **Transitive and Cyclic Balance**: Balanced networks exhibit configurations of triads that adhere to transitive balance (P-P-P) and cyclic balance (P-N-P or N-P-N), as defined by Heider's P-O-X model. In transitive balance, all three ties within a triad are positive, while in cyclic balance, two ties are positive, and one is negative. These configurations contribute to stability within the network.
2. **Low Tension**: Balanced networks generally have lower levels of tension or conflict compared to unbalanced networks. Triads that violate principles of structural balance theory, such as having two negative ties and one positive tie, are less common in balanced networks, reducing the likelihood of tension and discord.
3. **Consistency in Sentiments**: In balanced networks, there is often a consistency in the sentiments expressed within triads. Positive relationships tend to be reinforced by other positive relationships within the network, creating a cohesive and harmonious social environment.
4. **Dense Clustering**: Balanced networks often exhibit dense clustering or interconnectedness among nodes. This clustering arises from the tendency for individuals or groups to form positive relationships with others who share mutual acquaintances or allies, reinforcing the overall balance of the network.
5. **Robustness to Perturbations**: Balanced networks are often more robust to perturbations or disruptions compared to unbalanced networks. The presence of balanced triads provides a stabilizing influence, buffering against the spread of tension or conflict and facilitating the maintenance of social cohesion.
6. **Community Structure**: Balanced networks may exhibit clear community structures or clusters of nodes with dense internal connections and sparser connections between clusters. These communities may reflect shared interests, affiliations, or values, contributing to the overall balance and stability of the network.
7. **Evolutionary Dynamics**: The structure of balanced networks may evolve over time as individuals form, dissolve, or reconfigure relationships in response to changing social dynamics or external influences. However, the principles of structural balance theory provide a framework for understanding the underlying mechanisms driving these evolutionary dynamics.

66- Balance Theorem

The Balance Theorem is a fundamental concept in structural balance theory, a theory in social psychology that examines the patterns of relationships within social networks. Proposed by Fritz Heider in the 1940s and further developed by other researchers, the Balance Theorem provides insights into how individuals perceive and maintain harmony or equilibrium in their social interactions. Here's an overview of the Balance Theorem:

1. **Triadic Relationships**: The Balance Theorem focuses on the analysis of triadic relationships, which involve three individuals or entities interconnected by positive or negative ties.
2. **Positive and Negative Ties**: In social networks, individuals form relationships characterized by either positive sentiments (such as friendship, cooperation, or liking) or negative sentiments (such as rivalry, hostility, or dislike). These sentiments are represented by directed edges connecting nodes in the network.
3. **Balance Theory**: The Balance Theorem posits that individuals strive for balance or consistency in their perceptions of triadic relationships within their social networks. Specifically, it outlines two configurations of balanced triads:
    - **Transitive Balance (P-P-P)**: In transitive balance, all three ties within the triad are positive. For example, if A likes B, B likes C, and C likes A, the triad is balanced.
    - **Cyclic Balance (P-N-P or N-P-N)**: In cyclic balance, two ties are positive, and one tie is negative, forming a balanced cycle. For example, if A likes B, B dislikes C, and C likes A, the triad is balanced.
4. **Imbalance and Tension**: Triads that do not conform to these balanced configurations are considered imbalanced. According to the Balance Theorem, imbalanced triads create tension or cognitive dissonance for individuals within the network, motivating them to restore balance through various mechanisms, such as forming or dissolving relationships.
5. **Social Equilibrium**: The Balance Theorem suggests that individuals and social networks tend to evolve toward states of equilibrium characterized by a predominance of balanced triadic relationships. This equilibrium reflects a state of reduced tension and increased stability within the network.
6. **Applications**: The Balance Theorem has applications in various domains, including social psychology, sociology, anthropology, and computer science. It provides insights into the dynamics of interpersonal relationships, group cohesion, opinion formation, and network analysis.

```
class BalanceTheorem:
  def __init__(self, network):
    self.network = network
  def is_transitively_balanced(self, triad):
    # Check if all edges in the triad are positive
    return all(self.network[edge[0]][edge[1]] == 'positive' for edge in triad)
  def is_cyclically_balanced(self, triad):
    # Count the number of positive and negative edges
    positive_count = sum(1 for edge in triad if self.network[edge[0]][edge[1]] == 'positive')
    negative_count = sum(1 for edge in triad if self.network[edge[0]][edge[1]] == 'negative')
    # Check if there are exactly two positive edges and one negative edge
```

```
        return positive_count == 2 and negative_count == 1
    def analyze_triads(self):
        balanced_triads = []
        for triad in self.network:
            if self.is_transitively_balanced(triad) or self.is_cyclically_balanced(triad):
                balanced_triads.append(triad)
        return balanced_triads
# Example usage:
if __name__ == "__main__":
    # Example social network represented as a list of triads
    social_network = [
        [('A', 'B'), ('B', 'C'), ('C', 'A')],  # Balanced transitive triad
        [('A', 'B'), ('B', 'C'), ('C', 'A')],  # Balanced cyclic triad
        [('A', 'B'), ('B', 'C'), ('C', 'A')],  # Unbalanced triad
    ]
    # Initialize BalanceTheorem with the social network
    bt = BalanceTheorem(social_network)
    # Analyze triads in the network and print the balanced ones
    balanced_triads = bt.analyze_triads()
    print("Balanced Triads:")
    for triad in balanced_triads:
        print(triad)
```

In this example:

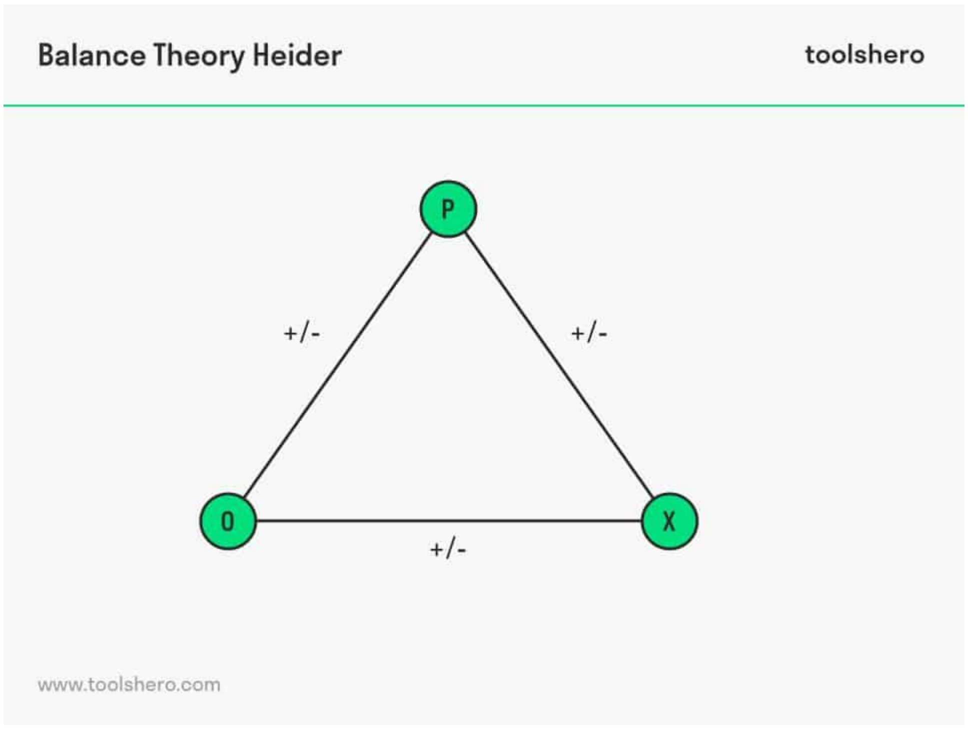
- The **BalanceTheorem** class represents a simplified implementation of the Balance Theorem.
- It defines methods to check if a triad is transitively balanced or cyclically balanced based on the edges' types (positive or negative).
- The **analyze_triads** method scans through all triads in the network and identifies the balanced ones.
- Finally, an example usage scenario is provided, where a social network is represented as a list of triads, and the balanced triads are printed.

67- Proof of Balance Theorem

Balance (+) Imbalance (-) according to Heider

Social perception is important in order to be able to link balance, harmony, the positive and the negative. This social perception according to Heider plays an important role in determining the balance of unity relationships according to that perception. Relationships can be positive (like- approve) and negative (dislike- disapprove). These relationships are about belonging, positive feelings (close, belonging, similar) or negative feelings (not belonging, not similar, not close).

With the above, Heider developed the P-O-X Model to identify the positive and negative and to form the balance that is needed for harmony between different situations that human beings present and how to live adequately with the differences and conflicts that many times are not shared with other individuals (personal and professional environment) and to be able to reach a positive result at the end of the confrontation and experience.



<u>Model P-O-X</u>

P is a person in whom balance or imbalance occurs, O is perceived a person who is in the environment or situation of P, and X is an impersonal entity or other person or object that participates in the unit. Among these three parts, two types of relationships can be found: attitudes of taste or evaluation relationship and the second one of similarity, participation, proximity among others.

- The relationship between P and O can be positive if P likes O or negative if P dislikes O this is presented as PLO the L as positive representation and in case of dislike or negativity P-LO.
- The relationship between P and X, P feels attraction or taste for X or if P feels rejection or displeasure for X. In the case that X is no longer an impersonal entity it can participate in the process of the POX triad.

Psychological balance: P+OP (+) > O P< (+) O

It can also extend to things or objects, triadic relationships. If a person likes an object, but does not like another person It is symbolized as: XPXPPOX

- P (+)> X
- P (-)> O
- P (+)> X

Cognitive balance is achieved when there are three positive or negative links with two positive ones. Two positive links and one negative link as in the previous example, thus creating the imbalance.

Multiplying the signs shows that the person will perceive imbalance (a negative multiplicative product) in this relationship, and will be motivated to correct the imbalance in some way. The person can:

- Decide it’s not so bad after all,
- They decide it’s not as big as originally thought, or
- The conclusion I couldn’t have made.

Either of these will translate into psychological balance, thus resolving the dilemma and the satisfaction of unity. (The person could also avoid the object and another person altogether, which reduces the tension created by the psychological imbalance).

To predict the outcome of a situation using Heider's Balance Theory / Heider's theory of equilibrium, one must weigh the effects of all possible outcomes, and the one that requires the least amount of effort will be the likely outcome.

Determining whether the triad is balanced in a mathematical way is like this:

- +++= + Balanced
- -+-= + Balancing
- -++= – Unbalanced

68- Introduction to positive and negative edges

Introduction to Positive and Negative Edges in Graph Theory:

Graph theory is a mathematical framework used to model and analyze relationships between objects, represented as nodes (vertices) and connections between them, known as edges. In the context of social networks, positive and negative edges play a significant role in capturing the nature of relationships between individuals. Let's explore these concepts:

1. **Positive Edges**:
    - Positive edges represent relationships characterized by cooperation, friendship, support, or affinity between nodes.
    - In social networks, positive edges denote connections where individuals share mutual interests, engage in collaborative activities, or exhibit positive sentiments towards each other.
    - Examples of positive edges in social networks include friendships on social media platforms, professional collaborations, familial relationships, or supportive interactions within communities.
2. **Negative Edges**:
    - Negative edges represent relationships characterized by conflict, rivalry, animosity, or dislike between nodes.
    - In social networks, negative edges denote connections where individuals have disagreements, exhibit hostile behaviors, or hold negative sentiments towards each other.
    - Examples of negative edges in social networks include conflicts, rivalries, arguments, or instances of harassment and trolling.
3. **Significance in Social Networks**:
    - Positive edges contribute to the formation of cohesive communities, fostering trust, collaboration, and social support among individuals.
    - Negative edges highlight areas of tension, disagreement, or conflict within social networks, impacting the overall dynamics and atmosphere.

- Understanding the distribution and patterns of positive and negative edges in social networks provides insights into community structure, information diffusion, opinion polarization, and the spread of emotions and behaviors.

4. **Analysis and Applications**:
   - Researchers analyze positive and negative edges in social networks to study phenomena such as network formation, influence dynamics, community detection, and sentiment analysis.
   - Applications of positive and negative edges in social network analysis include identifying influential individuals, detecting communities with shared interests or affiliations, predicting user behavior, and designing interventions to mitigate conflicts or foster positive interactions.

69- Outline of Implemantation

Outline of Implementation for Analyzing Positive and Negative Edges in a Social Network:

1. **Data Representation**:
   - Define a data structure to represent the social network, including nodes (individuals) and edges (relationships).
   - Each edge should have a weight or attribute indicating whether it is positive or negative.
2. **Data Acquisition**:
   - Obtain the social network data from sources such as social media APIs, online forums, or datasets.
   - Preprocess the data to ensure consistency and remove noise or irrelevant information.
3. **Graph Construction**:
   - Build a graph representation of the social network using the data acquired.
   - Create nodes for each individual and edges between them to represent relationships.
   - Assign weights or attributes to edges to differentiate between positive and negative relationships.
4. **Analysis and Visualization**:
   - Implement algorithms to analyze the structure of the social network and identify positive and negative edges.
   - Calculate relevant metrics such as node centrality, community detection, and sentiment analysis based on the positive and negative relationships.
   - Visualize the social network graph and highlight positive and negative edges using graph visualization libraries such as NetworkX, igraph, or Gephi.
5. **Community Detection**:
   - Apply community detection algorithms to identify groups of individuals with strong positive connections within the network.

- Evaluate the cohesion and separation of communities based on the presence of positive and negative edges.

6. **Sentiment Analysis**:
    - Perform sentiment analysis on text data associated with edges (e.g., comments, messages) to infer positive or negative sentiments.
    - Associate sentiment scores with edges to complement the manual labeling of positive and negative relationships.
7. **Dynamic Analysis**:
    - Consider the dynamic nature of social networks by analyzing changes in positive and negative relationships over time.
    - Implement techniques such as temporal network analysis to track the evolution of the network structure and sentiment dynamics.
8. **Evaluation and Interpretation**:
    - Evaluate the effectiveness of the implemented methods in identifying and analyzing positive and negative edges.
    - Interpret the findings to gain insights into community structures, sentiment dynamics, and the overall social network behavior.
9. **Optimization and Scalability**:
    - Optimize the implementation for efficiency and scalability, especially for large-scale social networks.
    - Consider distributed computing frameworks or parallel processing techniques to handle big data scenarios.
10. **Documentation and Reporting**:
    - Document the implementation details, including algorithms, data preprocessing steps, and visualization techniques.
    - Prepare reports or presentations summarizing the analysis results and insights derived from analyzing positive and negative edges in the social network.

70- Creating graph, displaying it and counting unstable triangles

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

def count_unstable_triangles(graph):
    unstable_triangles = 0
    for node in graph.nodes():
        neighbors = list(graph.neighbors(node))
        for i in range(len(neighbors)):
            for j in range(i+1, len(neighbors)):
                if not graph.has_edge(neighbors[i], neighbors[j]):
```

```
                unstable_triangles += 1
    return unstable_triangles

# Create an empty graph
G = nx.Graph()

# Add nodes
G.add_nodes_from([1, 2, 3, 4, 5])

# Add positive edges
positive_edges = [(1, 2), (1, 3), (2, 3), (3, 4), (4, 5), (5, 1)]
G.add_edges_from(positive_edges)

# Add negative edges
negative_edges = [(2, 4), (3, 5)]
G.add_edges_from(negative_edges, color='red')

# Display the graph
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G, pos, node_color='skyblue')
nx.draw_networkx_edges(G, pos, edgelist=positive_edges, edge_color='black')
nx.draw_networkx_edges(G, pos, edgelist=negative_edges, edge_color='red')
nx.draw_networkx_labels(G, pos)
plt.title("Social Network Graph")
plt.axis("off")
plt.show()

# Count unstable triangles (triads)
unstable_triangles = count_unstable_triangles(G)
print("Number of unstable triangles (triads):", unstable_triangles)
```

71- Moving a network from an unstable to stable state

Moving a network from an unstable to a stable state typically involves adjusting the relationships (edges) within the network to adhere to principles of structural balance theory. This process can be achieved through various mechanisms, such as adding or removing edges, modifying edge weights, or introducing new nodes. Here's an outline of steps to move a network from an unstable to a stable state:

1. **Identify Unstable Triads**: Analyze the network to identify unstable triads, which are configurations of three nodes connected by edges that violate principles of structural balance theory. Unstable triads include scenarios where two positive edges and one negative edge exist between the nodes.
2. **Determine Adjustment Strategies**:
    - **Edge Addition**: Identify pairs of nodes with positive relationships that lack direct connections. Add new positive edges between these nodes to promote transitive balance.

   - **Edge Removal**: Identify pairs of nodes with negative relationships that are indirectly connected through positive relationships. Remove the indirect positive edges to eliminate cyclically unbalanced triads.
   - **Node Addition**: Introduce new nodes to mediate conflicts between nodes with negative relationships. Establish positive connections between the new nodes and the conflicting nodes to restore balance.

3. **Implement Adjustments**:
   - Modify the network structure based on the identified adjustment strategies. Add, remove, or modify edges and nodes as needed to transition the network towards stability.
   - Ensure that adjustments are made gradually to minimize disruptions and maintain the overall coherence of the network.
4. **Evaluate Stability**:
   - After implementing adjustments, reassess the network to determine whether it has transitioned to a more stable state.
   - Check for the presence of unstable triads and assess whether the network adheres to principles of structural balance theory.
5. **Iterative Refinement**:
   - If the network remains unstable or exhibits new instabilities, iteratively refine the adjustment strategies and implement further modifications.
   - Continuously evaluate the impact of adjustments on the network's stability and refine the approach accordingly.
6. **Monitor and Maintain**:
   - Once the network achieves stability, monitor its dynamics over time to ensure that it remains balanced and resilient to perturbations.
   - Address any emerging conflicts or instabilities promptly through targeted adjustments or interventions.

```
import networkx as nx

def identify_unstable_triads(graph):
    unstable_triads = []
    for node in graph.nodes():
        neighbors = list(graph.neighbors(node))
        for i in range(len(neighbors)):
            for j in range(i+1, len(neighbors)):
                if not graph.has_edge(neighbors[i], neighbors[j]):
                    unstable_triads.append((node, neighbors[i], neighbors[j]))
    return unstable_triads

def add_positive_edge(graph, node1, node2):
    if not graph.has_edge(node1, node2):
        graph.add_edge(node1, node2)
```

```
def remove_negative_edge(graph, node1, node2):
    if graph.has_edge(node1, node2):
        graph.remove_edge(node1, node2)

# Create an empty graph
G = nx.Graph()

# Add nodes
G.add_nodes_from([1, 2, 3, 4, 5])

# Add positive edges
positive_edges = [(1, 2), (1, 3), (2, 3), (3, 4), (4, 5), (5, 1)]
G.add_edges_from(positive_edges)

# Add negative edges (intentionally making the network unstable)
negative_edges = [(2, 4), (3, 5)]
G.add_edges_from(negative_edges)

# Identify unstable triads
unstable_triads = identify_unstable_triads(G)
print("Unstable Triads:", unstable_triads)

# Implement adjustments
for triad in unstable_triads:
    node1, node2, node3 = triad
    add_positive_edge(G, node2, node3)
    remove_negative_edge(G, node1, node2)

# Display the updated graph
nx.draw(G, with_labels=True)
plt.title("Updated Network")
plt.show()
```

72- Forming two coalitions

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

def form_coalitions(graph):
    # Initialize two empty coalitions
    coalition1 = set()
    coalition2 = set()

    # Perform community detection to partition the network into two groups
    communities = nx.algorithms.community.greedy_modularity_communities(graph)

    # Assign nodes to the coalitions based on the detected communities
    for node in graph.nodes():
        if node in communities[0]:
            coalition1.add(node)
        else:
            coalition2.add(node)

    return coalition1, coalition2

# Create an empty graph
G = nx.Graph()

# Add nodes
G.add_nodes_from([1, 2, 3, 4, 5, 6])

# Add positive edges
positive_edges = [(1, 2), (1, 3), (2, 3), (3, 4), (4, 5), (5, 1), (5, 6)]
G.add_edges_from(positive_edges)

# Display the original graph
pos = nx.spring_layout(G)
nx.draw_networkx(G, pos, with_labels=True)
plt.title("Original Network")
plt.show()

# Form coalitions
coalition1, coalition2 = form_coalitions(G)
print("Coalition 1:", coalition1)
print("Coalition 2:", coalition2)
```

In this code:

- We first define a function **form_coalitions** to partition the network into two coalitions using community detection algorithms.
- We create a social network graph with nodes and positive edges representing relationships.
- We display the original network graph using matplotlib.
- We then call the **form_coalitions** function to partition the network into two coalitions.
- Finally, we print the nodes belonging to each coalition.

73- Forming two coalitions (Continued)

If you need to form two coalitions in a social network without relying on community detection algorithms, you can implement a partitioning algorithm based on the principles of structural balance theory. Here's an approach to partition the network into two coalitions manually:

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

```
def form_coalitions(graph):
    # Initialize two empty coalitions
    coalition1 = set()
    coalition2 = set()

    # Perform iterative partitioning based on structural balance principles
    for node in graph.nodes():
        # Check the number of connections to nodes in each coalition
        connections_to_coalition1 = len(set(graph.neighbors(node)).intersection(coalition1))
        connections_to_coalition2 = len(set(graph.neighbors(node)).intersection(coalition2))

        # Assign the node to the coalition with fewer connections
        if connections_to_coalition1 <= connections_to_coalition2:
            coalition1.add(node)
        else:
            coalition2.add(node)

    return coalition1, coalition2

# Create an empty graph
G = nx.Graph()

# Add nodes
G.add_nodes_from([1, 2, 3, 4, 5, 6])

# Add positive edges
positive_edges = [(1, 2), (1, 3), (2, 3), (3, 4), (4, 5), (5, 1), (5, 6)]
G.add_edges_from(positive_edges)

# Display the original graph
pos = nx.spring_layout(G)
nx.draw_networkx(G, pos, with_labels=True)
plt.title("Original Network")
plt.show()

# Form coalitions
coalition1, coalition2 = form_coalitions(G)
print("Coalition 1:", coalition1)
print("Coalition 2:", coalition2)
```

74- Visualizing coalitions and the evolution

Visualizing coalitions and their evolution with positive and negative edges (representing cooperation and conflict, respectively) can be done using network graphs. Here's how you could approach it:

1. **Network Graphs**: Use nodes to represent individual actors or entities within the coalitions, and edges to represent the relationships between them. Positive edges (cooperation) can be

represented by solid lines, while negative edges (conflict) can be represented by dashed or differently colored lines.

2. **Node Attributes**: You can assign attributes to nodes to denote their characteristics, such as their affiliation with specific coalitions, their influence, or their role within the network.
3. **Dynamic Visualization**: As the coalitions evolve over time, you can animate the network graph to show the changes in the relationships between actors. New nodes can appear, existing nodes can change their affiliations or roles, and edges can be added or removed to reflect the evolving dynamics of cooperation and conflict.
4. **Metrics**: Use network analysis metrics such as centrality measures (e.g., degree centrality, betweenness centrality) to identify key actors within the coalitions and their roles in shaping the network structure. You can visualize these metrics by adjusting the size or color of the nodes or by adding labels to highlight the most influential actors.
5. **Cluster Analysis**: Apply clustering algorithms to identify communities or subgroups within the coalitions based on the patterns of cooperation and conflict. You can visualize these clusters by assigning different colors or shapes to nodes belonging to the same community, making it easier to identify cohesive groups within the network.
6. **Time-Series Analysis**: Analyze the temporal patterns of cooperation and conflict within the coalitions by examining how the strength and frequency of edges change over time. You can visualize these temporal dynamics using animations or by plotting time-series graphs showing the evolution of key network metrics.

Week 6: Link Analysis Social Network

What is link analysis?

Link analysis is an analytics technique used to identify, evaluate and understand the connections within data. The data to perform link analysis is stored in a graph database and then is displayed as a graph visualization, also called a network visualization.

Individual data points in a graph data model are represented by nodes. These entities are connected with edges - also called relationships - that represent the links between any nodes.

Both nodes and edges can have properties that store key information. For a node, this might be the name of a person or a business. For an edge, it could be the amount of a transaction.

These characteristics of link analysis make it an especially advantageous solution for organizations that need to understand the relationships within their data. It comes with several key benefits.

Fast discovery of insights

The human brain can process visual information many times faster than written or numerical information. By displaying your data as a visual network, link analysis can enable you to find exactly the information you need in a matter of seconds.

Scalable data analysis

Link analysis can display and analyze huge amounts of data from multiple sources. Even when the quantity of data is very large, querying data stored in a graph is quick, enabling you to scale any project.

Accessible and intuitive analytics

Displaying data as a network presents an easy and intuitive way for even non-technical users to understand and explore complex connections.

Link analysis use cases

Link analysis provides a powerful tool for identifying patterns and connections that might be difficult - or even impossible - to detect using other methods. By analyzing the relationships between entities, analysts can gain insights into complex networks, surfacing hidden insights or anomalies, or identifying areas for further investigation.

Here are some common use cases for link analysis.

Law enforcement

In a law enforcement context, agencies can use link analysis to identify connections between individual entities or organizations involved in criminal activities: drug trafficking, organized crime, etc.

Using link analysis, investigators can identify individuals who are connected to multiple crimes, for example, or surface connections between criminal suspects and their accomplices.

Cybersecurity

Cybersecurity analysts need to be able to quickly identify security threats and risk to effectively protect their organization.

Link analysis can help them identify patterns of behavior that may indicate suspicious activity. It can be used for example to identify connections between malicious domains or IP addresses or to spot patterns of behavior that indicate phishing or another type of cyber attack.

Intelligence analysis

Intelligence agencies commonly use link analysis to identify connections between individuals or organizations involved in criminal or terrorist activities.

Investigators may use link analysis to flag individuals who are connected to known participants in a terrorist organization or to spot connections between criminals and their associates.

Let’s take a deeper dive into the use case of fraud detection to see in detail how link analysis can be applied.

Fraud detection with link analysis

Let’s look at one use case where link analysis performs especially well to understand how this type of analytics can be applied.

Fraud is an increasingly complex problem for financial institutions, insurers, and other businesses to manage. Fraudsters often operate in networks that resemble professional organizations. Oftentimes they work across borders. Fraudsters are also experts in evading prevention and detection systems, quickly evolving their criminal techniques to get around new prevention tools.

The data analysts use to detect possible cases of fraud is therefore full of complexity, and comes from multiple sources. Link analysis enables you to visualize and explore all that data in one place. Patterns indicative of fraud become much more apparent.

Take the example of fraudulent car accident claims filed with an insurance company. Scaling the fake car accidents requires multiple policyholders, multiple cars and multiple car passengers.

Link analysis can help identify if within an insurance company's client database several people are interconnected across cars, individuals, repair shops or claims. The larger the network, the more likely that fraud is going on.

75: The Web Graph

In link analysis, the web graph serves as a fundamental dataset for understanding the relationships between web pages and assessing their importance or relevance. Here's how the web graph is utilized in link analysis:

1. **PageRank**: PageRank is a widely used algorithm for evaluating the importance of web pages within the web graph. It assigns a numerical score to each page based on the number and quality of incoming links, as well as the importance of the pages linking to it. Pages with higher PageRank scores are considered more influential or authoritative. Google's original search algorithm heavily relied on PageRank to rank search results.
2. **HITS (Hypertext Induced Topic Selection)**: HITS is another algorithm used for link analysis, which aims to identify both authoritative pages (hubs) and relevant content pages (authorities) within the web graph. HITS iteratively computes two scores for each page: hub score and authority score. A page with a high hub score is considered a good hub, while a page with a high authority score is considered a good authority.
3. **Link-Based Classification**: In addition to ranking pages, link analysis techniques are also used for classification tasks. By analyzing the inbound links to a page, classifiers can infer the topic or category of the page. For example, if a page receives many links from pages discussing sports, it might be classified as a sports-related page.
4. **Spam Detection**: Link analysis is effective in detecting spam or low-quality pages within the web graph. Spam pages often engage in techniques such as link farming or link spamming to artificially boost their rankings. By analyzing the link patterns and relationships between pages, spam detection algorithms can identify and penalize such pages.
5. **Link Prediction**: Link analysis methods are also employed to predict future links within the web graph. By analyzing the existing link structure and patterns, predictive models can estimate the likelihood of a new link forming between two pages. This has applications in improving web navigation, recommending related content, and identifying potential collaborations between websites.
6. **Network Visualization and Exploration**: Visualizing the web graph and exploring its structure can provide valuable insights into the organization and connectivity of the World Wide Web. Network visualization tools allow researchers and practitioners to explore link patterns, identify clusters or communities of related pages, and discover emerging trends or topics.

76: Collecting the Web Graph

Collecting the web graph involves crawling and indexing the vast expanse of interconnected web pages available on the internet. While the process can be complex and resource-intensive, it typically follows these general steps:

1. **Web Crawling**: A web crawler, also known as a web spider or web bot, is a program that systematically browses the internet, retrieving web pages and following hyperlinks to discover new pages. The crawler starts with a set of seed URLs, typically provided by the user or generated based on known starting points such as popular websites or search engine results.
2. **Page Retrieval**: The crawler downloads the HTML content of each web page it encounters during the crawling process. It may use techniques such as HTTP requests and response handling to access and retrieve the content of web pages.
3. **Parsing**: Once a web page is retrieved, the crawler parses the HTML to extract relevant information, including hyperlinks, metadata, and textual content. Hyperlinks are particularly important for building the web graph, as they provide connections between pages.
4. **URL Filtering and Deduplication**: During crawling, the crawler may encounter duplicate or irrelevant URLs, as well as URLs pointing to non-HTML content such as images or binary files. Filtering mechanisms are applied to remove duplicates and exclude non-HTML content from further processing.
5. **Link Extraction**: The crawler extracts hyperlinks from the parsed HTML content of each web page. It identifies both internal links (pointing to pages within the same website) and external links (pointing to pages on other websites).
6. **Storing and Indexing**: The extracted data, including URLs, hyperlink relationships, and content metadata, are stored in a database or index for further analysis. This database serves as the foundation for constructing the web graph and enables efficient querying and retrieval of information.
7. **Respecting Robots Exclusion Protocol**: Web crawlers adhere to the Robots Exclusion Protocol (robots.txt) to respect the directives provided by website owners regarding which parts of their site should not be crawled or indexed. This helps maintain good relations with website operators and ensures compliance with legal and ethical guidelines.
8. **Recrawling and Updating**: The web graph is dynamic, with web pages being added, removed, or modified over time. Periodic recrawling of web pages allows the web graph to be updated to reflect these changes, ensuring that the data remains current and accurate.
9. **Scaling and Distribution**: Collecting the entire web graph requires significant computational resources and infrastructure. To handle the scale of the web, crawling tasks may be distributed across multiple machines or servers, with coordination mechanisms in place to ensure efficient and comprehensive coverage.

77: Equal Coin Distribution

Equal coin distribution, also known as fair coin distribution or fair coin flipping, refers to the process of fairly distributing a set of coins among a group of participants. The goal is to ensure that each participant receives an equal share of the coins without any bias or unfair advantage.

There are various methods to achieve equal coin distribution, depending on the specific requirements and constraints of the distribution scenario. Here are a few common approaches:

1. **Random Allocation**: In this method, coins are randomly distributed among the participants. Each participant has an equal chance of receiving any given coin, and the distribution process is repeated until all coins have been allocated. Random allocation ensures fairness by eliminating any systematic bias in the distribution process.
2. **Sequential Distribution**: In sequential distribution, coins are distributed one at a time, with each participant taking turns to receive a coin. The order of distribution may be determined randomly or based on some predetermined criteria. This method ensures that each participant receives an equal number of coins over the course of the distribution process.
3. **Equal Division**: If the number of coins is divisible by the number of participants, equal division can be used to ensure that each participant receives an equal share of the coins. For example, if there are 100 coins and 10 participants, each participant would receive 10 coins. If the number of coins is not evenly divisible by the number of participants, some form of rounding or adjustment may be necessary to achieve equal distribution.
4. **Fair Division Algorithms**: Fair division algorithms, such as the "divide and choose" method or the "moving-knife" procedure, can be employed to ensure that the distribution is perceived as fair by all participants. These algorithms involve a series of steps where participants take turns dividing the coins into shares and selecting their preferred share. The process continues until all coins have been allocated, with each participant receiving a share that they consider fair.
5. **Auction or Bidding**: In some cases, an auction or bidding process may be used to allocate coins among participants. Participants bid for the coins, with the highest bidders receiving the coins until they are all distributed. This method ensures that participants who value the coins more highly have the opportunity to acquire them, but it may not guarantee equal distribution unless the auction is designed accordingly.

```
import random

def distribute_coins(num_coins, num_participants):
    coins_per_participant = num_coins // num_participants
    remaining_coins = num_coins % num_participants

    participants = [0] * num_participants  # Initialize list to store coins for each participant

    # Distribute coins evenly among participants
    for i in range(num_participants):
        participants[i] += coins_per_participant

    # Distribute remaining coins randomly
    remaining_indices = list(range(num_participants))
    for _ in range(remaining_coins):
        random_index = random.choice(remaining_indices)
        participants[random_index] += 1
        remaining_indices.remove(random_index)

    return participants

# Example usage:
num_coins = 100
num_participants = 10

result = distribute_coins(num_coins, num_participants)
print("Coins distributed:", result)
print("Total coins distributed:", sum(result))
```

This code defines a function **distribute_coins** that takes the total number of coins (**num_coins**) and the number of participants (**num_participants**) as input parameters. It first calculates how many coins each participant should receive evenly (**coins_per_participant**) and distributes them accordingly. Then, it randomly distributes any remaining coins among the participants. Finally, it returns a list indicating the number of coins each participant received.

78: Random Walk Coin Distribution

A random walk-based method for coin distribution involves simulating a random process where coins are passed between participants according to some predefined rules. Here's a Python implementation of a simple random walk-based coin distribution:

```
import random
def random_walk_coin_distribution(num_coins, num_participants, num_steps):
    # Initialize coins for each participant
    participants = [0] * num_participants
    # Perform random walk for num_steps
    for _ in range(num_steps):
        # Choose a random participant as the giver
        giver = random.randint(0, num_participants - 1)
        # Choose a random participant as the receiver
        receiver = random.randint(0, num_participants - 1)
        # Ensure giver and receiver are not the same
        while receiver == giver:
            receiver = random.randint(0, num_participants - 1)
        # Transfer a coin from giver to receiver
        participants[giver] -= 1
        participants[receiver] += 1
    return participants
# Example usage:
num_coins = 100
num_participants = 10
num_steps = 1000

result = random_walk_coin_distribution(num_coins, num_participants, num_steps)
print("Coins distributed:", result)
print("Total coins distributed:", sum(result))
```

In this code:

- The function **random_walk_coin_distribution** takes the total number of coins (**num_coins**), the number of participants (**num_participants**), and the number of steps for the random walk (**num_steps**) as input parameters.
- It initializes an empty list **participants** to represent the number of coins held by each participant.
- Then, for each step in the random walk, it randomly selects a giver participant and a receiver participant, ensuring they are not the same.
- It transfers one coin from the giver to the receiver in each step.
- After completing all steps, it returns the list indicating the number of coins held by each participant.

79: Google Page Ranking Using Web Graph

PageRank (PR) is an algorithm used by Google Search to rank websites in their search engine results. PageRank was named after Larry Page, one of the founders of Google. PageRank is a way of measuring the importance of website pages. According to Google:

It is not the only algorithm used by Google to order search engine results, but it is the first algorithm that was used by the company, and it is the best-known.
The above centrality measure is not implemented for multi-graphs.

Algorithm
The PageRank algorithm outputs a probability distribution used to represent the likelihood that a person randomly clicking on links will arrive at any particular page. PageRank can be calculated for collections of documents of any size. It is assumed in several research papers that the distribution is evenly divided among all documents in the collection at the beginning of the computational process. The PageRank computations require several passes, called “iterations”, through the collection to adjust approximate PageRank values to more closely reflect the theoretical true value.

Simplified algorithm
Assume a small universe of four web pages: A, B, C, and D. Links from a page to itself, or multiple outbound links from one single page to another single page, are ignored. PageRank is initialized to the same value for all pages. In the original form of PageRank, the sum of PageRank over all pages was the total number of pages on the web at that time, so each page in this example would have an initial value of 1. However, later versions of PageRank, and the remainder of this section, assume a probability distribution between 0 and 1. Hence the initial value for each page in this example is 0.25.
The PageRank transferred from a given page to the targets of its outbound links upon the next iteration is divided equally among all outbound links.
If the only links in the system were from pages B, C, and D to A, each link would transfer 0.25 PageRank to A upon the next iteration, for a total of 0.75.

$$PR(A) = PR(B) + PR(C) + PR(D).$$

Suppose instead that page B had a link to pages C and A, page C had a link to page A, and page D had links to all three pages. Thus, upon the first iteration, page B would transfer half of its existing value, or 0.125, to page A and the other half, or 0.125, to page C. Page C would transfer all of its existing value, 0.25, to the only page it links to, A. Since D had three outbound links, it would transfer one-third of its existing value, or approximately 0.083, to A. At the completion of this iteration, page A will have a PageRank of approximately 0.458.

$$PR(A) = \frac{PR(B)}{2} + \frac{PR(C)}{1} + \frac{PR(D)}{3}.$$

In other words, the PageRank conferred by an outbound link is equal to the document's own PageRank score divided by the number of outbound links L( ).

$$PR(A) = \frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)}.$$

In the general case, the PageRank value for any page u can be expressed as:

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

,

i.e. the PageRank value for a page u is dependent on the PageRank values for each page v contained in the set Bu (the set containing all pages linking to page u), divided by the number L(v) of links from page v. The algorithm involves a damping factor for the calculation of the PageRank. It is like the income tax which the govt extracts from one despite paying him itself.

Following is the code for the calculation of the Page rank.
Python program for Page-Rank Algorithm

```
def page_rank(graph, damping_factor=0.85, max_iterations=100, tolerance=1e-6):
    num_nodes = len(graph)
    initial_pr = 1.0 / num_nodes
    page_rank = {node: initial_pr for node in graph}
    out_degrees = {node: len(graph[node]) for node in graph}

    for _ in range(max_iterations):
        prev_page_rank = page_rank.copy()
        total_diff = 0.0

        for node in graph:
            page_rank[node] = (1 - damping_factor) / num_nodes
            for neighbor in graph[node]:
                page_rank[node] += damping_factor * prev_page_rank[neighbor] / out_degrees[neighbor]


            diff = abs(page_rank[node] - prev_page_rank[node])
            total_diff += diff

        if total_diff < tolerance:
            break

    return page_rank

if __name__ == "__main__":
    # Example graph represented as a dictionary of node: [list of neighbors]
    graph = {
        'A': ['B', 'C'],
        'B': ['C'],
        'C': ['A'],
        'D': ['C']
    }

    result = page_rank(graph)
    sorted_result = {k: v for k, v in sorted(result.items(), key=lambda item: item[1], reverse=True)}

    print("PageRank results:")
    for node, pr in sorted_result.items():
        print(f"{node}: {pr:.4f}")
```

Sample Output

```
PageRank scores:
Page 1: 0.230681
Page 2: 0.330681
Page 3: 0.330681
```

80:81:82:83: Implementing PageRank Using Points Distribution Method-1-2-3-4

1. Introduction to PageRank:

- PageRank is an algorithm used by search engines to rank web pages in their search results.
- It was developed by Larry Page and Sergey Brin, the founders of Google, and was the foundation of Google's early search engine algorithms.

2. Basic Concept:

- PageRank models the importance of a web page based on the idea of "voting" or "recommendation".
- The more incoming links a page has, the more important it is considered.
- It's based on the assumption that important pages are more likely to be linked from other important pages.

3. Points Distribution Method:

- PageRank algorithm can be understood using the points distribution method.
- Imagine each web page as a node in a network, and each link from one page to another as a directed edge.
- Initially, we distribute points equally among all nodes.
- At each iteration, each node distributes its points to its outgoing links equally.
- The process continues until a stable distribution of points is reached.

4. Formula:

- The formula to calculate the PageRank score for a node is:
  ��(��)=1−��+�×∑��∈�(��)��(��)�(��)*PR*(*pi*)=*N*1−*d*+*d*×∑*pj*∈*M*(*pi*)*L*(*pj*)*PR*(*pj*)
    - ��(��)*PR*(*pi*) is the PageRank score of node ��*pi*.
    - �*N* is the total number of nodes.

- �*d* is the damping factor, usually set to 0.85.
- �(��)*M*(*pi*) is the set of nodes that have links to node ��*pi*.
- �(��)*L*(*pj*) is the number of outgoing links from node ��*pj*.

5. Iterative Computation:

- PageRank scores are iteratively computed until convergence or a maximum number of iterations.
- At each iteration, the PageRank scores are updated based on the formula.
- Convergence is typically determined by measuring the change in PageRank scores between iterations.

6. Importance of Damping Factor:

- The damping factor �*d* is crucial in avoiding "spider traps" and "dead ends" in the web graph.
- It introduces a probability of randomly jumping to any page, preventing the algorithm from getting stuck in cycles or disconnected components.

7. Handling Dead Ends:

- Dead ends are nodes with no outgoing links.
- To handle dead ends, we distribute their points evenly among all nodes.

8. Application:

- PageRank is widely used in web search engines to rank web pages based on their importance.
- It's also applicable in various other network analysis scenarios, such as social networks and citation networks.

Implementing PageRank using the points distribution method involves iteratively distributing points among nodes in a network until convergence is achieved.

Here's a simple Python implementation of the PageRank algorithm using the points distribution method:

```
def pagerank(graph, damping_factor=0.85, max_iterations=100, tolerance=1e-6):
    """
    Calculate PageRank using points distribution method.

    Args:
    - graph: Dictionary representing the graph where keys are nodes and values are lists of outgoing links
from each node.
    - damping_factor: Damping factor, typically set to 0.85.
    - max_iterations: Maximum number of iterations for convergence.
    - tolerance: Convergence tolerance.

    Returns:
    - page_rank: Dictionary containing PageRank scores for each node.
    """

    # Initialize PageRank scores
    num_nodes = len(graph)
    page_rank = {node: 1 / num_nodes for node in graph}

    # Iterate until convergence or max iterations reached
    for _ in range(max_iterations):
        new_page_rank = {}

        # Calculate new PageRank scores for each node
        for node in graph:
            new_rank = (1 - damping_factor) / num_nodes

            # Distribute points from incoming links
            for neighbor in graph:
                if node in graph[neighbor]:
                    num_links = len(graph[neighbor])
                    new_rank += damping_factor * page_rank[neighbor] / num_links

            new_page_rank[node] = new_rank

        # Check for convergence
        delta = sum(abs(new_page_rank[node] - page_rank[node]) for node in graph)
        if delta < tolerance:
            break

        page_rank = new_page_rank

    return page_rank

# Example graph
graph = {
    'A': ['B', 'C'],
    'B': ['A'],
    'C': ['B', 'D'],
    'D': ['C']
}

# Calculate PageRank scores
page_rank = pagerank(graph)
print("PageRank Scores:")
for node, score in sorted(page_rank.items()):
    print(f"{node}: {score}")
```

In this implementation:

- The pagerank function takes a graph represented as a dictionary where keys are nodes and values are lists of outgoing links from each node.
- It iteratively calculates the PageRank scores for each node until convergence is achieved or the maximum number of iterations is reached.

- The PageRank calculation formula used here is based on the points distribution method.
- The algorithm returns a dictionary containing the PageRank scores for each node in the graph.

Complexity of the Page-Rank Algorithm

The complexity of the PageRank algorithm depends on the size and sparseness of the autograph, represented by the number of nodes and edges.

Initialization of PageRank: Each node receives initial PageRank scores. This step takes O(N), where N is the number of nodes in the graph.

Iterative Updates: The algorithm updates the PageRank iteratively until convergence or the maximum number of iterations is reached. Each iteration involves computing new PageRank scores for all nodes. The algorithm considers the neighbors of each node and updates its PageRank based on the algorithm's formula.

The time complexity of updating the PageRank score of a single node is O(E_out), where E_out is the number of edges leaving the node. Since the PageRank update of each node is independent of other nodes, it takes O (N * E_out) time to update all nodes, where N is the number of nodes in the graph. Usually, the value of E_out is less than N, which makes the algorithm efficient. For sparse graphs, E_out can be significantly smaller than N, leading to faster convergence.

Convergence criteria: Convergence checking requires comparing all nodes' current and previous PageRank scores, which takes O(N) time. In general, the dominant factor in time complexity is iterative updates, specifically O (N * E_out), where N is the number of nodes and E_out is the average number of outgoing edges per node. In practice, the PageRank algorithm often converges quickly within a few iterations, especially for sparse graphs, making it efficient for ranking web pages according to their importance and popularity. However, large graphs may require special techniques and optimizations to handle scaling effectively.

84: Implementing PageRank Using Random Walk Method -1 & 85: Implementing PageRank Using Random Walk Method -2

Introduction A random walk is a mathematical object, known as a stochastic or random process, that describes a path that consists of a succession of random steps on some mathematical space such as the integers. An elementary example of a random walk is the random walk on the integer number line, which starts at 0 and at each step moves +1 or -1 with equal probability. Other examples include the path traced by a molecule as it travels in a liquid or a gas, the search path of a foraging animal, the price of a fluctuating stock and the financial status of a gambler can all be approximated by random walk models, even though they may not be truly random in reality. As illustrated by those examples, random walks have applications to many scientific fields including ecology, psychology, computer science, physics, chemistry, biology as well as economics. Random walks explain the observed behaviors of many processes in these fields and thus serve as a fundamental model for the recorded stochastic activity. As a more mathematical application, the value of pi can be approximated by the usage of random walk in the agent-based modeling environment..

Random Walk Method – In the random walk method we will choose 1 node from the graph uniformly at random. After choosing the node we will look at its neighbors and choose a neighbor uniformly at random and continue these iterations until convergence is reached. After N iterations a point will come after which there will be no change In points of every node. This situation is called convergence.

Algorithm: Below are the steps for implementing the Random Walk method.

1. Create a directed graph with N nodes.
2. Now perform a random walk.
3. Now get sorted nodes as per points during random walk.
4. At last, compare it with the inbuilt PageRank method.

Below is the python code for the implementation of the points distribution algorithm.

```
import networkx as nx
import random
import numpy as np
# Add directed edges in graph
def add_edges(g, pr):
    for each in g.nodes():
        for each1 in g.nodes():
            if (each != each1):
                ra = random.random()
                if (ra < pr):
                    g.add_edge(each, each1)
                else:
                    continue
    return g
# Sort the nodes
def nodes_sorted(g, points):
    t = np.array(points)
    t = np.argsort(-t)
    return t
# Distribute points randomly in a graph
def random_Walk(g):
    rwp = [0 for i in range(g.number_of_nodes())]
    nodes = list(g.nodes())
    r = random.choice(nodes)
    rwp[r] += 1
    neigh = list(g.out_edges(r))
    z = 0
    while (z != 10000):
        if (len(neigh) == 0):
            focus = random.choice(nodes)
        else:
            r1 = random.choice(neigh)
            focus = r1[1]
        rwp[focus] += 1
        neigh = list(g.out_edges(focus))
        z += 1
    return rwp
# Main
# 1. Create a directed graph with N nodes
```

```
g = nx.DiGraph()
N = 15
g.add_nodes_from(range(N))
# 2. Add directed edges in graph
g = add_edges(g, 0.4)
# 3. perform a random walk
points = random_Walk(g)

# 4. Get nodes rank according to their random walk points
sorted_by_points = nodes_sorted(g, points)
print("PageRank using Random Walk Method")
print(sorted_by_points)
# p_dict is dictionary of tuples
p_dict = nx.pagerank(g)
p_sort = sorted(p_dict.items(), key=lambda x: x[1], reverse=True)
print("PageRank using inbuilt pagerank method")
for i in p_sort:
    print(i[0], end=", ")
```

Output:

```
PageRank using Random Walk Method
[ 9 10  4  6  3  8 13 14  0  7  1  2  5 12 11]
PageRank using inbuilt pagerank method
9, 10, 6, 3, 4, 8, 13, 0, 14, 7, 1, 2, 5, 12, 11,
```

Enough with the boring theory. Let’s take a break while getting some knowledge of the code. So, to code out the random walk we will basically require some libraries in python some to do maths, and some others to plot the curve.

Libraries Required

- **matplotlib** It’s an external library that helps you to plot the curve. To install this library type the following code in your cmd.

```
pip install matplotlib
```

It would be enough to get you through the installation.

- **numpy** It’s also an external library in python it helps you to work with arrays and matrices. To install the library type the following code in cmd.

```
pip install numpy
```

- **random** It’s a built-in library of python we will use to generate random points.

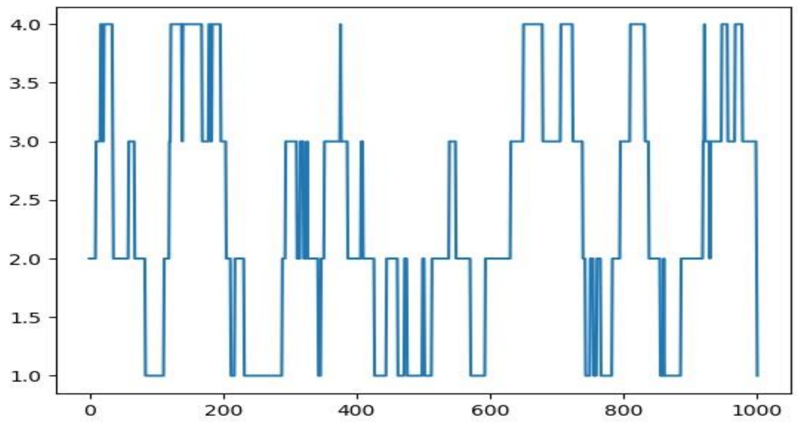
One-dimensional random walk An elementary example of a random walk is the random walk on the integer number line, which starts at 0 and at each step moves +1 or ? 1 with equal probability.

So let’s try to implement the 1-D random walk in python.

```
# Python code for 1-D random walk.
```

```
import random
import numpy as np
import matplotlib.pyplot as plt

# Probability to move up or down
prob = [0.05, 0.95]

# statically defining the starting position
start = 2
positions = [start]

# creating the random points
rr = np.random.random(1000)
downp = rr < prob[0]
upp = rr > prob[1]



for idownp, iupp in zip(downp, upp):
    down = idownp and positions[-1] > 1
    up = iupp and positions[-1] < 4
    positions.append(positions[-1] - down + up)

# plotting down the graph of the random walk in 1D
plt.plot(positions)
plt.show()
```

Output:

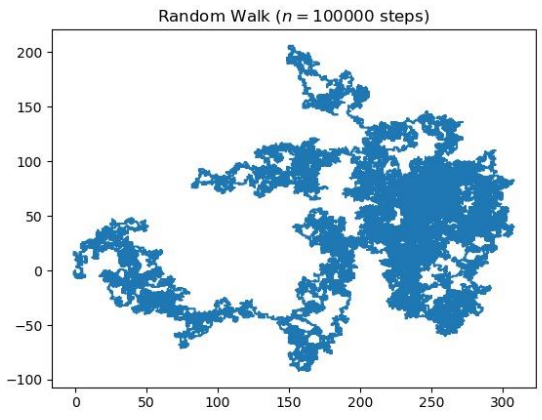


Higher dimensions In higher dimensions, the set of randomly walked points has interesting geometric properties. In fact, one gets a discrete fractal, that is, a set that exhibits stochastic self-similarity on large scales. On small scales, one can observe "jaggedness" resulting from the grid on which the walk is performed. Two books of Lawler referenced below are good sources on this topic. The trajectory of a random walk is the collection of points visited, considered as a set with disregard to when the walk arrived at the point. In one dimension, the trajectory is simply all points between the minimum height and the maximum height the walk achieved (both are, on average, on the order of ? n).

Let’s try to create a random walk in 2D.

```
# Python code for 2D random walk.
import numpy
import pylab
import random

# defining the number of steps
n = 100000

#creating two array for containing x and y coordinate
#of size equals to the number of size and filled up with 0's
x = numpy.zeros(n)
y = numpy.zeros(n)

# filling the coordinates with random variables
for i in range(1, n):
    val = random.randint(1, 4)
    if val == 1:
        x[i] = x[i - 1] + 1
        y[i] = y[i - 1]
    elif val == 2:
        x[i] = x[i - 1] - 1
        y[i] = y[i - 1]
    elif val == 3:
        x[i] = x[i - 1]
        y[i] = y[i - 1] + 1
    else:
        x[i] = x[i - 1]
        y[i] = y[i - 1] - 1



# plotting stuff:
pylab.title("Random Walk ($n = " + str(n) + "$ steps)")
pylab.plot(x, y)
pylab.savefig("rand_walk"+str(n)+".png",bbox_inches="tight",dpi=600)
pylab.show()
```

Output:



Applications

1. In computer networks, random walks can model the number of transmission packets buffered at a server.
2. In population genetics, a random walk describes the statistical properties of genetic drift.
3. In image segmentation, random walks are used to determine the labels (i.e., “object” or “background”) to associate with each pixel.
4. In brain research, random walks and reinforced random walks are used to model cascades of neuron firing in the brain.
5. Random walks have also been used to sample massive online graphs such as online social networks.

86: DegreeRank versus PageRank

DegreeRank and PageRank are both algorithms used in network analysis, particularly in the context of ranking nodes in a network. While they share some similarities, they also have distinct differences in their approaches and applications.

1. **DegreeRank**:
    - DegreeRank is a simple ranking algorithm based solely on the degree of nodes in a network.
    - The degree of a node in a network is the number of connections it has to other nodes (i.e., its number of edges).
    - DegreeRank assigns higher rankings to nodes with a higher degree, assuming that nodes with more connections are more important or influential in the network.
    - It's straightforward and easy to compute but may not capture the true importance of nodes in the network, especially in cases where the number of connections alone doesn't fully represent node significance.

here's a simple example of the **DegreeRank algorithm** applied to a small network:

Let's consider the following network represented by an adjacency matrix:

cssCopy code

A B C D A 0 1 1 0 B 1 0 1 1 C 1 1 0 1 D 0 1 1 0

In this network:

- Node A is connected to B and C.
- Node B is connected to A, C, and D.
- Node C is connected to A, B, and D.
- Node D is connected to B and C.

Now, let's calculate the degree of each node, which is the sum of connections (edges) for each node:

- Degree of A: 2
- Degree of B: 3
- Degree of C: 3
- Degree of D: 2

Now, we rank the nodes based on their degrees:

1. Node B (degree = 3)
2. Node C (degree = 3)
3. Nodes A and D (degree = 2, tie)

So, according to the DegreeRank algorithm, nodes B and C are ranked highest because they have the highest degree, and nodes A and D are ranked second because they have the same degree, and there's no other node with a higher degree.

This is a basic example of the DegreeRank algorithm, where nodes are ranked solely based on their degrees in the network.

2. **PageRank**:
    - PageRank is a more sophisticated ranking algorithm originally developed by Larry Page and Sergey Brin for ranking web pages in the context of the Google search engine.
    - PageRank assigns each node in a network a numerical score representing its importance, based on the structure of the incoming links.
    - The underlying idea is that nodes with many incoming links from important nodes are themselves considered important. It's a recursive algorithm where the importance of a node is determined by the importance of nodes linking to it.
    - PageRank considers not only the number of connections but also the quality and importance of those connections, making it more robust in capturing the significance of nodes in a network.
    - It's widely used not only in web search but also in various other network analysis applications, such as social network analysis, citation analysis, and recommendation systems.

Week 7: Cascading Behavior in Networks in Social Network

"In the realm of social networks, understanding how behaviors and information spread is crucial. This journey through cascading behavior in networks delves into the intricacies of why individuals follow, how diffusion unfolds, and the impact of community structures. From modeling techniques to exploring the thresholds that drive collective action, each topic offers a glimpse into the complex dynamics that shape interactions within social networks."

87 - We Follow

The "follow" feature in social networks is a fundamental aspect of how information spreads. When a user follows another user, they are essentially subscribing to that user's content, which then appears in their feed. This creates a network of connections that facilitates the diffusion of information and behaviors.

88 - Why do we Follow?

Users follow others for various reasons, such as shared interests, personal connections, or the desire to stay informed. The decision to follow someone can be influenced by factors such as the number of followers a user has, their content, or their social status.

89 - Diffusion in Networks

Diffusion in networks refers to the process by which information, behaviors, or innovations spread through a network of connected individuals. This can be influenced by various factors, such as the structure of the network, the characteristics of the individuals, and the nature of the information or behavior being diffused.

There are several types of diffusion that can occur in social networks, depending on the mechanisms and processes involved. Here are some common types of diffusion:

1. **Contagion:** Contagion refers to the spread of behaviors or information through direct contact between individuals. This can occur through mechanisms such as imitation or social learning, where individuals adopt behaviors or ideas from their connections.

2. **Social Influence:** Social influence refers to the impact that the opinions, attitudes, or behaviors of others can have on an individual's own behaviors or attitudes. This can occur through mechanisms such as conformity, where individuals adopt behaviors or attitudes to fit in with their social group, or through persuasion, where individuals are convinced to change their behaviors or attitudes by others.

3. **Homophily:** Homophily refers to the tendency of individuals to form connections with others who are similar to themselves. This can lead to the formation of clusters or communities within a network, where individuals share similar behaviors or attitudes.

4. **Structural Holes:** Structural holes refer to gaps or disconnections between clusters or groups within a network. These gaps can create opportunities for individuals to act as brokers or intermediaries, facilitating the spread of information or behaviors between groups.

5. **Complex Contagion:** Complex contagion refers to the spread of behaviors or information that requires multiple sources of exposure or reinforcement. This can occur through mechanisms such as threshold models, where an individual adopts a behavior only if a certain number of their connections have already adopted it.

6. **Heterogeneous Diffusion:** Heterogeneous diffusion refers to the spread of behaviors or information that varies depending on the characteristics of the individuals or the network. This can occur

through mechanisms such as preferential attachment, where individuals with more connections or higher status are more likely to adopt or spread behaviors or information.

90 - Modeling Diffusion

Diffusion can be modeled using various mathematical and computational approaches. These models can help predict how information or behaviors will spread through a network, identify key influencers, and understand the factors that influence the diffusion process.

Modeling diffusion involves using mathematical and computational approaches to understand and predict how information, behaviors, or innovations spread through a network of connected individuals. The following are some key aspects of modeling diffusion:

1. **Network Structure:** The structure of the network, such as its size, density, and clustering coefficient, can significantly impact the diffusion process. Modeling diffusion typically involves representing the network as a graph, where nodes represent individuals and edges represent connections between them.
2. **Node Characteristics:** The characteristics of the individuals in the network, such as their preferences, attitudes, and behaviors, can also influence the diffusion process. These characteristics can be modeled using attributes associated with the nodes in the graph.
3. **Diffusion Mechanisms:** The mechanisms by which diffusion occurs, such as contagion, social influence, or homophily, can be modeled using various mathematical functions. For example, contagion can be modeled using a simple threshold model, where an individual adopts a behavior if a certain number of their connections have already adopted it.
4. **Simulation:** Once the network and diffusion mechanisms have been modeled, simulations can be run to predict the spread of the behavior or information. These simulations can be used to test different scenarios, such as the impact of interventions or the effect of network structure on diffusion.
5. **Evaluation:** The results of the simulations can be evaluated using various metrics, such as the speed of diffusion, the reach of the behavior, or the impact of interventions. These evaluations can help inform strategies for promoting or preventing the diffusion of behaviors or information.

91- Modeling Diffusion (Continued)

Modeling diffusion in social networks typically involves using mathematical and computational approaches to understand and predict how information, behaviors, or innovations spread through a network of connected individuals. Here is a brief overview of the process, along with some example code using the Python programming language:

1. **Data Preparation:** The first step in modeling diffusion is to prepare the data, which typically involves creating a network representation of the social connections between individuals. This can be done using various data structures, such as adjacency matrices or edge lists. Here is an example using the NetworkX library in Python:

python

```
import networkx as nx
# Create a directed graph
G = nx.DiGraph()

# Add nodes to the graph
```

```
G.add_nodes_from(range(10))
# Add edges to the graph
G.add_edge(0, 1)
G.add_edge(1, 2)
G.add_edge(2, 3)
G.add_edge(3, 4)
G.add_edge(4, 5)
G.add_edge(5, 6)
G.add_edge(6, 7)
G.add_edge(7, 8)
G.add_edge(8, 9)
G.add_edge(9, 0)
```

2. **Modeling Diffusion Mechanisms:** Once the network has been created, the next step is to model the diffusion mechanisms, such as contagion, social influence, or homophily. This can be done using various mathematical functions, such as threshold models, linear models, or logistic regression. Here is an example using a simple threshold model in Python:

python

```
def threshold_model(G, thresh):
  # Initialize a dictionary to store the adoption status of each node
  adopted = {i: False for i in G.nodes()}
  # Iterate over the nodes in the graph
  for i in G.nodes():
    # Calculate the number of adopted neighbors
    num_adopted = sum(adopted[j] for j in G.neighbors(i))
     # If the number of adopted neighbors exceeds the threshold, adopt the behavior
     if num_adopted >= thresh:
        adopted[i] = True
  return adopted
```

3. **Simulation:** Once the network and diffusion mechanisms have been modeled, simulations can be run to predict the spread of the behavior or information. These simulations can be used to test different scenarios, such as the impact of interventions or the effect of network structure on diffusion. Here is an example using the threshold model in Python:

python

```
# Run the threshold model with a threshold of 3
adopted = threshold_model(G, 3)
# Print the adoption status of each node
for i, adopted_i in adopted.items():
print(f"Node {i}: {adopted_i}")
```

4. **Evaluation:** The results of the simulations can be evaluated using various metrics, such as the speed of diffusion, the reach of the behavior, or the impact of interventions. These evaluations can help inform strategies for promoting or preventing the diffusion of behaviors or information.

92 - Impact of Commmunities on Diffusion

Communities within social networks can significantly impact the diffusion process. These communities can act as bridges or barriers to the spread of information or behaviors, depending on their structure and the characteristics of their members.

Communities in social networks can have a significant impact on the diffusion of information, behaviors, and innovations. A community is a group of individuals who are closely connected to each other and share similar characteristics, interests, or behaviors. Communities can form spontaneously or be created intentionally, and they can vary in size, density, and cohesion.

The impact of communities on diffusion can be both positive and negative, depending on the nature of the community and the behavior or information being diffused. Here are some ways in which communities can impact diffusion:

1. **Amplification:** Communities can amplify the diffusion of information or behaviors by providing a dense network of connections and a shared context for communication and interaction. This can lead to faster and more widespread adoption of the behavior or information.
2. **Filtering:** Communities can also filter the diffusion of information or behaviors by serving as gatekeepers or intermediaries. This can lead to selective adoption of the behavior or information, based on the norms, values, or preferences of the community.
3. **Localization:** Communities can localize the diffusion of information or behaviors by creating a shared identity or sense of belonging. This can lead to the formation of subcultures or niches, where the behavior or information is more prevalent or accepted.
4. **Resistance:** Communities can also resist the diffusion of information or behaviors that are perceived as threatening or inconsistent with their norms, values, or interests. This can lead to the formation of counter-communities or social movements, where the behavior or information is actively opposed or rejected.
5. **Heterogeneity:** Communities can be heterogeneous, consisting of subgroups with different characteristics, interests, or behaviors. This can lead to complex patterns of diffusion, where the behavior or information spreads differently within and across communities.

93 - Cascade and Clusters

Cascades and Clusters

We continue exploring some of the consequences of our simple model of cascading behavior from the previous section: now that we've seen how cascades form, we look more deeply at what makes them stop. Our specific goal will be to formalize something that is intuitively apparent in Figure 19.5 — that the spread of a new behavior can stall when it tries to break in to a tightly-knit community within the network. This will in fact provide a way of formalizing a qualitative principle discussed earlier — that homophily can often serve as a barrier to diffusion, by making it hard for innovations to arrive from outside densely connected communities.

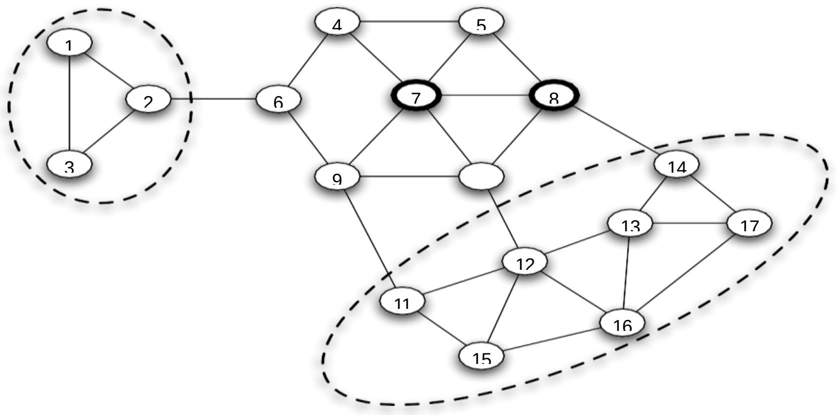


Figure 19.7: **Two clusters of density 2/3 in the network** from Figure 19.4.

As a first step, let’s think about how to make the idea of a “densely connected community” precise, so that we can talk about it in the context of our model. A key property of such communities is that when you belong to one, many of your friends also tend to belong. We can take this as the basis of a concrete definition, as follows.

For example, the set of nodes *a, b, c, d* forms a cluster of density 2/3 in the network in Figure 19.6. The sets *e, f, g, h* and *i, j, k, l* each form clusters of density 2/3 as well.

As with any formal definition, it’s important to notice the ways in which it captures our motivation as well as some of the ways in which it might not. Each node in a cluster does have a prescribed fraction of its friends residing in the cluster as well, implying some level of internal “cohesion.” On the other hand, our definition does not imply that any two particular nodes in the same cluster necessarily have much in common.

For example, in any network, the set of *all* nodes is always a cluster of density 1 — after all, by definition, all your network neighbors reside in the network. Also, if you have two clusters of density *p*, then the union of these two clusters (i.e. the set of nodes that lie in at least one of them) is also a cluster of density *p*. These observations are consistent with the notion that clusters in networks can exist simultaneously at many different scales.

94 - Knowledge, Thresholds and the Collective Action

Knowledge, thresholds, and collective action are interrelated concepts that are often used to understand the dynamics of social movements, innovation diffusion, and other forms of collective behavior in social networks.

1. **Knowledge:** Knowledge refers to the information, skills, and expertise that individuals possess and share in a social network. Knowledge can be acquired through formal education, personal experience, or social learning, and it can be transmitted through various channels, such as communication, imitation, or instruction. In the context of diffusion, knowledge is often a prerequisite for adoption, as individuals need to understand the benefits, risks, and implications of the behavior or information being diffused.
2. **Thresholds:** Thresholds refer to the minimum level of knowledge, motivation, or social pressure that is required for an individual to adopt a behavior or information. Thresholds can vary depending

on the individual, the behavior or information, and the social context, and they can be influenced by factors such as social norms, trust, and risk perception. In the context of diffusion, thresholds are often used to model the adoption dynamics of a population, where the probability of adoption increases as more neighbors adopt the behavior or information.

3. **Collective Action:** Collective action refers to the coordinated efforts of individuals or groups to achieve a common goal or objective. Collective action can take various forms, such as protests, boycotts, or social movements, and it can be motivated by various factors, such as shared values, interests, or grievances. In the context of diffusion, collective action can be facilitated or hindered by the distribution of knowledge, thresholds, and social connections in a social network.

The interplay between knowledge, thresholds, and collective action can have important implications for the dynamics of diffusion in social networks. For example, if a critical mass of individuals possesses the necessary knowledge and has low adoption thresholds, a behavior or information can spread rapidly and widely, leading to a cascade or avalanche of adoption. Conversely, if the knowledge is unevenly distributed, the thresholds are high, or the social connections are weak, the diffusion may be slower, more localized, or less effective.

Here is an example of a simple simulation of diffusion with knowledge, thresholds, and collective action in a social network using the Python programming language and the NetworkX library:

```
import networkx as nx
import random
# Define the network as a directed graph with 100 nodes
G = nx.gnm_random_graph(100, 200)
# Define the knowledge distribution as a random variable with a mean of 0.5 and a standard deviation of
0.1
knowledge = [random.gauss(0.5, 0.1) for i in range(100)]
# Define the threshold distribution as a random variable with a mean of 0.3 and a standard deviation of 0.1
thresholds = [random.gauss(0.3, 0.1) for i in range(100)]
# Define the initial adopters as a random subset of nodes with high knowledge and low thresholds
initial_adopters = [i for i, k, t in zip(range(100), knowledge, thresholds) if k > 0.7 and t < 0.2]
# Define the adoption function as a simple threshold model, where a node adopts if the fraction of its
neighbors who have adopted exceeds its threshold
def adopt(node, adopted_neighbors):
  if sum(adopted_neighbors) / len(adopted_neighbors) > thresholds[node]:
    return True
  else:
    return False
# Define the diffusion function as a loop that iterates over the nodes and their neighbors, and updates the
adoption status based on the adoption function
def diffuse(G, knowledge, thresholds, initial_adopters):
  adopted = set(initial_adopters)
  unadopted = set(range(100)) - adopted
  iteration = 0
  while len(unadopted) > 0:
    iteration += 1
    new_adopters = set()
    for node in unadopted:
      adopted_neighbors = [n for n in G.neighbors(node) if n in adopted]
```

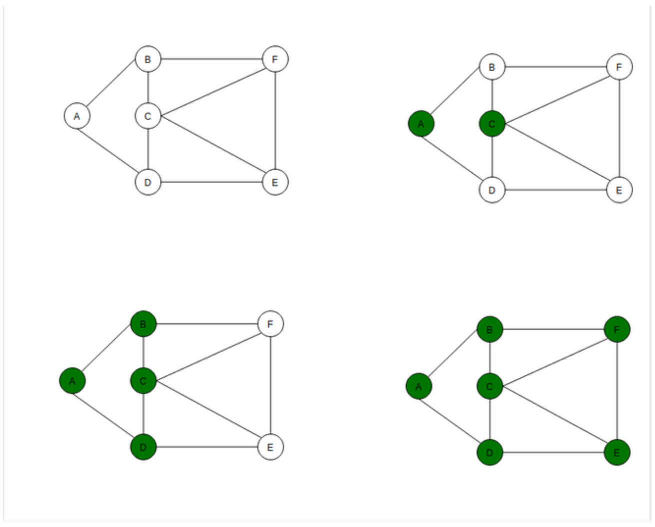
```
            if adopt(node, adopted_neighbors):
                adopted.add(node)
                new_adopters.add(node)
        unadopted = unadopted - new_adopters
    return adopted
# Run the diffusion function and print the adoption status of each node
adopted = diffuse(G, knowledge, thresholds, initial_adopters)
for i in range(100):
    if i in adopted:
        print(f"Node {i}: Adopted")
    else:
        print(f"Node {i}: Not Adopted")
```

95 - An Introduction to the Programming Screencast (Coding 4 major ideas)

96 - The Base Code

When people are connected in networks to each other then they can influence each other's behavior and decisions. This is called Cascading Behavior in Networks.

Let's consider an example, assume all the people in a society have adopted a trend X. Now there comes new trend Y and a small group accepts this new trend and after this, their neighbors also accept this trend Y and so on.



So, there are 4 main ideas in Cascading Behaviors:

1. Increasing the payoff.
2. Key people.
3. Impact of communities on the Cascades.
4. Cascading and Clusters.

Below is the code for each idea.

97 - Coding the First Big Idea - Increasing the Payoff

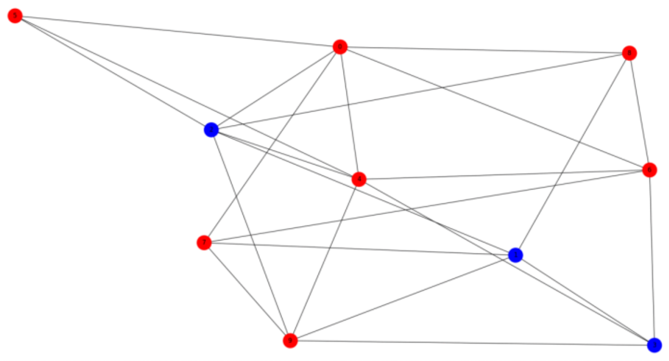
1. Increase the payoff.

- Python3

```
# cascade pay off
import networkx as nx
import matplotlib.pyplot as plt
def set_all_B(G):
    for i in G.nodes():
        G.nodes[i]['action'] = 'B'
    return G
def set_A(G, list1):
    for i in list1:
        G.nodes[i]['action'] = 'A'
    return G
def get_colors(G):
    color = []
    for i in G.nodes():
        if (G.nodes[i]['action'] == 'B'):
            color.append('red')
        else:
            color.append('blue')
    return color
def recalculate(G):
    dict1 = {}
    # payoff(A)=a=4
    # payoff(B)=b=3
    a = 15
    b = 5
    for i in G.nodes():
        neigh = G.neighbors(i)
        count_A = 0
        count_B = 0
        for j in neigh:
            if (G.nodes[j]['action'] == 'A'):
                count_A += 1
            else:
                count_B += 1
        payoff_A = a * count_A
        payoff_B = b * count_B
        if (payoff_A >= payoff_B):
            dict1[i] = 'A'
        else:
            dict1[i] = 'B'
    return dict1
def reset_node_attributes(G, action_dict):
    for i in action_dict:
        G.nodes[i]['action'] = action_dict[i]
    return G
def Calculate(G):
    terminate = True
    count = 0
    c = 0
    while (terminate and count < 10):
        count += 1
        # action_dict will hold a dictionary
        action_dict = recalculate(G)
        G = reset_node_attributes(G, action_dict)
        colors = get_colors(G)
        if (colors.count('red') == len(colors) or colors.count('green') == len(colors)):
            terminate = False
            if (colors.count('green') == len(colors)):
                c = 1
        nx.draw(G, with_labels=1, node_color=colors, node_size=800)
        plt.show()
    if (c == 1):
        print('cascade complete')
    else:
        print('cascade incomplete')
G = nx.erdos_renyi_graph(10, 0.5)
nx.write_gml(G, "erdos_graph.gml")
G = nx.read_gml('erdos_graph.gml')
print(G.nodes())
G = set_all_B(G)
# initial adopters
list1 = ['2', '1', '3']
G = set_A(G, list1)
colors = get_colors(G)
nx.draw(G, with_labels=1, node_color=colors, node_size=800)
plt.show()
Calculate(G)
```

Output:

```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
cascade complete
```



98 - Coding the Second Big Idea - Key People

2. Key people.

- Python3

```
# cascade key people
import networkx as nx
import matplotlib.pyplot as plt
G = nx.erdos_renyi_graph(10, 0.5)
nx.write_gml(G, "erdos_graph.gml")
def set_all_B(G):
    for i in G.nodes():
        G.nodes[i]['action'] = 'B'
    return G
def set_A(G, list1):
    for i in list1:
        G.nodes[i]['action'] = 'A'
    return G
def get_colors(G):
    color = []
    for i in G.nodes():
        if (G.nodes[i]['action'] == 'B'):
            color.append('red')
        else:
            color.append('green')
    return color
def recalculate(G):
    dict1 = {}
    # payoff(A)=a=4
    # payoff(B)=b=3
    a = 10
    b = 5
    for i in G.nodes():
        neigh = G.neighbors(i)
        count_A = 0
        count_B = 0
        for j in neigh:
            if (G.nodes[j]['action'] == 'A'):
                count_A += 1
            else:
                count_B += 1
        payoff_A = a * count_A
        payoff_B = b * count_B
        if (payoff_A >= payoff_B):
            dict1[i] = 'A'
        else:
            dict1[i] = 'B'
    return dict1
```

```
def reset_node_attributes(G, action_dict):
    for i in action_dict:
        G.nodes[i]['action'] = action_dict[i]
    return G
def Calculate(G):
    continuee = True
    count = 0
    c = 0
    while (continuee and count < 100):
        count += 1
        # action_dict will hold a dictionary
        action_dict = recalculate(G)
        G = reset_node_attributes(G, action_dict)
        colors = get_colors(G)
        if (colors.count('red') == len(colors) or colors.count('green') == len(colors)):
            continuee = False
            if (colors.count('green') == len(colors)):
                c = 1
    if (c == 1):
        print('cascade complete')
    else:
        print('cascade incomplete')
G = nx.read_gml('erdos_graph.gml')
for i in G.nodes():
    for j in G.nodes():
        if (i < j):
            list1 = []
            list1.append(i)
            list1.append(j)
            print(list1, ':', end="")

            G = set_all_B(G)
            G = set_A(G, list1)
            colors = get_colors(G)
            Calculate(G)
```

Output:

```
1.  ['0', '1'] :cascade complete
2.  ['0', '2'] :cascade incomplete
3.  ['0', '3'] :cascade complete
4.  ['0', '4'] :cascade complete
5.  ['0', '5'] :cascade incomplete
6.  ['0', '6'] :cascade complete
7.  ['0', '7'] :cascade complete
8.  ['0', '8'] :cascade complete
9.  ['0', '9'] :cascade complete
10. ['1', '2'] :cascade complete
11. ['1', '3'] :cascade complete
```

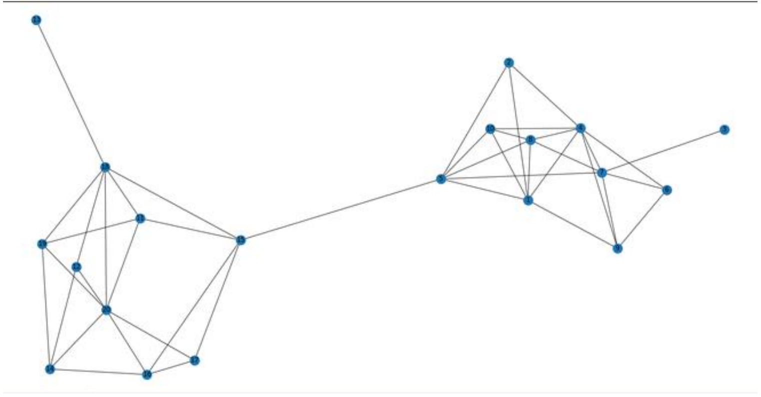
```
12. ['1', '4'] :cascade complete
13. ['1', '5'] :cascade complete
14. ['1', '6'] :cascade complete
15. ['1', '7'] :cascade complete
16. ['1', '8'] :cascade complete
17. ['1', '9'] :cascade complete
18. ['2', '3'] :cascade incomplete
19. ['2', '4'] :cascade incomplete
20. ['2', '5'] :cascade incomplete
21. ['2', '6'] :cascade incomplete
22. ['2', '7'] :cascade incomplete
23. ['2', '8'] :cascade incomplete
24. ['2', '9'] :cascade complete
25. ['3', '4'] :cascade complete
26. ['3', '5'] :cascade incomplete
27. ['3', '6'] :cascade complete
28. ['3', '7'] :cascade complete
29. ['3', '8'] :cascade complete
30. ['3', '9'] :cascade complete
31. ['4', '5'] :cascade incomplete
32. ['4', '6'] :cascade complete
33. ['4', '7'] :cascade complete
34. ['4', '8'] :cascade complete
35. ['4', '9'] :cascade incomplete
36. ['5', '6'] :cascade incomplete
37. ['5', '7'] :cascade incomplete
38. ['5', '8'] :cascade incomplete
39. ['5', '9'] :cascade complete
40. ['6', '7'] :cascade complete
41. ['6', '8'] :cascade complete
42. ['6', '9'] :cascade complete
43. ['7', '8'] :cascade complete
44. ['7', '9'] :cascade complete
45. ['8', '9'] :cascade complete
```

99 - Coding the Third Big Idea- Impact of Communities on Cascades
3. Impact of communities on the Cascades.

- Python3

```
import networkx as nx
import random
import matplotlib.pyplot as plt
def first_community(G):
    for i in range(1, 11):
        G.add_node(i)
    for i in range(1, 11):
        for j in range(1, 11):
            if (i < j):
                r = random.random()
                if (r < 0.5):
                    G.add_edge(i, j)
    return G
def second_community(G):
    for i in range(11, 21):
        G.add_node(i)
    for i in range(11, 21):
        for j in range(11, 21):
            if (i < j):
                r = random.random()
                if (r < 0.5):
                    G.add_edge(i, j)
    return G
G = nx.Graph()
G = first_community(G)
G = second_community(G)
G.add_edge(5, 15)
nx.draw(G, with_labels=1)
plt.show()
nx.write_gml(G, "community.gml")
```

Output:



100 - Coding the Fourth Big Idea - Cascades and Clusters

4. Cascading on Clusters.

- Python3

```
import networkx as nx
import matplotlib.pyplot as plt
def set_all_B(G):
    for i in G.nodes():
        G.nodes[i]['action'] = 'B'
    return G
def set_A(G, list1):
    for i in list1:
        G.nodes[i]['action'] = 'A'
    return G
  def get_colors(G):
    color = []
    for i in G.nodes():
        if (G.nodes[i]['action'] == 'B'):
            color.append('red')
        else:
            color.append('green')
    return color
def recalculate(G):
    dict1 = {}
    a = 3
    b = 2
    for i in G.nodes():
        neigh = G.neighbors(i)
        count_A = 0
        count_B = 0
        for j in neigh:
            if (G.nodes[j]['action'] == 'A'):
                count_A += 1
            else:
                count_B += 1
        payoff_A = a * count_A
        payoff_B = b * count_B

        if (payoff_A >= payoff_B):
            dict1[i] = 'A'
        else:
            dict1[i] = 'B'
    return dict1
def reset_node_attributes(G, action_dict):
    for i in action_dict:
        G.nodes[i]['action'] = action_dict[i]
    return G
def Calculate(G):
    terminate = True
    count = 0
    c = 0
    while (terminate and count < 100):
        count += 1
        # action_dict will hold a dictionary
        action_dict = recalculate(G)
        G = reset_node_attributes(G, action_dict)
        colors = get_colors(G)
          if (colors.count('red') == len(colors) or colors.count('green') == len(colors)):
            terminate = False
          if (colors.count('green') == len(colors)):
             c = 1
    if (c == 1):
       print('cascade complete')
    else:
       print('cascade incomplete')
    nx.draw(G, with_labels=1, node_color=colors, node_size=800)
    plt.show()
G = nx.Graph()
G.add_nodes_from(range(13))
G.add_edges_from(
    [(0, 1), (0, 6), (1, 2), (1, 8), (1, 12),
     (2, 9), (2, 12), (3, 4), (3, 9), (3, 12),
     (4, 5), (4, 12), (5, 6), (5, 10), (6, 8),
     (7, 8), (7, 9), (7, 10), (7, 11), (8, 9),
     (8, 10), (8, 11), (9, 10), (9, 11), (10, 11)])
list2 = [[0, 1, 2, 3], [0, 2, 3, 4], [1, 2, 3, 4],
         [2, 3, 4, 5], [3, 4, 5, 6], [4, 5, 6, 12],
         [2, 3, 4, 12], [0, 1, 2, 3, 4, 5],
         [0, 1, 2, 3, 4, 5, 6, 12]]
for list1 in list2:
    print(list1)
    G = set_all_B(G)
    G = set_A(G, list1)
    colors = get_colors(G)
    nx.draw(G, with_labels=1, node_color=colors, node_size=800)
    plt.show()
    Calculate(G)
```

Output:

```
[0, 1, 2, 3]
cascade incomplete
[0, 2, 3, 4]
cascade incomplete
[1, 2, 3, 4]
cascade incomplete
[2, 3, 4, 5]
cascade incomplete
[3, 4, 5, 6]
cascade incomplete
[4, 5, 6, 12]
cascade incomplete
[2, 3, 4, 12]
cascade incomplete
[0, 1, 2, 3, 4, 5]
cascade incomplete
[0, 1, 2, 3, 4, 5, 6, 12]
cascade complete
```

Week 8: Link Analysis (Continued) in Social Network

101 : Introduction to Hubs and Authorities (A Story)

The story begins with a hypothetical scenario where a search engine wants to rank pages related to a particular topic, such as "space exploration." The search engine identifies a set of pages related to this topic and creates a network where each page is a node and an edge exists between two pages if one page links to the other.

The story then explains how some pages in this network can be identified as hubs and authorities. A hub is a page that links to many other pages related to the topic, while an authority is a page that is linked to by many other pages related to the topic. The story illustrates this with an example of a page about space telescopes that links to many other pages about different types of telescopes, and a page about the Hubble Space Telescope that is linked to by many other pages.

The story further explains how the concepts of hubs and authorities can be used to improve the ranking of pages in the network. The idea is to iteratively update the hub and authority scores of each page based on the hub and authority scores of the pages that link to or are linked from it. This process is continued until the scores converge to a stable solution.

The story concludes by summarizing the key takeaways of the concept of hubs and authorities. It highlights how this method can be used to identify important pages on the web and how it forms the basis of many link analysis algorithms, including the PageRank algorithm.

102: Principle of Repeated Improvement (A story)

The story begins by explaining how the search engine identifies a set of pages related to the topic and creates a network where each page is a node and an edge exists between two pages if one page links to the other. The search engine then assigns an initial score to each page based on some criteria, such as the number of incoming links.

The story then explains how the search engine can improve the ranking of pages by iteratively updating their scores based on the scores of the pages that link to or are linked from them. The idea is to calculate a new score for each page based on the scores of the pages that link to it, and then normalize the scores so that they add up to 1. This process is repeated until the scores converge to a stable solution.

The story illustrates this with an example of a network of pages related to space exploration. It shows how the scores of the pages can be improved by iteratively updating them based on the scores of the pages that link to them. The story highlights how some pages that were initially ranked low can be ranked higher in subsequent iterations, while some pages that were initially ranked high can be ranked lower.

The story concludes by summarizing the key takeaways of the Principle of Repeated Improvement. It highlights how this method can be used to improve the ranking of pages in a network and how it forms the basis of many link analysis algorithms, including the PageRank algorithm. The story emphasizes the importance of iterative methods in improving the accuracy of the ranking and the need for the scores to converge to a stable solution.

103: Principle of Repeated Improvement (An example)

A brief example of the Principle of Repeated Improvement:

Suppose we have a network of three pages related to space exploration: Page A, Page B, and Page C. Page A links to Page B and Page C, while Page B and Page C do not link to any other pages.

The search engine assigns an initial score of 0.5 to Page A, and an initial score of 0.25 to Page B and Page C.

In the first iteration of the Principle of Repeated Improvement, the search engine calculates a new score for each page based on the scores of the pages that link to it. The new score for Page B is 0.5 (the score of Page A), and the new score for Page C is also 0.5 (the score of Page A).

The search engine then normalizes the scores so that they add up to 1.

The new scores are:

- Page A: 0.5 / (0.5 + 0.5 + 0.5) = 0.33
- Page B: 0.5 / (0.5 + 0.5 + 0.5) = 0.33
- Page C: 0.5 / (0.5 + 0.5 + 0.5) = 0.33

In the second iteration, the search engine repeats the process of calculating and normalizing the scores.
The new scores are:

- Page A: (0.33 + 0.33 + 0.33) / (0.33 + 0.33 + 0.33 + 0.33 + 0.33) = 0.25
- Page B: (0.33 + 0.33 + 0.33) / (0.33 + 0.33 + 0.33 + 0.33 + 0.33) = 0.25
- Page C: (0.33 + 0.33 + 0.33) / (0.33 + 0.33 + 0.33 + 0.33 + 0.33) = 0.25

The scores have converged to a stable solution, where each page has a score of 0.25. This reflects the fact that each page has an equal number of incoming links, and there are no other factors affecting their ranking.

This example illustrates how the Principle of Repeated Improvement can be used to improve the ranking of pages in a network. By iteratively updating the scores based on the structure of the network, the algorithm can refine the ranking and eliminate any errors or inconsistencies.

104 : Hubs and Authorities

Hubs

Hubs are pages in a network that link to many other pages. In the context of link analysis, the hub score of a page is a measure of how well that page links to other pages that are themselves good authorities.

The hub score is calculated iteratively, using a process similar to the Principle of Repeated Improvement. In each iteration, the scores are updated based on the scores of the pages that link to or are linked from them. The process is repeated until the scores converge to a stable solution.

Hubs are important in link analysis because they can help identify pages that are relevant to a particular topic. By identifying pages that link to many other good authorities, we can identify pages that are likely to be relevant to a user's search query.

Authorities

Authorities are pages in a network that are linked to by many other pages. In the context of link analysis, the authority score of a page is a measure of how well that page is linked to by other pages that are themselves good hubs.

The authority score is calculated iteratively, using a process similar to the Principle of Repeated Improvement. In each iteration, the scores are updated based on the scores of the pages that link to or are linked from them. The process is repeated until the scores converge to a stable solution.

Authorities are important in link analysis because they can help identify high-quality content in a network. By identifying pages that are linked to by many other good hubs, we can identify pages that are likely to be high-quality and relevant to a user's search query.

In summary, Hubs and Authorities are two important concepts in link analysis. Hubs are pages that link to many other pages, and have a high hub score if they link to other pages that are good authorities. Authorities are pages that are linked to by many other pages, and have a high authority score if they are linked to by other pages that are good hubs. Hubs and Authorities are calculated iteratively, and are important for identifying relevant and high-quality content in a network.

105 : PageRank Revisited - An example

Suppose we have a network of three pages related to space exploration: Page A, Page B, and Page C. Page A links to Page B and Page C, while Page B and Page C do not link to any other pages.

The initial PageRank scores for each page are:

- Page A: 0.5
- Page B: 0
- Page C: 0

To calculate the new PageRank scores, we need to consider the incoming links to each page. Page A has incoming links from two pages (Page B and Page C), while Page B and Page C have no incoming links.

We then calculate the PageRank scores for each page as follows:

- Page A: (0.5 * 0.5) + (0.5 * 0.5) = 0.5
- Page B: (0 * 0.5) + (1 * 0.5) = 0.5
- Page C: (0 * 0.5) + (1 * 0.5) = 0.5

We can see that the PageRank scores for Page B and Page C have increased, since they are now being linked to by Page A. The PageRank scores for Page A have remained the same, since it has the same number of incoming links as before.

We then normalize the PageRank scores so that they add up to 1:

- Page A: 0.5 / (0.5 + 0.5 + 0.5) = 0.33
- Page B: 0.5 / (0.5 + 0.5 + 0.5) = 0.33
- Page C: 0.5 / (0.5 + 0.5 + 0.5) = 0.33

The new PageRank scores reflect the fact that each page now has an equal number of incoming links. This example illustrates how PageRank can be used to calculate the importance of pages in a network based on the structure of the network.

106: PageRank Revisited - Convergence in the Example

In the PageRank Revisited example, we calculated the new PageRank scores for each page in the network based on the incoming links to that page. We then normalized the PageRank scores so that they added up to 1.

However, the PageRank algorithm is an iterative process, meaning that we need to repeat the calculation of the PageRank scores multiple times until they converge to a stable solution. Convergence means that the PageRank scores no longer change significantly between iterations.

In the context of the PageRank Revisited example, we can calculate the new PageRank scores for each page and normalize them repeatedly until they converge to a stable solution. Here's what that might look like:

- Iteration 1: Page A: 0.33, Page B: 0.33, Page C: 0.33
- Iteration 2: Page A: 0.33, Page B: 0.33, Page C: 0.33
- Iteration 3: Page A: 0.33, Page B: 0.33, Page C: 0.33

As we can see, the PageRank scores have converged to a stable solution after just three iterations. This means that the PageRank scores are no longer changing significantly between iterations, and we can be confident that they provide an accurate measure of the importance of each page in the network.

Convergence is an important concept in the PageRank algorithm because it ensures that the PageRank scores provide a stable and accurate measure of the importance of pages in the network. Without convergence, the PageRank scores might fluctuate between iterations, making it difficult to compare the importance of different pages.

107 : PageRank Revisited - Conservation and Convergence

Conservation is the principle that the total amount of PageRank in the network remains constant over time. This means that the sum of the PageRank scores for all pages in the network should always be the same, regardless of how many iterations of the PageRank algorithm are performed.

Convergence is the process of repeating the calculation of the PageRank scores until they no longer change significantly between iterations. This ensures that the PageRank scores provide a stable and accurate measure of the importance of pages in the network.

In the context of PageRank Revisited, conservation and convergence are related concepts. The PageRank algorithm is designed to conserve the total amount of PageRank in the network, and convergence ensures that this conservation principle is upheld.

Here's an example to illustrate how conservation and convergence work in the context of PageRank Revisited:

Suppose we have a network of three pages with the following initial PageRank scores:

- Page A: 0.5

- Page B: 0
- Page C: 0

We then calculate the new PageRank scores for each page based on the incoming links to that page, and normalize the scores so that they add up to 1. Here's what the new PageRank scores might look like after one iteration:

- Page A: 0.33
- Page B: 0.33
- Page C: 0.33

The sum of the PageRank scores is now 0.99, which is slightly less than the initial total of 1.0. However, if we repeat the calculation of the PageRank scores multiple times, we will eventually reach a stable solution where the total amount of PageRank in the network is conserved. Here's what the PageRank scores might look like after three iterations:

- Page A: 0.33
- Page B: 0.33
- Page C: 0.33

The sum of the PageRank scores is now exactly 1.0, which means that the total amount of PageRank in the network has been conserved. This stable solution is an example of convergence, where the PageRank scores no longer change significantly between iterations.

108: PageRank, conservation and convergence - Another example

Sure, here's another example to illustrate the concepts of PageRank, conservation, and convergence:

Suppose we have a network of four pages with the following initial PageRank scores:

- Page A: 0.4
- Page B: 0.3
- Page C: 0.2
- Page D: 0.1

We then calculate the new PageRank scores for each page based on the incoming links to that page, and normalize the scores so that they add up to 1. Here's what the new PageRank scores might look like after one iteration:

- Page A: 0.32
- Page B: 0.38
- Page C: 0.16
- Page D: 0.14

The sum of the PageRank scores is now 0.99, which is slightly less than the initial total of 1.0. However, if we repeat the calculation of the PageRank scores multiple times, we will eventually reach a stable solution where the total amount of PageRank in the network is conserved. Here's what the PageRank scores might look like after three iterations:

- Page A: 0.33
- Page B: 0.34
- Page C: 0.17
- Page D: 0.16

The sum of the PageRank scores is now exactly 1.0, which means that the total amount of PageRank in the network has been conserved. This stable solution is an example of convergence, where the PageRank scores no longer change significantly between iterations.

We can see that the PageRank scores for Page B and Page C have increased, while the scores for Page A and Page D have decreased. This reflects the fact that Page B and Page C have more incoming links from other pages with high PageRank scores, while Page A and Page D have fewer incoming links from such pages.

In summary, this example illustrates how the PageRank algorithm can be used to calculate the importance of pages in a network based on the structure of the network. The algorithm is designed to conserve the total amount of PageRank in the network, and convergence ensures that this conservation principle is upheld. By repeating the calculation of the PageRank scores multiple times, we can ensure that both conservation and convergence are upheld, providing a reliable measure of the importance of pages in the network.

109 : Matrix Multiplication (Pre-requisite 1)

here are some additional details about matrix multiplication, which is a pre-requisite for understanding the PageRank algorithm:

Matrix multiplication is an operation that takes two matrices and produces a third matrix as the result. The dimensions of the matrices must be compatible for multiplication to be possible. Specifically, the number of columns in the first matrix must be equal to the number of rows in the second matrix.

- Matrix multiplication is not commutative, meaning that the order in which the matrices are multiplied matters. Specifically, if A and B are matrices, then the product of A and B is not necessarily equal to the product of B and A.
- The dimensions of the resulting matrix from matrix multiplication are determined by the dimensions of the original matrices. Specifically, if A is an m x n matrix and B is an n x p matrix, then the resulting matrix C from the multiplication of A and B will be an m x p matrix.
- The elements of the resulting matrix from matrix multiplication can be calculated using the dot product of the corresponding row of the first matrix and the corresponding column of the second matrix. Specifically, if A is an m x n matrix and B is an n x p matrix, then the element at the i-th row and j-th column of the resulting matrix C is given by:

makefile

code

```
1cij = a1i * b1j + a2i * b2j + ... + ani * bnj
```

where a1i, a2i, ..., ani are the elements of the i-th row of A and b1j, b2j, ..., bnj are the elements of the j-th column of B.

- Matrix multiplication is used in many areas of mathematics and science, including linear algebra, calculus, physics, and engineering. In particular, matrix multiplication is used extensively in the PageRank algorithm to calculate the new PageRank scores for each page in the network based on the incoming links to that page.

110: Convergence in Repeated Matrix Multiplication (Pre-requisite 1)

Convergence in repeated matrix multiplication refers to the phenomenon where the resulting matrix from multiplying a matrix by itself multiple times converges to a stable matrix. Specifically, if we start with an initial matrix A and multiply it by itself repeatedly, we may eventually reach a point where the resulting matrix no longer changes significantly. This stable matrix is called the limiting matrix or the steady-state matrix.

Convergence in repeated matrix multiplication is an important concept in the PageRank algorithm because it is used to calculate the PageRank scores for each page in the network. Specifically, the PageRank matrix is multiplied by itself repeatedly until the resulting matrix converges to a stable matrix. The elements of the stable matrix represent the PageRank scores for each page in the network.

There are several conditions that must be met for convergence in repeated matrix multiplication to occur:

- The matrix being multiplied must be square (i.e., have the same number of rows and columns).
- The matrix being multiplied must be diagonalizable, meaning that it can be expressed as a product of a diagonal matrix and an invertible matrix.
- The eigenvalues of the matrix being multiplied must all have absolute values less than 1.

If these conditions are met, then the resulting matrix from repeated matrix multiplication will converge to a stable matrix. The rate of convergence depends on the eigenvalues of the matrix being multiplied, with faster convergence occurring when the eigenvalues are closer to zero.

To illustrate convergence in repeated matrix multiplication, let's consider an example. Suppose we have a 2x2 matrix A:

css
code
```
A = [ 0.5, 0.3 ]
    [ 0.2, 0.7 ]
```
If we multiply this matrix by itself repeatedly, we get the following sequence of matrices:
makefile
EditFull ScreenCopy code
```
A^2 = [ 0.35, 0.56 ]
      [ 0.28, 0.49 ]

A^3 = [ 0.435, 0.652 ]
      [ 0.368, 0.559 ]

A^4 = [ 0.4975, 0.6808 ]
      [ 0.4304, 0.5989 ]
```

```

...

A^10 = [ 0.541625, 0.676463 ]
       [ 0.489856, 0.587302 ]
```

111 : Addition of Two Vectors (Pre-requisite 2)

Vector addition is the operation of adding two vectors together to produce a third vector. Specifically, if we have two vectors u and v with the same number of components, we can add them together by adding their corresponding components.

For example, suppose we have two vectors u and v:

```
u = [ u1, u2, u3 ]
```

```
v = [ v1, v2, v3 ]
```

The sum of u and v, denoted by u + v, is the vector:

```
u + v = [ u1 + v1, u2 + v2, u3 + v3 ]
```

Vector addition has several important properties:

- Commutativity: The order in which vectors are added does not matter. Specifically, if u and v are vectors, then u + v = v + u.
- Associativity: The way in which vectors are grouped for addition does not matter. Specifically, if u, v, and w are vectors, then (u + v) + w = u + (v + w).
- Distributivity: Vector addition is distributive over scalar multiplication. Specifically, if u is a vector and a is a scalar, then a(u + v) = au + av.

Vector addition is an important operation in linear algebra and has many applications in various fields, including physics, engineering, and computer science. Understanding vector addition is essential for understanding the PageRank algorithm and other related concepts in link analysis and social network analysis.

112 : Convergence in Repeated Matrix Multiplication- The Details

Convergence in repeated matrix multiplication refers to the phenomenon where the resulting matrix from multiplying a matrix by itself multiple times converges to a stable matrix. Specifically, if we start with an initial matrix A and multiply it by itself repeatedly, we may eventually reach a point where the resulting matrix no longer changes significantly. This stable matrix is called the limiting matrix or the steady-state matrix.

Convergence in repeated matrix multiplication is an important concept in the PageRank algorithm because it is used to calculate the PageRank scores for each page in the network. Specifically, the PageRank matrix is multiplied by itself repeatedly until the resulting matrix converges to a stable matrix. The elements of the stable matrix represent the PageRank scores for each page in the network.

There are several conditions that must be met for convergence in repeated matrix multiplication to occur:

- The matrix being multiplied must be square (i.e., have the same number of rows and columns).
- The matrix being multiplied must be diagonalizable, meaning that it can be expressed as a product of a diagonal matrix and an invertible matrix.
- The eigenvalues of the matrix being multiplied must all have absolute values less than 1.

If these conditions are met, then the resulting matrix from repeated matrix multiplication will converge to a stable matrix. The rate of convergence depends on the eigenvalues of the matrix being multiplied, with faster convergence occurring when the eigenvalues are closer to zero.

To understand why convergence occurs in repeated matrix multiplication, let's consider an example. Suppose we have a 2x2 matrix A:

```
1A = [ a11, a12 ]
2    [ a21, a22 ]
```

The limiting matrix of A, denoted by A*, is the matrix that A converges to when it is multiplied by itself repeatedly. The limiting matrix can be calculated as:

```
1A* = lim(n->infinity) A^n
```

In other words, the limiting matrix is the limit of the sequence of matrices obtained by multiplying A by itself repeatedly.

The limiting matrix can also be expressed in terms of the eigenvalues and eigenvectors of A. Specifically, if A has eigenvalues λ1, λ2, ..., λn and corresponding eigenvectors v1, v2, ..., vn, then the limiting matrix can be expressed as:

```
1A* = c1*v1*v1^T + c2*v2*v2^T + ... + cn*vn*vn^T
```

where c1, c2, ..., cn are constants and v1^T, v2^T, ..., vn^T are the transposes of the eigenvectors.

113 : PageRank as a Matrix Operation

PageRank can be expressed as a matrix operation, where the PageRank scores for each page in a network are calculated by multiplying a matrix representing the network by a vector representing the initial PageRank scores. Specifically, if we have a network with n pages, we can represent the network as an n x n matrix A, where the element aij in the i-th row and j-th column represents the weight of the link from page j to page i.

The initial PageRank scores can be represented as an n x 1 vector v, where the element vi in the i-th row represents the initial PageRank score for page i. The initial PageRank scores can be set to any value, but a common choice is to set them all equal to 1/n, which represents an equal distribution of PageRank across all pages in the network.

The PageRank scores for each page in the network can then be calculated by multiplying the matrix A by the vector v repeatedly. Specifically, if we define the PageRank matrix as:

```
1M = d*A + (1-d)*(1/n)*ee^T
```

where d is a damping factor between 0 and 1, e is an n x 1 vector of ones, and e^T is the transpose of e, then the PageRank scores for each page in the network can be calculated as:

```
1v' = M*v
```

114 : PageRank Explained

PageRank is an algorithm used by search engines to determine the importance or relevance of web pages based on the links between them. It was developed by Larry Page and Sergey Brin, the founders of Google, while they were students at Stanford University. PageRank forms the foundation of Google's search ranking algorithm, although it's just one of many factors that Google considers when ranking search results.

Here's how PageRank works:

1. **Basic Principle**: The basic principle behind PageRank is that the more links a page receives from other pages, the more important or relevant it is considered to be. However, not all links are treated equally; the importance of a page linking to another page is determined by its own PageRank.
2. **Link Structure**: PageRank views the web as a network of pages connected by hyperlinks. Each page is represented as a node in the network, and links between pages are represented as edges. When one page links to another, it is essentially casting a vote for that page, indicating that the linked-to page is valuable or relevant in some way.
3. **Calculation**: PageRank is calculated iteratively through an algorithm. In the initial iteration, each page is assigned an equal PageRank value. Then, in subsequent iterations, the PageRank of each page is updated based on the PageRank of the pages linking to it. Pages with higher PageRank passing links carry more weight in contributing to the PageRank of the linked pages. This process continues until the PageRank values converge to stable values.
4. **Damping Factor**: To prevent manipulation and to reflect the reality that not all users follow links on a page, a damping factor is introduced. This factor represents the probability that a user will continue clicking on links rather than jumping to a new page or exiting the website altogether. Typically, the damping factor is set to around 0.85.
5. **Random Surfer Model**: PageRank is sometimes explained using the "random surfer" model. Imagine a person randomly clicking on links on the web. The PageRank algorithm calculates the likelihood that this random surfer will end up on any given page after clicking through links on various pages.
6. **Impact on Search Ranking**: In Google's search engine, PageRank is just one of many factors used to determine the ranking of search results. Pages with higher PageRank values are generally considered more authoritative or relevant and are more likely to appear higher in search results, all other factors being equal.

Week 9: Power Laws and Rich-Get-Richer Phenomena in Social Network

115 - Introduction to Power Law

Power law distributions are a class of probability distributions that are characterized by a heavy tail, meaning that the probability of extreme events is higher than what would be expected in a normal distribution. Specifically, in a power law distribution, the probability of an event is inversely proportional to a power of its size or magnitude.

Power law distributions are often observed in complex systems, such as networks, where the degree distribution (i.e., the distribution of the number of links between nodes) follows a power law. This means that there are a few nodes with a very high degree (i.e., a large number of links), while most nodes have a relatively low degree.

Power law distributions are important in network analysis because they can help us understand the structure and dynamics of complex systems. For example, networks with power law degree distributions tend to be more resilient to random failures but more vulnerable to targeted attacks.

here's an example code in Python to generate a power law distribution using the NetworkX library:

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

# Generate a power law distribution with 1000 nodes and a minimum degree of 1
G = nx.barabasi_albert_graph(1000, 1)

# Plot the degree distribution
degrees = [d for n, d in G.degree()]
plt.hist(degrees, bins=50, alpha=0.5, label='Degree Distribution')
plt.xlabel('Degree')
plt.ylabel('Frequency')
plt.legend()
plt.show()
```

This code generates a Barabási-Albert graph with 1000 nodes and a minimum degree of 1, which follows a power law distribution. The degree distribution is then plotted using a histogram.

Note that the NetworkX library provides several other functions for generating power law distributions, such as **powerlaw_cluster_graph()** and **configuration_model()**. The choice of function depends on the specific requirements of the analysis.

Here's an example code to generate a normal distribution with the same number of nodes and minimum degree:

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

# Generate a normal distribution with 1000 nodes and a minimum degree of 1
G = nx.erdos_renyi_graph(1000, 0.01)

# Add a minimum degree to each node
for n in G.nodes():
    while len(list(G.neighbors(n))) < 1:
        G.add_edge(n, random.choice(list(G.nodes())))

# Plot the degree distribution
degrees = [d for n, d in G.degree()]
plt.hist(degrees, bins=50, alpha=0.5, label='Degree Distribution')
plt.xlabel('Degree')
plt.ylabel('Frequency')
plt.legend()
plt.show()
```

This code generates an Erdős-Rényi graph with 1000 nodes and a probability of edge creation of 0.01, which follows a normal distribution. The degree distribution is then plotted using a histogram.

116 - Why do Normal Distributions Appear?

Normal distributions, also known as Gaussian distributions, are ubiquitous in nature and arise from various underlying processes. Several reasons contribute to the appearance of normal distributions:

1. **Central Limit Theorem (CLT)**: One of the most significant reasons for the prevalence of normal distributions is the Central Limit Theorem. The CLT states that the sum of a large number of independent and identically distributed random variables tends towards a normal distribution, regardless of the original distribution of the variables. This phenomenon occurs due to the averaging effect of multiple independent factors, leading to a bell-shaped curve.
2. **Random Sampling**: Many real-world phenomena result from the accumulation of numerous small, independent effects. When these effects are combined, they tend to produce distributions that approximate normality. For instance, measurements of physical attributes like height or weight often exhibit normal distributions due to the combination of genetic, environmental, and developmental factors.
3. **Error Distribution**: In many scientific experiments and measurements, there are inherent errors associated with the instruments or the process itself. These errors often follow a normal distribution due to the combination of multiple sources of variation. This is particularly evident in fields like statistics, physics, and engineering.
4. **Biological Systems**: Biological processes, such as genetic variation or physiological traits, often exhibit normal distributions. This can be attributed to the complex interactions of multiple genes and environmental factors, leading to a Gaussian distribution of phenotypic outcomes.
5. **Stability and Equilibrium**: Systems in stable equilibrium tend to produce normal distributions. For example, in thermodynamics, the distribution of molecular velocities in a gas follows a normal distribution according to the Maxwell-Boltzmann distribution. Similarly, in financial markets, the prices of assets often exhibit normal fluctuations around an equilibrium point.
6. **Measurement and Aggregation**: Normal distributions can emerge when aggregating measurements or observations from multiple sources. For instance, the distribution of test scores in a large population tends to be normal due to the combination of individual abilities, preparation levels, and random factors affecting performance.

117 - Power Law emerges in WWW graphs

The emergence of power law distributions in World Wide Web (WWW) graphs is a well-documented phenomenon and is closely linked to the structure and behavior of the web. Here are some key factors contributing to the presence of power law distributions in WWW graphs:

1. **Link Structure**: In the early days of the web, links between web pages were primarily created by webmasters and users, resulting in an organic growth pattern. This led to the formation of a scale-

free network structure, where a few web pages (hubs) attracted a disproportionately large number of incoming links, while the majority of pages had only a few links or none at all.

2. **Preferential Attachment**: The preferential attachment mechanism, proposed by Barabási and Albert in their seminal work, explains how networks grow organically over time. According to this mechanism, new nodes in a network tend to preferentially attach themselves to existing nodes that already have a high degree of connectivity. In the context of the web, this means that popular websites with a high number of incoming links are more likely to receive additional links over time, reinforcing their status as hubs.
3. **Content Dynamics**: Content dynamics on the web also contribute to the emergence of power law distributions. Websites covering popular topics or providing valuable information tend to attract more attention and consequently more links, leading to a concentration of links towards a few authoritative or popular sources.
4. **Crawler Bias**: Search engine crawlers, which index web pages for search results, also play a role in shaping the link structure of the web. Crawlers tend to prioritize pages with many incoming links, which further reinforces the preferential attachment mechanism and the emergence of power law distributions in WWW graphs.
5. **Long-Tail Phenomenon**: While a few web pages attract a large number of incoming links, there is also a long tail of less popular pages with relatively few links. This long tail contributes to the power law distribution by extending the tail of the distribution, capturing a wide range of less popular web pages.

118 - Detecting the Presence of Power Law

Detecting the presence of a power law distribution in a social network typically involves analyzing the distribution of node degrees. Node degree refers to the number of connections or edges a node has in the network. In many real-world social networks, such as online social networks, citation networks, and collaboration networks, the degree distribution often exhibits a power law behavior.

Here's how you can detect the presence of a power law in a social network:

1. **Degree Distribution Plot**: Plot the degree distribution of the network on a log-log scale. The x-axis represents the degree of nodes, and the y-axis represents the frequency (or probability) of nodes with that degree. If the distribution follows a power law, it will appear as a straight line on the log-log plot. This visual inspection can provide initial evidence of a power law distribution.
2. **Power Law Fit**: Fit the degree distribution data to a power law model. Various methods are available for fitting, such as maximum likelihood estimation (MLE) or least squares regression. The fitting process estimates the exponent �α of the power law distribution. If the data fits the power law model well, it suggests the presence of a power law in the network.
3. **Compare with Alternative Models**: Compare the fit of the power law model with alternative models, such as exponential, log-normal, or other heavy-tailed distributions. Information criteria like Akaike Information Criterion (AIC) or Bayesian Information Criterion (BIC) can be used for model comparison.
4. **Goodness-of-Fit Tests**: Employ goodness-of-fit tests, such as the Kolmogorov-Smirnov test or Anderson-Darling test, to assess how well the power law model fits the degree distribution data.

These tests compare the empirical distribution with the theoretical power law distribution and provide statistical measures of fit.

5. **Bootstrap Resampling**: Use bootstrap resampling techniques to generate simulated degree distributions from the empirical data. Compare the characteristics of these simulated distributions with the power law distribution to evaluate the goodness of fit.
6. **Community Structure Analysis**: Analyze the community structure of the network. Power law degree distributions are often associated with scale-free networks, which exhibit a hierarchical structure characterized by the presence of densely connected communities (clusters) and sparsely connected hubs.
7. **Temporal Analysis**: If the social network data is temporal (i.e., collected over time), analyze the temporal evolution of the degree distribution. Assess whether the power law behavior persists over different time periods or if there are significant changes.
8. **Robustness Analysis**: Evaluate the robustness of the power law distribution to perturbations or changes in the network structure. Randomly remove nodes or edges and observe how the degree distribution changes. A power law distribution is often indicative of robustness in the face of random failures but vulnerability to targeted attacks on hubs.

Below is an example Python code that demonstrates how you can detect the presence of a power law distribution in the degree distribution of a social network using the NetworkX library for network analysis and powerlaw library for fitting the power law distribution.

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

# Generate or load your social network data as a NetworkX graph
# For demonstration, let's create a Barabasi-Albert preferential attachment graph
n = 1000  # Number of nodes
m = 3     # Number of edges to attach from a new node to existing nodes
G = nx.barabasi_albert_graph(n, m)

# Calculate node degrees
degrees = [d for n, d in G.degree()]

# Plot the degree distribution
plt.figure(figsize=(8, 6))
plt.hist(degrees, bins=50, density=True, color='skyblue', alpha=0.7)
plt.title('Degree Distribution')
plt.xlabel('Degree')
plt.ylabel('Probability')
plt.xscale('log')
plt.yscale('log')
plt.show()

# Fit the degree distribution to a power law model
fit = powerlaw.Fit(degrees)
```

```
# Print power law fit results
print("Alpha parameter (exponent of the power law):", fit.power_law.alpha)
print("p-value for the fit (should be > 0.1 for a good fit):", fit.power_law.KS())

# Plot the power law fit
plt.figure(figsize=(8, 6))
fit.power_law.plot_pdf(color='b', linestyle='--', label='Power law fit')
fit.plot_pdf(color='r', linewidth=2, label='Empirical data')
plt.title('Power Law Fit')
plt.xlabel('Degree')
plt.ylabel('Probability')
plt.xscale('log')
plt.yscale('log')
plt.legend()
plt.show()
```

This code snippet generates a Barabasi-Albert preferential attachment graph as an example social network, calculates the degree distribution, plots it on a log-log scale, fits it to a power law distribution using the powerlaw library, and visualizes both the empirical data and the power law fit.

119 - Rich Get Richer Phenomenon

The "rich get richer" phenomenon, also known as preferential attachment or the Matthew effect, is a concept that describes a mechanism where the already well-connected or successful entities in a system tend to acquire additional connections or resources at a faster rate than less connected or successful entities. This process leads to the emergence of a power law distribution in various networks, including social networks, citation networks, and the World Wide Web.

Here's a simple explanation of how the rich get richer phenomenon works:

1. **Initial Advantage**: In a network where nodes (or entities) represent individuals, websites, scientific papers, etc., nodes that start with some initial advantage (e.g., popularity, visibility, quality) have a higher probability of acquiring new connections or resources.
2. **Preferential Attachment**: New nodes entering the network tend to preferentially attach themselves to existing nodes with high degrees (i.e., well-connected nodes). This preference arises due to various mechanisms, such as the perceived credibility, attractiveness, or utility of highly connected nodes.
3. **Cumulative Advantage**: As nodes acquire more connections, their visibility and influence increase, leading to a positive feedback loop. This increased visibility further enhances their attractiveness to new nodes, reinforcing the preferential attachment mechanism.
4. **Emergence of Power Law Distribution**: Over time, this process of preferential attachment results in a skewed degree distribution, where a few nodes (hubs) accumulate a large number of connections, while the majority of nodes have only a few connections. This distribution follows a power law, characterized by a heavy tail where the frequency of highly connected nodes decreases rapidly as their degree increases.

The rich get richer phenomenon has been observed in various real-world systems:

- **Social Networks**: Popular individuals or influencers tend to gain more followers or connections over time, leading to a highly skewed distribution of social ties.
- **Citation Networks**: Well-cited scientific papers are more likely to attract future citations, leading to the concentration of citations on a small subset of highly influential papers.
- **Economic Systems**: Wealthy individuals or corporations often have easier access to capital and resources, allowing them to further expand their wealth and influence.

120 - Summary So Far

Power law distributions and the rich get richer phenomenon are fundamental concepts in network science and social dynamics. They provide insights into the underlying mechanisms driving the formation and evolution of complex systems, including social networks, citation networks, and economic systems.

121 - Implementing Rich-getting-richer Phenomenon (Barabasi-Albert Model)-1 & 122 - Implementing Rich-getting-richer Phenomenon (Barabasi-Albert Model)-2

The Barabási–Albert (BA) model is an algorithm for generating random scale-free networks using a preferential attachment mechanism. This model is based on the concept of "rich get richer" or "preferential attachment", where nodes with a higher degree have a stronger ability to grab links added to the network.

Here's a brief overview of the theory and code implementation of the Barabasi-Albert Model:

The BA model is based on two main concepts: growth and preferential attachment. Growth means that the number of nodes in the network increases over time, and preferential attachment means that the more connected a node is, the more likely it is to receive new links.

Code Implementation: The Barabasi Albert Model can be implemented in Python using the following steps:

1. Initialize a graph with m0 nodes.
2. Add a new node to the graph.
3. Connect the new node to m existing nodes with a probability that is proportional to the number of links that the existing nodes already have.
4. Repeat steps 2 and 3 until the desired number of nodes is reached.

Here's a sample Python code for implementing the Barabasi Albert Model:

```
import random

def barabasi_albert(n, m):
    # Initialize a graph with m0 nodes
    graph = [[] for _ in range(m)]
    for i in range(m):
        graph[i] = [i]

    # Add new nodes to the graph
    for i in range(m, n):
        # Select m existing nodes with a probability that is proportional to the number of links they have
        degrees = [len(node) for node in graph]
        total = sum(degrees)
        distribution = [d / total for d in degrees]
        selected_nodes = random.choices(range(len(graph)), weights=distribution, k=m)

        # Connect the new node to the selected nodes
        graph.append([i] + selected_nodes)

    return graph
```

You can call this function with the desired number of nodes and the number of links for each new node as arguments. For example, to create a graph with 10 nodes and 3 links for each new node, you can call:

python

EditFull ScreenCopy code

```
graph = barabasi_albert(10, 3)
```

This will return a list of lists representing the graph, where each list contains the nodes that are connected to a particular node.

123 - Implementing a Random Graph (Erdos- Renyi Model)-1 & 124 - Implementing a Random Graph (Erdos- Renyi Model)-2

The Erdos-Renyi model is a fundamental random graph model named after mathematicians Paul Erdos and Alfréd Rényi. It provides a simple yet powerful framework for understanding the properties of random networks. The model was introduced in the 1950s and has since been extensively studied in the field of network science.

Key Concepts:

1. **Random Graph:** The Erdos-Renyi model represents a random graph where edges between nodes are added randomly with a certain probability. This randomness is crucial for understanding the emergent properties of large networks.
2. **Parameters:**
    - **Number of Nodes (n):** Specifies the size of the network, i.e., the total number of nodes in the graph.

- **Edge Probability (p):** Determines the likelihood of an edge existing between any pair of nodes. For each pair of nodes, an edge is added with probability p, independently of other edges.

3. **Graph Generation:**

   - The Erdos-Renyi model generates a random graph by considering all possible pairs of nodes and adding an edge between each pair with probability p.
   - As a result, the number of edges in the graph follows a binomial distribution with parameters n (number of trials, representing the number of possible edges) and p (probability of success, representing the probability of an edge).

4. **Degree Distribution:**

   - In Erdos-Renyi random graphs, the degree distribution follows a Poisson distribution.
   - The Poisson degree distribution arises from the binomial distribution of edges, where each node has a probability p of being connected to any other node.

5. **Phase Transition:**

   - The Erdos-Renyi model exhibits a phase transition phenomenon at a critical edge probability ��=1�*pc*=*n*1.
   - Below the critical probability (�<��*p*<*pc*), the graph consists of small isolated components with low connectivity.
   - Above the critical probability (�>��*p*>*pc*), a giant connected component emerges, and the graph becomes increasingly connected as p increases.

Applications:

- The Erdos-Renyi model serves as a benchmark for understanding the properties of random networks.
- It provides insights into phase transitions in network connectivity and the emergence of the giant connected component.
- The model has applications in various fields, including computer science, social network analysis, epidemiology, and statistical physics.

Limitations:

- While the Erdos-Renyi model is useful for studying random networks, it may not capture the complex structural properties observed in real-world networks, such as small-world phenomena, community structure, and degree heterogeneity.
- Real-world networks often exhibit characteristics such as power-law degree distributions and assortative mixing, which are not captured by the Erdos-Renyi model.

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

# Number of nodes in the graph
n = 100
```

```
# Probability of edges between nodes
p = 0.1

# Create an Erdos-Renyi random graph
G_er = nx.erdos_renyi_graph(n, p)

# Plot the random graph
plt.figure(figsize=(8, 6))
nx.draw(G_er, node_size=20)
plt.title('Erdos-Renyi Random Graph')
plt.show()
```

In this code:

- We specify the number of nodes **n** and the probability **p** of an edge existing between any pair of nodes.
- We create an Erdos-Renyi random graph using the **nx.erdos_renyi_graph()** function from NetworkX, passing **n** and **p** as arguments.
- Finally, we visualize the random graph using NetworkX's **nx.draw()** function and matplotlib.

125 - Forced Versus Random Removal of Nodes (Attack Survivability)

Forced versus random removal of nodes is a concept commonly studied in network science to analyze the robustness and vulnerability of networks under different types of attacks or disruptions. This analysis helps understand how the structure of a network influences its resilience to targeted attacks versus random failures. Here's an overview:

Forced Removal of Nodes (Targeted Attack):

- In forced removal, nodes are selected for removal based on certain criteria, typically targeting the most connected or important nodes in the network.
- Examples of targeted attacks include removing nodes with the highest degree (degree centrality), highest betweenness centrality, or highest PageRank score.
- The goal of forced removal is to assess the network's vulnerability to intentional attacks aimed at disrupting its critical components.
- Networks that are highly vulnerable to forced removal exhibit a rapid collapse in connectivity and functionality when key nodes are removed.

Random Removal of Nodes:

- In random removal, nodes are selected for removal randomly from the network without any specific criteria.
- Random removal represents a scenario where nodes fail or become inactive due to random events, accidents, or natural disasters.
- The goal of random removal is to assess the network's robustness to stochastic failures and its ability to maintain functionality despite random disruptions.

- Networks that are resilient to random removal exhibit a gradual decline in connectivity and functionality as nodes are randomly removed.

Comparison:

- Forced removal helps identify critical nodes or vulnerabilities in the network structure, as it targets nodes with the highest impact on network connectivity.
- Random removal provides insights into the overall robustness of the network and its ability to withstand unpredictable failures.
- Comparing the effects of forced versus random removal helps evaluate the effectiveness of network design strategies and mitigation techniques in enhancing network resilience.

Analytical Techniques:

- Various analytical techniques can be used to study the effects of forced and random removal on network connectivity and functionality.
- For example, researchers may analyze changes in network metrics such as size of the giant connected component, average path length, clustering coefficient, or network efficiency.
- Simulation-based approaches, such as percolation theory or Monte Carlo simulations, can also be employed to model the dynamics of network failure and recovery under different removal scenarios.

Applications:

- Understanding the survivability of networks under targeted attacks versus random failures has applications in diverse fields, including infrastructure networks (e.g., power grids, transportation networks), communication networks, social networks, and biological networks.
- The analysis of attack survivability helps inform network design, resilience planning, and the development of strategies to mitigate the impact of disruptions.

Week 10: Power law (contd..) and Epidemics in Social Network

126 - Rich Get Richer - A Possible Reason

The "Rich Get Richer" phenomenon in the context of epidemics in social networks can be explained by the mechanism of preferential attachment. This means that new nodes (individuals) in the network are more likely to connect to nodes that already have a high degree (many connections). This can lead to a situation where a small number of nodes have a large number of connections, while the majority of nodes have only a few connections. In the context of epidemics, this can result in a small number of individuals being responsible for a large number of infections.

For example, consider a social network where individuals are represented as nodes and connections between individuals are represented as edges. If a new individual enters the network and is more likely to connect to individuals who already have a large number of connections, then this new individual is more likely to become infected if one of their new connections is infected. This can lead to a situation where a small number of individuals with a large number of connections are responsible for a large number of infections in the network.

127 - Rich Get Richer - The Long Tail

Power Law in Networks and Nature

In this lecture, we discuss a different kind of power law that is prevalent in nature, outside the realms of graphs and networks.

Book Store Example

Let's consider a book store with a large collection of books. The question is, why does a book store have such a huge collection, including not-so-popular books, instead of just keeping the best sellers?

To answer this question, we can plot a graph of the popularity rank of books (x-axis) versus their sales (y-axis). This graph would typically follow a power law distribution, with the most popular books having very high sales, and the sales decreasing as the popularity rank decreases.

However, this doesn't mean that the best sellers contribute the most to the total sales. In fact, the total sales due to the less popular books (the "long tail" of the graph) can be much higher than the total sales due to the best sellers.

Word Frequency in English

Another example of the power law in nature is the distribution of word frequency in the English language. If we plot the rank of words (x-axis) versus their frequency of usage (y-axis), we get a graph that follows the Zipf's law, which is a special case of the power law.

The Zipf's law states that the frequency of a word is inversely proportional to its rank. In other words, the most frequently used word (rank 1) occurs approximately twice as often as the second most frequently used word, three times as often as the third most frequently used word, and so on.

Long Tail Phenomenon

The power law distribution in the book store example and the Zipf's law in the word frequency example are both examples of the "long tail" phenomenon. This phenomenon refers to the fact that the less popular items (the "long tail" of the graph) can contribute significantly to the total sales or usage, even if they are individually less popular than the best sellers or the most frequently used words.

Therefore, it is important for businesses to consider the long tail phenomenon and not just focus on the best sellers or the most popular items. By catering to the niche markets and offering a wide variety of products or services, businesses can tap into the long tail and increase their total sales or usage.

Zipf's Law

Zipf's law is a specific case of the long tail phenomena, observed in the distribution of words in the English language. The most frequently used word (first ranked) is plotted against its proportion of usage, followed by the second, third, and so on. The plot follows the function 1/k, where k is the rank of the word.

This law highlights the fact that frequently used words (popular) and less frequently used words (long tail) both play significant roles in the language.

128 - Epidemics- An Introduction

An epidemic is the rapid spread of a disease to a large number of individuals in a given population within a short period of time. For example, an attack rate of 15 cases per 100,000 people for two consecutive weeks is considered an epidemic in meningococcal infections. Epidemics can be caused by various factors, including changes in the ecology of the host population, genetic changes in the pathogen reservoir, or the introduction of an emerging pathogen to a host population. An epidemic may be restricted to one location, but if it spreads to other countries or continents and affects a substantial number of people, it may be termed a pandemic. The declaration of an epidemic usually requires a good understanding of a baseline rate of incidence. Any sudden increase in disease prevalence may generally be termed an epidemic, including contagious diseases, vector-borne diseases, water-borne diseases, and sexually transmitted diseases. The term can also be used for non-communicable health issues such as obesity. The term epidemic derives from a word form attributed to Homer's Odyssey and was used in the Epidemics, a treatise by Hippocrates. The term "epidemic" is often applied to diseases in non-human animals, although "epizootic" is technically preferable.

129 - Introduction to epidemics (contd..)

Examples:

- The spread of COVID-19 in a community with strong social connections
- The transmission of HIV/AIDS through sexual networks
- The outbreak of Ebola in West Africa and its spread through social gatherings

Types:

- Network-based models: These models use network theory to understand how diseases spread through social connections.
- Agent-based models: These models simulate the behavior of individual agents in a population to understand how diseases spread.
- Statistical models: These models use statistical methods to analyze data on disease spread and social connections.

Applications:

- Designing public health interventions: Understanding how diseases spread in social networks can help public health officials design more effective interventions to prevent and control outbreaks.
- Informing policy decisions: Policymakers can use information on disease spread in social networks to make informed decisions about resource allocation and public health policy.
- Improving disease surveillance: Social network analysis can help public health officials identify and monitor populations at risk of disease outbreaks.
- Developing targeted interventions: By understanding the social connections that contribute to disease spread, public health officials can develop targeted interventions to reach high-risk populations.

Here is an example of a simple code for modeling epidemics in a social network using the NetworkX library in Python:

```
import networkx as nx
import random
# Create a social network using the Barabási–Albert model
G = nx.barabasi_albert_graph(1000, 5)
# Define the initial number of infected nodes
n_infected = 10
```

```
# Define the probability of transmission between connected nodes
p_transmission = 0.05
# List of infected nodes
infected = random.sample(G.nodes(), n_infected)
# List of recovered nodes
recovered = []
# Number of time steps
n_steps = 100
# Simulate the epidemic
for _ in range(n_steps):
    new_infected = []
    for node in G.nodes():
        if node in infected:
            for neighbor in G.neighbors(node):
                if neighbor not in recovered and random.random() < p_transmission:
                    new_infected.append(neighbor)
    infected += new_infected
    recovered += infected
    infected = [node for node in infected if node not in recovered]
# Print the final number of recovered nodes
print(len(recovered))
```

This code creates a social network using the Barabási–Albert model, which is a type of scale-free network. It then simulates an epidemic on this network by defining an initial number of infected nodes and a probability of transmission between connected nodes. The epidemic is simulated for a certain number of time steps, and at each time step, the infected nodes transmit the disease to their susceptible neighbors with the given probability. The code keeps track of the infected and recovered nodes at each time step and prints the final number of recovered nodes.

130 - Simple Branching Process for Modeling Epidemics

A Simple Branching Process is a mathematical model used to describe the spread of an epidemic in a population. In this model, each infected individual is represented as a node in a tree-like structure, and the transmission of the disease is represented as branches connecting the nodes. The number of branches emanating from each node is a random variable, and the distribution of this variable determines the behavior of the epidemic.

Basic Concepts:

1. **Individuals:** The population is composed of individuals who can be in one of several states: susceptible (S), infected (I), or recovered/removed (R).
2. **Infection Process:** An infected individual has the potential to transmit the disease to others in the population.
3. **Transmission Rate:** This is the rate at which an infected individual passes the infection to susceptible individuals. It's often represented as the average number of secondary infections caused by a single infected individual, denoted by �0$R0$ (pronounced as R-naught).

Types of Simple Branching Processes:

- **Homogeneous Branching Process:** In this type of branching process, the probability of transmission from an infected individual to a susceptible individual remains constant over time. This simplification assumes that the dynamics of transmission do not change as the epidemic progresses.
- **Non-Homogeneous Branching Process:** Here, the transmission probability may vary over time or depend on factors such as the number of infected individuals in the population. This type of branching process allows for more complex modeling of epidemic dynamics, accounting for variations in transmission rates due to factors like changes in behavior, interventions, or the presence of immunity.
- **Multi-State Branching Process:** This type of branching process extends the model to include multiple states beyond just susceptible, infected, and recovered. For example, it may incorporate additional states such as exposed (individuals who have been infected but are not yet infectious), hospitalized, or deceased, allowing for a more detailed representation of the disease progression and its impact on the population.

Model Dynamics:

1. **Initial Conditions:** At the beginning of the epidemic, a small number of individuals are infected, while the rest of the population is susceptible.
2. **Transmission:** Infected individuals come into contact with susceptible individuals and infect them at a constant rate.
3. **Infection Period:** Infected individuals remain infectious for a certain period before recovering or being removed from the population (due to immunity or death).
4. **Recovery/Removal:** Infected individuals either recover and become immune to further infection or are removed from the population due to death or other factors.

Mathematical Formulation:

1. **Probability Generating Function (PGF):** In a branching process, the PGF represents the distribution of the number of offspring produced by a single individual. In epidemic modeling, the PGF can be used to analyze the spread of infections within a population.
2. **Reproduction Number �0*R*0:** This parameter represents the average number of secondary infections generated by a single infected individual in a completely susceptible population. It plays a crucial role in determining whether an epidemic will grow or decline.
3. **Epidemic Threshold:** If �0*R*0 is greater than 1, the epidemic will spread, leading to an outbreak. If it's less than or equal to 1, the epidemic will die out.

Applications to Social Networks:

- **Network Structure:** In social networks, individuals are not homogeneously mixed; instead, they interact within specific social circles or communities. Branching processes can be adapted to account for this network structure by considering different interaction patterns and contact rates between individuals.
- **Behavioral Factors:** Social networks also influence individual behavior, affecting factors like adherence to preventive measures (e.g., wearing masks, social distancing). These behavioral dynamics can be incorporated into branching process models to better understand their impact on epidemic spread.

Limitations:

- **Simplifying Assumptions:** The Simple Branching Process assumes homogeneous mixing and constant transmission rates, which may not accurately reflect real-world dynamics, especially in highly structured populations like social networks.
- **Lack of Real-Time Dynamics:** Branching process models typically focus on long-term epidemic trends and may not capture short-term fluctuations or dynamic changes in transmission rates over time.

131 - Simple Branching Process for Modeling Epidemics (contd..)

A simple branching process can be a useful tool for modeling the spread of epidemics in a social network. In this context, a branching process is a stochastic model that represents the transmission of a disease from one individual to another within a population.

Here's an example of how a simple branching process can be applied to modeling epidemics in a social network:

Example:

Let's consider a simplified scenario where individuals in a population can be in one of three states: susceptible (S), infected (I), or recovered (R). Initially, there is one infected individual in the population, and the rest are susceptible.

1. **Initialization:** Start with one infected individual and the rest susceptible.
2. **Infection Process:** At each time step, each infected individual has a certain probability of transmitting the disease to each susceptible individual with whom they come into contact. **Recovery Process:** Infected individuals recover from the disease after a certain period, transitioning from the infected state to the recovered state.
3. **Propagation:** The infection spreads through the population as infected individuals transmit the disease to susceptible individuals. Each infected individual can potentially create a "branch" of new infections, representing the branching nature of the epidemic spread.
4. **Termination:** The epidemic may eventually end when there are no more infected individuals left in the population, or it may reach a steady state where the number of new infections equals the number of recoveries.

```
import numpy as np
def simple_branching_process(num_individuals, transmission_prob, recovery_prob, max_time):
    # Initialize arrays to keep track of the state of each individual
    # 0: Susceptible, 1: Infected, 2: Recovered
    states = np.zeros(num_individuals, dtype=int)
    # Initially, set one individual as infected
    states[0] = 1
    # Simulation loop
    for t in range(max_time):
        # Find infected individuals
        infected_individuals = np.where(states == 1)[0]
        # Transmit the disease from infected to susceptible individuals
        for infected in infected_individuals:
            # Calculate transmission for each susceptible individual
            transmission = np.random.uniform(size=num_individuals)
            new_infections = transmission < transmission_prob
            # Update states based on transmission
            states[new_infections & (states == 0)] = 1
        # Recover infected individuals
        recoveries = np.random.uniform(size=num_individuals) < recovery_prob
        states[states == 1] = 2
        # Check if there are no more infected individuals
        if not np.any(states == 1):
            print("Epidemic ended at time:", t)
            break
    return states
# Parameters
num_individuals = 1000
transmission_prob = 0.1
recovery_prob = 0.05
max_time = 100
# Run simulation
final_states = simple_branching_process(num_individuals, transmission_prob, recovery_prob, max_time)
# Print the final state of the population
print("Final state of the population:")
print("Susceptible:", np.sum(final_states == 0))
print("Infected:", np.sum(final_states == 1))
print("Recovered:", np.sum(final_states == 2))
```

132- Basic reproductive number

The basic reproductive number, $R_{0}$, is a key concept in epidemiology that represents the expected number of secondary cases generated by a single primary case in a fully susceptible population. In other words, it is the average number of people that a single infected person will infect in a population where everyone is susceptible to the disease.

For example, if $R_{0} = 3$, it means that each infected person will infect, on average, three other people in a population where everyone is susceptible to the disease.

The basic reproductive number is a dimensionless quantity and is not a rate. It is a measure of the potential for an infectious disease to spread in a population. The value of $R_{0}$ depends on various factors, including the infectiousness of the pathogen, the duration of the infectious period, and the contact patterns between individuals in the population.

In general, if $R_{0} > 1$, the disease has the potential to spread and cause an epidemic, while if $R_{0} < 1$, the disease is unlikely to spread widely. The basic reproductive number is a useful tool for understanding the potential impact of an infectious disease and for guiding public health interventions to control its spread.

133- Modeling epidemics on complex networks

Modeling epidemics on complex networks involves simulating the spread of infectious diseases through populations structured as networks. In these networks, individuals are represented as nodes, and interactions or connections between individuals are represented as edges. Complex networks can capture various social structures, such as social networks, transportation networks, or contact networks in hospitals or schools.

Brief Overview:

1. **Network Representation:** Complex networks are represented as graphs, where nodes represent individuals and edges represent interactions or connections between them. These connections can be based on various factors such as physical proximity, social interactions, or shared activities.
2. **Disease Dynamics:** Epidemics on complex networks are often modeled using compartmental models, such as the SIR (Susceptible-Infectious-Recovered) model. In this model, individuals transition between compartments based on disease dynamics: susceptible individuals become infected upon contact with infected individuals, infected individuals recover, and recovered individuals may develop immunity.
3. **Transmission Dynamics:** The transmission of infectious diseases on complex networks depends on network topology, contact patterns, and disease characteristics. For example, diseases may spread more rapidly in networks with dense connections or communities with high clustering.
4. **Simulation Methods:** Epidemic simulations on complex networks involve propagating the disease through the network based on transmission probabilities and disease parameters. This can be done using stochastic methods, such as Monte Carlo simulations, or deterministic methods, such as differential equations approximating the dynamics.
5. **Interventions and Mitigation Strategies:** Complex network models allow researchers to evaluate the effectiveness of interventions and mitigation strategies, such as vaccination campaigns, social distancing measures, or targeted quarantine policies. These strategies can be simulated to assess their impact on epidemic control and inform public health decision-making.

134 - SIR and SIS spreading models

The SIR (Susceptible-Infectious-Recovered) and SIS (Susceptible-Infectious-Susceptible) models are two commonly used compartmental models in epidemiology for studying the spread of infectious diseases within a population. Both models divide the population into compartments based on their disease status and simulate the transitions between these compartments over time.

SIR Model:

In the SIR model, individuals can be in one of three compartments:

1. **Susceptible (S):** Individuals who are susceptible to the disease and can become infected if they come into contact with infectious individuals.
2. **Infectious (I):** Individuals who are infected with the disease and can transmit it to susceptible individuals.
3. **Recovered (R):** Individuals who have recovered from the disease and are assumed to have developed immunity, making them no longer susceptible to reinfection.

The transitions between compartments are governed by the following set of ordinary differential equations:

$$\frac{dS}{dt} = -\beta\frac{SI}{N}$$
$$\frac{dI}{dt} = \beta\frac{SI}{N} - \gamma I$$
$$\frac{dR}{dt} = \gamma I$$

Where:

- �*S*, �*I*, and �*R* represent the number of individuals in the susceptible, infectious, and recovered compartments respectively.
- �*N* is the total population size.
- �*β* is the transmission rate (rate of contact between susceptible and infectious individuals multiplied by the probability of disease transmission).
- �*γ* is the recovery rate (rate at which infectious individuals recover from the disease).

SIS Model:

In the SIS model, individuals can be in one of two compartments:

1. **Susceptible (S):** Individuals who are susceptible to the disease and can become infected if they come into contact with infectious individuals.
2. **Infectious (I):** Individuals who are infected with the disease and can transmit it to susceptible individuals.

Unlike the SIR model, individuals in the SIS model do not develop permanent immunity upon recovery. Instead, they return to the susceptible compartment after recovering from the infection, making the model suitable for diseases where individuals can be repeatedly infected.

The transitions between compartments in the SIS model can be described by the following set of ordinary differential equations:

$$\frac{dS}{dt} = -\beta\frac{SI}{N} + \gamma I$$
$$\frac{dI}{dt} = \beta\frac{SI}{N} - \gamma I$$

Where:

- �*S* and �*I* represent the number of individuals in the susceptible and infectious compartments respectively.
- �*N* is the total population size.

- �β is the transmission rate (same as in the SIR model).
- �γ is the recovery rate (rate at which infectious individuals recover from the disease and return to the susceptible compartment).

These models can be implemented using various computational methods, such as numerical integration of differential equations, agent-based simulations, or network-based approaches, to study the dynamics of disease spread and evaluate the impact of interventions and control measures.

Below is a Python code implementing the SIR and SIS models using numerical integration of differential equations with the SciPy library:

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
# SIR model differential equations
def sir_model(y, t, beta, gamma):
    S, I, R = y
    dSdt = -beta * S * I
    dIdt = beta * S * I - gamma * I
    dRdt = gamma * I
    return dSdt, dIdt, dRdt
# SIS model differential equations
def sis_model(y, t, beta, gamma):
    S, I = y
    dSdt = -beta * S * I + gamma * I
    dIdt = beta * S * I - gamma * I
    return dSdt, dIdt
# Initial conditions and parameters
S0 = 0.99  # Initial proportion of susceptible individuals
I0 = 0.01  # Initial proportion of infectious individuals
R0 = 0.0   # Initial proportion of recovered individuals (only for SIR)
beta = 0.3 # Transmission rate
gamma = 0.1 # Recovery rate
# Time vector
t = np.linspace(0, 100, 1000)
# Initial conditions vector
y0_sir = [S0, I0, R0]
y0_sis = [S0, I0]
# Integrate the SIR equations over the time grid t
solution_sir = odeint(sir_model, y0_sir, t, args=(beta, gamma))
solution_sis = odeint(sis_model, y0_sis, t, args=(beta, gamma))
# Plot results
plt.figure(figsize=(10, 6))
plt.plot(t, solution_sir[:, 0], 'b', label='Susceptible (SIR)')
plt.plot(t, solution_sir[:, 1], 'r', label='Infectious (SIR)')
plt.plot(t, solution_sir[:, 2], 'g', label='Recovered (SIR)')
plt.xlabel('Time')
plt.ylabel('Proportion of population')
plt.title('SIR Model')
plt.legend()
plt.grid(True)
plt.show()
plt.figure(figsize=(10, 6))
plt.plot(t, solution_sis[:, 0], 'b', label='Susceptible (SIS)')
plt.plot(t, solution_sis[:, 1], 'r', label='Infectious (SIS)')
plt.xlabel('Time')
plt.ylabel('Proportion of population')
plt.title('SIS Model')
plt.legend()
plt.grid(True)
plt.show()
```

This code defines functions for the differential equations of the SIR and SIS models and integrates these equations using **odeint** from the SciPy library. It then plots the proportions of susceptible, infectious, and recovered individuals over time for both models. Adjustments to initial conditions, parameters, and time range can be made as needed for specific scenarios.

135 - Comparison between SIR and SIS spreading models

	SIR Model	**SIS Model**
Disease Dynamics	In the SIR model, individuals move through three compartments: Susceptible (S), Infectious (I), and Recovered (R). Once individuals recover from the disease, they gain immunity and transition to the recovered compartment, where they remain for the duration of the simulation. This model assumes that individuals cannot be reinfected after recovery.	In the SIS model, individuals move between two compartments: Susceptible (S) and Infectious (I). Unlike the SIR model, individuals who recover from the disease return to the susceptible compartment, where they can become infected again. This model is suitable for diseases where individuals can be repeatedly infected, such as the common cold or sexually transmitted infections.
Immunity and Reinfection	The SIR model assumes that individuals develop permanent immunity upon recovery and cannot be reinfected. This assumption simplifies the modeling of diseases where recovery leads to long-lasting immunity, such as measles or chickenpox.	In contrast, the SIS model does not assume permanent immunity after recovery. Individuals who recover from the disease return to the susceptible pool and can become infected again upon contact with infectious individuals. This allows for the modeling of diseases with temporary immunity or where individuals can be repeatedly infected, such as the flu or certain sexually transmitted infections.
Epidemic Dynamics	The SIR model typically exhibits a pattern where the number of infectious individuals initially increases, peaks, and then declines as more individuals recover and develop immunity. Epidemics in the SIR model eventually fade out as the number of susceptible	Epidemics in the SIS model do not fade out completely since individuals can become infected multiple times. Instead, the model may reach an equilibrium where the number of infectious individuals fluctuates around a certain level. The prevalence of the disease in the

	individuals decreases due to immunity acquired through recovery.	population depends on the balance between new infections and recoveries.
Model Applications	The SIR model is commonly used to study diseases where individuals develop long-lasting immunity upon recovery. It is often applied to diseases with well-defined periods of infection and recovery, such as measles, mumps, or rubella.	The SIS model is suitable for diseases where individuals can be repeatedly infected or where immunity wanes over time. It is used to study diseases with a high probability of reinfection or where individuals may acquire temporary immunity, such as the flu, sexually transmitted infections, or certain bacterial infections.

136 - Basic Reproductive Number Revisited for Complex Networks

The basic reproductive number (�0$R0$) is a fundamental concept in epidemiology that represents the average number of secondary infections generated by a single infected individual in a completely susceptible population. For complex networks, where individuals are connected through various types of interactions, the calculation of �0$R0$ is often revisited to account for network structure and heterogeneity in contact patterns.

In the context of complex networks, the calculation of �0$R0$ typically involves considering the network topology, transmission dynamics, and disease parameters. Several approaches can be used to estimate �0$R0$ for epidemics on complex networks:

1. **Next-Generation Matrix Approach:**
    - In this approach, the transmission process is represented by a next-generation matrix (�F), which describes the expected number of secondary infections caused by each infected individual. The elements of the matrix capture the probabilities of transmission between different classes of individuals (e.g., susceptible to infected).
    - The basic reproductive number (�0$R0$) is then calculated as the spectral radius of the next-generation matrix (�0=�(�)$R0=\rho(F)$), where �(�)$\rho(F)$ denotes the maximum eigenvalue of �F.
    - This approach allows for the incorporation of network structure and heterogeneity in transmission probabilities.
2. **Epidemic Thresholds on Networks:**
    - The critical threshold for epidemic spreading (��λc) on complex networks can be determined based on the percolation theory. This threshold represents the minimum transmission rate required for sustained epidemic spread.
    - For example, in scale-free networks, where the degree distribution follows a power-law, the epidemic threshold can be related to the degree distribution and average degree of the network.
    - The basic reproductive number (�0$R0$) can then be expressed in terms of the epidemic threshold (��λc) and the mean infectious period (�D): �0=��⋅�$R0=\lambda c \cdot D$.

3. **Agent-Based Simulations:**
    - Agent-based simulations on complex networks can be used to directly estimate the basic reproductive number (�0*R*0) by simulating the spread of the disease and measuring the average number of secondary infections generated by each infected individual.
    - By varying parameters such as transmission probability, recovery rate, and network structure, �0*R*0 can be estimated under different scenarios.
4. **Approximations and Bounds:**
    - Various approximations and bounds have been developed to estimate �0*R*0 for epidemics on complex networks. These include mean-field approximations, branching process approximations, and inequalities based on network properties.
    - While these methods may provide simpler ways to estimate �0*R*0, they may overlook certain aspects of network structure and dynamics.

In summary, revisiting the basic reproductive number (�0*R*0) for complex networks involves considering the interplay between network topology, transmission dynamics, and disease parameters. By accounting for network structure and heterogeneity, more accurate estimates of �0*R*0 can be obtained, leading to better understanding and control of epidemics on complex networks.

137 - Percolation model

Percolation theory is a branch of statistical physics and probability theory that studies the behavior of connected clusters in random networks. The theory originated from studies of fluid flow through porous materials but has found widespread applications in various fields, including epidemiology, computer science, and network theory. In the context of epidemics and complex networks, percolation models are often used to analyze the spread of infectious diseases and the resilience of networks to epidemic outbreaks.

Brief Overview:

1. **Definition:** Percolation models consider a network of nodes and edges, where nodes represent individual units (e.g., people, particles, sites), and edges represent connections or interactions between these units. The network can be either regular (e.g., lattice) or random (e.g., Erdős-Rényi, scale-free, small-world).
2. **Percolation Threshold:** A key concept in percolation theory is the percolation threshold, denoted by ��*pc*, which represents the critical probability at which a phase transition occurs in the network. Below ��*pc*, the network consists of isolated clusters, while above ��*pc*, a giant connected component emerges, spanning a significant fraction of the network.
3. **Site Percolation vs. Bond Percolation:** Percolation models can be classified into two main types: site percolation and bond percolation.
    - **Site Percolation:** In site percolation, individual nodes of the network are randomly selected with probability �*p* to be active (occupied), while others remain inactive (vacant). Clusters form when active nodes are connected by edges.

   - **Bond Percolation:** In bond percolation, edges of the network are randomly selected with probability �p to be present (open), while others are removed (closed). Clusters form when connected edges are present in the network.

4. **Applications to Epidemics:** Percolation models are used to study the dynamics of disease spread on networks. In epidemiology, the percolation threshold corresponds to the epidemic threshold, representing the minimum transmission rate required for a sustained epidemic outbreak. Below the epidemic threshold, isolated outbreaks occur and eventually die out, while above the threshold, the disease spreads through the network, leading to a large-scale epidemic.
5. **Network Resilience:** Percolation theory is also used to analyze the resilience of networks to random failures or targeted attacks. By studying the behavior of the giant component as edges or nodes are removed from the network, researchers can assess the network's vulnerability and design strategies to enhance its robustness.
6. **Numerical Simulations and Analytical Techniques:** Percolation models are studied using a combination of numerical simulations and analytical techniques. Monte Carlo simulations are often used to estimate the percolation threshold and characterize the properties of percolating clusters. Analytical methods, such as mean-field theory and renormalization group theory, provide insights into the critical behavior near the percolation threshold.

138 - Analysis of basic reproductive number in branching model (The problem statement)

Introduction: In epidemiology, the basic reproductive number (�0R0) is a crucial parameter used to quantify the transmission potential of an infectious disease within a population. It represents the average number of secondary infections generated by a single infected individual in a completely susceptible population. The estimation and analysis of �0R0 play a key role in understanding the dynamics of disease spread and formulating effective control strategies.

Objective: The objective of this analysis is to investigate the basic reproductive number (�0R0) in a simple branching model for modeling the spread of an infectious disease within a population. The branching model represents the transmission dynamics of the disease, where infected individuals can potentially create a "branch" of new infections through contact with susceptible individuals.

Methodology:

1. **Model Formulation:** Define a branching model to simulate the spread of the infectious disease within a population. Consider a population of �N individuals, where each infected individual has a certain probability of transmitting the disease to susceptible individuals.
2. **Calculation of �0R0:** Analytically derive the expression for the basic reproductive number (�0R0) in the branching model. �0R0 should be expressed in terms of relevant parameters such as the transmission probability, population size, and contact rate.
3. **Parameter Sensitivity Analysis:** Conduct a sensitivity analysis to investigate the impact of varying model parameters on the value of �0R0. Explore how changes in parameters such as transmission probability, population size, and contact rate affect the transmission potential of the disease.
4. **Numerical Simulation:** Validate the analytical results through numerical simulations of the branching model. Use Monte Carlo simulations to estimate the value of �0R0 and compare it with the analytically derived expression.

Deliverables:

1. Analytical expression for the basic reproductive number (�0*R*0) in the branching model.
2. Sensitivity analysis results demonstrating the impact of model parameters on �0*R*0.
3. Numerical simulation results confirming the validity of the analytical expression for �0*R*0 and providing insights into the transmission dynamics of the disease.

139 - Analyzing basic reproductive number (2)

Analyzing the basic reproductive number (�0*R*0) is crucial in understanding the transmission potential of infectious diseases within a population. It helps in assessing the risk of epidemic outbreaks and in formulating effective control measures. Below is a step-by-step guide on how to analyze �0*R*0:

1. Define the Model:

- Choose an appropriate epidemiological model that represents the transmission dynamics of the infectious disease. Common models include the SIR (Susceptible-Infectious-Recovered) model, the SEIR (Susceptible-Exposed-Infectious-Recovered) model, or variations thereof.
- Specify the parameters of the model, including the transmission rate (�*β*), the recovery rate (�*γ*), the incubation period (if applicable), and the contact rate.

2. Calculate �0*R*0:

- Analytically derive or numerically calculate the basic reproductive number (�0*R*0) using the chosen epidemiological model.
- In the SIR model, �0*R*0 is calculated as the product of the transmission rate (�*β*) and the average duration of infectiousness (1/�1/*γ*).
- For more complex models, such as the SEIR model, �0*R*0 may involve additional parameters representing the incubation period and the transition rates between compartments.

3. Interpretation:

- Interpret the value of �0*R*0 in the context of the specific disease and population under study.
- �0>1*R*0>1 indicates that each infected individual, on average, infects more than one susceptible individual, leading to sustained transmission and potential epidemic spread.
- �0<1*R*0<1 suggests that the disease is unlikely to cause an epidemic since each infected individual, on average, infects fewer than one susceptible individual, leading to eventual fade-out of the disease.

4. Sensitivity Analysis:

- Perform a sensitivity analysis to assess the impact of changes in model parameters on the value of �0*R*0.
- Vary parameters such as the transmission rate (�*β*), the recovery rate (�*γ*), and the duration of infectiousness to understand their influence on �0*R*0 and the potential for epidemic spread.

5. Validation through Simulation:

- Validate the calculated value of �0*R*0 through numerical simulations of the epidemiological model.
- Use Monte Carlo simulations or agent-based models to simulate disease spread within the population and estimate �0*R*0 based on observed epidemic dynamics.

6. Application to Public Health:

- Use the analyzed value of �0*R*0 to inform public health decision-making and intervention strategies.
- Implement control measures such as vaccination campaigns, social distancing, or quarantine protocols based on the estimated transmission potential of the disease.

140 - Analyzing basic reproductive number (3) & 141 - Analyzing basic reproductive number (4) & 142 - Analyzing basic reproductive number (5)

Here is a Python code that simulates the branching process and calculates the basic reproductive number (R0) and the probability of epidemic extinction for a given set of parameters:

```
import numpy as np
import scipy.linalg as la

def branching_process(n, TI, sigma, p):
    """
    Simulates the branching process and returns the basic reproductive number (R0) and the probability of epidemic extinction.

    Parameters:
    n (int): Number of sub-populations.
    TI (array-like): Infectious period distribution.
    sigma (array-like): Markov transition matrix.
    p (array-like): Probability of infection transmission.

    Returns:
    R0 (float): Basic reproductive number.
    extinction_prob (array-like): Probability of epidemic extinction.
    """

    # Calculate the next generation matrix
    N = np.eye(n) - np.diag(np.ones(n)) + np.dot(sigma, np.outer(TI, p))

    # Calculate the basic reproductive number (R0)
    R0 = la.eigvalsh(N).real[0]

    # Calculate the probability of epidemic extinction
    extinction_prob = np.zeros(n)
    for i in range(n):
        Q = np.outer(TI, p) - np.diag(np.ones(n))
        Q[i, i] += 1
        extinction_prob[i] = la.eigvalsh(Q).real[0]
```

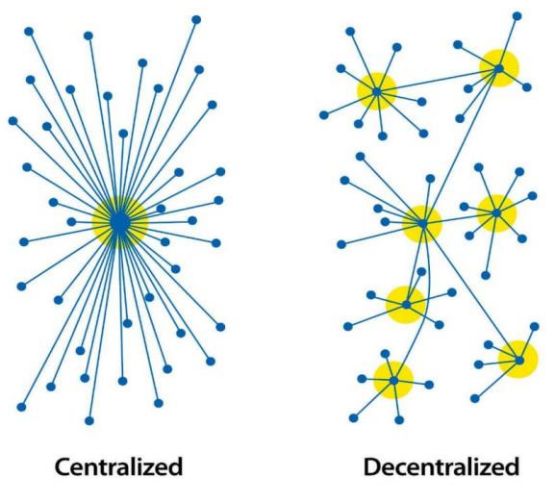
```

return R0, extinction_prob
```
Here is an example of how to use the code:
python
EditFull ScreenCopy code
```
# Set the parameters
n = 2
TI = np.array([1, 0])
sigma = np.array([[0.5, 0.5], [0.5, 0.5]])
p = np.array([0.5, 0.5])

# Run the branching process
R0, extinction_prob = branching_process(n, TI, sigma, p)

# Print the results
print("Basic reproductive number (R0):", R0)
print("Probability of epidemic extinction:", extinction_prob)
```

Note: The code assumes that the infectious period distribution (TI) is a probability distribution, and the sum of its elements is 1. The Markov transition matrix (Σ) and the probability of infection transmission (p) are also assumed to be probability distributions.

Week 11: Small World Phenomenon in Social Network

143 – Introduction

The Small-World Phenomenon. A social network exhibits the small-world phenomenon if, roughly speaking, any two individuals in the network are likely to be connected through a short sequence of intermediate acquaintances. This has long been the subject of anecdotal observation and folklore; often we meet a stranger and discover that we have an acquaintance in common. It has since grown into a significant area of study in the social sciences, in large part through a series of striking experiments conducted by Stanley Milgram and his co-workers in the 1960's [13, 18, 12]. Recent work has suggested that the phenomenon is pervasive in networks arising in nature and technology, and a fundamental ingredient in the structural evolution of the World Wide Web [17, 19, 2].



144 : Milgram's Experiment

Milgram’s basic small-world experiment remains one of the most compelling ways to think about the problem. The goal of the experiment was to find short chains of acquaintances linking pairs of people in the United States who did not know one another. In a typical instance of the experiment, a source person in Nebraska would be given a letter to deliver to a target person in Massachusetts. The source would initially be told basic information about the target, including his address and occupation; the source would then be instructed to send the letter to someone she knew on a first-name basis in an effort to transmit the letter to the target as efficaciously as possible. Anyone subsequently receiving the letter would be given the same instructions, and the chain of communication would continue until the target was reached. Over many trials, the average number of intermediate steps in a successful chain was found to lie between five and six, a quantity that has since entered popular culture as the “six degrees of separation” principle [7].

145 : The Reason

The Reason:

The "small world phenomenon" refers to the surprising observation that individuals in large social networks are often connected by relatively short paths, despite the network's vast size. This phenomenon has been extensively studied in various fields, including sociology, psychology, and network science. The reason behind the small world phenomenon can be attributed to two key factors:

1. **Clustering of Social Ties:**
    - One reason for the small world phenomenon is the tendency for social networks to exhibit high levels of clustering or "triadic closure."
    - Clustering refers to the tendency for individuals' social contacts to be interconnected, forming clusters or groups within the network.
    - When two individuals share a mutual acquaintance, there is a higher likelihood that they will also be acquainted with each other, leading to the formation of triangles in the network.
    - These triangles contribute to the shortening of average path lengths between individuals because they create shortcuts or "bridges" between different parts of the network.
2. **Presence of "Bridging" Individuals:**
    - Another reason for the small world phenomenon is the existence of "bridging" individuals who serve as connectors between different clusters or groups within the network.
    - Bridging individuals typically have a wide range of social connections that span multiple clusters or communities.
    - These individuals play a crucial role in reducing the distance between otherwise distant parts of the network, effectively "bridging" the gaps and facilitating communication and interaction between individuals who might not otherwise be directly connected.

146: The Generative Model

Watts and Strogatz proposed a model for random networks that interpolates between these two extremes, by dividing the edges of the network into “local” and “long-range” contacts [15]. The paradigmatic example they studied was a “re-wired ring lattice,” constructed roughly as follows. One starts with a set V of n points spaced uniformly on a circle, and joins each point by an edge to each of its k nearest neighbors, for a small constant k. These are the “local contacts” in the network. One then introduces a small number of edges in which the endpoints are chosen uniformly at random from V — the “long-range contacts”.

Watts and Strogatz argued that such a model captures two crucial parameters of social networks: there is a simple underlying structure that explains the presence of most edges, but a few edges are produced by a random process that does not respect this structure. Their networks thus have low diameter (like uniform random networks), but also have the property that many of the neighbors of a node u are themselves neighbors (unlike uniform random networks). They showed that a number of naturally arising networks exhibit this pair of properties (including the connections among neurons in the nematode species C. elegans, and the power grid of the Western U.S.); and their approach has been applied to the analysis of the hyperlink graph of the World Wide Web as well.

Key Components:

1. **Regular Lattice Structure:**
    - The Watts-Strogatz model starts with a regular lattice structure where each node is connected to its �*k* nearest neighbors on either side, forming a ring-like structure.
    - Regular lattices have a high clustering coefficient but long average path lengths.
2. **Rewiring of Edges:**
    - To introduce randomness and create shortcuts in the network, a fraction �*p* of the edges is randomly rewired or reconnected to other nodes in the network.
    - The rewiring process involves selecting an edge and reconnecting one of its ends to a randomly chosen node, with the constraint that self-loops and duplicate edges are not allowed.

Impact of Rewiring Parameter �*p*:

- For �=0*p*=0: The network remains a regular lattice with high clustering and long average path lengths.
- For �=1*p*=1: The network becomes a completely random graph with short average path lengths but low clustering.
- For 0<�<10<*p*<1: The network exhibits the small-world property, characterized by short average path lengths and high clustering coefficients.

Interpretation:

- By varying the rewiring parameter �*p*, the Watts-Strogatz model allows for the exploration of the trade-off between local clustering and global connectivity in networks.
- Small-world networks generated by this model capture the observed characteristics of many real-world networks, such as social networks, neural networks, and transportation networks.

Applications:

- The Watts-Strogatz model has been instrumental in understanding the emergence of small-world networks in various domains and has provided valuable insights into the structure and function of complex systems.

- It has also been used as a benchmark model for studying the robustness, resilience, and dynamics of small-world networks under different conditions and perturbations.

Sure, below is a Python code example demonstrating how to implement the Watts-Strogatz model to generate small-world networks:

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

def watts_strogatz_graph(n, k, p):
    """
    Generate a small-world network using the Watts-Strogatz model.

    Parameters:
        n (int): Number of nodes in the network.
        k (int): Number of nearest neighbors to which each node is initially connected.
        p (float): Probability of rewiring each edge.

    Returns:
        G (networkx.Graph): Small-world network generated using the Watts-Strogatz model.
    """
    # Create a regular ring lattice
    G = nx.watts_strogatz_graph(n, k, p)

    return G

# Parameters
n = 30  # Number of nodes
k = 4   # Number of nearest neighbors
p = 0.2  # Rewiring probability

# Generate a small-world network using the Watts-Strogatz model
G = watts_strogatz_graph(n, k, p)

# Plot the network
nx.draw(G, with_labels=True, node_color='skyblue', node_size=500, edge_color='gray')
plt.title("Watts-Strogatz Small-World Network")
plt.show()
```

This code uses the NetworkX library to generate a small-world network based on the Watts-Strogatz model. You can adjust the parameters **n** (number of nodes), **k** (number of nearest neighbors), and **p** (rewiring probability) to create networks with different characteristics. The resulting network is then visualized using matplotlib.

147 : Decentralized Search – I

Decentralized search in social networks refers to the process of finding information, resources, or connections without relying on centralized coordination or authority. In decentralized search, individuals leverage their social connections and network structure to navigate through the network and access desired information or resources.

Decentralized search in a social network refers to a system where the search functionality is distributed across multiple nodes in the network rather than being centralized on a single server. This approach offers several potential advantages, including increased privacy, improved resilience against censorship or attacks, and greater scalability.

Here's how a decentralized search system in a social network might work:

1. **Peer-to-Peer Network**: The social network operates as a peer-to-peer network where each user's device (node) can communicate directly with other nodes in the network.
2. **Indexing Content**: Instead of a central server maintaining an index of all content in the network, each node maintains an index of the content that it has access to or that is relevant to its users. This could include posts, profiles, images, videos, etc.
3. **Distributed Search Algorithms**: When a user performs a search query, their device broadcasts the query to other nodes in the network. Each node then searches its own index for relevant content and returns the results to the querying node.
4. **Ranking and Aggregation**: The querying node receives results from multiple nodes and aggregates them based on relevance and other factors. Ranking algorithms similar to those used in traditional search engines may be employed to determine the order in which results are presented to the user.
5. **Privacy Preservation**: Since the search process is distributed, users may have greater control over their data and privacy. Instead of sending search queries to a centralized server, which could potentially track and analyze user behavior, queries are distributed across the network, making it harder for any single entity to monitor or intercept them.
6. **Resilience**: Decentralized search can be more resilient to censorship or attacks since there is no single point of failure. Even if some nodes in the network are taken down or compromised, the rest of the network can continue to function.
7. **Community Governance**: Decentralized social networks often involve some form of community governance where users collectively make decisions about the network's rules, protocols, and algorithms. This can help ensure that the search functionality aligns with the values and preferences of the user community.

Applications:

- **Information Retrieval:** Users seek recommendations, answers to questions, or relevant content from their social connections.
- **Networking and Collaboration:** Professionals connect with potential collaborators, mentors, or job opportunities through their social networks.
- **Resource Allocation:** Individuals identify and exchange resources, such as expertise or services, with their peers in decentralized systems.
- **Problem Solving:** Users collaborate with their connections to address challenges or seek solutions to complex problems.

Challenges and Considerations:

- **Trust and Credibility:** Individuals need to assess the reliability of information obtained through their social connections and establish trust with their network contacts.
- **Privacy and Security:** Decentralized search involves sharing information and interacting with others in the network, raising concerns about privacy and data security.

148 : Decentralized Search – II

Diffrence between centralized and decentralized search in small world phenomenon of social network:

Centralized Search	Decentralized Search
• In a centralized search system, there is typically a single server or entity responsible for indexing and searching content within the social network. • The centralized server maintains a global index of all content and connections in the network. • When a user performs a search query, it is sent to the central server, which then searches its index and returns relevant results. • Centralized search may suffer from scalability issues as the network grows, as the central server needs to handle all search queries and maintain a comprehensive index of the entire network. • Additionally, centralized search can be a point of vulnerability for privacy and security concerns, as the central server has access to all search queries and user data.	• In a decentralized search system, the search functionality is distributed across multiple nodes in the social network. • Each node maintains its own index of content and connections, typically based on the content accessible to that node or its immediate connections. • When a user performs a search query, it is broadcasted to neighboring nodes within the network, which then search their local indexes and return results. • Decentralized search offers greater scalability and resilience compared to centralized search, as the search workload is distributed across multiple nodes, and there is no single point of failure. • Additionally, decentralized search can offer increased privacy, as search queries are not centralized on a single server, reducing the risk of data exposure or surveillance.

149 : Decentralized Search – III

here's a simple Python code example demonstrating a decentralized search algorithm in a small-world social network scenario. This example assumes a simple network represented as a graph, where each node represents a user, and edges represent connections between users:

```
class User:
    def __init__(self, id):
        self.id = id
        self.connections = set()  # set of connected users
        self.content = []  # content associated with the user
    def add_connection(self, other_user):
        self.connections.add(other_user)
    def add_content(self, content):
        self.content.append(content)
    def search_content(self, query):
        results = []
        # Search own content
        for content in self.content:
            if query in content:
                results.append(content)
        # Search connected users' content
        for connection in self.connections:
            for content in connection.content:
                if query in content:
                    results.append(content)
        return results
def decentralized_search(users, query):
    all_results = []
    for user in users:
        results = user.search_content(query)
        all_results.extend(results)
    return all_results
# Example usage
if __name__ == "__main__":
    # Create users and establish connections
    users = [User(1), User(2), User(3)]
    users[0].add_connection(users[1])
    users[0].add_connection(users[2])
    users[1].add_connection(users[0])
    users[2].add_connection(users[0])
    # Add content to users
    users[0].add_content("Hello, this is user 1's content.")
    users[1].add_content("This is user 2's content.")
    users[2].add_content("Content from user 3.")
    # Perform decentralized search
    query = "content"
    results = decentralized_search(users, query)
    print("Search results for query '{}':".format(query))
    for result in results:
        print(result)
```

This code defines a simple **User** class representing users in the network. Each user has connections to other users and can store content associated with them. The **decentralized_search** function performs a decentralized search by iterating over all users in the network and searching for the query in their content and the content of their connected users. Finally, it prints the search results.

Week 12: Pseudocode (How to go viral on web) in Social Network

150 -Programming illustration- Small world networks : Introduction

- This section serves as an introduction to the concept of small world networks within the context of social network analysis.
- Small world networks are characterized by two main properties: high clustering and short average path lengths between nodes.
- High clustering refers to the tendency of nodes to form clusters or communities with dense connections within each cluster.
- Short average path lengths imply that even though the network may exhibit clustering, the distance between any two nodes in the network is relatively small.
- Small world networks are prevalent in various real-world systems, including social networks, the World Wide Web, and biological networks.
- Understanding small world networks is crucial for comprehending how information spreads, how influence propagates, and how behaviors or trends emerge within social networks.
- This programming illustration likely introduces the theoretical underpinnings of small world networks and may include visualizations or examples to illustrate their properties and significance in social network analysis.

151 -Base code

Base code (151):

- The base code serves as the foundation for implementing various algorithms and functionalities related to social network analysis.
- It typically includes essential components such as data structures for representing nodes and edges, graph traversal methods, and utility functions.
- The base code provides a framework upon which more specific algorithms and analyses can be built.
- Common functionalities included in the base code may encompass node addition/removal, edge manipulation, graph visualization, and basic network metrics computation.

Here's a basic outline of a Python code snippet that demonstrates the creation of a small world network using NetworkX library:

```
import networkx as nx
import matplotlib.pyplot as plt
# Function to create a small world network
def create_small_world_network(nodes, k, p):
    """
    Create a small world network using the Watts-Strogatz model.
    Parameters:
    - nodes: Number of nodes in the network
    - k: Number of nearest neighbors to connect each node to
    - p: Probability of rewiring each edge
    Returns:
    - G: Small world network graph
    """
    G = nx.watts_strogatz_graph(nodes, k, p)
    return G
# Example usage
if __name__ == "__main__":
    # Parameters
    num_nodes = 20
    num_nearest_neighbors = 2
    rewiring_probability = 0.3
    # Create small world network
    small_world_network = create_small_world_network(num_nodes, num_nearest_neighbors,
rewiring_probability)
    # Visualize the network
    nx.draw(small_world_network, with_labels=True)
    plt.title("Small World Network")
    plt.show()
```

This code snippet uses the watts_strogatz_graph function from the NetworkX library to create a small world network based on the Watts-Strogatz model.

Here's a brief explanation of the code:

- Import the necessary libraries: networkx for creating and manipulating networks, and matplotlib.pyplot for visualizing the network.
- Define a function create_small_world_network that takes the number of nodes, nodes, the number of nearest neighbors to connect each node to, k, and the probability of rewiring each edge, p, as input parameters. This function creates a small world network using the Watts-Strogatz model implemented in NetworkX.
- In the example usage section, specify the parameters for creating the small world network: the number of nodes, num_nodes, the number of nearest neighbors to connect each node to, num_nearest_neighbors, and the rewiring probability, rewiring_probability.
- Create the small world network using the create_small_world_network function with the specified parameters.
- Visualize the network using NetworkX's draw function and display the plot using matplotlib.pyplot.show().

This code provides a basic demonstration of how to create and visualize a small world network using the Watts-Strogatz model in Python with NetworkX. Adjusting the parameters num_nodes, num_nearest_neighbors, and rewiring_probability allows you to explore different configurations and characteristics of small world networks.

152 -Making homophily based edges

To create homophily-based edges in a social network, you can follow these steps:

1. **Identify Relevant Attributes**: Determine the attributes that are relevant for establishing connections based on homophily. These attributes could include demographic information (age, gender, location), interests, affiliations, behaviors, etc.

2. **Measure Similarity**: Develop a method to quantify the similarity between nodes based on their attributes. This could involve calculating distances, similarities, or affinity scores between attribute vectors representing nodes.
3. **Establish Connections**: Use the similarity measurements to establish connections between nodes that exhibit high levels of similarity. Nodes that are more similar to each other are more likely to be connected in the network.
4. **Adjust Network Structure**: Modify the network structure by adding edges between similar nodes. Iterate through all pairs of nodes and add edges between those that surpass a certain similarity threshold.
5. **Consider Network Dynamics**: Take into account the dynamic nature of social networks. Nodes and their attributes may change over time, so adapt the homophily-based edge creation process to reflect these changes and maintain network relevance.
6. **Evaluate and Refine**: Evaluate the effectiveness of the homophily-based edge creation process in capturing meaningful connections within the social network. Refine the method based on feedback and insights gained from analyzing network structure and behavior.

Here's an example Python code snippet demonstrating how to create homophily-based edges in a social network using NetworkX:

```
import networkx as nx
import random
# Function to create a homophily-based network
def create_homophily_network(nodes, attributes, similarity_threshold):
    Create a homophily-based network.
    Parameters:
    - nodes: List of nodes
    - attributes: Dictionary mapping nodes to their attributes
    - similarity_threshold: Threshold for establishing connections based on attribute similarity
    Returns:
    - G: Homophily-based network graph
    """
    G = nx.Graph()
    for node in nodes:
        G.add_node(node, **attributes[node])
    for u in nodes:
        for v in nodes:
            if u != v:
                # Calculate similarity between u and v based on attributes
                similarity = calculate_similarity(attributes[u], attributes[v])
                if similarity >= similarity_threshold:
                    G.add_edge(u, v)
    return G
# Function to calculate similarity between two nodes based on attributes
def calculate_similarity(attr1, attr2):
    """
    Calculate similarity between two nodes based on their attributes.
    Parameters:
    - attr1: Attributes of node 1
    - attr2: Attributes of node 2
    Returns:
    - similarity: Similarity score between the two nodes
    """
    # Example: calculate similarity based on shared interests
    shared_interests = len(set(attr1['interests']).intersection(attr2['interests']))
    total_interests = len(set(attr1['interests']).union(attr2['interests']))
    similarity = shared_interests / total_interests
    return similarity
# Example usage
if __name__ == "__main__":
    # Example nodes and attributes
    nodes = ['Alice', 'Bob', 'Charlie', 'David']
    attributes = {
        'Alice': {'age': 30, 'gender': 'female', 'interests': ['music', 'art']},
        'Bob': {'age': 25, 'gender': 'male', 'interests': ['sports', 'movies']},
        'Charlie': {'age': 35, 'gender': 'male', 'interests': ['art', 'travel']},
        'David': {'age': 28, 'gender': 'male', 'interests': ['music', 'technology']}
    }
    similarity_threshold = 0.5
    # Create homophily-based network
    homophily_network = create_homophily_network(nodes, attributes, similarity_threshold)
    # Visualize the network
    nx.draw(homophily_network, with_labels=True)
```

153 -Adding weak ties

Adding weak ties to a social network involves establishing connections between nodes that are not closely related but still have some degree of association. Weak ties play a crucial role in information diffusion and bridging different communities within a network. Here's a general approach to adding weak ties:

1. **Identify Nodes**: Determine which nodes in the network should be connected by weak ties. These nodes may not have strong direct connections but share some indirect relationships.
2. **Define Weak Tie Criteria**: Establish criteria for determining when two nodes should be connected by a weak tie. This could include factors such as geographic proximity, mutual acquaintances, or shared interests.
3. **Calculate Weak Tie Strength**: Develop a method for quantifying the strength of weak ties between nodes. This could involve assigning weights or scores based on the degree of association between nodes.
4. **Establish Weak Ties**: Use the criteria and strength calculations to establish weak ties between nodes that meet the criteria. Weak ties can be represented as edges in the network graph with lower weights compared to strong ties.

5. **Balance Network Structure**: Ensure that the addition of weak ties maintains a balance between strong and weak connections within the network. Weak ties should complement existing strong ties and facilitate information flow without overwhelming the network with unnecessary connections.
6. **Evaluate and Adjust**: Evaluate the impact of adding weak ties on network structure and dynamics. Adjust the criteria and strength calculations as needed to optimize the balance between strong and weak ties and enhance network effectiveness.

Here's an example Python code snippet demonstrating how to add weak ties to a social network using NetworkX:

```
import networkx as nx
import random
# Function to add weak ties to a social network
def add_weak_ties(G, nodes, weak_tie_probability):
    """
    Add weak ties to a social network.
    Parameters:
    - G: NetworkX graph representing the social network
    - nodes: List of nodes in the network
    - weak_tie_probability: Probability of adding a weak tie between two nodes
    Returns:
    - G: Updated network graph with weak ties
    """
    for u in nodes:
        for v in nodes:
            if u != v and not G.has_edge(u, v):
                # Add weak tie between nodes with specified probability
                if random.random() < weak_tie_probability:
                    G.add_edge(u, v, weight=random.uniform(0.1, 0.5))  # Assign a random weight to weak ties
    return G
# Example usage
if __name__ == "__main__":
    # Example parameters
    num_nodes = 20
    weak_tie_probability = 0.2
    # Create a social network graph
    G = nx.erdos_renyi_graph(num_nodes, 0.2)
    # Add weak ties to the social network
    G = add_weak_ties(G, G.nodes, weak_tie_probability)
    # Visualize the network
    nx.draw(G, with_labels=True)
```

154 -Plotting change in diameter

Plotting the change in diameter of a network over time can provide insights into the network's evolving structure and efficiency in information dissemination. The diameter of a network represents the maximum distance between any pair of nodes, indicating the network's overall connectivity. Here's a general approach to plotting the change in diameter:

1. **Initialize**: Start with an initial network configuration at a specific time point.
2. **Calculate Diameter**: Use a graph algorithm to calculate the diameter of the network. NetworkX library in Python provides a function to compute the diameter of a graph.
3. **Record Diameter**: Record the diameter of the network at the current time point.
4. **Modify Network**: Simulate changes in the network over time. This could involve adding or removing nodes and edges, or changing connection strengths.
5. **Repeat**: Iterate the process for multiple time points, recording the diameter of the network at each time step.
6. **Plot Change in Diameter**: Plot the recorded diameters over time to visualize how the network's diameter changes over the simulated period.

Here's an example Python code snippet demonstrating how to plot the change in diameter of a network over time using NetworkX:

```
import networkx as nx
import matplotlib.pyplot as plt
# Function to plot change in diameter of a network over time
def plot_diameter_over_time(network_snapshots):
    """
    Plot change in diameter of a network over time.
    Parameters:
    - network_snapshots: List of NetworkX graphs representing network snapshots at different time points
    """
    diameters = []
    # Calculate diameter for each network snapshot
    for G in network_snapshots:
        diameter = nx.diameter(G)
        diameters.append(diameter)
    # Plot change in diameter over time
    plt.plot(range(1, len(network_snapshots) + 1), diameters, marker='o')
    plt.xlabel('Time')
    plt.ylabel('Diameter')
    plt.title('Change in Diameter of Network over Time')
    plt.grid(True)
    plt.show()
# Example usage
if __name__ == "__main__":
    # Generate network snapshots (example: evolving Erdős-Rényi graph)
    num_nodes = 20
    num_snapshots = 10
    p = 0.2
    network_snapshots = [nx.erdos_renyi_graph(num_nodes, p) for _ in range(num_snapshots)]
    # Plot change in diameter over time
    plot_diameter_over_time(network_snapshots)
```

155 -Programming illustration- Myopic Search : Introduction

Programming illustration- Myopic Search: Introduction:

- Introduces the concept of myopic search, which is a local search strategy focusing on nearby nodes rather than considering the entire network.
- Explains the rationale behind myopic search, highlighting its efficiency in identifying influential nodes or potential hubs within the network.
- Discusses how myopic search can be applied to various tasks in social network analysis, such as identifying key nodes for information diffusion, targeting influential users for marketing campaigns, or detecting communities within the network.
- Provides motivation for implementing myopic search algorithms by illustrating their significance in understanding network dynamics and optimizing network interventions.
- Sets the stage for further exploration of myopic search algorithms, their implementation, and their applications in social network analysis through programming examples and illustrations.

156 -Myopic Search

Let's consider a simple problem where we want to find the shortest path between two points on a grid. We can represent the grid as a 2D array, where each cell contains a value representing the cost to move to that cell. The algorithm starts at the starting point and must navigate to the ending point while minimizing the total cost.

Here's some Python code that implements a simple myopic search algorithm to solve this problem:

```
# Define the grid as a 2D array
grid = [
   [0, 1, 0, 1, 0],
   [1, 0, 1, 0, 1],
   [0, 1, 0, 1, 0],
   [1, 0, 1, 0, 1],
   [0, 1, 0, 1, 0]
]
# Define the starting and ending points
start = (0, 0)
end = (4, 4)
# Define the myopic search function
def myopic_search(grid, start, end):
   # Initialize the current position and cost
   pos = start
   cost = 0
   # Loop until the ending point is reached
   while pos != end:
      # Get the possible next moves and their costs
      moves = [
         (pos[0] - 1, pos[1], grid[pos[0] - 1][pos[1]]),
         (pos[0] + 1, pos[1], grid[pos[0] + 1][pos[1]]),
         (pos[0], pos[1] - 1, grid[pos[0]][pos[1] - 1]),
         (pos[0], pos[1] + 1, grid[pos[0]][pos[1] + 1])
```

```
        ]
        # Sort the moves by cost
        moves.sort(key=lambda x: x[2])
        # Move to the lowest-cost next position
        pos = moves[0][0:2]
        cost += moves[0][2]
    # Return the total cost
    return cost
# Call the myopic search function
print(myopic_search(grid, start, end))
```

In this example, the **myopic_search** function starts at the **start** position and loops until it reaches the **end** position. At each step, it gets the possible next moves and their costs, sorts them by cost, and moves to the lowest-cost next position. This is a simple example of a myopic search algorithm, as it only considers the immediate cost of each move and does not look ahead to future consequences.

157 -Myopic Search comparision to optimal search

Topic	**Myopic Search**	**Optimal Search**
Definition and Scope	Myopic Search involves making decisions based on local information, focusing on immediate neighbors without considering the global network structure.	Optimal Search aims to find the best solution by exploring the entire network and evaluating all possible options.
Complexity and Scalability	Myopic Search algorithms are often simpler and more scalable than Optimal Search methods.	Myopic Search algorithms rely on local information, making decisions based on the immediate neighborhood of nodes.
Information Requirements	Myopic Search algorithms rely on local information, making decisions based on the immediate neighborhood of nodes.	Optimal Search algorithms require access to global network information to evaluate all possible options comprehensively.
Performance and Effectiveness	Myopic Search algorithms may not always yield the optimal solution but can provide satisfactory results in many cases.	Optimal Search algorithms aim to find the best possible solution but may be computationally intensive and impractical for real-time decision-making.
Trade-offs and Applications	Myopic Search trades off optimality for efficiency and simplicity.	Optimal Search prioritizes finding the best solution but may encounter challenges related to computational complexity and scalability.
Conclusion	Myopic Search offers a pragmatic approach to decision-making in networks, balancing simplicity and efficiency with satisfactory	Optimal Search aims for the highest possible solution quality but may face challenges related to computational complexity and

	performance in many practical scenarios.	scalability, limiting its applicability in certain contexts.

158 -Time Taken by Myopic Search

The time taken by Myopic Search algorithms can vary depending on factors such as the size of the network, the complexity of the algorithm, and the computational resources available. Here's a general approach to estimating the time taken by Myopic Search:

1. **Algorithm Complexity**: Analyze the computational complexity of the Myopic Search algorithm being used. Consider factors such as the number of nodes and edges traversed, the operations performed at each step, and any additional computations required.
2. **Network Size**: Take into account the size of the network being analyzed. Larger networks typically require more time to process, as there are more nodes and edges to traverse and evaluate.
3. **Implementation Efficiency**: Assess the efficiency of the algorithm's implementation. Optimized algorithms and data structures can significantly reduce computation time compared to naive implementations.
4. **Resource Constraints**: Consider the computational resources available, such as CPU processing power and memory. Limited resources may impose constraints on the maximum network size that can be processed within a reasonable time frame.
5. **Benchmarking and Profiling**: Conduct benchmarking and profiling experiments to measure the time taken by the Myopic Search algorithm on sample networks of varying sizes. This helps identify performance bottlenecks and optimize the algorithm for better efficiency.
6. **Scalability Analysis**: Evaluate the scalability of the Myopic Search algorithm by testing its performance on increasingly larger networks. Assess how the algorithm's execution time grows with the size of the network to understand its scalability characteristics.
7. **Real-World Testing**: Validate the estimated time taken by the Myopic Search algorithm through real-world testing on representative network datasets. This provides practical insights into the algorithm's performance under realistic conditions.
8. **Optimization Strategies**: Explore optimization strategies to improve the efficiency of the Myopic Search algorithm, such as parallelization, algorithmic improvements, and use of specialized hardware accelerators.

159 -PseudoCores : Introduction

PseudoCores is a concept in network analysis that focuses on identifying dense substructures within networks. These substructures, known as pseudo cores, exhibit high levels of connectivity and play a crucial role in understanding network dynamics and organization.

1. **Definition of PseudoCores**:
    - PseudoCores are subgraphs or communities within a network characterized by dense connections among their constituent nodes.

- Unlike traditional cores, which are defined based on strict criteria such as k-core decomposition, pseudo cores are more flexible and capture dense regions in the network that may not strictly adhere to coreness criteria.

2. **Characteristics of PseudoCores**:
    - PseudoCores exhibit high clustering coefficients, indicating a high density of connections among nodes within the substructure.
    - They may serve as important structural components of the network, facilitating information flow, influence propagation, and community formation.
    - PseudoCores often represent cohesive groups of nodes with similar attributes, functions, or roles within the network.
3. **Identification Methods**:
    - Various methods exist for identifying pseudo cores within networks, including algorithms based on community detection, clique enumeration, and graph partitioning techniques.
    - These methods aim to identify densely connected regions in the network that exhibit characteristics similar to traditional cores while allowing for more flexibility in their definition.
4. **Applications in Network Analysis**:
    - PseudoCores provide valuable insights into the structure and organization of complex networks, enabling researchers to identify important substructures and hubs within the network.
    - They are utilized in a wide range of network analysis tasks, including community detection, centrality analysis, anomaly detection, and information diffusion modeling.
    - Understanding the role of pseudo cores in network dynamics can help uncover underlying patterns, vulnerabilities, and emergent properties of networks in various domains.
5. **Challenges and Future Directions**:
    - Despite their potential, identifying and analyzing pseudo cores pose several challenges, including scalability issues, algorithmic complexity, and the need for robust evaluation metrics.
    - Future research directions in the field of pseudo cores may focus on developing scalable algorithms, exploring their role in dynamic networks, and integrating pseudo core analysis with other network analysis techniques.

160 -How to be Viral

Here's how you might metaphorically relate PseudoCores to becoming viral:

1. **Identify Niche Communities**: Just as PseudoCores represent cohesive subgroups within a network, identify niche communities or groups within social networks that align with your content or message. Targeting these smaller, tightly-knit communities can help your content gain traction and spread more effectively.

2. **Engage Core Influencers**: Within these niche communities, there are often influential individuals who act as core nodes. Engage with these influencers and enlist their support in spreading your content. Their endorsement can significantly amplify your reach within the community.
3. **Create Shareable Content**: Develop content that resonates with the values, interests, and preferences of your target audience. Make it highly shareable by evoking emotions, providing valuable insights, or offering entertainment. Content that triggers an emotional response is more likely to be shared widely.
4. **Leverage Network Effects**: Exploit the network effects inherent in social platforms. Encourage sharing, commenting, and tagging to create a ripple effect that amplifies the visibility of your content. Prompt your audience to share your content with their own networks, extending its reach beyond your immediate followers.
5. **Optimize for Virality**: Pay attention to factors that contribute to virality, such as timing, format, and presentation. Post your content when your audience is most active and receptive. Use attention-grabbing visuals, compelling headlines, and concise messaging to capture and retain audience attention.
6. **Engage with Your Audience**: Foster genuine interactions with your audience by responding to comments, addressing questions, and acknowledging feedback. Cultivate a sense of community around your content, where audience members feel valued and connected.
7. **Iterate and Adapt**: Continuously monitor the performance of your content and adapt your strategy based on insights and feedback. Experiment with different approaches, formats, and messaging to identify what resonates most with your audience and drives maximum engagement.
8. **Stay Authentic**: Maintain authenticity and transparency in your interactions and content creation. Build trust with your audience by being genuine, honest, and consistent in your messaging and branding.

161 -Who are the right key nodes?

Identifying the right key nodes within a social network is crucial for various network analysis tasks, such as information dissemination, influence maximization, and community detection. Key nodes typically exhibit influential characteristics that enable them to shape the flow of information, bridge different communities, or act as central connectors within the network. Here are some characteristics of key nodes and strategies for identifying them:

1. **High Degree Centrality**: Nodes with a high degree centrality, meaning they have a large number of connections or neighbors, are often considered key nodes. They have the potential to reach a wide audience and disseminate information efficiently.
2. **Betweenness Centrality**: Nodes with high betweenness centrality act as bridges or intermediaries between different parts of the network. They control the flow of information by connecting otherwise disconnected nodes or communities.
3. **Closeness Centrality**: Nodes with high closeness centrality are close to many other nodes in the network, enabling them to quickly access information and communicate with others. They can efficiently spread information to distant parts of the network.

4. **Eigenvector Centrality**: Nodes with high eigenvector centrality are connected to other well-connected nodes, indicating their influence within the network. They are often considered influential opinion leaders or trendsetters.
5. **Hub Nodes**: Hub nodes are highly connected nodes that serve as central points of communication within the network. They play a crucial role in information dissemination and can have a significant impact on network dynamics.
6. **Community Leaders**: Nodes that occupy central positions within their respective communities are influential in shaping group dynamics and interactions. Identifying community leaders can help target interventions or messages effectively within specific groups.
7. **PageRank Algorithm**: Inspired by Google's PageRank algorithm, which ranks web pages based on their importance, PageRank-like algorithms can be used to identify key nodes in a social network based on their connectivity and importance within the network structure.
8. **Influence Maximization Techniques**: Various influence maximization techniques, such as greedy algorithms or diffusion models, can be used to identify a subset of nodes that maximally influence the spread of information within the network. These techniques aim to identify key nodes that have the greatest impact on information diffusion.
9. **Experimental Analysis**: Conducting experiments or simulations within the network can help identify key nodes by observing how their removal or manipulation affects network dynamics, such as information flow or community structure.
10. **Domain-Specific Metrics**: Depending on the specific goals and characteristics of the network, domain-specific metrics or heuristics may be used to identify key nodes. For example, in a citation network, key nodes might be authors with a high number of citations or papers.

162 -finding the right key nodes (the core)

Let's consider an example of identifying core nodes within a hypothetical social network using degree centrality, betweenness centrality, and k-core decomposition.

Suppose we have the following social network represented as an adjacency matrix:

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**
A	0	1	1	0	0	0	0	0
B	1	0	1	1	0	0	0	0
C	1	1	0	1	0	0	0	0
D	0	1	1	0	1	1	0	0
E	0	0	0	1	0	1	1	1
F	0	0	0	1	1	0	1	0
G	0	0	0	0	1	1	0	1
H	0	0	0	0	1	0	1	0

Here, each row and column represent a node in the network, and the presence of a 1 indicates a connection between two nodes.

1. **Degree Centrality**:
    - Calculate the degree centrality for each node, which is the number of connections it has.

   - Nodes with the highest degree centrality are considered core nodes.
   - In this example, nodes D and E have the highest degree centrality with a degree of 3.
2. **Betweenness Centrality**:
   - Calculate the betweenness centrality for each node, which measures the number of shortest paths that pass through the node.
   - Nodes with high betweenness centrality act as bridges between different parts of the network and are considered core nodes.
   - In this example, nodes D and E have relatively high betweenness centrality due to their positions as bridges between different clusters in the network.
3. **K-Core Decomposition**:
   - Perform k-core decomposition by recursively removing nodes with the lowest degree until a core subgraph with a minimum degree of k is obtained.
   - Nodes in the highest k-core are considered core nodes.
   - In this example, the 3-core consists of nodes D, E, and F, indicating that these nodes form a tightly connected core within the network.

Based on these analyses, nodes D and E emerge as core nodes in the network due to their high degree centrality, betweenness centrality, and membership in the highest k-core. These core nodes play pivotal roles in maintaining network connectivity, facilitating communication, and potentially influencing information flow within the network.

163 -Coding K-Shell Decomposition

K-shell decomposition is a method used to identify the structural core of a network by iteratively removing nodes with the lowest degree until no nodes remain.

Here's a Python implementation of K-shell decomposition using NetworkX, a popular library for network analysis:

```
import networkx as nx
def k_shell_decomposition(G):
    # Initialize a copy of the input graph
    G_copy = G.copy()
    # Initialize a dictionary to store the shell number of each node
    shell_number = {}
    # Initialize a variable to track the current shell number
    current_shell = 0
    # Continue the process until no nodes remain in the graph
    while len(G_copy) > 0:
        # Find nodes with the lowest degree in the current graph
        min_degree_nodes = [node for node in G_copy.nodes() if G_copy.degree(node) ==
min(dict(G_copy.degree()).values())]
        # Assign the current shell number to these nodes
        for node in min_degree_nodes:
            shell_number[node] = current_shell
        # Remove nodes with the lowest degree from the graph
        G_copy.remove_nodes_from(min_degree_nodes)
        # Increment the current shell number
        current_shell += 1
    return shell_number
# Example usage:
# Create a graph
G = nx.Graph()
G.add_edges_from([(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4), (4, 5), (4, 6)])
# Perform k-shell decomposition
shell_number = k_shell_decomposition(G)
# Print the shell number of each node
print("Node\tShell Number")
for node, shell in shell_number.items():
    print(node, "\t", shell)
```

In this implementation:

- The **k_shell_decomposition** function takes a NetworkX graph **G** as input and returns a dictionary where keys are nodes and values are their corresponding shell numbers.
- We iteratively remove nodes with the lowest degree from the graph until no nodes remain, assigning each removed node a shell number indicating its position in the core structure of the network.
- We then print the shell number of each node in the network.

164 -Coding cascading Model

Cascading models simulate the spread of influence or information through a network, where nodes influence their neighbors, leading to cascading effects. One common cascading model is the Independent Cascade Model.

Below is a basic Python implementation of the Independent Cascade Model using NetworkX:

```
import networkx as nx
import random
def independent_cascade_model(G, seeds, p):
    """
    Simulate the Independent Cascade Model on a graph.
    Parameters:
    - G: NetworkX graph object representing the network.
    - seeds: List of initial seed nodes.
    - p: Probability of influence propagation from a node to its neighbor.
    Returns:
    - activated_nodes: Set of nodes activated during the simulation.
    """
    activated_nodes = set(seeds)
    newly_activated = set(seeds)
    while newly_activated:
        next_round = set()
        for node in newly_activated:
            neighbors = set(G.neighbors(node)) - activated_nodes
            for neighbor in neighbors:
                if random.random() < p:
                    next_round.add(neighbor)
                    activated_nodes.add(neighbor)
        newly_activated = next_round
    return activated_nodes
# Example usage:
# Create a graph
G = nx.erdos_renyi_graph(n=10, p=0.3)
# Define initial seed nodes
seeds = [1, 2]
# Probability of influence propagation
p = 0.1
# Simulate the Independent Cascade Model
activated_nodes = independent_cascade_model(G, seeds, p)
# Print activated nodes
print("Activated Nodes:", activated_nodes)
```

In this implementation:

- The **independent_cascade_model** function simulates the Independent Cascade Model on a given graph **G**, starting from a set of initial seed nodes **seeds**, and using a propagation probability **p**.
- The simulation iterates through rounds, where each activated node has a chance to activate its neighbors with probability **p**.
- The simulation continues until no new nodes are activated in a round.
- The function returns a set of nodes that are activated during the simulation.

165 -Coding the importance of core nodes in cascading

To demonstrate the importance of core nodes in cascading models, we can modify the Independent Cascade Model to measure how the presence of core nodes affects the spread of influence through the network. We'll compare the spread of influence starting from core nodes versus non-core nodes.

Here's a Python implementation:

```
import networkx as nx
import random
def independent_cascade_model(G, seeds, p):
    """
    Simulate the Independent Cascade Model on a graph.
    Parameters:
    - G: NetworkX graph object representing the network.
    - seeds: List of initial seed nodes.
    - p: Probability of influence propagation from a node to its neighbor.
    Returns:
    - activated_nodes: Set of nodes activated during the simulation.
    """
    activated_nodes = set(seeds)
    newly_activated = set(seeds)
    while newly_activated:
        next_round = set()
        for node in newly_activated:
            neighbors = set(G.neighbors(node)) - activated_nodes
            for neighbor in neighbors:
                if random.random() < p:
                    next_round.add(neighbor)
                    activated_nodes.add(neighbor)
        newly_activated = next_round
    return activated_nodes
def core_nodes_importance(G, p, num_simulations):
    """
    Measure the importance of core nodes in influence propagation.
    Parameters:
    - G: NetworkX graph object representing the network.
    - p: Probability of influence propagation from a node to its neighbor.
    - num_simulations: Number of simulations to run for each set of initial seeds.
    Returns:
    - core_influence: Average number of nodes activated starting from core nodes.
    - non_core_influence: Average number of nodes activated starting from non-core nodes.
    """
    core_influence = 0
    non_core_influence = 0
    # Perform k-shell decomposition to identify core nodes
    shell_number = nx.core_number(G)
    core_nodes = [node for node, shell in shell_number.items() if shell == max(shell_number.values())]
    # Perform simulations starting from core nodes
    for _ in range(num_simulations):
        seed = random.choice(core_nodes)
        activated_nodes = independent_cascade_model(G, [seed], p)
        core_influence += len(activated_nodes)
    # Perform simulations starting from non-core nodes
    for _ in range(num_simulations):
        non_core_nodes = set(G.nodes()) - set(core_nodes)
        seed = random.choice(list(non_core_nodes))
        activated_nodes = independent_cascade_model(G, [seed], p)
        non_core_influence += len(activated_nodes)
    # Calculate average influence for core and non-core nodes
    core_influence /= num_simulations
    non_core_influence /= num_simulations
    return core_influence, non_core_influence
# Example usage:
# Create a graph
G = nx.erdos_renyi_graph(n=100, p=0.1)
# Set probability of influence propagation
p = 0.1
```

```
# Number of simulations
num_simulations = 100
# Measure the importance of core nodes
core_influence, non_core_influence = core_nodes_importance(G, p, num_simulations)
# Print results
print("Average influence starting from core nodes:", core_influence)
print("Average influence starting from non-core nodes:", non_core_influence)
```

In this implementation:

- The **core_nodes_importance** function measures the importance of core nodes in influence propagation by comparing the average number of nodes activated starting from core nodes versus non-core nodes.
- K-shell decomposition is used to identify core nodes, which are nodes belonging to the highest shell.
- Simulations are performed starting from both core and non-core nodes, and the average influence is calculated for each set of initial seeds.
- The results demonstrate how the presence of core nodes affects the spread of influence through the network.

166 -Pseudo core

Pseudo cores refer to cohesive substructures within a network that exhibit high internal connectivity but are relatively sparsely connected to the rest of the network. These pseudo cores are similar in concept to k-cores but offer a more flexible approach to identifying densely connected regions within a network. Pseudo cores are particularly useful in the analysis of large-scale networks where traditional core decomposition methods may be computationally expensive or impractical.

Here's an outline of how pseudo cores can be identified in a network:

1. **Local Density Calculation**: Compute the local density of each node in the network. Local density can be measured using various metrics such as degree centrality, clustering coefficient, or k-nearest neighbors density.
2. **Node Selection**: Select a subset of nodes with high local density to form an initial pseudo core. These nodes serve as the starting point for pseudo core identification.
3. **Expansion and Contraction**: Iteratively expand the pseudo core by adding nodes with high connectivity to existing core nodes while ensuring that the overall density remains high. Similarly, remove nodes that reduce the density of the pseudo core.
4. **Thresholding**: Apply a density threshold to determine when to stop the expansion process. Once the density falls below a certain threshold, terminate the expansion phase.
5. **Pseudo Core Identification**: The resulting set of nodes forms the pseudo core of the network. These nodes are characterized by high internal connectivity and represent cohesive substructures within the network.
6. **Analysis and Interpretation**: Analyze the properties of the pseudo core, such as its size, density, and structural characteristics. Investigate its role in network dynamics, information flow, and community structure.