2023

Lab 3 NetworkX

SEIF ELDIN MAHMOUD

ID: 6773, GROUP NO: 1, SECTION 1

Introduction:

As a computer and communication engineer, I have developed a Python program that utilizes Dijkstra's algorithm to calculate the shortest paths in an undirected graph. The program then creates forwarding tables for each node in the graph and outputs them to the console. Additionally, it visualizes the graph using the NetworkX and Matplotlib libraries.

Dijkstra's algorithm is a well-known algorithm used to determine the shortest path between nodes in a graph. The algorithm starts from a source node and iteratively computes the shortest path to all its neighboring nodes. The algorithm uses a priority queue to keep track of the nodes with the smallest tentative distance.

Implementation:

The program reads the input data from a text file named 'input.txt'. The file's first line contains two integers, n, and m, representing the number of nodes and edges in the graph, respectively. The next m lines contain the edges, with each line containing the source node, destination node, and edge weight, separated by commas.

The program uses the NetworkX library to create and manipulate the graph data structure. The matplotlib library is used to visualize the graph.

The program then adds the edges to the graph and creates an empty graph using the NetworkX library. Since the graph is undirected, the program adds both directions for each edge.

For each node in the graph, the program computes the shortest path to every other node using Dijkstra's algorithm. The program constructs a forwarding table for each node containing the shortest path and the next hop for each destination node.

Finally, the program outputs the forwarding tables to the console and visualizes the graph using Matplotlib.

Conclusion:

In conclusion, the Python program I have developed implements Dijkstra's algorithm to compute the shortest paths in an undirected graph. The program creates forwarding tables for each node in the graph and outputs them to the console. The program utilizes the NetworkX and Matplotlib libraries to manipulate and visualize the graph. The program is useful for various applications, including network routing and pathfinding algorithms.

Explanation and code implementation:

The code is an implementation of Dijkstra's algorithm for finding the shortest path in a graph, with the objective of constructing forwarding tables for each node in the graph.

The code first imports the necessary libraries, `matplotlib.pyplot` for graph visualization, `networkx` for creating and manipulating graphs, and `heapq` for creating a priority queue.

The `dijkstra()` function takes two parameters, a graph and a source node. It returns two dictionaries, `dist` and `pred`, which represent the shortest distance from the source node to every other node in the graph, and the predecessor of each node on the shortest path, respectively. The function uses a priority queue to visit nodes in order of increasing distance from the source node. It initializes the distance of every node to infinity, except for the source node, which is set to 0. The function then repeatedly extracts the node with the smallest distance from the priority queue and updates the distances of its neighbors if a shorter path is found.

The code reads the input from a file named `input.txt`. The first line of the file contains two integers, `n` and `m`, which represent the number of nodes and edges in the graph, respectively. The following `m` lines contain the edges of the graph, each line consisting of three values separated by commas: the source node, the destination node, and the weight of the edge.

The code creates an empty graph `G` using `networkx`. It then loops through the edges and adds them to the graph using the `add_edge()` method. Since the graph is undirected, the code adds edges in both directions.

The code then loops through every node in the graph and computes the shortest path from the node to every other node using `dijkstra()`. For each source node, the code constructs a forwarding table that maps each destination node to the shortest path and the next hop on that path. The forwarding tables are stored in a dictionary `forwarding_tables`.

Finally, the code prints the forwarding tables for each node in the graph. It then uses `networkx` and `matplotlib.pyplot` to visualize the graph with edge weights, and displays the graph using `plt.show()`.

Simple Run:

Forwarding table for node u:

v: u -> v (next hop: v)

w: u -> x -> y -> w (next hop: x)

x: u -> x (next hop: x)

y: u -> x -> y (next hop: x)

z: u -> x -> y -> z (next hop: x)

Forwarding table for node v:

u: v -> u (next hop: u)

w: v -> w (next hop: w)

x: v -> x (next hop: x)

y: v -> x -> y (next hop: x)

z: v -> x -> y -> z (next hop: x)

Forwarding table for node w:

u: w -> y -> x -> u (next hop: y)

v: w -> v (next hop: v)

x: w -> y -> x (next hop: y)

y: w -> y (next hop: y)

z: w -> y -> z (next hop: y)

Forwarding table for node x:

u: x -> u (next hop: u)

v: x -> v (next hop: v)

w: x -> y -> w (next hop: y)

y: x -> y (next hop: y)

z: x -> y -> z (next hop: y)

Forwarding table for node y:

u: y -> x -> u (next hop: x)

v: y -> x -> v (next hop: x)

w: y -> w (next hop: w)

x: y -> x (next hop: x)

z: y -> z (next hop: z)

Forwarding table for node z:

u: z -> y -> x -> u (next hop: y)

v: z -> y -> x -> v (next hop: y)

w: z -> y -> w (next hop: y)

x: z -> y -> x (next hop: y)

y: z -> y (next hop: y)