Network Graph Implementation and Analysis

IoT Networks and Graph Theory Applications

Minor in AI - IIT Ropar

June 4, 2025

Contents

1 Introduction

Network graphs are fundamental structures in computer science and IoT systems, representing relationships between entities through nodes (vertices) and edges. This document covers the implementation and analysis of network graphs using Python's NetworkX library, with practical applications in IoT network simulation.

2 Graph Creation and Manipulation

2.1 Basic Graph Construction

The foundation of graph analysis begins with creating nodes and edges. In NetworkX, graphs can be constructed incrementally:

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

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

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

# Add edges
G.add_edges_from([(1,2), (2,3), (1,3)])

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

Listing 1: Basic Graph Creation

2.2 Creating Connected Graphs with Random Nodes

For larger networks, we need algorithms to ensure connectivity while maintaining randomness:

```python
import random

def create_graph(G, N):
    # Add N nodes to the graph
    for i in range(N):
        G.add_node(i)
        G.nodes[i]["Id"] = i

    # Ensure connectivity by adding random edges
    while True:
        i = random.randint(0, N-1)
        j = random.randint(0, N-1)

        if nx.is_connected(G):
            break
        else:
            if i != j:
                G.add_edge(i, j)

    return G
```

Listing 2: Connected Graph Generation Algorithm

Algorithm Analysis:

- **Time Complexity:** $O(N^2)$ in worst case
- **Space Complexity:** $O(N + E)$ where E is the number of edges
- **Connectivity Check:** Uses DFS/BFS internally with $O(N + E)$ complexity

2.3 Graph Completion Algorithms

The connectivity algorithm ensures that every node is reachable from every other node, creating a connected component. The process continues until the graph satisfies the connectivity property.

Algorithm 1 Graph Connectivity Algorithm

```
1: Initialize empty graph $G$
2: Add $N$ nodes to $G$
3: while $G$ is not connected do
4:     Select random nodes $i, j$ where $i \neq j$
5:     Add edge $(i, j)$ to $G$
6: end while
7: return $G$
```

3 Node Properties and Visualization

3.1 Color Coding Nodes Based on Properties

Nodes can be differentiated based on mathematical or logical properties. A common example is distinguishing prime numbers:

```
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

# Color nodes: red for prime, green for non-prime
node_colors = [’red’ if is_prime(node) else ’green’
               for node in gen_graph.nodes]

```

```python
nx.draw(gen_graph, with_labels=True,
        node_color=node_colors, font_color='white')
plt.show()
```

Listing 3: Prime Number Detection and Node Coloring

3.2 Node Labels and Positioning

Graph visualization requires careful consideration of:

- **Layout Algorithms:** Spring layout, circular layout, random layout
- **Node Attributes:** Size, color, shape, labels
- **Edge Properties:** Weight, color, style

4 Minimum Spanning Tree (MST)

4.1 MST Implementation Using Kruskal's Algorithm

A Minimum Spanning Tree connects all vertices with the minimum total edge weight:

```python
# Generate MST using Kruskal's algorithm
mst = nx.minimum_spanning_tree(G, algorithm='kruskal')

# Position nodes for visualization
pos = nx.spring_layout(mst)

# Draw the MST
nx.draw(mst, pos, with_labels=True,
        node_color=node_colors, edge_color='gray')

# Display edge weights
edge_labels = nx.get_edge_attributes(mst, 'weight')
nx.draw_networkx_edge_labels(mst, pos, edge_labels=edge_labels)
plt.show()
```

Listing 4: MST Implementation

4.2 Properties of MST

Key Properties:

1. **Acyclic:** Contains no cycles
2. **Connected:** All vertices are reachable
3. **Minimal:** Has exactly $V - 1$ edges for V vertices
4. **Optimal:** Minimum total weight among all spanning trees

Kruskal's Algorithm Complexity:

- **Time:** $O(E \log E)$ due to edge sorting
- **Space:** $O(V)$ for union-find data structure

5 Node Communication Range

5.1 Euclidean Distance Calculation

For spatial networks, connectivity depends on physical distance:

```python
import math

def euclidean(p1, p2):
    return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

# Create nodes with random positions and communication ranges
N = 20
G = nx.Graph()
positions = {}

for i in range(N):
    G.add_node(i)
    G.nodes[i]["Id"] = i
    G.nodes[i]["Role"] = "Prime" if is_prime(i) else "Normal"
    G.nodes[i]["Tx_range"] = random.randint(10, 60)
    G.nodes[i]["Rx_range"] = random.randint(10, 60)
    positions[i] = (random.uniform(0, 100), random.uniform(0,
  100))
```

Listing 5: Distance-Based Connectivity

5.2 Transmission and Receiving Ranges

Communication between nodes requires both transmission and reception capabilities:

```python
# Add edges based on communication ranges
for i in range(N):
    for j in range(i + 1, N):
        dist = euclidean(positions[i], positions[j])
        tx_i = G.nodes[i]["Tx_range"]
        rx_i = G.nodes[i]["Rx_range"]
        tx_j = G.nodes[j]["Tx_range"]
        rx_j = G.nodes[j]["Rx_range"]

        # Edge exists if distance allows bidirectional
  communication
        if (dist <= tx_i and dist <= rx_j and
            dist <= tx_j and dist <= rx_i):
            G.add_edge(i, j)
```

Listing 6: Range-Based Edge Creation

Communication Conditions: For nodes i and j to communicate:

$$d(i,j) \leq \min(Tx_i, Rx_j) \quad (1)$$
$$d(i,j) \leq \min(Tx_j, Rx_i) \quad (2)$$

where $d(i,j)$ is the Euclidean distance between nodes i and j.

5.3 Path Finding Between Nodes

Once the graph is constructed, we can find shortest paths:

```python
try:
    source = int(input("Enter the Source Node (0 to N-1): "))
    destination = int(input("Enter the Destination Node (0 to N-1): "))

    if source not in G.nodes or destination not in G.nodes:
        print("One or both nodes are not in the graph")
    else:
        if nx.has_path(G, source, destination):
            path = nx.shortest_path(G, source, destination)
            print(f"Destination node {destination} is reachable from source {source}")
            print(f"Shortest Path: {path}")
        else:
            print("Path doesn't exist")
except ValueError:
    print("Invalid input...")
```

Listing 7: Shortest Path Finding

6 IoT Network Simulation

```python

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

# ---------- EDGE DEVICE SIMULATION ----------

def generate_sensor_data(city):
    return {
    "city": city,
    "temperature": [round(random.uniform(20, 35), 1) for _ in range(5)],
    "humidity": [round(random.uniform(40, 80), 1) for _ in range(5)]
    }
# Edge Devices (Cities)
edge1 = generate_sensor_data("Delhi")
edge2 = generate_sensor_data("Mumbai")

# ---------- NETWORK TOPOLOGY SETUP ----------
def create_iot_network(edge1_data, edge2_data):
    G = nx.Graph()
    # Add nodes with type and data
    G.add_node("Delhi", type="edge", data=edge1_data)
    G.add_node("Mumbai", type="edge", data=edge2_data)
    G.add_node("Fog", type="fog", data={"forwarded": True})
    G.add_node("Cloud", type="cloud", data={"received": []}) # will be filled later
    # Add edges
    G.add_edge("Delhi", "Fog")
    G.add_edge("Mumbai", "Fog")
    G.add_edge("Fog", "Cloud")
    return G

# ---------- FOG NODE (forwarding logic) ----------
def fog_node(G):
    # Get data from edge nodes and attach it to cloud node
    forwarded_data = []
    for node in G.nodes:
        if G.nodes[node]["type"] == "edge":
            forwarded_data.append(G.nodes[node]["data"])
    G.nodes["Cloud"]["data"]["received"] = forwarded_data
    return forwarded_data

# ---------- DRAW NETWORK ----------
def draw_network(G):
    pos = {
        "Delhi": (0, 1),
        "Mumbai": (0, -1),
        "Fog": (2, 0),
        "Cloud": (4, 0)
    }

    node_colors = []
    for _, attrs in G.nodes(data=True):
        node_type = attrs["type"]
        if node_type == "edge":
            node_colors.append("lightgreen")
        elif node_type == "fog":
            node_colors.append("orange")
        elif node_type == "cloud":
            node_colors.append("skyblue")

    nx.draw(G, pos, with_labels=True, node_color=node_colors,
            node_size=1200, font_weight='bold', arrows=True)

    plt.title("IoT Network Topology (Edge -> Fog -> Cloud)")
    plt.show()

# ---------- CLOUD VISUALIZATION ----------
def cloud_visualization(cloud_data):
    plt.figure(figsize=(10, 8))

    for i, device in enumerate(cloud_data):
        time_points = list(range(1, 6))
        city = device["city"]

        # Temperature plot
        plt.subplot(2, 2, i * 2 + 1)
        plt.plot(time_points, device["temperature"], marker='o', color='red')
        plt.title(f"{city} - Temperature")
        plt.xlabel("Time")
        plt.ylabel("Temperature ( C )")
        plt.grid(True)

        # Humidity plot
        plt.subplot(2, 2, i * 2 + 2)
        plt.plot(time_points, device["humidity"], marker='s', color='blue')
        plt.title(f"{city} - Humidity")
        plt.xlabel("Time")
        plt.ylabel("Humidity (%)")
        plt.grid(True)

    plt.tight_layout()
    plt.suptitle("Cloud Visualization of IoT Sensor Data", fontsize=16, y=1.02)
    plt.show()

# ---------- RUN FULL SIMULATION ----------
iot_network = create_iot_network(edge1, edge2)
draw_network(iot_network)
fog_data = fog_node(iot_network)  # Forward data and store in cloud
cloud_visualization(fog_data)
```
Listing 8: IoT Network Topology

7 Mathematical Foundations

7.1 Graph Theory Definitions

Graph: $G = (V, E)$ where V is the set of vertices and E is the set of edges.

Connectivity: A graph is connected if there exists a path between every pair of vertices.

Spanning Tree: A subgraph that includes all vertices and is both connected and acyclic.

7.2 Complexity Analysis

8 Conclusion

Network graph implementation encompasses various algorithms and data structures essential for modeling real-world systems. The combination of graph theory, visualization

Operation	**Time Complexity**	**Space Complexity**
Add Node	$O(1)$	$O(1)$
Add Edge	$O(1)$	$O(1)$
Check Connectivity	$O(V+E)$	$O(V)$
Shortest Path (Dijkstra)	$O((V+E)\log V)$	$O(V)$
MST (Kruskal)	$O(E\log E)$	$O(V)$

Table 1: Algorithm Complexities

techniques, and practical applications in IoT networks demonstrates the versatility and importance of these concepts in modern computing.

Key Takeaways:

- Graph connectivity algorithms ensure network reliability
- Node properties enable sophisticated visualization and analysis
- MST algorithms optimize network topology
- Distance-based connectivity models real-world constraints
- IoT architectures benefit from graph-based modeling