Write a Python program to calculate the betweenness centrality of nodes in a given network graph.

Using NetworkX

```
import networkx as nx
from collections import deque
from itertools import combinations

# Given graph (Adjacency List Representation)
graph = {
    "Alice": ["Bob", "Charlie", "David"],
    "Bob": ["Alice", "David", "Eve", "Hank"],
    "Charlie": ["Alice", "David", "Frank", "Grace"],
    "David": ["Alice", "Bob", "Charlie", "Eve", "Frank"],
    "Eve": ["Bob", "David", "Frank", "Ivy"],
    "Frank": ["Charlie", "David", "Eve", "Grace", "Ivy", "Jack"],
    "Grace": ["Charlie", "Frank", "Hank"],
    "Hank": ["Bob", "Grace", "Ivy", "Jack"],
    "Ivy": ["Eve", "Frank", "Hank", "Jack", "Kelly"],
    "Jack": ["Frank", "Hank", "Ivy", "Kelly", "Leo"],
    "Kelly": ["Ivy", "Jack", "Leo"],
    "Leo": ["Jack", "Kelly"]
}

# Convert the adjacency list to a NetworkX graph
G = nx.Graph(graph)

print("Betweenness Centrality (NetworkX):")
bet_centrality_nx = nx.betweenness_centrality(G)
print(bet_centrality_nx)

Betweenness Centrality (NetworkX):
{'Alice': 0.00909090909090909, 'Bob': 0.08075757575757576, 'Charlie':
0.05666666666666664, 'David': 0.07166666666666666, 'Eve':
0.05121212121212121216, 'Frank': 0.25878787878787873, 'Grace':
0.02424242424242424, 'Hank': 0.13242424242424242, 'Ivy':
0.1265151515151515, 'Jack': 0.20984848484848484, 'Kelly':
0.01515151515151515, 'Leo': 0.0}
```

Without using NetworkX

```
def bfs_shortest_paths(source, graph):
    queue = deque([source])
    paths = {source: 1}
    level = {source: 0}
    parents = {source: []}

    while queue:
```

```
        node = queue.popleft()
        for neighbor in graph[node]:
            if neighbor not in level:
                queue.append(neighbor)
                level[neighbor] = level[node] + 1
                paths[neighbor] = paths[node]
                parents[neighbor] = [node]
            elif level[neighbor] == level[node] + 1:
                paths[neighbor] += paths[node]
                parents[neighbor].append(node)

    return paths, parents

def compute_betweenness(graph):
    betweenness = {node: 0 for node in graph}

    for node in graph:
        paths, parents = bfs_shortest_paths(node, graph)
        dependency = {n: 0 for n in graph}

        for n in sorted(parents.keys(), key=lambda x: -paths[x]):
            for parent in parents[n]:
                dependency[parent] += (paths[parent] / paths[n]) * (1
+ dependency[n])
                if n != node:
                    betweenness[n] += dependency[n]

    for node in betweenness:
        betweenness[node] /= 2  # Each edge is counted twice
    return betweenness

print("\nBetweenness Centrality (Manual Calculation):")
bet_centrality_manual = compute_betweenness(graph)
print(bet_centrality_manual)


Betweenness Centrality (Manual Calculation):
{'Alice': 0.5, 'Bob': 2.2708333333333335, 'Charlie':
1.9500000000000002, 'David': 4.1625000000000005, 'Eve':
3.141666666666667, 'Frank': 7.083333333333333, 'Grace':
0.3333333333333333, 'Hank': 4.074999999999999, 'Ivy':
6.895833333333332, 'Jack': 4.833333333333333, 'Kelly':
0.8333333333333333, 'Leo': 0.0}
```

Write a Python program to apply the Girvan-Newman Algorithm to detect communities in a given network graph.

Using NetworkX

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

```python
def networkx_girvan_newman(G):
    communities = next(girvan_newman(G))
    return [list(c) for c in communities]

print("\nCommunities Detected (NetworkX):")
print(networkx_girvan_newman(G))
```

```
Communities Detected (NetworkX):
[['Grace', 'Alice', 'David', 'Bob', 'Frank', 'Charlie', 'Eve'],
['Hank', 'Ivy', 'Kelly', 'Leo', 'Jack']]
```

Without using NetworkX

```python
def edge_betweenness(graph):
    edge_btw = {tuple(sorted((u, v))): 0 for u in graph for v in
graph[u]}

    for node in graph:
        paths, parents = bfs_shortest_paths(node, graph)
        dependency = {n: 0 for n in graph}

        for n in sorted(parents.keys(), key=lambda x: -paths[x]):
            for parent in parents[n]:
                edge = tuple(sorted((parent, n)))
                edge_btw[edge] += (paths[parent] / paths[n]) * (1 +
dependency[n])
                dependency[parent] += (paths[parent] / paths[n]) * (1
+ dependency[n])

    return edge_btw

def remove_edge(edge, graph_copy):
    u, v = edge
    graph_copy[u].remove(v)
    graph_copy[v].remove(u)

def dfs(node, community, visited, graph_copy):
    stack = [node]
    while stack:
        n = stack.pop()
        if n not in visited:
            visited.add(n)
            community.append(n)
            stack.extend(graph_copy[n] - visited)

def girvan_newman_manual(graph):
    graph_copy = {node: set(neighbors) for node, neighbors in
graph.items()}

    while True:
```

```python
        edge_btw = edge_betweenness(graph_copy)
        if not edge_btw:
            break
        edge_to_remove = max(edge_btw, key=edge_btw.get)
        remove_edge(edge_to_remove, graph_copy)

        visited = set()
        communities = []

        for node in graph_copy:
            if node not in visited:
                community = []
                dfs(node, community, visited, graph_copy)
                communities.append(community)

        if len(communities) > 1:
            return communities

print("\nCommunities Detected (Manual Girvan-Newman):")
print(girvan_newman_manual(graph))
```

```
Communities Detected (Manual Girvan-Newman):
[['Alice', 'Bob', 'Eve', 'David', 'Charlie', 'Grace', 'Frank',
'Hank'], ['Ivy', 'Jack', 'Kelly', 'Leo']]
```