University of Dhaka

Department of Computer Science and Engineering

CSE-3111 : Computer Networking Lab

Lab Report 7: Implementation of Link State Routing Algorithm

Submitted By:

Name: Meherun Farzana

Roll No : 05

Name: Mohd. Jamal Uddin Mallick

Roll No : 07

Submitted On :

March 28, 2024

Submitted To :

Dr. Md. Abdur Razzaque

Dr. Md. Mamun Or Rashid

Dr. Muhammad Ibrahim

Redwan Ahmed Rizvee

1 Introduction

The Link State Algorithm, also known as the shortest path first algorithm, is a key method in computer networking for determining the most efficient route between two nodes within a network. This algorithm operates by ensuring each router within the network maintains a comprehensive layout of the network's structure, including details about each connection's status, like its bandwidth and latency. Upon receiving updates regarding any connection's status, routers adjust their network maps and compute the shortest paths to all other nodes using Dijkstra's algorithm. These updated maps are then disseminated to all routers within the network, enabling them to refine their own maps and compute fresh shortest paths. Widely utilized in expansive networks such as the Internet, the Link State Algorithm excels in handling networks with numerous nodes, offering superior scalability compared to alternatives like the Distance Vector Algorithm. Unlike the latter, which necessitates routers broadcasting their entire routing tables, the Link State Algorithm only transmits updates pertaining to changes in network topology. This approach minimizes network congestion and enhances overall performance. In this lab exercise, you will create and execute a program emulating the Link State Algorithm, showcasing its operation and gaining firsthand experience with its routing capabilities.

1.1 Objectives

- Acquire hands-on experience with the Link State Routing Algorithm through its implementation within a simulated network setting.
- Explore the core concepts underlying the Link State Routing Algorithm, such as distributing network structure details and computing the most efficient paths.
- Evaluate the behavior of Dijkstra's algorithm across various network topologies, ranging from small-scale to large-scale setups.
- Assess performance metrics related to memory usage and processing time to gauge the algorithm's efficiency.

2 Theory

The Link State Algorithm, rooted in the concept of shortest-path routing, identifies the most efficient route within a network. While multiple paths may exist to reach a destination, the shortest path is defined by the fewest hops or the lowest cost, considering metrics like bandwidth, delay, or congestion. To establish this, the algorithm constructs a comprehensive network map and employs Dijkstra's algorithm to compute the shortest path to all nodes.

Each router within the network maintains a repository containing details about link statuses, encompassing metrics such as bandwidth and delay. Utilizing this information, routers build and store a network topology map. Subsequently, Dijkstra's algorithm is executed by each router to determine the shortest paths to all network nodes.

In instances of link state changes, like link failures or new link additions, affected routers update their databases and propagate this information to all other routers. Consequently, each router updates its database and reevaluates the shortest paths using Dijkstra's algorithm.

Compared to alternatives like the Distance Vector Algorithm, the Link State Algorithm offers several advantages. It effectively manages large-scale networks with numerous nodes and operates more efficiently by broadcasting only updates to network topology, reducing network congestion and enhancing overall performance.

3 Methodology

1. **Devise a network structure:** Define the configuration of routers, their connections, and the associated costs for each link.

2. **Develop a network simulation program:** Utilize a programming language to construct a simulator encompassing essential elements such as routers, connections, data packets, and routing protocols.

3. **Implement the Link State Algorithm:** Integrate the requisite code into the simulation program to enact the Link State Algorithm. This entails generating Link State Packets (LSPs) for individual routers and disseminating them to neighboring routers through flooding. Subsequently, each router calculates the shortest paths to all other nodes in the network using Dijkstra's algorithm.

4. **Validate program functionality:** Employ the simulator to verify the efficacy of the Link State Algorithm by altering the network structure and assessing the resulting shortest paths. This can involve implementing a timer mechanism to periodically modify link costs.

5. **Evaluate algorithm performance:** Document the algorithm's time complexity and memory utilization across diverse network topologies, contrasting these metrics with those of alternative routing algorithms. This assessment entails measuring the time required for shortest-path calculations and quantifying memory consumption for storing network topologies.

6. **Display incremental path updates:** Throughout the simulation process, output the evolving path updates for each node, facilitating the monitoring and validation of path computation progress.

4 About the Program

Here, we present a thorough outline of our program. We begin by introducing the definitions of variables and functions. Following that, we delve into a detailed explanation of their roles and functionalities within the program.

4.1 Functions

4.1.1 broadcast(message:bytes,neighbors: list[str]):

Sends a message to all neighboring nodes. The message is in bytes, and the neighbors are specified as a list of strings representing their identifiers.

4.1.2 get_adj(*router_id*):

Retrieves the adjacency list for the specified router. This list contains information about the directly connected neighbors and the cost of the links to them.

4.1.3 shortest_path(*parents, start, target*):

Determines the shortest path from the start node to the target node using the information in the 'parents' data structure, which is typically the output of a pathfinding algorithm like Dijkstra's.

4.1.4 dijkstra(*graph, start*):

Implements Dijkstra's algorithm to compute the shortest paths from the start node to all other nodes in the graph. The graph is represented as a data structure containing nodes and edges with associated costs.

4.1.5 encode_message(*router_id: str, serial: int, adj: dict[str, dict[str, int]], ttl=32) -¿ bytes:*

Encodes a message containing the router's identifier, a serial number, and its adjacency list into bytes. The 'ttl' parameter represents the time-to-live for the message.

4.1.6 decode_message(*message: bytes*):

Decodes a message from bytes back into its constituent parts, such as the router's identifier, serial number, and adjacency list.

4.1.7 updateLinkWeight():

Updates the weights of the links in the graph, which may be necessary due to changes in the network or dynamic routing decisions.

4.1.8 send_msg():

A generic function to send a message across the network. The specifics of the message and its destination are not detailed here.

4.2 Flow of the program

The program initializes various parameters and configurations essential for routing purposes. These include unique identifiers for routers, their respective IP addresses, and the network topology specified within individual text files corresponding to each router. The network's structure is represented through an adjacency list, where each entry denotes the weight or cost associated with the connection between two routers. The primary function of the program revolves around constructing an initial graph based on the provided network topology file. It continuously updates routing information by employing Dijkstra's algorithm to compute the shortest paths to all other routers within the network whenever there is a modification in link weights and upon receiving updated information. Subsequently, this routing data is

disseminated to neighboring routers through broadcast transmissions.

To efficiently manage the tasks at hand, the program operates in a multi-threaded manner. It handles incoming packets, updates link weights at intervals of 30 seconds, and facilitates communication with neighboring routers. Upon receipt of a packet from a neighboring router, the program promptly updates its adjacency list and triggers a broadcast to notify other routers of the alteration.

5 Experimental Result

The experiments involved a fixed number of nodes within a network, each with diverse link configurations. The network topology underwent modifications every 30 seconds through the utilization of the updateLinkWeight() function, aimed at assessing the efficacy of the Link State Algorithm across different network scenarios.

The findings revealed that the Link State Algorithm adeptly computed the shortest path in all tested scenarios. Furthermore, experiments were conducted to assess the algorithm's performance under network congestion conditions. The results indicated the algorithm's ability to adapt to evolving network conditions while consistently providing the shortest path. It demonstrated robustness and reliability across all cases.

In summary, the experiments underscored the effectiveness and efficiency of the Link State Algorithm in determining the shortest path within a network. Its versatility extends to various applications, including routing protocols in computer networks, transportation systems, and logistics optimization.

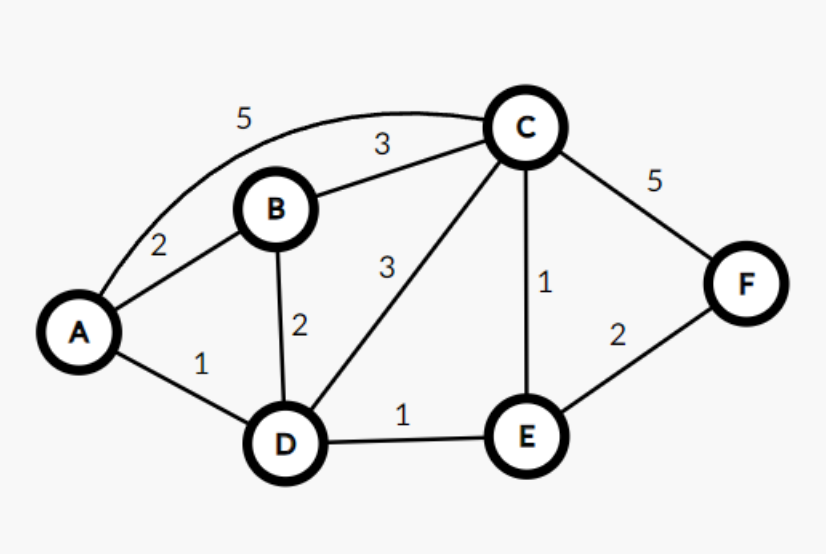

Figure 1: Initial Graph

5.1 Demonstrating Console Outputs

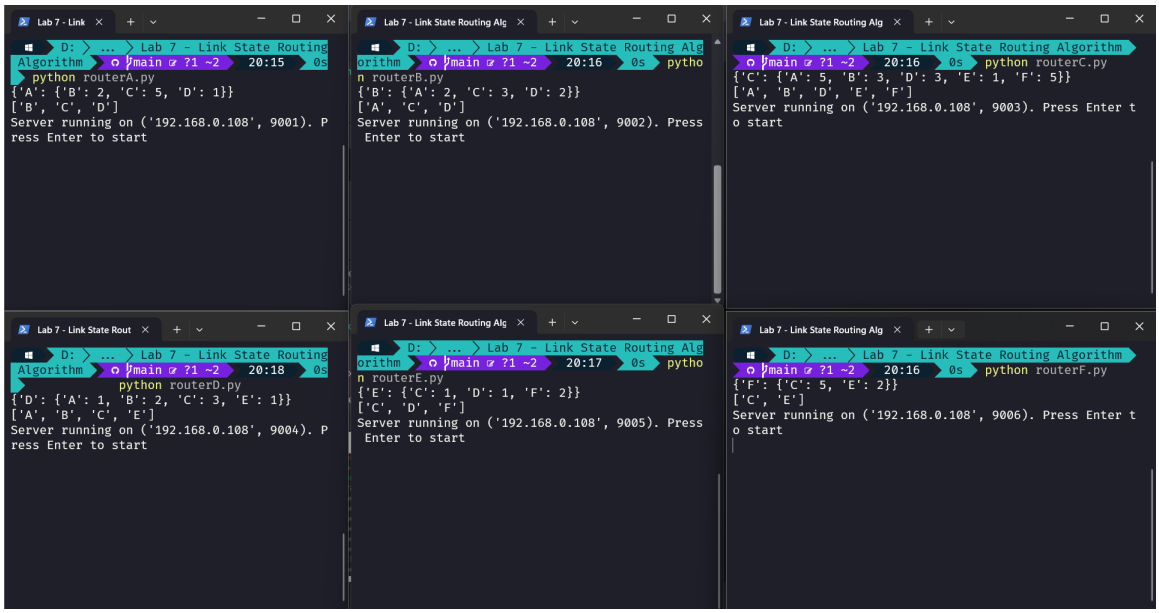

Figure 2: Initializing the routers

```
A → A: ['A']
Distance: 0

A → B: ['A', 'B']
Distance: 2

A → C: ['A', 'D', 'E', 'C']
Distance: 3

A → D: ['A', 'D']
Distance: 1

A → E: ['A', 'D', 'E']
Distance: 2

A → F: ['A', 'D', 'E', 'F']
Distance: 4
```

Figure 3: Showing shortest distance information at a router

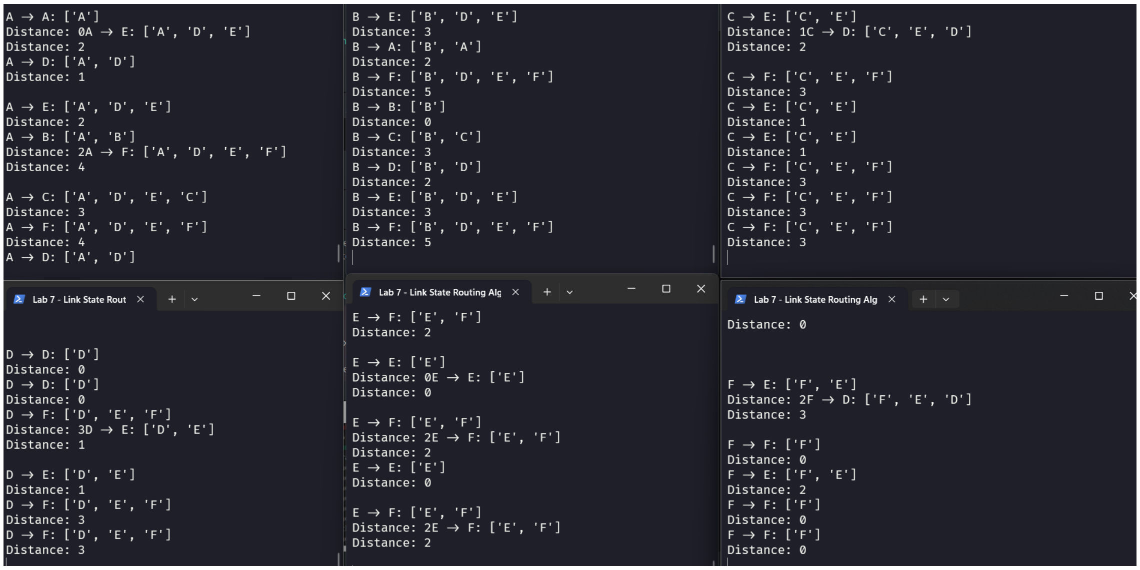

Figure 4: Showing all pair shortest path

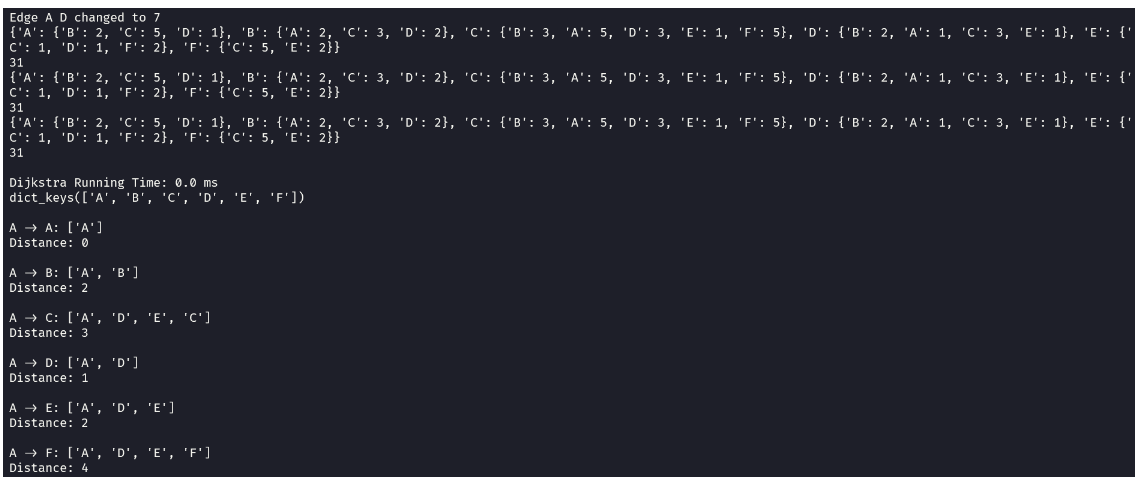

Figure 5: Recalculation after update

5.2 Graphical Analysis of the Algorithmic Performance

5.2.1 Number of Nodes VS. Time

The graph below illustrates the correlation between the quantity of nodes and the duration necessary for executing Dijkstra's Algorithm, measured in milliseconds.

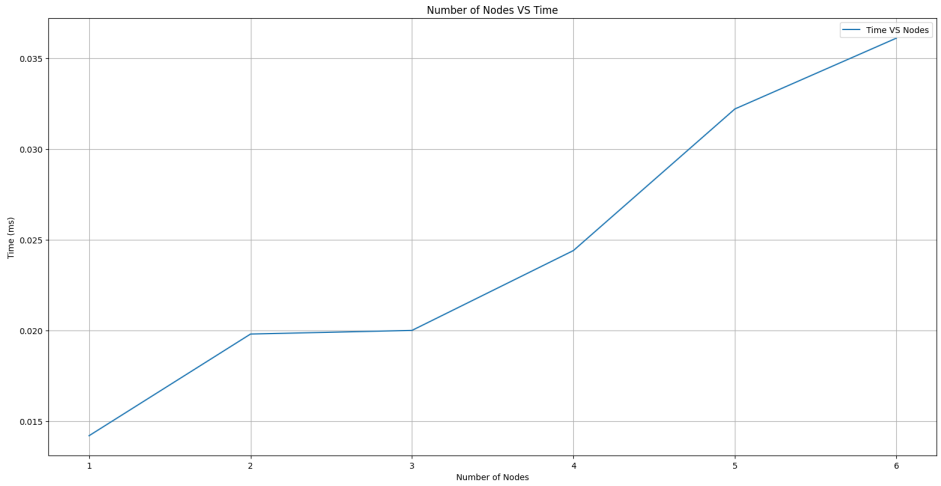

Figure 6: Effect of the load of nodes on the time complexity

5.3 Number of Nodes VS. Memory Usage

The graph below illustrates the connection between the number of nodes and the additional memory capacity needed for executing Dijkstra's Algorithm, measured in bytes.

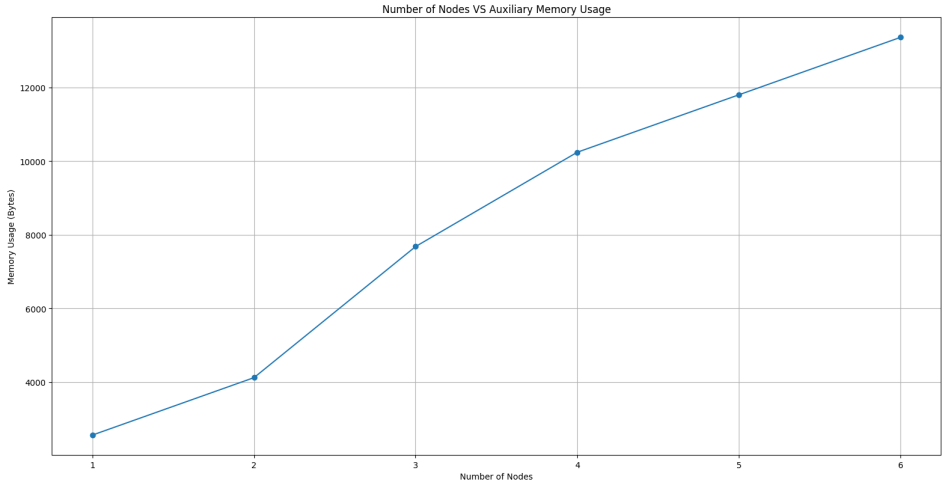

Figure 7: Effect of the load of nodes on consumed memory

The provided code implements Dijkstra's algorithm to determine the shortest path in a weighted graph. The time complexity of this algorithm is denoted as O((E+V)logV), where E represents the number of edges and V denotes the number of vertices in the graph. This implementation utilizes a heap data structure to efficiently manage the minimum cost node and its neighboring nodes, resulting in a logarithmic time complexity for accessing the minimum cost node.

The memory usage of the algorithm is influenced by the graph's size and the data structures employed to store the graph and intermediate outcomes. In this implementation, a dictionary

data structure represents the graph, with memory usage proportional to the number of edges and vertices. Additionally, the algorithm employs a priority queue (heap) to store nodes alongside their respective costs, contributing to the overall memory consumption.

Regarding network communication, the algorithm communicates with its neighbors to update its distance table. The volume of network traffic generated by the algorithm hinges on the update frequency and the number of neighbors present in the graph.

In essence, the algorithm's time and memory complexity are contingent on the graph's scale and the update frequency. While Dijkstra's algorithm proves efficient for small to medium-sized graphs in practice, it may become impractical for very large graphs featuring millions of nodes and edges.

6 Discussion

A significant hurdle involved managing network interactions among the nodes effectively. This process necessitated utilizing sockets, which brought forth several connection and message transmission complications that demanded attention. We undertook thorough testing and refinement of the network communication capabilities to tackle these issues, which included managing exceptions and mistakes.

Additionally, we faced the task of dealing with spontaneous changes to the edges within the graph. It was crucial to maintain the graph's integrity and prevent these modifications from disrupting Dijkstra's algorithm. To overcome this obstacle, we established an independent thread dedicated to graph updates and integrated necessary validations to preserve the graph's integrity.

7 Conclusion

To sum up, our study provided valuable insights into the deployment of the Link State Algorithm (LSA) within computer networks. Our practical tests demonstrated that LSA is adept at identifying the most direct routes between network nodes by keeping a precise and current representation of the network layout. The Dijkstra's Link State routing algorithm is a fundamental element in networking, delivering a reliable and superior method for pinpointing the most efficient routes across the network. Its importance is underscored by its rapid computation of the shortest path from one node to all others, an essential operation in routing protocols. Ultimately, Dijkstra's algorithm is crucial for enhancing network efficiency and ensuring smooth data flow among connected devices.