Report Lab3 Networks

NAME : AHMED MOAHMED AHMED ALI AHMED SALEM
ID:7197

Shortest Path Algorithm using Dijkstra

This is a Python implementation of a network routing algorithm based on Dijkstra's algorithm. It computes the shortest path between nodes in a network using either Dijkstra's algorithm or the shortest path algorithm.

Requirements

- Python 3.x
- networkx module
- matplotlib module

Usage

1 - Clone or download the repository

2 - Install the required modules using pip

- pip install networkx
- pip install matplotlib

3 - Create a text file named input_file.txt in the same directory with the following format: n,m src1,dst1,weight1 src2,dst2,weight2 ... srcm,dstm,weightm

- n: number of nodes
- m: number of edges
- srci: source node of edge i
- dsti: destination node of edge i
- weighti: weight of edge i

4 - Run the script

- python shortest_path_algorithm.py

5 - Choose an option:

- d: Compute forwarding tables using Dijkstra's algorithm
- s: Compute forwarding tables using the shortest path algorithm
- p: Visualize the topology
- e: Exit the program

Analysis Methods

The script provides two methods for computing forwarding tables:

Dijkstra's Algorithm: This method uses Dijkstra's algorithm to find the shortest path between each pair of nodes in the network. It then generates a forwarding table for each node that lists the next hop for each destination node. To use this method, choose the d option when prompted.

Shortest Path Algorithm: This method uses the built-in shortest_path function of the networkx library to find the shortest path between each pair of nodes in the network. It then generates a forwarding table for each node that lists the next hop for each destination node. To use this method, choose the s option when prompted.

Code Overview:

```python
def create_graph():
    # Read input from file
    with open('input_file.txt', 'r') as f:
        lines = f.readlines()

    n, m = map(int, lines[0].strip().split(','))

    # Create a new empty graph
    G = nx.Graph()

    # Add edges with weights to the graph
    for i in range(1, m+1):
        src, dst, weight = lines[i].strip().split(',')
        G.add_edge(src, dst, weight=int(weight))

    return G
```

This function is used to create a graph object from an input file. Here's a step-by-step explanation of how it works:

1. The function begins by opening the input file named 'input_file.txt' in read mode using the open() function and assigning it to the variable f.
2. The readlines() method is then called on the file object f to read all the lines from the file and store them as a list of strings in the variable lines. Each string represents a line from the file.
3. The first line of the file is assumed to contain two integers separated by a comma. The map() function is used to apply the int() function to each element obtained by splitting the first line using the strip() and split() methods. The resulting integers are assigned to the variables n and m, representing the number of nodes and the number of edges in the graph, respectively.
4. The function creates an empty graph object using nx.Graph(). This graph will be used to store the nodes and edges of the graph read from the input file.

5. The function then iterates over the remaining lines in the lines list, starting from index 1 and ending at index m. Each iteration represents an edge in the graph.
6. Within the loop, the current line is stripped of leading and trailing whitespaces using the strip() method, and then split using the split() method with a comma as the delimiter. The resulting three elements are assigned to the variables src, dst, and weight, representing the source node, destination node, and weight of the edge, respectively.
7. The add_edge() method is called on the graph object G to add an edge between the source and destination nodes, with the weight specified as an integer using the int() function.
8. After all the edges have been added to the graph, the function returns the graph object G.

Overall, this function reads the input file, extracts the number of nodes and edges, and creates a graph object with weighted edges based on the information in the file.

```python
def dijkstra(G, source, dest):
    # Initialize distance dictionary with infinite distances for all nodes
    dist = {node: float('inf') for node in G.nodes()}
    dist[source] = 0

    # Initialize empty set to store visited nodes
    visited = set()

    # Loop through all nodes to find the shortest path
    while len(visited) < len(G.nodes()):
        # Find the node with the minimum distance
        min_dist = float('inf')
        min_node = None
        for node in G.nodes():
            if node not in visited and dist[node] < min_dist:
                min_dist = dist[node]
                min_node = node

        # If no node found, break out of the loop
        if min_node is None:
            break

        # Mark the node as visited
        visited.add(min_node)

        # Update the distances of its neighbors
        for neighbor in G.neighbors(min_node):
            if neighbor not in visited:
                new_dist = dist[min_node] + G[min_node][neighbor]['weight']
                if new_dist < dist[neighbor]:
                    dist[neighbor] = new_dist

    # Return the shortest distance from source to dest
    return dist[dest]
```

This function implements Dijkstra's algorithm to find the shortest path between a source node and a destination node in a graph. Here's a step-by-step explanation of how it works:

1. The function takes three parameters: G (the graph object), source (the source node), and dest (the destination node).
2. The function begins by initializing a dictionary dist to store the minimum distances from the source node to all other nodes in the graph. Initially, all distances are set to infinity except for the distance from the source node, which is set to 0.

3. An empty set visited is initialized to keep track of visited nodes.
4. The function enters a loop that continues until all nodes have been visited. In each iteration, it selects the node with the minimum distance from the source among the unvisited nodes.
5. Within the loop, the minimum distance and its corresponding node are initialized to infinity and None, respectively.
6. The function iterates over all nodes in the graph using for node in G.nodes(). For each node, it checks if the node has not been visited and if its distance is smaller than the current minimum distance. If both conditions are met, the minimum distance and corresponding node are updated.
7. After finding the node with the minimum distance, it is marked as visited by adding it to the visited set.
8. The function then updates the distances of the unvisited neighbors of the current node. It iterates over the neighbors of the current node using for neighbor in G.neighbors(min_node). For each neighbor, it checks if the neighbor has not been visited. If true, it calculates the new distance by adding the weight of the edge between the current node and the neighbor to the distance of the current node. If the new distance is smaller than the current distance of the neighbor, it updates the neighbor's distance in the dist dictionary.
9. The loop continues until all nodes have been visited or until no node with a finite distance is found.
10. Finally, the function returns the shortest distance from the source node to the destination node, which is obtained from the dist dictionary.

In summary, this function applies Dijkstra's algorithm to find the shortest path in a graph by maintaining a dictionary of minimum distances from the source node to all other nodes. It iteratively explores the nodes, updates the distances, and marks visited nodes until it reaches the destination or exhausts all possibilities.

```
def forward_Table(G, source):
    print("Forwarding table for " + source + ": ")
    least_cost_path = nx.shortest_path(G, source, weight='weight')
    print('+-------------+--------+')
    print('| Destination | Link   |')
    print('+-------------+--------+')
    for i, elem in enumerate(least_cost_path):
        if elem != source:
            print('|      ' + elem + '      | (' +
                  least_cost_path[elem][0] + ',' + least_cost_path[elem][1] + ')  |')
    print('+-------------+--------+')
    print()
```

This function, forward_Table(G, source), is used to print the forwarding table for a given source node in a graph. Here's a breakdown of how it works:

1. The function takes two parameters: G (the graph object) and source (the source node for which the forwarding table is generated).
2. The function starts by printing a header line indicating the forwarding table is for the specified source node.
3. It then computes the least-cost path from the source node to all other nodes in the graph using the nx.shortest_path() function from the NetworkX library. The weight='weight' parameter indicates that the weights of the graph edges are considered while finding the shortest path.
4. Next, the function prints a table header line containing the column headers "Destination" and "Link".
5. Within a loop, the function iterates over the elements in the least_cost_path dictionary, which represents the least-cost paths from the source node to other nodes. Each element consists of a destination node as the key and a list representing the link as the value.
6. For each element, the function checks if the element is not the source node itself (to exclude the source node from the table). If it's not the source node, it prints a table row containing the destination node and the link information.
7. The table rows are printed using formatted strings, aligning the destination node and link information to specific column widths.
8. Finally, the function prints a line of dashes to signify the end of the table.

In summary, this function generates and prints a forwarding table for a given source node in a graph. The table includes destination nodes and the

corresponding links, representing the least-cost paths from the source node to each destination node.

```
def forward_Table_dijkstra(G, source):
    print("Forwarding table for " + source +
          ":(Using Dijkstra) ")
    print('+-------------+--------+')
    print('| Destination |  Link  |')
    print('+-------------+--------+')
    for dest in sorted(G.nodes()):
        if dest != source:
            next_hop = None
            min_dist = float('inf')

            for neighbor in G.neighbors(source):
                new_dist = G[source][neighbor]['weight'] + \
                    dijkstra(G, neighbor, dest)
                if new_dist < min_dist:
                    min_dist = new_dist
                    next_hop = neighbor
            print('|      ' + dest + '      |   (' +
                  source + ',' + next_hop + ')  |')
    print('+-------------+--------+')
    print()
```

This function, forward_Table_dijkstra(G, source), is used to print the forwarding table for a given source node in a graph using Dijkstra's algorithm. Here's an explanation of how it works:

1. The function takes two parameters: G (the graph object) and source (the source node for which the forwarding table is generated).
2. The function begins by printing a header line indicating that the forwarding table is computed using Dijkstra's algorithm and is for the specified source node.
3. It then prints a table header line containing the column headers "Destination" and "Link".
4. Within a loop, the function iterates over the sorted list of destination nodes in the graph.

5. For each destination node, it checks if the node is not the same as the source node to exclude the source node from the table.
6. Inside the loop, the function initializes the next_hop variable as None and the min_dist variable as infinity. These variables will be used to keep track of the next hop and minimum distance.
7. The function then enters another loop, iterating over the neighbors of the source node using for neighbor in G.neighbors(source).
8. Within this nested loop, it calculates the new distance to reach the destination node by adding the weight of the edge between the source node and the current neighbor to the distance obtained from the dijkstra() function. The dijkstra() function is invoked to find the shortest path distance from the neighbor node to the destination node.
9. If the new distance is smaller than the current minimum distance (min_dist), it updates the min_dist and next_hop variables accordingly.
10. After evaluating all neighbors, the function prints a table row containing the destination node and the link information (source node and next hop) for the current destination node.
11. The table rows are printed using formatted strings, aligning the destination node and link information to specific column widths.
12. Once the loop is complete, the function prints a line of dashes to signify the end of the table.

In summary, this function generates and prints a forwarding table for a given source node in a graph using Dijkstra's algorithm. The table includes destination nodes and the corresponding links, representing the next hop node for each destination node in order to achieve the shortest path from the source node.

```python
while True:
    print("\nPlease choose an option:")
    print("d - Compute forwarding tables using Dijkstra's algorithm")
    print("s - Compute forwarding tables using shortest path algorithm")
    print("p - Visualize the topology")
    print("e - Exit the program")
    method = input("> ")

    # handle user input
    if method == 'e':
        break
    elif method == 'd':
        print("Computing forwarding tables using Dijkstra's algorithm...")
        for src in sorted(G.nodes()):  # type: ignore
            forward_Table_dijkstra(G, src)  # type: ignore
    elif method == 's':
        print("Computing forwarding tables using shortest path algorithm...")
        for src in sorted(G.nodes()):  # type: ignore
            forward_Table(G, src)  # type: ignore
    elif method == 'p':
        print("Visualizing the topology...")
        pos = nx.spring_layout(G)  # type: ignore
        nx.draw(G, pos, with_labels=True)  # type: ignore
        labels = nx.get_edge_attributes(G, 'weight')  # type: ignore
        nx.draw_networkx_edge_labels(  # type: ignore
            G, pos, edge_labels=labels)  # type: ignore
        plt.show()
    else:
        print("Invalid option. Please try again.")
```

This main program implements a menu-driven interface where the user can choose between different options. Here's a summary of its functionality:

1. The program presents a menu to the user, offering options to compute forwarding tables using Dijkstra's algorithm ('d'), compute forwarding tables using the shortest path algorithm ('s'), visualize the topology ('p'), or exit the program ('e').
2. Based on the user's input, the program performs the following actions:
    - If 'e' is selected, the program breaks out of the loop and terminates.
    - If 'd' is selected, forwarding tables are computed for each source node using Dijkstra's algorithm. The forward_Table_dijkstra(G, src) function is called for each source node.
    - If 's' is selected, forwarding tables are computed for each source node using the shortest path algorithm. The forward_Table(G, src) function is called for each source node.

- If 'p' is selected, the program visualizes the topology of the graph. The graph is displayed using nx.draw() and nx.draw_networkx_edge_labels() functions.

3. If the input doesn't match any of the available options, an error message is displayed.
4. The program continues to prompt the user for input and perform the selected actions until the user chooses to exit the program.

In summary, this main program provides a user-friendly interface to compute forwarding tables using different algorithms and visualize the graph topology.

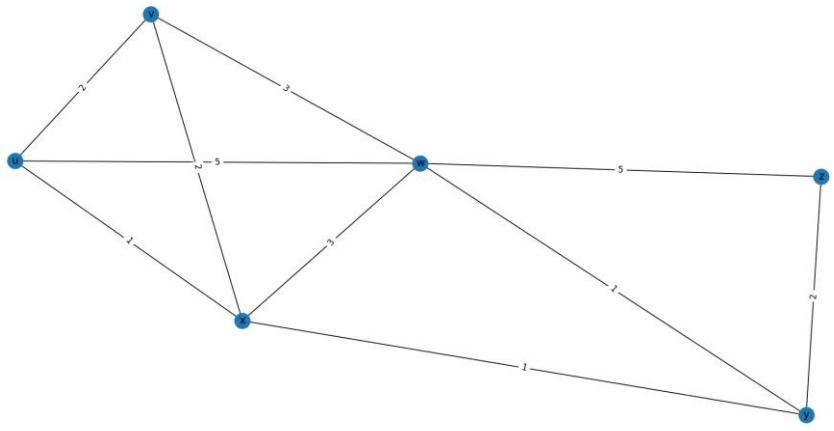
Output

```
Please choose an option:
d - Compute forwarding tables using Dijkstra's algorithm
s - Compute forwarding tables using shortest path algorithm
p - Visualize the topology
e - Exit the program
> s
Computing forwarding tables using shortest path algorithm...
Forwarding table for u:
+-------------+--------+
| Destination |  Link  |
+-------------+--------+
|      v      |  (u,v) |
|      w      |  (u,x) |
|      x      |  (u,x) |
|      y      |  (u,x) |
|      z      |  (u,x) |
+-------------+--------+

Forwarding table for v:
+-------------+--------+
| Destination |  Link  |
+-------------+--------+
|      u      |  (v,u) |
|      x      |  (v,x) |
|      w      |  (v,w) |
|      y      |  (v,x) |
|      z      |  (v,x) |
+-------------+--------+

Forwarding table for w:
+-------------+--------+
| Destination |  Link  |
+-------------+--------+
|      u      |  (w,y) |
|      v      |  (w,v) |
|      x      |  (w,y) |
|      y      |  (w,y) |
|      z      |  (w,y) |
+-------------+--------+

Forwarding table for x:
+-------------+--------+
| Destination |  Link  |
+-------------+--------+
|      u      |  (x,u) |
|      v      |  (x,v) |
|      w      |  (x,y) |
|      y      |  (x,y) |
|      z      |  (x,y) |
+-------------+--------+

Forwarding table for y:
+-------------+--------+
| Destination |  Link  |
+-------------+--------+
|      w      |  (y,w) |
|      x      |  (y,x) |
|      z      |  (y,z) |
|      u      |  (y,x) |
|      v      |  (y,x) |
+-------------+--------+

Forwarding table for z:
+-------------+--------+
| Destination |  Link  |
+-------------+--------+
|      w      |  (z,y) |
|      y      |  (z,y) |
|      x      |  (z,y) |
|      u      |  (z,y) |
|      v      |  (z,y) |
+-------------+--------+
```

1- Forwarding table for each node in the network (using shortest path algorithm):

```
Please choose an option:
d - Compute forwarding tables using Dijkstra's algorithm
s - Compute forwarding tables using shortest path algorithm
p - Visualize the topology
e - Exit the program
> d
Computing forwarding tables using Dijkstra's algorithm...
Forwarding table for u:(Using Dijkstra)
+-------------+--------+
| Destination |  Link  |
+-------------+--------+
|      v      |  (u,v) |
|      w      |  (u,x) |
|      x      |  (u,x) |
|      y      |  (u,x) |
|      z      |  (u,x) |
+-------------+--------+

Forwarding table for v:(Using Dijkstra)
+-------------+--------+
| Destination |  Link  |
+-------------+--------+
|      u      |  (v,u) |
|      w      |  (v,w) |
|      x      |  (v,x) |
|      y      |  (v,x) |
|      z      |  (v,x) |
+-------------+--------+

Forwarding table for w:(Using Dijkstra)
+-------------+--------+
| Destination |  Link  |
+-------------+--------+
|      u      |  (w,y) |
|      v      |  (w,v) |
|      x      |  (w,y) |
|      y      |  (w,y) |
|      z      |  (w,y) |
+-------------+--------+

Forwarding table for x:(Using Dijkstra)
+-------------+--------+
| Destination |  Link  |
+-------------+--------+
|      u      |  (x,u) |
|      v      |  (x,v) |
|      w      |  (x,y) |
|      y      |  (x,y) |
|      z      |  (x,y) |
+-------------+--------+

Forwarding table for y:(Using Dijkstra)
+-------------+--------+
| Destination |  Link  |
+-------------+--------+
|      u      |  (y,x) |
|      v      |  (y,x) |
|      w      |  (y,w) |
|      x      |  (y,x) |
|      z      |  (y,z) |
+-------------+--------+

Forwarding table for z:(Using Dijkstra)
+-------------+--------+
| Destination |  Link  |
+-------------+--------+
|      u      |  (z,y) |
|      v      |  (z,y) |
|      w      |  (z,y) |
|      x      |  (z,y) |
|      y      |  (z,y) |
+-------------+--------+
```

2- Forwarding table for each node in the network (Dijkstra's algorithm):



3- Visualization of the network topology (if the user chooses the 'p' option)