Dynamic Network Routing Protocol

Computer Networks Project

A Project Report Submitted by

Thallapally Nimisha

CS22B1082

B.Tech in Computer Science and Engineering
IIITDM Kancheepuram

November 20, 2024

Contents

1 Aim

The aim of this project is to implement and compare the performance of three popular shortest path algorithms—Dijkstra's Algorithm, Bellman-Ford Algorithm, and Floyd-Warshall Algorithm—using Python. The project also explores the effects of node failures on the routing process and measures round-trip time for client-server communication during pathfinding requests.

2 Introduction

Shortest path algorithms are crucial in computer networks for determining the most efficient paths between nodes in a graph, which can represent networks, transportation systems, and various other systems. This project focuses on implementing and analyzing three key algorithms—Dijkstra's, Bellman-Ford, and Floyd-Warshall. These algorithms are essential for routing and network optimization tasks.

The project will compare their computational efficiencies and ability to handle dynamic changes in the network, such as node failures.

3 System Design

3.1 Architecture

The system is designed with a client-server model where:

- The server implements the shortest path algorithms.
- Clients send requests for shortest paths between two nodes.
- A node failure simulation is included to test the resilience of the algorithms.

Each client communicates with a server on a specific port number. The system uses Python's socket programming for client-server communication.

3.2 Protocol/Concept Details

- **Dijkstra's Algorithm**: Efficient for graphs with non-negative edge weights, providing the shortest path from a source node to all other nodes.
- **Bellman-Ford Algorithm**: Capable of handling negative edge weights and detecting negative weight cycles.
- **Floyd-Warshall Algorithm**: Computes the shortest paths between all pairs of nodes in a graph.

3.3 Tools and Technologies

- Programming Language: Python
- Communication Protocol: TCP (Socket Programming)
- Libraries: socket, time

4 Implementation Details

4.1 Client-Side Code

The client code sends a request for the shortest path to a target node from a given server and calculates the round-trip time. Below is the implementation of the common client-side code:

Listing 1: Client-Side Code for Requesting Shortest Paths

```
import socket
import time

def client_request(server_id, target_node):
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect(('localhost', 8000 + server_id))
    client_socket.send(str(target_node).encode('utf-8'))
    start_time = time.time()

    response = client_socket.recv(1024).decode('utf-8')

    end_time = time.time()
    round_trip_time = end_time - start_time

    print(f"Client received: {response}")
    print(f"Round-trip time for request: {round_trip_time:.6f} seconds"
        )

    client_socket.close()

# Simulate clients requesting shortest paths to different target nodes
def run_clients():
    client_request(0, 4)
    client_request(1, 4)
    client_request(2, 4)

if __name__ == "__main__":
    run_clients()
```

4.2 Server-Side Code

Here is the placeholder for the server-side code for each algorithm. Replace the corresponding sections with the actual code for Dijkstra, Bellman-Ford, and Floyd-Warshall algorithms.

4.2.1 Dijkstra's Algorithm

Listing 2: Server-Side Code for Dijkstra's Algorithm

```
import socket
import threading
import heapq
import random
import time
import networkx as nx
```

```python
import matplotlib.pyplot as plt


def generate_large_sparse_graph(nodes, edges):
    graph = {i: {} for i in range(nodes)}
    edge_count = 0
    current_edge = 0


    while edge_count < edges:
        u = current_edge % nodes
        v = (current_edge + 1) % nodes
        weight = (current_edge + 1) % 10 + 1
        if u != v and v not in graph[u]:
            graph[u][v] = weight
            graph[v][u] = weight
        current_edge += 1


        if current_edge > nodes * 2:
            print("Unable to generate enough edges due to constraints.")
            break

    return graph


def plot_graph(graph, failed_nodes=None):
    """Visualizes the graph with optional highlighting of failed nodes."""
    G = nx.Graph()
    for node, neighbors in graph.items():
        for neighbor, weight in neighbors.items():
            G.add_edge(node, neighbor, weight=weight)

    pos = nx.spring_layout(G)
    plt.figure(figsize=(8, 6))
    nx.draw(
        G,
        pos,
        with_labels=True,
        node_color="skyblue",
        edge_color="gray",
        node_size=700,
        font_size=10,
        font_weight="bold",
    )
    if failed_nodes:
        nx.draw_networkx_nodes(G, pos, nodelist=failed_nodes, node_color="red")

    labels = nx.get_edge_attributes(G, "weight")
    nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)

    plt.title("Network Graph")
    plt.show()


```

```python
# Dijkstra's algorithm to find the shortest path
def dijkstra(graph, start_node, failed_node=None):

    # Step 1: Initialize distances to all nodes as infinity, except the
        start node
    queue = [(0, start_node)]  # (cost, node)
    distances = {node: float("inf") for node in graph}
    distances[start_node] = 0
    previous_nodes = {node: None for node in graph}

    # Step 2: Modify graph to exclude failed node if specified
    if failed_node is not None:
        graph = {
            node: {
                neighbor: weight
                for neighbor, weight in neighbors.items()
                if neighbor != failed_node
            }
            for node, neighbors in graph.items()
            if node != failed_node
        }

    while queue:
        current_distance, current_node = heapq.heappop(queue)

        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous_nodes[neighbor] = current_node
                heapq.heappush(queue, (distance, neighbor))

    return distances, previous_nodes


# Server code for each node
class Server:
    def __init__(self, node_id, graph):
        self.node_id = node_id
        self.graph = graph  # Network topology as graph
        self.server_socket = socket.socket(socket.AF_INET, socket.
            SOCK_STREAM)
        self.server_socket.bind(("localhost", 8000 + node_id))
        self.server_socket.listen(5)
        self.failed_nodes = set()
        self.routing_table = {}
        self.is_active = True
        print(f"Server {node_id} started on port {8000 + node_id}")
        self.recalculate_routing_table()

    def recalculate_routing_table(self):
        """Recalculate the routing table for all nodes."""
        print(f"Recalculating routing table for Server {self.node_id}")
        self.routing_table, _ = dijkstra(self.graph, self.node_id)
        print(f"Updated Routing Table for Server {self.node_id}:")
```

```
        for node, distance in self.routing_table.items():
            print(f"  To Node {node}: Distance {distance}")

    def handle_client(self, client_socket):
        try:
            if not self.is_active:
                # Notify client that the server is down
                print(f"Server {self.node_id} is down. Rejecting request.")
                client_socket.send(
                    f"Server {self.node_id} is currently down.".encode("utf-8")
                )
                return

            data = client_socket.recv(1024).decode("utf-8")
            if data:
                target_node = int(data)  # Target node ID
                print(
                    f"Server {self.node_id} received request for shortest path to node {target_node}"
                )
                server_start_time = time.time()

                # Retrieve the shortest path from the precomputed routing table
                distance = self.routing_table.get(target_node, float("inf"))
                response = f"Shortest path to node {target_node}: {distance}"
                server_end_time = time.time()
                server_computation_time = server_end_time - server_start_time  # Time taken to process the request
                print(f"Server {self.node_id} computation time: {server_computation_time:.6f} seconds")

                client_socket.send(response.encode("utf-8"))
        finally:
            client_socket.close()

    def run(self):
        while True:
            client_socket, _ = self.server_socket.accept()
            client_thread = threading.Thread(
                target=self.handle_client, args=(client_socket,)
            )
            client_thread.start()

    def simulate_failure(self, failed_node):
        print(f"Simulating failure of Node {failed_node} in Server {self.node_id}")


        if self.node_id == failed_node:
            self.is_active = False
            print(f"Server {self.node_id} is now marked as down.")
            return
```

```
166
167
168         self.failed_nodes.add(failed_node)
169
170
171         for node in self.graph:
172             if failed_node in self.graph[node]:
173                 del self.graph[node][failed_node]
174             if node == failed_node:
175                 self.graph[node] = {}
176
177
178         print(f"Node {failed_node} failure simulated. Recalculating
                routing table.")
179         self.recalculate_routing_table()
180         # plot_graph(self.graph, failed_nodes=self.failed_nodes)
181
182
183 # Start server for each node in the network
184 def start_servers():
185     # graph = {
186     #     0: {1: 10},
187     #     1: {0: 10, 2: 10, 3: 50},
188     #     2: {1: 10, 3: 30},
189     #     3: {2: 30, 4: 40, 1: 50},
190     #     4: {3: 40},
191     # }
192     nodes = 100
193     edges = 200
194     graph=generate_large_sparse_graph(nodes,edges)
195     servers = []
196     # plot_graph(graph)
197     # Start servers
198     for node_id in range(len(graph)):
199         server = Server(node_id, graph)
200         threading.Thread(target=server.run).start()
201         servers.append(server)
202
203     # Simulate failure of a node
204     failed_node = 2
205     time.sleep(5)
206     for server in servers:
207         server.simulate_failure(failed_node)
208
209
210     print("\nSimulating client trying to access a failed server:")
211     client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
212     try:
213         client_socket.connect(("localhost", 16000 + failed_node))
214         client_socket.send("3".encode("utf-8"))
215         response = client_socket.recv(1024).decode("utf-8")
216         print(f"Response from Server {failed_node}: {response}")
217     except ConnectionRefusedError:
218         print(f"Server {failed_node} is unavailable.")
219     finally:
220         client_socket.close()
221
222
```

```
223 if __name__ == "__main__":
224     start_servers()
```

4.2.2 Bellman-Ford Algorithm

Listing 3: Server-Side Code for Bellman-Ford Algorithm

```
1  import socket
2  import threading
3  import time
4  import random
5  import networkx as nx
6  import matplotlib.pyplot as plt
7
8  def generate_large_sparse_graph(nodes, edges):
9      graph = {i: {} for i in range(nodes)}
10     edge_count = 0
11     current_edge = 0
12
13
14     while edge_count < edges:
15         u = current_edge % nodes
16         v = (current_edge + 1) % nodes
17         weight = (current_edge + 1) % 10 + 1
18
19         if u != v and v not in graph[u]:
20             graph[u][v] = weight
21             graph[v][u] = weight
22             edge_count += 1
23
24
25         current_edge += 1
26
27
28         if current_edge > nodes * 2:
29             print("Unable to generate enough edges due to constraints."
                   )
30             break
31
32     return graph
33
34 def plot_graph(graph, failed_nodes=None):
35     """Visualizes the graph with optional highlighting of failed nodes.
           """
36     G = nx.Graph()
37     for node, neighbors in graph.items():
38         for neighbor, weight in neighbors.items():
39             G.add_edge(node, neighbor, weight=weight)
40
41     pos = nx.spring_layout(G)
42     plt.figure(figsize=(8, 6))
43     nx.draw(
44         G,
45         pos,
46         with_labels=True,
47         node_color="skyblue",
48         edge_color="gray",
```

```
        node_size=700,
        font_size=10,
        font_weight="bold",
    )
    if failed_nodes:
        nx.draw_networkx_nodes(G, pos, nodelist=failed_nodes,
            node_color="red")

    labels = nx.get_edge_attributes(G, "weight")
    nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)

    plt.title("Network Graph")
    plt.show()

# Bellman-Ford algorithm to find the shortest path
def bellman_ford(graph, start_node):

    distances = {node: float('inf') for node in graph}
    distances[start_node] = 0
    previous_nodes = {node: None for node in graph}


    for _ in range(len(graph) - 1):
        for node in graph:
            for neighbor, weight in graph[node].items():
                if distances[node] + weight < distances[neighbor]:
                    distances[neighbor] = distances[node] + weight
                    previous_nodes[neighbor] = node


    for node in graph:
        for neighbor, weight in graph[node].items():
            if distances[node] + weight < distances[neighbor]:
                print("Graph contains a negative weight cycle")

    return distances, previous_nodes

# Server code for each node
class Server:
    def __init__(self, node_id, graph):
        self.node_id = node_id
        self.graph = graph  # Network topology as graph
        self.server_socket = socket.socket(socket.AF_INET, socket.
            SOCK_STREAM)
        self.server_socket.bind(("localhost", 8000 + node_id))
        self.server_socket.listen(5)
        self.failed_nodes = set()
        self.routing_table = {}
        self.is_active = True
        print(f"Server {node_id} started on port {8000 + node_id}")
        self.recalculate_routing_table()

    def recalculate_routing_table(self):
        """Recalculate the routing table for all nodes."""
        print(f"Recalculating routing table for Server {self.node_id}")
        self.routing_table, _ = bellman_ford(self.graph, self.node_id)
        print(f"Updated Routing Table for Server {self.node_id}:")
        for node, distance in self.routing_table.items():
```

```
105             print(f"  To Node {node}: Distance {distance}")
106
107     def handle_client(self, client_socket):
108         try:
109             if not self.is_active:
110                 # Notify client that the server is down
111                 print(f"Server {self.node_id} is down. Rejecting
                        request.")
112                 client_socket.send(
113                     f"Server {self.node_id} is currently down.".encode(
                            "utf-8")
114                 )
115                 return
116
117             data = client_socket.recv(1024).decode("utf-8")
118             if data:
119                 target_node = int(data)  # Target node ID
120                 print(
121                     f"Server {self.node_id} received request for
                            shortest path to node {target_node}"
122                 )
123                 server_start_time = time.time()
124
125                 # Retrieve the shortest path from the precomputed
                        routing table
126                 distance = self.routing_table.get(target_node, float("
                        inf"))
127                 response = f"Shortest path to node {target_node}: {
                        distance}"
128                 server_end_time = time.time()
129                 server_computation_time = server_end_time -
                        server_start_time  # Time taken to process the
                        request
130                 print(f"Server {self.node_id} computation time: {
                        server_computation_time:.6f} seconds")
131
132                 client_socket.send(response.encode("utf-8"))
133         finally:
134             client_socket.close()
135
136     def run(self):
137         while True:
138             client_socket, _ = self.server_socket.accept()
139             client_thread = threading.Thread(
140                 target=self.handle_client, args=(client_socket,)
141             )
142             client_thread.start()
143
144     def simulate_failure(self, failed_node):
145         print(f"Simulating failure of Node {failed_node} in Server {
                self.node_id}")
146
147
148         if self.node_id == failed_node:
149             self.is_active = False
150             print(f"Server {self.node_id} is now marked as down.")
151             return
152
```

```
153
154         self.failed_nodes.add(failed_node)
155
156
157         for node in self.graph:
158             if failed_node in self.graph[node]:
159                 del self.graph[node][failed_node]
160             if node == failed_node:
161                 self.graph[node] = {}
162
163
164         print(f"Node {failed_node} failure simulated. Recalculating
                routing table.")
165         self.recalculate_routing_table()
166         # plot_graph(self.graph, failed_nodes=self.failed_nodes)
167
168
169 # Start server for each node in the network
170 def start_servers():
171     # graph = {
172     #     0: {1: 10},
173     #     1: {0: 10, 2: 10, 3: 50},
174     #     2: {1: 10, 3: 30},
175     #     3: {2: 30, 4: 40, 1: 50},
176     #     4: {3: 40},
177     # }
178     nodes = 100
179     edges = 200
180     graph=generate_large_sparse_graph(nodes,edges)
181     servers = []
182     # plot_graph(graph)
183     # Start servers
184     for node_id in range(len(graph)):
185         server = Server(node_id, graph)
186         threading.Thread(target=server.run).start()
187         servers.append(server)
188
189     # Simulate failure of a node
190     failed_node = 2
191     time.sleep(5)
192     for server in servers:
193         server.simulate_failure(failed_node)
194
195
196     print("\nSimulating client trying to access a failed server:")
197     client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
198     try:
199         client_socket.connect(("localhost", 16000 + failed_node))
200         client_socket.send("3".encode("utf-8"))
201         response = client_socket.recv(1024).decode("utf-8")
202         print(f"Response from Server {failed_node}: {response}")
203     except ConnectionRefusedError:
204         print(f"Server {failed_node} is unavailable.")
205     finally:
206         client_socket.close()
207
208
209 if __name__ == "__main__":
```

```
    start_servers()
```

4.2.3 Floyd-Warshall Algorithm

Listing 4: Server-Side Code for Floyd-Warshall Algorithm

```
import socket
import threading
import heapq
import random
import time
import networkx as nx
import matplotlib.pyplot as plt


def generate_large_sparse_graph(nodes, edges):
    graph = {i: {} for i in range(nodes)}
    edge_count = 0
    current_edge = 0


    while edge_count < edges:
        u = current_edge % nodes
        v = (current_edge + 1) % nodes
        weight = (current_edge + 1) % 10 + 1

        if u != v and v not in graph[u]:
            graph[u][v] = weight
            graph[v][u] = weight
            edge_count += 1


        current_edge += 1


        if current_edge > nodes * 2:
            print("Unable to generate enough edges due to constraints."
                )
            break

    return graph
def floyd_warshall(graph, start_node, failed_node=None):
    # Step 1: Initialize distances matrix
    nodes = list(graph.keys())
    num_nodes = len(nodes)
    distances = {node: {neighbor: float("inf") for neighbor in nodes}
       for node in nodes}
    previous_nodes = {node: None for node in nodes}

    # Set initial distances based on the graph
    for u in graph:
        distances[u][u] = 0  # Distance to itself is 0
        for v, weight in graph[u].items():
            distances[u][v] = weight

    # Step 2: Modify graph to exclude failed node if specified
    if failed_node is not None:
```

```
        distances = {
            node: {
                neighbor: distances[node][neighbor]
                for neighbor in distances[node]
                if neighbor != failed_node
            }
            for node in distances
            if node != failed_node
        }

    # Step 3: Apply Floyd-Warshall algorithm
    for k in distances:
        for i in distances:
            for j in distances:
                if distances[i][k] + distances[k][j] < distances[i][j]:
                    distances[i][j] = distances[i][k] + distances[k][j]

    # Extract the distances from the start_node
    single_source_distances = distances[start_node]
    return single_source_distances, previous_nodes


def plot_graph(graph, failed_nodes=None):
    """Visualizes the graph with optional highlighting of failed nodes.
       """
    G = nx.Graph()
    for node, neighbors in graph.items():
        for neighbor, weight in neighbors.items():
            G.add_edge(node, neighbor, weight=weight)

    pos = nx.spring_layout(G)
    plt.figure(figsize=(8, 6))
    nx.draw(
        G,
        pos,
        with_labels=True,
        node_color="skyblue",
        edge_color="gray",
        node_size=700,
        font_size=10,
        font_weight="bold",
    )
    if failed_nodes:
        nx.draw_networkx_nodes(G, pos, nodelist=failed_nodes,
           node_color="red")

    labels = nx.get_edge_attributes(G, "weight")
    nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)

    plt.title("Network Graph")
    plt.show()


# Server code for each node
class Server:
    def __init__(self, node_id, graph):
        self.node_id = node_id
        self.graph = graph  # Network topology as graph
```

```
106         self.server_socket = socket.socket(socket.AF_INET, socket.
                SOCK_STREAM)
107         self.server_socket.bind(("localhost", 8000 + node_id))
108         self.server_socket.listen(5)
109         self.failed_nodes = set()
110         self.routing_table = {}  # Stores distances to all nodes
111         self.is_active = True  # Indicates if the server is up or down
112         print(f"Server {node_id} started on port {8000 + node_id}")
113         self.recalculate_routing_table()  # Calculate initial routing
                table
114 
115     def recalculate_routing_table(self):
116         """Recalculate the routing table for all nodes."""
117         print(f"Recalculating routing table for Server {self.node_id}")
118         self.routing_table, _ = floyd_warshall(self.graph, self.node_id
                )
119         print(f"Updated Routing Table for Server {self.node_id}:")
120         for node, distance in self.routing_table.items():
121             print(f"  To Node {node}: Distance {distance}")
122 
123     def handle_client(self, client_socket):
124         try:
125             if not self.is_active:
126                 # Notify client that the server is down
127                 print(f"Server {self.node_id} is down. Rejecting
                        request.")
128                 client_socket.send(
129                     f"Server {self.node_id} is currently down.".encode(
                            "utf-8")
130                 )
131                 return
132 
133             data = client_socket.recv(1024).decode("utf-8")
134             if data:
135                 target_node = int(data)  # Target node ID
136                 print(
137                     f"Server {self.node_id} received request for
                            shortest path to node {target_node}"
138                 )
139                 server_start_time = time.time()
140 
141                 # Retrieve the shortest path from the precomputed
                        routing table
142                 distance = self.routing_table.get(target_node, float("
                        inf"))
143                 response = f"Shortest path to node {target_node}: {
                        distance}"
144                 server_end_time = time.time()
145                 server_computation_time = server_end_time -
                        server_start_time  # Time taken to process the
                        request
146                 print(f"Server {self.node_id} computation time: {
                        server_computation_time:.6f} seconds")
147                 # client_socket.send(response.encode("utf-8"))
148                 client_socket.send(response.encode("utf-8"))
149         finally:
150             client_socket.close()
151 
```

```
152     def run(self):
153         while True:
154             client_socket, _ = self.server_socket.accept()
155             client_thread = threading.Thread(
156                 target=self.handle_client, args=(client_socket,)
157             )
158             client_thread.start()
159 
160     def simulate_failure(self, failed_node):
161         print(f"Simulating failure of Node {failed_node} in Server {
                self.node_id}")
162 
163         # If this server is the one that failed, mark it as inactive
164         if self.node_id == failed_node:
165             self.is_active = False
166             print(f"Server {self.node_id} is now marked as down.")
167             return
168 
169         # Mark the failed node as unreachable and update the graph
170         self.failed_nodes.add(failed_node)
171 
172         # Update the graph for the failure
173         for node in self.graph:
174             if failed_node in self.graph[node]:
175                 del self.graph[node][failed_node]
176             if node == failed_node:
177                 self.graph[node] = {}
178 
179         # Recalculate the routing table for all servers after the
                failure
180         print(f"Node {failed_node} failure simulated. Recalculating
                routing table.")
181         self.recalculate_routing_table()
182         # plot_graph(self.graph, failed_nodes=self.failed_nodes)
183 
184 
185 # Server code for each node
186 class Server:
187     def __init__(self, node_id, graph):
188         self.node_id = node_id
189         self.graph = graph  # Network topology as graph
190         self.server_socket = socket.socket(socket.AF_INET, socket.
                SOCK_STREAM)
191         self.server_socket.bind(("localhost", 8000 + node_id))
192         self.server_socket.listen(5)
193         self.failed_nodes = set()
194         self.routing_table = {}
195         self.is_active = True
196         print(f"Server {node_id} started on port {8000 + node_id}")
197         self.recalculate_routing_table()
198 
199     def recalculate_routing_table(self):
200         """Recalculate the routing table for all nodes."""
201         print(f"Recalculating routing table for Server {self.node_id}")
202         self.routing_table, _ = floyd_warshall(self.graph, self.node_id
                )
203         print(f"Updated Routing Table for Server {self.node_id}:")
204         for node, distance in self.routing_table.items():
```

```
205             print(f"  To Node {node}: Distance {distance}")
206
207     def handle_client(self, client_socket):
208         try:
209             if not self.is_active:
210                 # Notify client that the server is down
211                 print(f"Server {self.node_id} is down. Rejecting
                        request.")
212                 client_socket.send(
213                     f"Server {self.node_id} is currently down.".encode(
                            "utf-8")
214                 )
215                 return
216
217             data = client_socket.recv(1024).decode("utf-8")
218             if data:
219                 target_node = int(data)  # Target node ID
220                 print(
221                     f"Server {self.node_id} received request for
                            shortest path to node {target_node}"
222                 )
223                 server_start_time = time.time()
224
225                 # Retrieve the shortest path from the precomputed
                        routing table
226                 distance = self.routing_table.get(target_node, float("
                        inf"))
227                 response = f"Shortest path to node {target_node}: {
                        distance}"
228                 server_end_time = time.time()
229                 server_computation_time = server_end_time -
                        server_start_time  # Time taken to process the
                        request
230                 print(f"Server {self.node_id} computation time: {
                        server_computation_time:.6f} seconds")
231
232                 client_socket.send(response.encode("utf-8"))
233         finally:
234             client_socket.close()
235
236     def run(self):
237         while True:
238             client_socket, _ = self.server_socket.accept()
239             client_thread = threading.Thread(
240                 target=self.handle_client, args=(client_socket,)
241             )
242             client_thread.start()
243
244     def simulate_failure(self, failed_node):
245         print(f"Simulating failure of Node {failed_node} in Server {
                self.node_id}")
246
247
248         if self.node_id == failed_node:
249             self.is_active = False
250             print(f"Server {self.node_id} is now marked as down.")
251             return
252
```

```
253
254         self.failed_nodes.add(failed_node)
255
256
257         for node in self.graph:
258             if failed_node in self.graph[node]:
259                 del self.graph[node][failed_node]
260             if node == failed_node:
261                 self.graph[node] = {}
262
263
264         print(f"Node {failed_node} failure simulated. Recalculating
                routing table.")
265         self.recalculate_routing_table()
266         # plot_graph(self.graph, failed_nodes=self.failed_nodes)
267
268
269 # Start server for each node in the network
270 def start_servers():
271     # graph = {
272     #     0: {1: 10},
273     #     1: {0: 10, 2: 10, 3: 50},
274     #     2: {1: 10, 3: 30},
275     #     3: {2: 30, 4: 40, 1: 50},
276     #     4: {3: 40},
277     # }
278     nodes = 100
279     edges = 200
280     graph=generate_large_sparse_graph(nodes,edges)
281     servers = []
282     # plot_graph(graph)
283     # Start servers
284     for node_id in range(len(graph)):
285         server = Server(node_id, graph)
286         threading.Thread(target=server.run).start()
287         servers.append(server)
288
289     # Simulate failure of a node
290     failed_node = 2
291     time.sleep(5)
292     for server in servers:
293         server.simulate_failure(failed_node)
294
295
296     print("\nSimulating client trying to access a failed server:")
297     client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
298     try:
299         client_socket.connect(("localhost", 16000 + failed_node))
300         client_socket.send("3".encode("utf-8"))
301         response = client_socket.recv(1024).decode("utf-8")
302         print(f"Response from Server {failed_node}: {response}")
303     except ConnectionRefusedError:
304         print(f"Server {failed_node} is unavailable.")
305     finally:
306         client_socket.close()
307
308
309 if __name__ == "__main__":
```

```
310     start_servers()
```

5 Testing and Results

5.1 Testing Strategy

To evaluate the project, the shortest path algorithms were implemented on separate servers. Each client sends requests for the shortest path between nodes in a weighted graph. The servers calculate the shortest paths using their respective algorithms. The round-trip time for each request was measured to assess performance.

5.2 Results

Algorithm	**Average Time**	**Node Failure Handling**	**Suitability for Networks**
Dijkstra's	Fast	Limited	Efficient for static networks with consistent routing needs
Bellman-Ford	Moderate	Effective	Ideal for dynamic networks prone to frequent topology changes
Floyd-Warshall	Slow	Limited	Suitable for fully connected networks requiring comprehensive path analysis

Table 1: Performance Comparison of Shortest Path Algorithms in Network Scenarios

5.3 Data Collected During Testing

During testing, the following metrics were recorded:

- Round-trip time for each client request.
- The shortest path returned by the server for a given source and target node.
- Server response times under different network conditions (e.g., increased load or simulated failures).

5.4 Analysis of Results

- **Performance Comparison:** Dijkstra's algorithm was the fastest among the three, followed by Bellman-Ford and Floyd-Warshall. The response times matched expectations based on the complexity of the algorithms.

- **Accuracy:** All algorithms produced accurate results for their respective scenarios, including handling negative weights and computing all-pairs shortest paths.
- **Network Delays:** The average round-trip time for client-server communication was consistent with minimal packet loss under normal conditions. However, under heavy load, delays increased, particularly for Floyd-Warshall.

5.5 Screenshots

Below are the screenshots demonstrating the client-server interaction and the output for each algorithm.

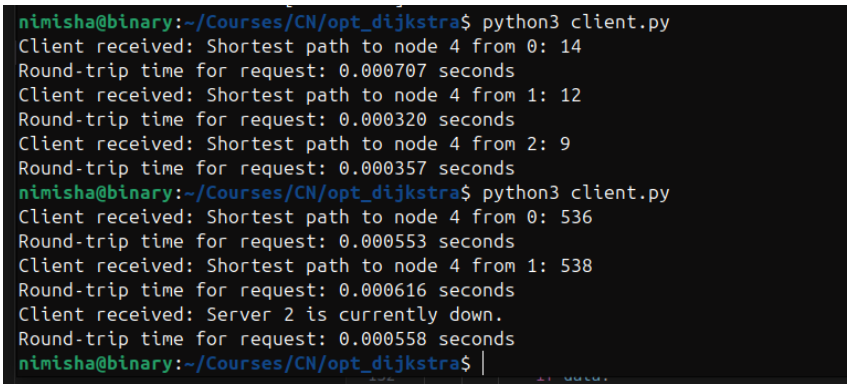

Figure 1: Output Screenshot 1: Client Request to Dijkstra Server

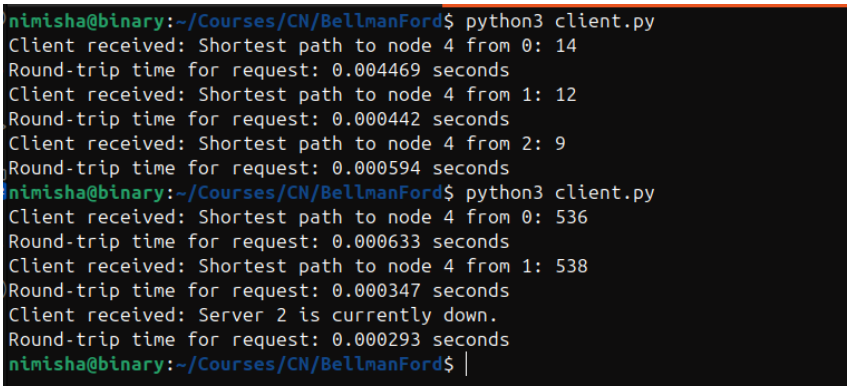

Figure 2: Output Screenshot 2: Client Request to Bellman-Ford Server

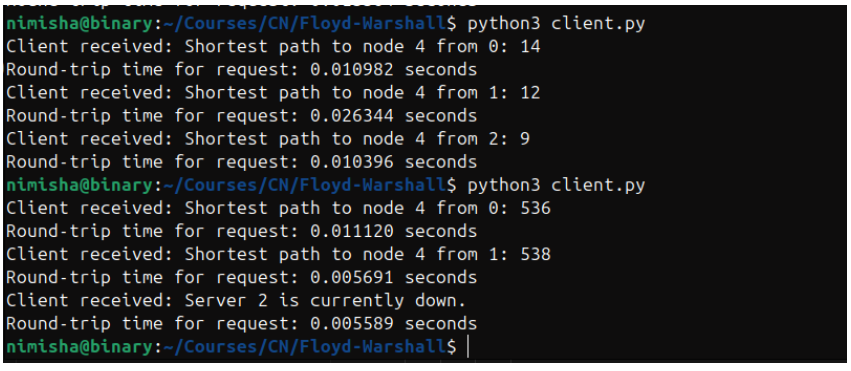

Figure 3: Output Screenshot 3: Client Request to Floyd-Warshall Server

6 Discussions

The implementation of each algorithm revealed specific strengths and weaknesses:

- **Dijkstra's Algorithm:** It performed efficiently in terms of computational time, making it suitable for scenarios requiring quick calculations of shortest paths. However, its performance depends on the data structure used for managing the priority queue.
- **Bellman-Ford Algorithm:** While slower than Dijkstra's algorithm, it demonstrated robustness in recomputing paths after network topology changes, such as node or link failures. This flexibility makes it well-suited for dynamic networks.
- **Floyd-Warshall Algorithm:** The algorithm provides a comprehensive solution for all-pairs shortest path problems. However, its high computational complexity limits its practicality for large-scale networks.

Other notable observations include:

- **Scalability:** As the size of the network increased, the computational overhead for Floyd-Warshall became significantly higher compared to the other algorithms. Dijkstra's algorithm scaled better for larger graphs when implemented with an efficient priority queue.
- **Real-Time Adaptability:** During node failure simulations, Bellman-Ford adapted to the changes more effectively than Dijkstra's algorithm, as it recalculated the paths without requiring a fresh initialization.
- **Algorithmic Trade-offs:** Each algorithm exhibited trade-offs between computational efficiency and comprehensiveness. The choice of the algorithm should be guided by the specific requirements of the use case, such as the need for real-time updates or comprehensive connectivity analysis.

Overall, the project highlighted the importance of understanding the context in which each algorithm is deployed, as their performance and utility vary significantly depending on the network's size and dynamic nature.

7 Future Enhancements

Future work can include the following:

- Implementing more efficient algorithms like A* for pathfinding in large graphs.
- Adding support for real-time dynamic updates to the network topology, such as node or link failures.
- Optimizing the performance of the Floyd-Warshall algorithm using parallel computing techniques.

8 References

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, 3rd Edition. This study is based on [2]...

2. Ming-Xin Yang Bing-Tong Wang Wen-Dong Guo , "Research on the performance of dynamic routing algorithm," *IEEE*. Available: link