Network Connectivity

it is time to get back to basics

John Major

Introduction

Connectivity is a fundamental concept in network analysis that describes how well-connected a network is and how information or resources can flow through it. Understanding connectivity helps us analyze network robustness, identify critical components, and understand the structure of complex systems.

Basic Concepts

Walks and Paths

Walk: A sequence of nodes and edges where each edge connects consecutive nodes in the sequence.

Mathematically, a walk can be defined as:

$$W = (v_0, e_1, v_1, e_2, v_2, ..., e_n, v_n)$$

where v_i is a node and e_i is an edge.

Path: A walk with distinct nodes (no repeated nodes).

Key Differences:

- A walk can visit the same node multiple times
- A path visits each node at most once (except start and end are the same)
- All paths are walks, but not all walks are paths

Path ⊂ Trail ⊂ Walk

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

def generate_path(G, length):
    nodes = list(G.nodes())
    current_node = nodes[0]
    path = [current_node]
    for _ in range(1, length):
        next_node = np.random.choice(nodes)
            while next_node in path:
                next_node = np.random.choice(nodes)
            path.append(next_node)
    return path

G = nx.Graph()
G.add_nodes_from(range(10))
for i in range(10):
    for j in range(i + 1, 10):
        G.add_edge(i, j)

path = generate_path(G, 5)
print(path)
```

Connected Components

A **connected component** is a maximal subgraph where every pair of nodes is connected by a path.

Types of Connectivity:

1. **Strongly Connected**: In directed graphs, every node can reach every other node
2. **Weakly Connected**: In directed graphs, the underlying undirected graph is connected
3. **Connected**: In undirected graphs, there exists a path between any two nodes

Graph laplacian of a connected graph has only one zero eigenvalue

```
# Create a graph (replace this with your actual graph creation)

G = nx.complete_bipartite_graph(3, 5)

# Calculate the Laplacian matrix

laplacian_matrix = nx.laplacian_matrix(G).toarray()

# Find the eigenvalues of the Laplacian matrix

eigenvalues = np.linalg.eigvals(laplacian_matrix)
print(eigenvalues)

# Check if there is exactly one eigenvalue equal to zero

num_zero_eigenvalues = np.count_nonzero(np.isclose(eigenvalues,
0))
    if num_zero_eigenvalues == 1:
        print("The graph is connected.")
    else:
        print("The graph is not connected.")

[0. 5. 8. 5. 3. 3. 3. 3.]
The graph is connected.
```

Connectivity Measures

1. Node Connectivity

Definition: The minimum number of nodes that must be removed to disconnect the graph.

Mathematical Definition:

$$\kappa(G) = \min_{S \subset V} |S|$$

where S is a node cut set that disconnects the graph.

Properties:

- $\kappa(G) = 0$ if and only if G is disconnected
- $\kappa(G) = 1$ if G has a cut vertex
- $\kappa(G) \leq \delta(G)$ where $\delta(G)$ is the minimum degree

2. Edge Connectivity

Definition: The minimum number of edges that must be removed to disconnect the graph.

Mathematical Definition:

$$\lambda(G) = \min_{F \subset E} |F|$$

where F is an edge cut set that disconnects the graph.

Properties:

- $\lambda(G) = 0$ if and only if G is disconnected
- $\lambda(G) = 1$ if G has a bridge
- $\lambda(G) \leq \delta(G)$ where $\delta(G)$ is the minimum degree

```
nx.minimum_node_cut()
nx.minimum_edge_cut()
```

3. Menger's Theorem

Node Version: The maximum number of node-disjoint paths between two nodes equals the minimum number of nodes whose removal disconnects them.

Edge Version: The maximum number of edge-disjoint paths between two nodes equals the minimum number of edges whose removal disconnects them.

Network Robustness

Giant Component Analysis

Definition: In large networks, the largest connected component is called the giant component.

Properties:

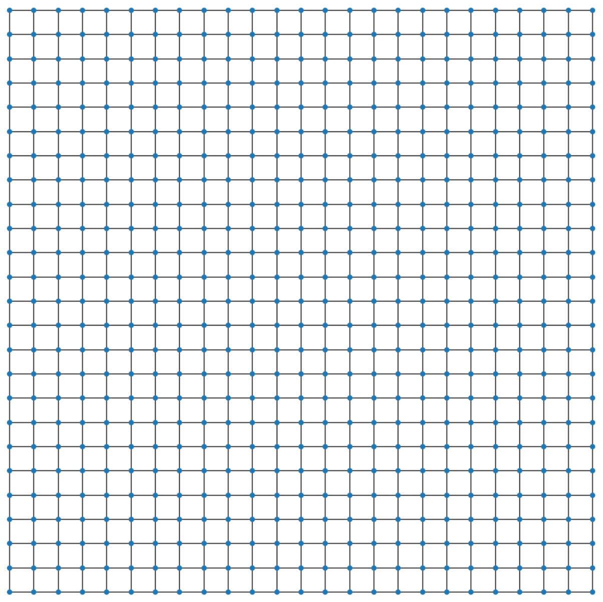
- Size scales with network size
- Emerges at a critical threshold
- Important for network functionality

Percolation Theory

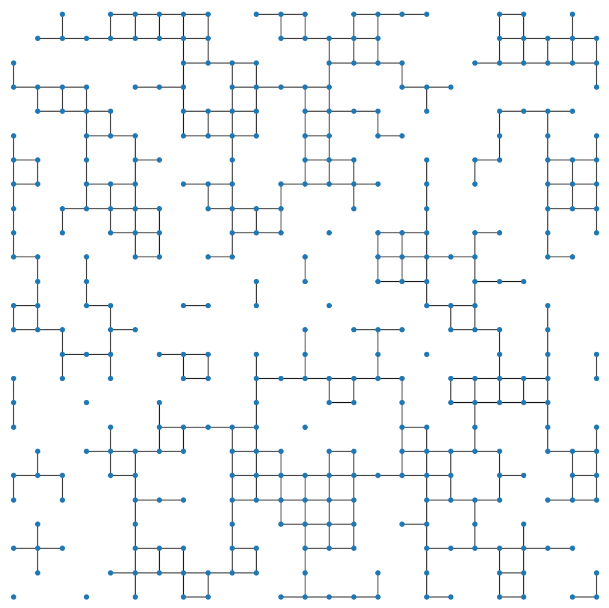
Site Percolation: Random removal of nodes **Bond Percolation**: Random removal of edges

Critical Threshold: The point at which the giant component emerges or disappears.

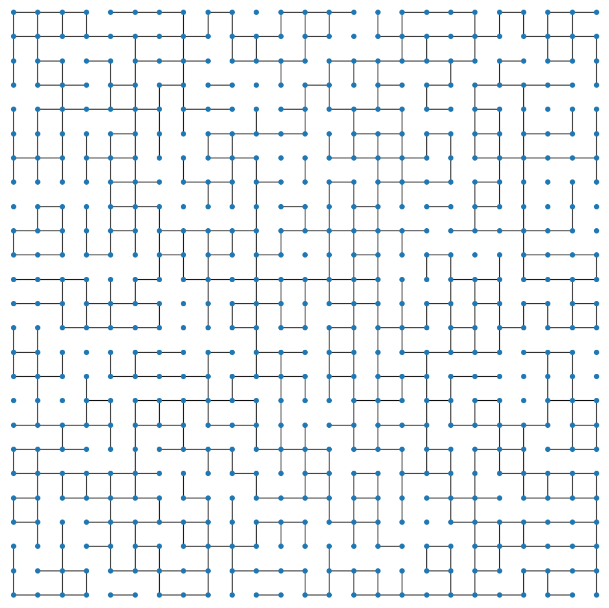
Base Lattice: 25×25 grid (625 nodes, 1200 edges)

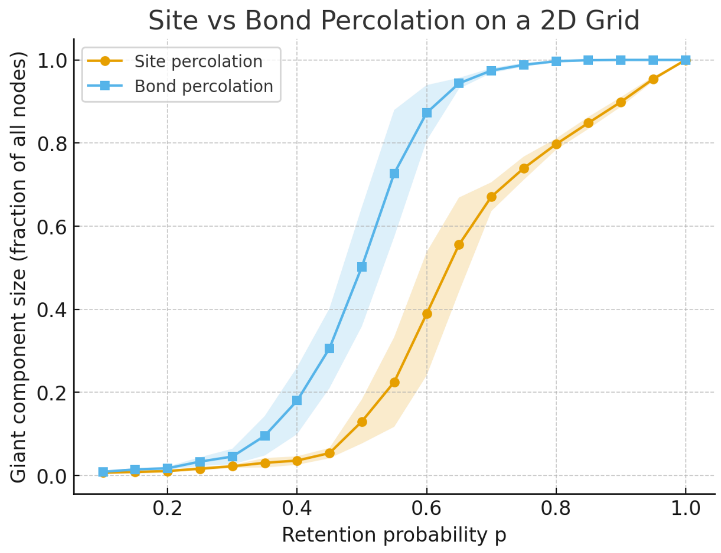


Site Percolation (nodes kept with p=0.6)
Giant component fraction ≈ 0.269 (normalized by N)



Bond Percolation (edges kept with p=0.6)
Giant component fraction ≈ 0.917 (normalized by N)





Assortativity

Degree Assortativity

Definition: The tendency of nodes to connect to other nodes with similar degrees.

Mathematical Definition:

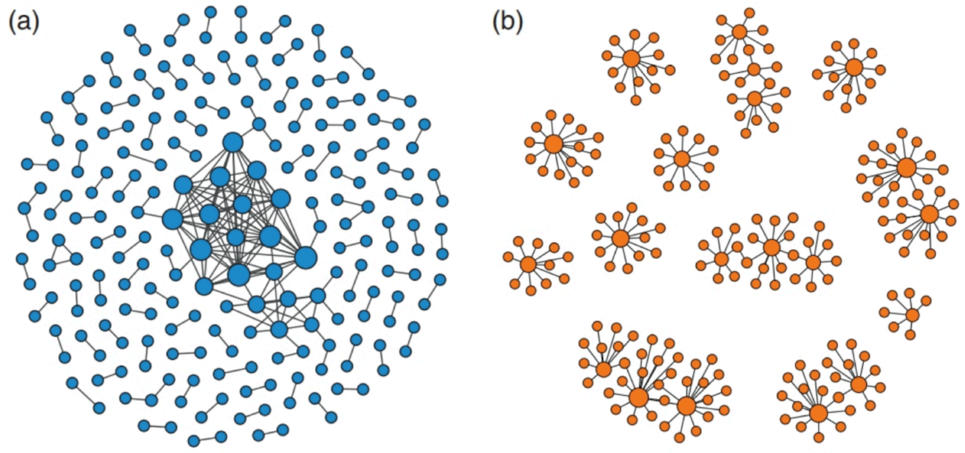
$$r = \frac{\sum_{xy} xy(e_{xy} - a_x b_y)}{\sigma_a \sigma_b}$$

where:

- e_{xy} is the fraction of edges connecting nodes of degree x and y
- $a_x = \sum_y e_{xy}$ and $b_y = \sum_x e_{xy}$
- σ_a and σ_b are standard deviations

Interpretation:

- $r > 0$: Assortative (high-degree nodes connect to high-degree nodes)
- $r < 0$: Disassortative (high-degree nodes connect to low-degree nodes)
- $r = 0$: No degree correlation



assortativity of different networks, courtesy of https://cambridgeuniversitypress.github.io/FirstCourseNetworkScience/

Types of Assortativity

1. **Assortative Networks**:
2. Social networks (people tend to connect to others with similar characteristics)
3. Collaboration networks
4. Examples: Facebook friendships, scientific collaborations
5. **Disassortative Networks**:
6. Technological networks (hubs connect to many low-degree nodes)
7. Biological networks
8. Examples: Internet, protein interaction networks

Connectivity Algorithms

Finding Connected Components

Depth-First Search (DFS) Algorithm:

Depth first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

```
def find_components_dfs(graph):
    visited = set()
    components = []

    for node in graph.nodes():
        if node not in visited:
            component = []
            dfs_visit(graph, node, visited, component)
            components.append(component)

    return components
```

Breadth-First Search (BFS) Algorithm:

Breadth first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'[1]) and explores the neighbor nodes first, before moving to the next level neighbors.

```
def find_components_bfs(graph):
    visited = set()
    components = []

    for node in graph.nodes():
        if node not in visited:
            component = []
            bfs_visit(graph, node, visited, component)
            components.append(component)

    return components
```

Applications

1. Network Design

- Designing robust communication networks
- Identifying critical infrastructure
- Planning transportation systems

2. Social Network Analysis

- Understanding community structure
- Identifying influential spreaders
- Analyzing information flow

3. Biological Networks

- Protein interaction networks
- Metabolic networks
- Gene regulatory networks

4. Technological Networks

- Internet topology
- Power grids
- Transportation networks

Computational Complexity

Problem	Time Complexity	Notes
Connected components	$O(V + E)$	Linear time
Node connectivity	$O(V^3 * E)$	Polynomial time
Edge connectivity	$O(V^3 * E)$	Polynomial time
Giant component	$O(V + E)$	Linear time
Assortativity	$O(E)$	Linear time

References

- https://cambridgeuniversitypress.github.io/FirstCourseNetworkScience/
- Bollobás, B. (2001). Random Graphs. Cambridge University Press.
- Albert, R., & Barabási, A. L. (2002). Statistical mechanics of complex networks. Reviews of Modern Physics, 74(1), 47.