In [1]:
```
import networkx as nx
import itertools
import random
import numpy as np
from collections import Counter
from itertools import combinations

file_path = 'edgelist_1.txt'
G = nx.read_edgelist(file_path, create_using=nx.DiGraph, nodetype=int)

# Generate all possible 3-node combinations
nodes = list(G.nodes)
subgraphs = set()

for triplet in combinations(nodes, 3):
    subgraph = G.subgraph(triplet)
    if nx.is_weakly_connected(subgraph):  # Check if the subgraph is conn
        subgraphs.add(tuple(sorted(subgraph.edges)))

# Display unique subgraphs
# for subgraph in subgraphs:
#     print(subgraph)
print(len(subgraphs))
```

24672

This Python implementation analyzes triad motifs within a directed graph using the concept of connected 3-node subgraphs (triads). The code begins by loading a graph from an edgelist file and then identifies all connected triads. It computes a canonical representation for each triad to efficiently track and classify them. The code generates random graphs with the same degree distribution as the original graph to serve as a baseline for motif significance. The Z-score is then calculated for each triad by comparing its frequency in the real graph against the random graphs. Triads are classified as motifs (significantly over-represented), anti-motifs (significantly under-represented), or neutral. The results are visualized and saved in a CSV file for further analysis. The program is designed to handle large graphs by processing nodes in batches, ensuring memory efficiency.

In [8]:
```
import networkx as nx
import numpy as np
import random
from collections import Counter, defaultdict
from itertools import combinations
import matplotlib.pyplot as plt
import time
import pandas as pd

def load_graph(file_path):
    """Load the directed graph from edgelist file."""
    G = nx.read_edgelist(file_path, create_using=nx.DiGraph(), nodetype=i
    print(f"Loaded graph with {len(G.nodes())} nodes and {len(G.edges())}
    return G

def get_triad_id(G, nodes):
    """
```

```
    Generate a canonical ID for a 3-node subgraph based on its adjacency
    This implementation is more efficient for large networks.
    """
    # Create a 3x3 adjacency matrix
    adj = np.zeros((3, 3), dtype=int)

    # Map nodes to indices 0, 1, 2
    node_to_idx = {node: i for i, node in enumerate(nodes)}

    # Fill the adjacency matrix
    for u, v in G.subgraph(nodes).edges():
        adj[node_to_idx[u]][node_to_idx[v]] = 1

    # Return as a hashable tuple
    return tuple(map(tuple, adj))

def enumerate_connected_triads(G):
    """
    Enumerate all connected 3-node subgraphs.
    This approach is more memory-efficient for large networks.
    """
    triad_counts = Counter()
    print("Enumerating connected triads...")

    # Process in batches of nodes to avoid memory issues
    nodes = list(G.nodes())
    nodes_count = len(nodes)
    batch_size = min(50, nodes_count)  # Adjust batch size based on memor

    total_processed = 0
    start_time = time.time()

    for i in range(0, nodes_count, batch_size):
        batch_nodes = nodes[i:min(i+batch_size, nodes_count)]

        # Process triplets involving at least one node from this batch
        for node1 in batch_nodes:
            neighbors = set(G.successors(node1)).union(set(G.predecessors
            neighbors = list(neighbors)

            # Consider triplets with node1 and two of its neighbors
            for j, node2 in enumerate(neighbors):
                for node3 in neighbors[j+1:]:
                    triplet = (node1, node2, node3)
                    subgraph = G.subgraph(triplet)

                    if nx.is_weakly_connected(subgraph):
                        # Get canonical ID and increment count
                        triad_id = get_triad_id(G, triplet)
                        triad_counts[triad_id] += 1

        total_processed += len(batch_nodes)
        elapsed = time.time() - start_time
        print(f"Processed {total_processed}/{nodes_count} nodes in {elaps

    # Divide by 6 because each triad is counted multiple times
    # (once for each node in the triad)
    for triad_id in triad_counts:
        triad_counts[triad_id] = triad_counts[triad_id] // 6
```

```
    print(f"Found {len(triad_counts)} unique connected triad patterns")
    return triad_counts

def generate_random_graph(G, preserving_method='configuration'):
    """
    Generate a random graph with the same degree sequence as G.
    """
    if preserving_method == 'configuration':
        # Configuration model preserves degree sequence
        in_degrees = [d for n, d in G.in_degree()]
        out_degrees = [d for n, d in G.out_degree()]

        try:
            R = nx.directed_configuration_model(in_degrees, out_degrees)
            R = nx.DiGraph(R)  # Remove parallel edges
            R.remove_edges_from(nx.selfloop_edges(R))  # Remove self-loop
        except Exception as e:
            print(f"Configuration model failed: {e}. Using edge swapping
            R = generate_random_graph(G, 'edge_swap')
    else:
        # Edge swapping preserves exact degree sequence
        R = G.copy()
        try:
            n_swaps = min(10 * len(G.edges()), 100000)  # Cap the number
            nx.algorithms.swap.directed_edge_swap(R, nswaps=n_swaps, max_
        except Exception as e:
            print(f"Edge swapping warning: {e}")

    return R

def calculate_motif_significance(G, num_random=10):
    """
    Calculate the significance of each triad motif using Z-scores.
    """
    # Count triads in the original network
    original_counts = enumerate_connected_triads(G)

    # Initialize arrays for random networks
    random_counts = defaultdict(list)

    # Generate random networks and count triads
    print(f"Generating {num_random} random networks...")
    for i in range(num_random):
        start_time = time.time()
        R = generate_random_graph(G)
        print(f"Random network {i+1} generated in {time.time() - start_ti

        start_time = time.time()
        r_counts = enumerate_connected_triads(R)
        print(f"Triad counting for random network {i+1} completed in {tim

        for triad_id, count in original_counts.items():
            random_counts[triad_id].append(r_counts.get(triad_id, 0))

    # Calculate z-scores
    results = []
    for triad_id, original_count in original_counts.items():
        random_values = random_counts[triad_id]
        mean_random = np.mean(random_values)
        std_random = np.std(random_values)
```

```
        # Calculate z-score with proper handling of zero std
        if std_random > 0:
            z_score = (original_count - mean_random) / std_random
        else:
            if original_count == mean_random:
                z_score = 0
            else:
                z_score = float('inf') if original_count > mean_random el

        results.append({
            'triad_id': triad_id,
            'original_count': original_count,
            'mean_random': mean_random,
            'std_random': std_random,
            'z_score': z_score
        })

    # Sort by absolute z-score
    results.sort(key=lambda x: abs(x['z_score']), reverse=True)
    return results

def visualize_triad(triad_id, index):
    """
    Visualize a 3-node subgraph from its adjacency matrix.
    """
    # Create a directed graph from the adjacency matrix
    G = nx.DiGraph()
    G.add_nodes_from([0, 1, 2])

    for i in range(3):
        for j in range(3):
            if triad_id[i][j] == 1:
                G.add_edge(i, j)

    # Position nodes in a triangle
    pos = {0: (0, 0), 1: (1, 0), 2: (0.5, 0.866)}

    plt.figure(figsize=(4, 4))
    nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=5
            arrowsize=20, font_weight='bold', font_size=16)
    plt.title(f"Triad {index+1}")
    plt.savefig(f"triad_{index+1}.png")
    plt.close()

def classify_triads(results, threshold=2.0):
    """
    Classify triads as motifs or anti-motifs based on z-scores.
    """
    motifs = [r for r in results if r['z_score'] > threshold]
    anti_motifs = [r for r in results if r['z_score'] < -threshold]
    neutral = [r for r in results if abs(r['z_score']) <= threshold]

    return motifs, anti_motifs, neutral

def print_results(motifs, anti_motifs, neutral):
    """
    Print results and visualize top motifs and anti-motifs.
    """
    print("\n===== NETWORK MOTIFS =====")
```

```
    print(f"Found {len(motifs)} motifs with Z-score > 2.0")
    for i, r in enumerate(motifs[:10]):  # Show top 10
        print(f"Motif {i+1}: Z-score = {r['z_score']:.2f}, Count = {r['or
        visualize_triad(r['triad_id'], i)

    print("\n===== NETWORK ANTI-MOTIFS =====")
    print(f"Found {len(anti_motifs)} anti-motifs with Z-score < -2.0")
    for i, r in enumerate(anti_motifs[:10]):  # Show top 10
        print(f"Anti-motif {i+1}: Z-score = {r['z_score']:.2f}, Count = {
        visualize_triad(r['triad_id'], i+len(motifs))

    print(f"\nNeutral triads: {len(neutral)}")

    # Create a summary table
    data = []
    for i, r in enumerate(motifs + anti_motifs):
        data.append({
            'Type': 'Motif' if i < len(motifs) else 'Anti-motif',
            'Z-score': r['z_score'],
            'Original Count': r['original_count'],
            'Random Mean': r['mean_random'],
            'Standard Deviation': r['std_random']
        })

    df = pd.DataFrame(data)
    print("\n===== SUMMARY TABLE =====")
    print(df)

    # Save to CSV
    df.to_csv('motif_analysis_results.csv', index=False)
    print("Results saved to motif_analysis_results.csv")

start_time = time.time()

# File path
file_path = 'edgelist_1.txt'

# Load graph
G = load_graph(file_path)

# Calculate motif significance
results = calculate_motif_significance(G, num_random=5)  # Reduced number

# Classify triads
motifs, anti_motifs, neutral = classify_triads(results)
```

```
Loaded graph with 300 nodes and 1600 edges
Enumerating connected triads...
Processed 50/300 nodes in 1.34 seconds
Processed 100/300 nodes in 1.69 seconds
Processed 150/300 nodes in 1.83 seconds
Processed 200/300 nodes in 1.92 seconds
Processed 250/300 nodes in 1.96 seconds
Processed 300/300 nodes in 1.98 seconds
Found 10 unique connected triad patterns
Generating 5 random networks...
Random network 1 generated in 0.01 seconds
Enumerating connected triads...
Processed 50/300 nodes in 1.40 seconds
Processed 100/300 nodes in 1.85 seconds
Processed 150/300 nodes in 2.06 seconds
Processed 200/300 nodes in 2.17 seconds
Processed 250/300 nodes in 2.20 seconds
Processed 300/300 nodes in 2.24 seconds
Found 32 unique connected triad patterns
Triad counting for random network 1 completed in 2.24 seconds
Random network 2 generated in 0.01 seconds
Enumerating connected triads...
Processed 50/300 nodes in 1.10 seconds
Processed 100/300 nodes in 1.47 seconds
Processed 150/300 nodes in 1.63 seconds
Processed 200/300 nodes in 1.71 seconds
Processed 250/300 nodes in 1.75 seconds
Processed 300/300 nodes in 1.78 seconds
Found 32 unique connected triad patterns
Triad counting for random network 2 completed in 1.78 seconds
Random network 3 generated in 0.01 seconds
Enumerating connected triads...
Processed 50/300 nodes in 1.24 seconds
Processed 100/300 nodes in 1.61 seconds
Processed 150/300 nodes in 1.79 seconds
Processed 200/300 nodes in 1.87 seconds
Processed 250/300 nodes in 1.91 seconds
Processed 300/300 nodes in 1.93 seconds
Found 31 unique connected triad patterns
Triad counting for random network 3 completed in 1.93 seconds
Random network 4 generated in 0.01 seconds
Enumerating connected triads...
Processed 50/300 nodes in 1.06 seconds
Processed 100/300 nodes in 1.34 seconds
Processed 150/300 nodes in 1.48 seconds
Processed 200/300 nodes in 1.54 seconds
Processed 250/300 nodes in 1.57 seconds
Processed 300/300 nodes in 1.59 seconds
Found 29 unique connected triad patterns
Triad counting for random network 4 completed in 1.59 seconds
Random network 5 generated in 0.01 seconds
Enumerating connected triads...
Processed 50/300 nodes in 0.95 seconds
Processed 100/300 nodes in 1.20 seconds
Processed 150/300 nodes in 1.32 seconds
Processed 200/300 nodes in 1.39 seconds
Processed 250/300 nodes in 1.41 seconds
Processed 300/300 nodes in 1.43 seconds
Found 29 unique connected triad patterns
Triad counting for random network 5 completed in 1.43 seconds
```

In [9]:
```
print_results(motifs, anti_motifs, neutral)

print(f"\nTotal execution time: {time.time() - start_time:.2f} seconds")
```

```
===== NETWORK MOTIFS =====
Found 8 motifs with Z-score > 2.0
Motif 1: Z-score = 63.88, Count = 135, Random Mean = 39.40
Motif 2: Z-score = 30.71, Count = 138, Random Mean = 49.00
Motif 3: Z-score = 29.61, Count = 142, Random Mean = 45.40
Motif 4: Z-score = 17.17, Count = 1000, Random Mean = 745.40
Motif 5: Z-score = 9.19, Count = 35, Random Mean = 22.00
Motif 6: Z-score = 5.11, Count = 1896, Random Mean = 1744.00
Motif 7: Z-score = 3.92, Count = 31, Random Mean = 25.80
Motif 8: Z-score = 3.04, Count = 38, Random Mean = 31.80

===== NETWORK ANTI-MOTIFS =====
Found 1 anti-motifs with Z-score < -2.0
Anti-motif 1: Z-score = -4.66, Count = 462, Random Mean = 558.00

Neutral triads: 1

===== SUMMARY TABLE =====
         Type    Z-score  Original Count  Random Mean  Standard Deviatio
n
0       Motif  63.875437             135         39.4            1.49666
3
1       Motif  30.707917             138         49.0            2.89827
5
2       Motif  29.614630             142         45.4            3.26190
1
3       Motif  17.171374            1000        745.4           14.82700
2
4       Motif   9.192388              35         22.0            1.41421
4
5       Motif   5.112313            1896       1744.0           29.73213
7
6       Motif   3.919647              31         25.8            1.32665
0
7       Motif   3.039800              38         31.8            2.03960
8
8  Anti-motif  -4.655589             462        558.0           20.62037
8
Results saved to motif_analysis_results.csv

Total execution time: 15.08 seconds
```

BIOLOGICAL/STRUCTURAL INTERPRETATION and INSIGHTS:

The motif analysis reveals key structural patterns in the network, with highly represented motifs (Z-scores > 2.0) indicating crucial, stable interactions, like protein complexes or regulatory pathways in biological systems. Moderately represented motifs suggest important but less dominant structures, possibly reflecting intermediate roles in larger complexes. The anti-motif (Z-score = -4.66) highlights an under-represented configuration, potentially signaling unstable or unfavorable interactions, such as incompatible protein interactions. Neutral triads are neither over- nor under-represented, indicating redundant or non-essential connections. Overall, the analysis helps identify

key functional modules, avoidable structures, and stable configurations critical for biological network functionality and stability.

The analysis identified 8 motifs and 1 anti-motif based on Z-scores derived from comparing the frequency of 3-node subgraphs (triads) in the original graph against random graphs with the same degree distribution.

Motifs: The motifs show significantly higher counts than expected in random graphs, with the most prominent motif having a Z-score of 63.88 (Motif 1), indicating a highly over-represented triad in the original graph. Other motifs also show strong significance, with Z-scores ranging from 3.04 to 63.88. These motifs might represent important structures or patterns in the graph that are significantly more common than in randomized networks.

Anti-motif: There is 1 anti-motif (Z-score of -4.66), where the triad is significantly under-represented compared to random graphs, suggesting that this particular structure is rare in the original graph.

Neutral Triads: Only 1 triad was found to be neutral, meaning its frequency in the original graph was similar to the random networks.

The results indicate that the original graph contains certain structural patterns (motifs) that are much more frequent than in random networks, and one pattern (anti-motif) that is significantly under-represented. These findings can provide insights into the underlying structure or key relationships within the network.

The results were saved to a CSV file for further review and potential visualization.