Name: **Rohan Prasad**
Email: rpp5524@psu.edu
PSU ID: **980707395**

Date: 12/17/2024

CSE 514
Computer Networks

CSE 514 Final Project Report

Title - Simulation of Routing Algorithms and Optimizations

INDEX

Introduction

Computer networks' dependability and performance depend on effective routing. By determining the optimal data transfer pathways, routing algorithms affect crucial variables including latency, throughput, and network utilization. recognizing the trade-offs and performance of these network algorithms as they grow in size and complexity.

By providing useful advice for choosing routing protocols in actual networks, the main experiments and conclusions seek to bridge the gap between theoretical knowledge and real-world application.

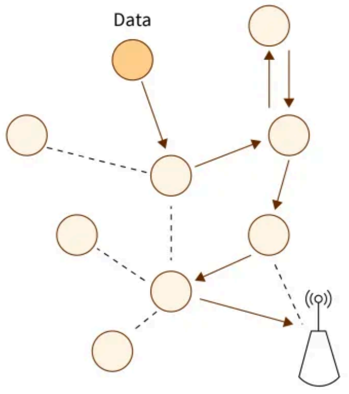


Figure 1. A Sample Network

Abstract

This project focuses on simulating and analyzing three core routing algorithms: **Distance Vector, Link State, and Path Vector**. These algorithms are evaluated across different network topologies, such as *linear, mesh, tree*, and others, to observe their behavior under varying conditions, including changes in network size and structure.

The study looks at critical performance parameters in a real-world network. This research shows the strengths and limitations of each method by visualizing and comparing the results, revealing information about their scalability and efficiency.

Interesting Aspects

Interesting Aspects of the project include the implementation of diverse routing algorithms, automated testing across a range of topologies, and visualization of performance trends. This project guides one on ***how network structure influences algorithmic efficiency and scalability, particularly in dynamic or large-scale systems***.

Salient Results

The salient results reveal distinct trade-offs among the algorithms. While Link State consistently demonstrated lower latency and better throughput in structured topologies like mesh and star, it incurred higher routing overhead due to global link-state flooding. Distance Vector showed efficiency in smaller networks but struggled to scale in larger configurations due to convergence delays. Path Vector performed well in preventing routing loops but exhibited increased execution time and overhead in dense topologies.

Significant Findings

A particularly significant finding is the impact of network topology on algorithm performance.

- For instance, tree topologies resulted in prolonged convergence times for Distance Vectors due to their hierarchical nature
- Mesh networks favored Link State for faster path discovery.
- Additionally, Path Vector demonstrated robustness in handling dynamic route updates but at the cost of higher overhead.

Main Idea of the Project / Problem Statement

The main goal of this project is to evaluate and compare the performance of three fundamental routing algorithms:

- Distance Vector algorithm
- Link State algorithm
- Path Vector algorithm

… across diverse network topologies. As modern computer networks become increasingly complex, selecting the most efficient routing algorithm is critical to ensuring optimal performance, scalability, and resilience.

This study addresses the following key problem:
How do routing algorithms perform in terms of latency, throughput, routing overhead, and network utilization across different network topologies and varying node sizes?

By simulating these algorithms on multiple topologies (e.g., mesh, tree, star, and grid) and analyzing their behavior for networks ranging from 10 to 200 nodes, the project aims to identify:

- The strengths and weaknesses of each algorithm.
- How network topology impacts algorithm efficiency and convergence time.
- Which algorithm is best suited for specific network conditions or requirements.

Impetus for pursuing this Project

The conclusions of this study will provide insights into the trade-offs between computing efficiency, resource utilization, and scalability, as well as practical suggestions for implementing routing protocols in real-world networks.

It aligns with my research interests in computer networks, specifically in understanding and optimizing data routing mechanisms. Previously, I worked on a project involving **Parallel Max Flow Network Algorithms**, which closely relates to concepts in computer networks, particularly in **Flow Control** and **Congestion Control**. That experience sparked my curiosity about how network traffic is managed efficiently in dynamic environments.

I feel that this project will not only build on my former expertise, but will also increase my knowledge of routing protocols and network optimization, complementing my previous computer network projects and research efforts.

Tools Used

1. **Python** is used as the primary programming language to implement and simulate the routing algorithms. Python 3.9.2
2. **NetworkX is** a Python library utilized for creating and managing network topologies (e.g., linear, mesh, tree, ring, grid) and applying graph-based computations.
3. **Pandas** is Used for organizing, processing, and analyzing simulation results, as well as for reading and writing data to CSV files for further analysis.
4. **Bash Scripting** is used to automate the process by modifying parameters and running experiments efficiently.

Figure 2. Tools Used

Details

Project System Design

The system was divided into the following components:

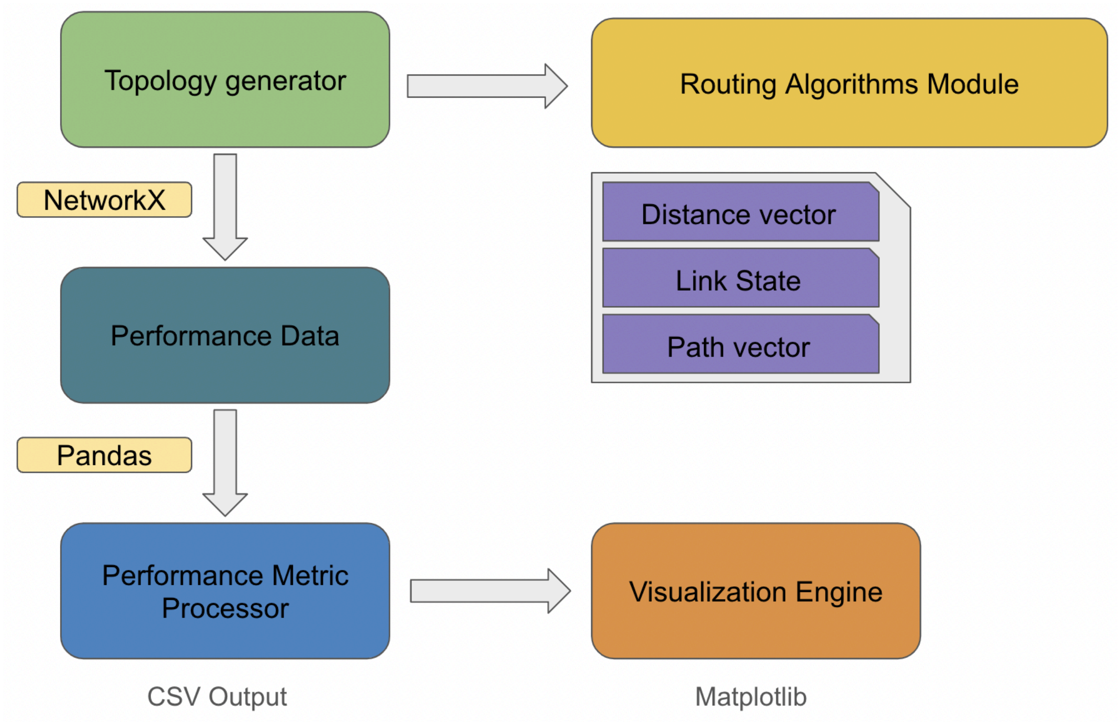


Figure 3: Routing Algorithm Simulation System Design

Network Topology Generator

This section of the project generates topologies dynamically using NetworkX for graph-based simulations. This generator constitutes 2 parts:

- Topology selector
- Adding Edges with:
    - Constant weight initialization
    - Random weight initialization
    - Real world weight initialization

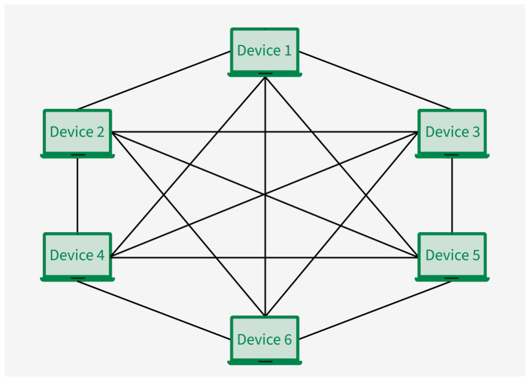


Figure 4: Mesh Topology

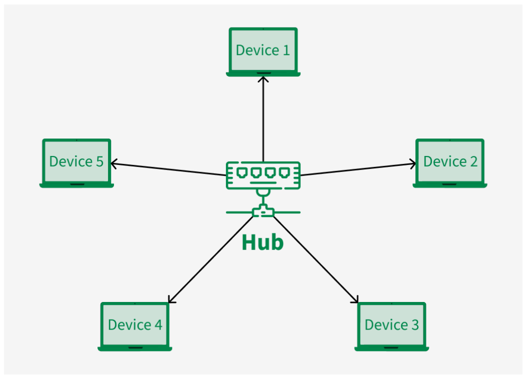


Figure 5: Star topology

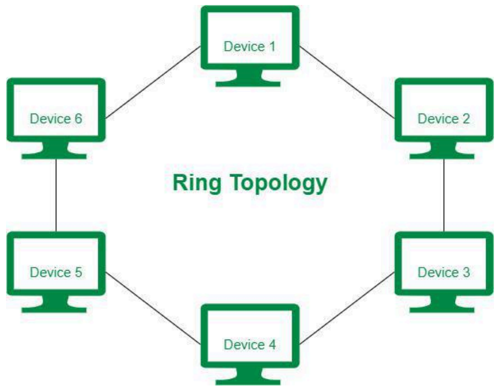


Figure 6: Ring Topology

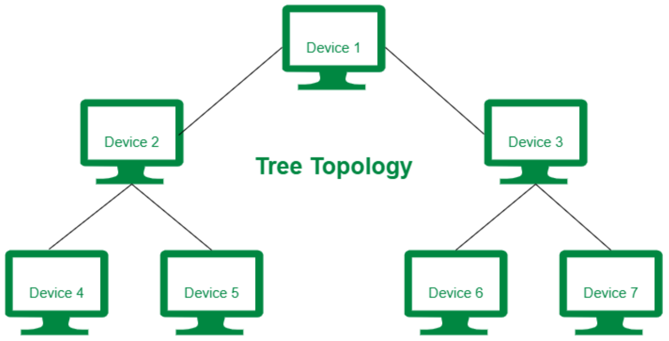


Figure 7: Tree topology

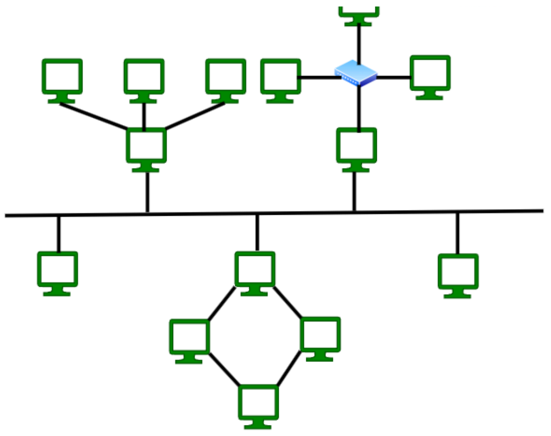


Figure 8: Hybrid Topology

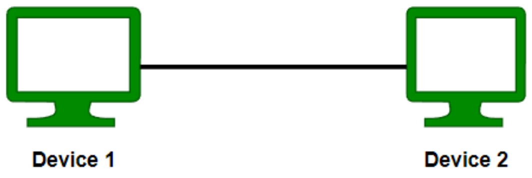


Figure 9: Linear topology

Real World Weight Initialization

I have come up with a way to initialize weights to networks to simulate real world networks that exist. This is how I have done it:

$$Weight = Distance + Congestion\, Factor/Bandwidth$$

The logic behind thisis to simulate real-world network conditions by combining multiple factors that affect link performance.

In real-world networks, physical distance plays a critical role in determining link performance. Higher bandwidth links are capable of transferring data more quickly, meaning they have lower weights in routing calculations.

The inverse relationship is modeled as:

$$Weight \propto 1/Bandwidth$$

To give you an idea: A link with 500 Mbps has a smaller contribution to weight than a link with 50 Mbps, as it can handle more data simultaneously.

Congestion represents additional delays caused by network traffic or load on a particular link. A congestion multiplier (randomly between 1 and 3) is used to simulate this.

Routing Algorithm Modules

These modules are implementations of the three algorithms were modularized and parameterized for flexibility:

Distance Vector Algorithm.

The Distance Vector Routing Algorithm is a **decentralized algorithm** where each node (router) maintains a routing table containing the **distance (cost)** and **next-hop information** to reach every other node in the network. Nodes periodically exchange their routing tables/path information with their neighbors.

$$Cost(N \rightarrow D) = neighbor\, Mmin(Cost(N \rightarrow M) + Cost(M \rightarrow D))$$

```
# Step 1: Initialize distance to self as 0
for node in nodes:
    distance[node][node] = 0
    for neighbor, weight in graph[node].items():
        distance[node][neighbor] = weight
        next_hop[node][neighbor] = neighbor

# Step 2: Bellman-Ford Update Rule
converged = False
while not converged:
    converged = True
    for u in nodes:
        for v in nodes:
            for neighbor in graph[u]:
                if distance[u][neighbor] + distance[neighbor][v] <
distance[u][v]:
                    distance[u][v] = distance[u][neighbor] +
distance[neighbor][v]
                    next_hop[u][v] = next_hop[u][neighbor]
                    converged = False

# Step 3: Return Routing Tables
return distance, next_hop
```

Algorithm 1: Distance Vector Algorithm (Bellman Ford Update)

Link State Algorithm

The shortest path cost D(v) from the source node S to a node v is updated as:

$$D(v) = u \in N min(D(u) + w(u, v))$$

```
distance = {node: float('inf') for node in graph}
previous = {node: None for node in graph}
distance[source] = 0
visited = set()
pq = [(0, source)]

while pq:
    current_distance, current_node = heapq.heappop(pq)
    if current_node in visited:
        continue
    visited.add(current_node)

    for neighbor, weight in graph[current_node].items():
        new_distance = current_distance + weight
        if new_distance < distance[neighbor]:
            distance[neighbor] = new_distance
            previous[neighbor] = current_node
            heapq.heappush(pq, (new_distance, neighbor))

return distance, previous
```
Algorithm 2: Link State Algorithm (Follows Dijkstra's Algorithm)

Path Vector Algorithm

```
nodes = list(graph.keys())
paths = {node: {n: (float('inf'), []) for n in nodes} for node in nodes}

# Step 1: Initialize direct neighbors
for node in nodes:
    paths[node][node] = (0, [node])
    for neighbor, weight in graph[node].items():
        paths[node][neighbor] = (weight, [node, neighbor])

# Step 2: Path updates
converged = False
while not converged:
    converged = True
    for u in nodes:
        for v in nodes:
            for neighbor in graph[u]:
                current_cost, current_path = paths[u][v]
                neighbor_cost, neighbor_path = paths[neighbor][v]
                new_cost = paths[u][neighbor][0] + neighbor_cost
                if new_cost < current_cost:
                    paths[u][v] = (new_cost, paths[u][neighbor][1] +
neighbor_path[1:])
                    converged = False

# Step 3: Return Path Tables
return paths
```
Algorithm 3: Path Vector Algorithm (Path Propagation)

Simulation Automation

Bash Scripting is used to automate experiments, run algorithms on different topologies, and generate results across varying node sizes.

1. The Bash script initializes simulation parameters, such as network sizes, topology types, and the algorithms to be executed.
2. The script generates the network graph using the Python NetworkX library.

3. The script iterates through each routing algorithm (Distance Vector, Link State, and Path Vector) and runs simulations for varying node sizes (10 to 200).
4. Capture metrics like latency, throughput, and routing overhead
5. Results are analyzed using Python scripts with Pandas, and performance trends are visualized with Matplotlib.
6. Logs, outputs, and graphs are saved for further analysis and reporting.

Performance Metric Processor Module

This is a python module written to assess data after an algorithm has been run. The algorithm returns JSON data in the form of nodes with its computed shortest path weights to the rest of the nodes.

Metric	Definition	Units	Significance
Latency	Total time for a packet to travel from source to destination	Milliseconds (ms)	Indicates how quickly packets are delivered.
Throughput	Rate of successful data transmission	Megabits per second (Mbps)	Measures network efficiency.
Routing Overhead	Number of control messages exchanged	Packets	Indicates the cost of maintaining routing tables.
Network Utilization	Ratio of bandwidth used to total available bandwidth	Percentage (%)	Measures network resource efficiency.
Execution Time	Total time for the algorithm to compute and converge	Seconds (s)	Indicates computational efficiency.

Metric Formulas

$$\text{Latency (ms)} = \sum weights \;/\; node\ pairs$$

$$\text{Throughput (Mbps)} = \sum(packets\_per\_second \;*\; packet\_size \;*\; 8) \;/\; (total\ time \;*\; 10^6)\ \text{Mbps}$$

$$\text{Routing Overhead (control messages)} = number\ of\ nodes \;*\; number\ of\ edges \;*\; 2$$

$$\text{Utilization} = (\sum(packets\_per\_second \;*\; packet\_size \;*\; 8) \;/\; total\_available\_bandwidth) \;*\; 100$$

Execution Time (s) = Result of execution of the bash script which monitor the time it took to run the algorithm for n nodes.

Assumptions

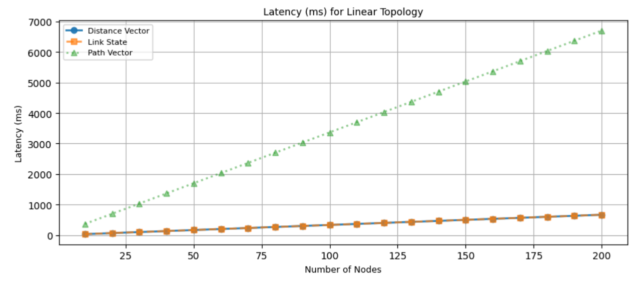
Several assumptions were made to standardize the test environment and focus on routing algorithm behaviors:

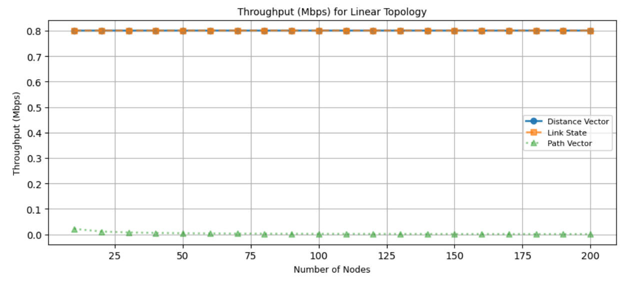
1. Link Costs: All links were initialized with either **constant weights (default: 10) or random weights** within a specified range for diversity.
2. Bandwidth: Each link was assumed to have a default **bandwidth** of **100 Mbps**.
3. Packet Size and Traffic: **Packets** were assumed to be **1000 bytes** in size, with a packet transmission rate of 100 packets per second.
4. Ideal Network Conditions: The network was assumed to have no packet loss or errors to isolate the impact of routing algorithms on performance
5. I am also assuming that in the performance analysis module, the routing overhead is the **number of messages exchanged to build routing tables**. (control messages essentially like initiate request messages or ACKs)
6. For the tree topology, data has been analyzed for a maximum of 100 nodes, the reason being that the computation time for performing Distance vector and Path vector algorithms exceed 10 hours beyond this node size.

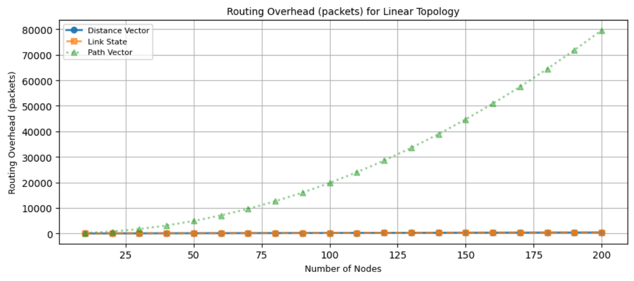
Experimental Results

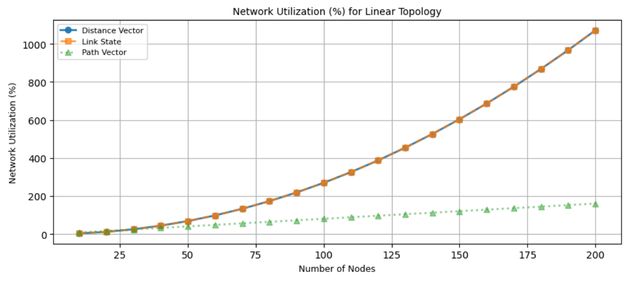
Graphs and Visualizations

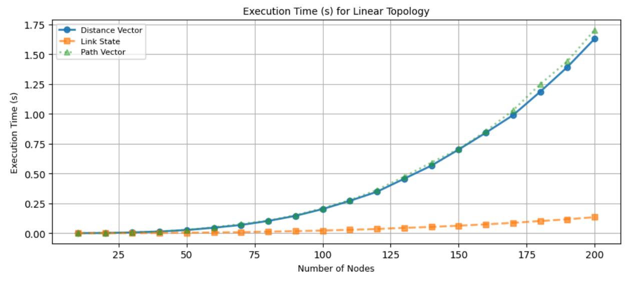
Linear Topology







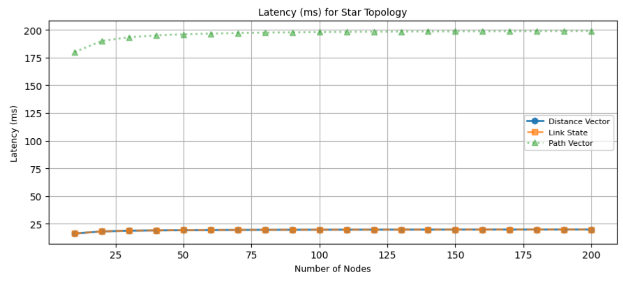


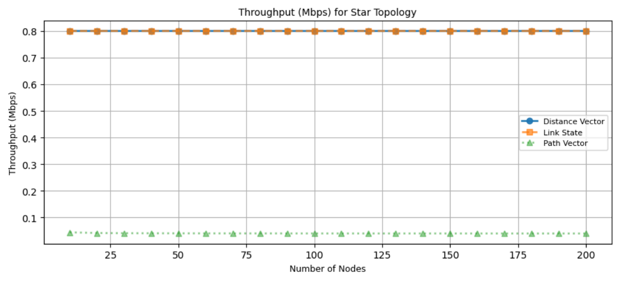


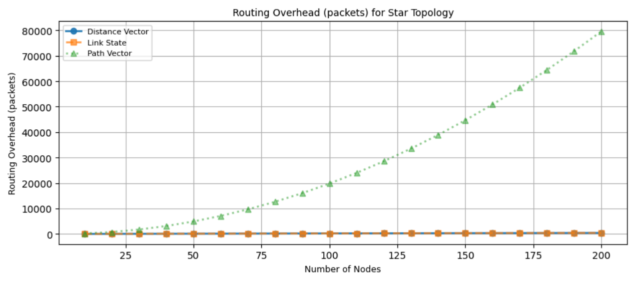
For a linear topology, **Link State Routing is generally the best algorithm**. This is because Link State utilizes **Dijkstra's algorithm** to compute the shortest paths efficiently, leveraging its global knowledge of the network. Since a linear topology has minimal complexity and predictable link relationships, the **overhead of flooding link-state packets is relatively low** compared to more complex topologies.

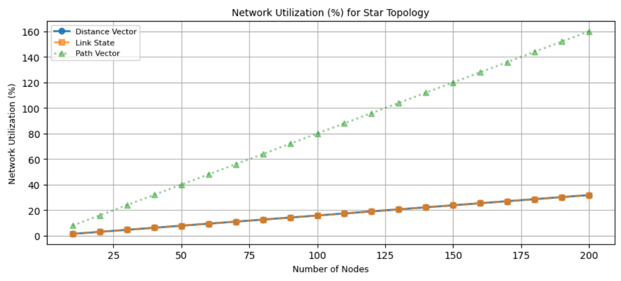
In contrast, **Path Vector**, while robust for preventing loops, **introduces unnecessary overhead** in this simple topology due to the transmission of entire paths. **Therefore, Link State** achieves the best balance of low latency, fast convergence, and minimal routing overhead in linear networks.

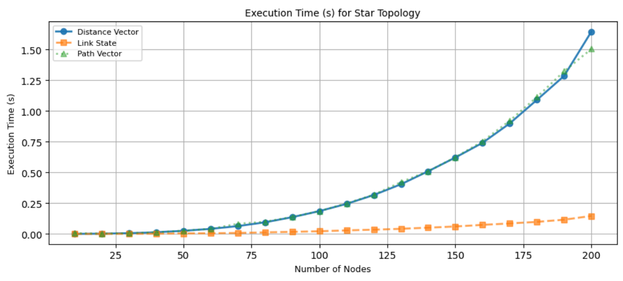
Star Topology







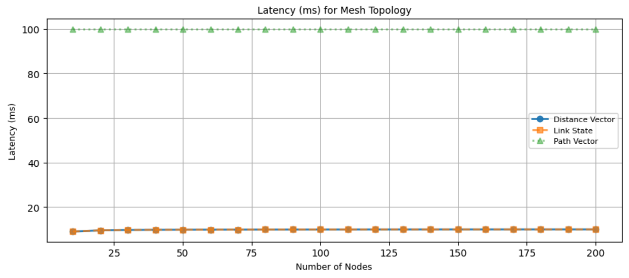


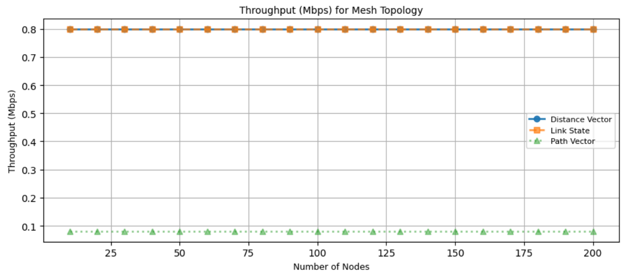


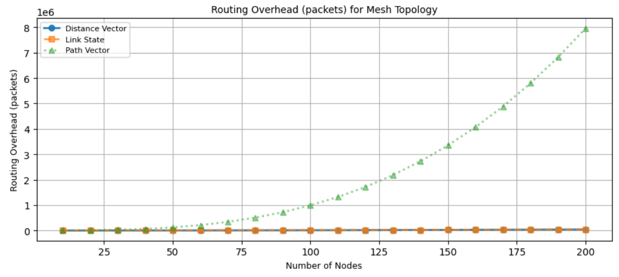
For a star topology, **Distance Vector Routing** is the most efficient algorithm. In a star configuration, all nodes are directly connected to a **central hub**, which simplifies routing decisions and minimizes the number of hops. Distance Vector performs well in this scenario because each node only needs to exchange routing updates with the central hub, resulting in **low overhead** and **fast convergence.**

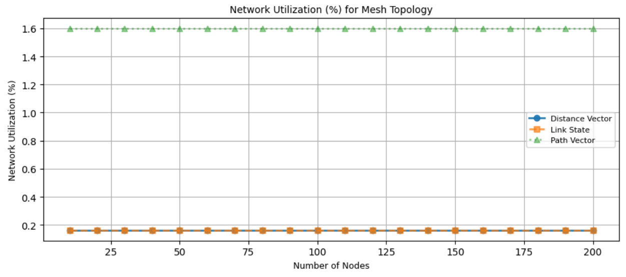
While Link State Routing would also work effectively, the flooding of link-state packets adds **needless overhead** to a basic, centralized topology. Path Vector, intended for more complicated networks, **adds needless expense**. Thus, Distance Vector is the optimum choice for star topologies due to its simplicity and **low communication requirements.**

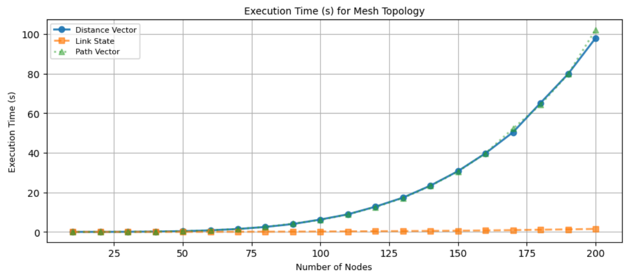
Mesh Topology





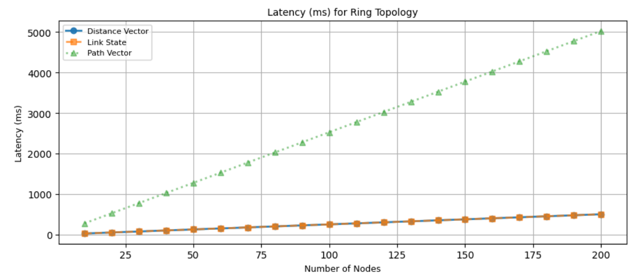


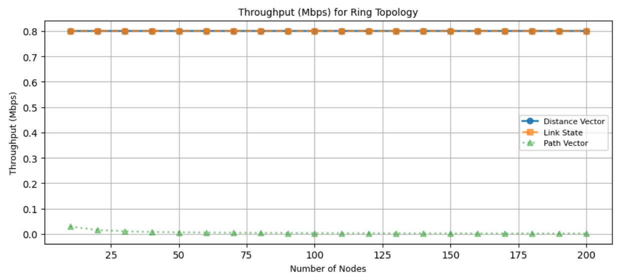


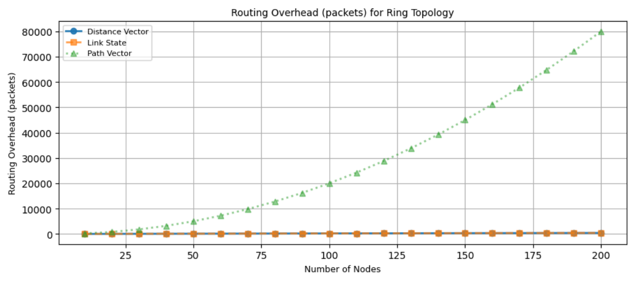


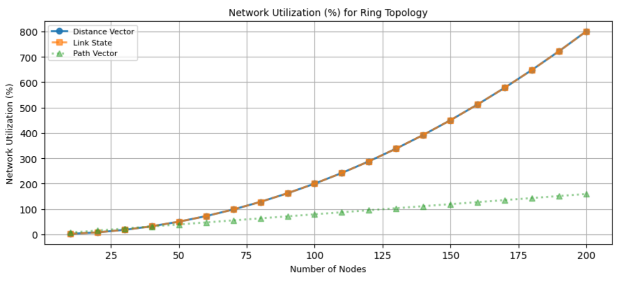
For a mesh topology, **Link State Routing** is the most effective algorithm. Mesh networks are highly connected, **offering multiple paths between nodes**, and Link State's global topology awareness allows it to efficiently compute the shortest paths using **Dijkstra's algorithm**. The algorithm converges quickly and avoids routing loops, which is particularly beneficial in dense, interconnected networks.

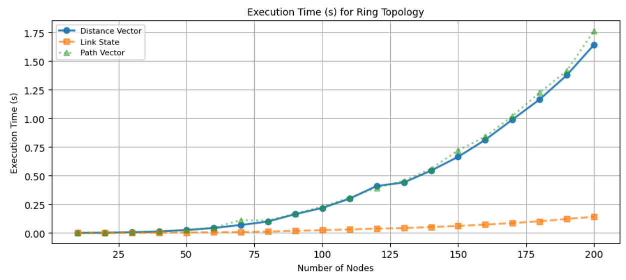
Ring Topology







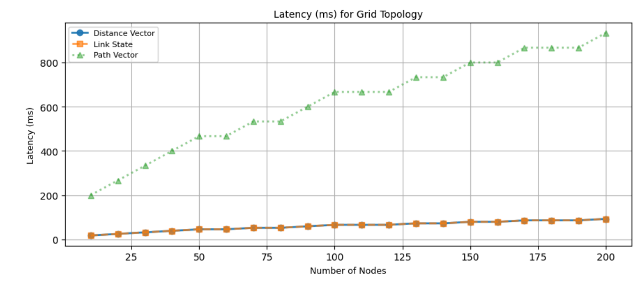


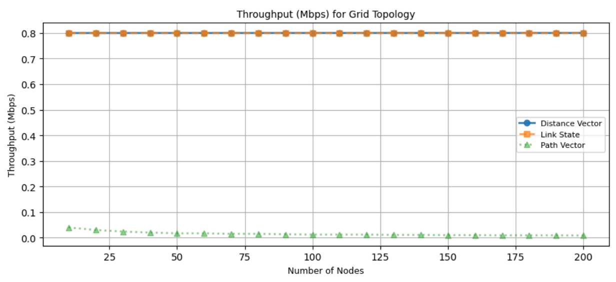


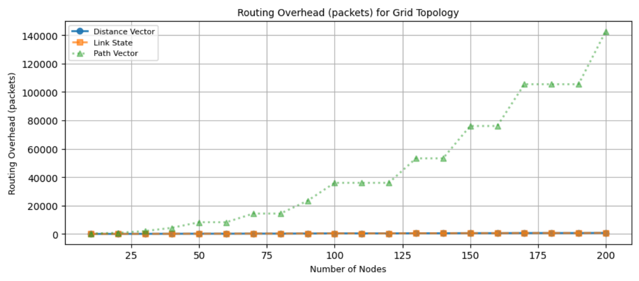
For a ring topology, **Link State Routing** is the most suitable algorithm. In a ring, nodes are connected in a circular manner, and Link State's ability to compute shortest paths using Dijkstra's algorithm ensures efficient routing with minimal latency. The global network view **prevents routing loops**, which are more likely in a ring topology, and allows **quick convergence.**

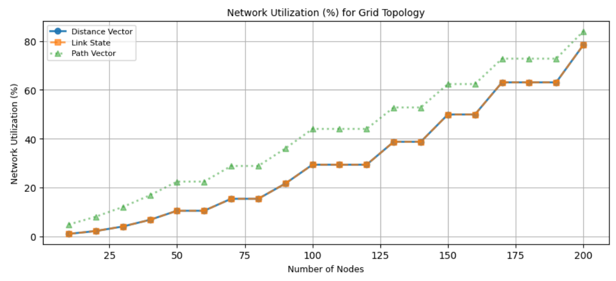
Distance Vector can face challenges such as the **count-to-infinity problem** when a link fails

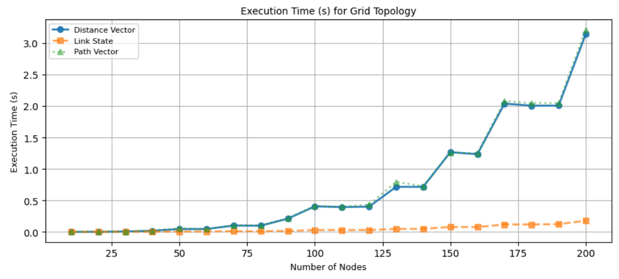
Grid Topology







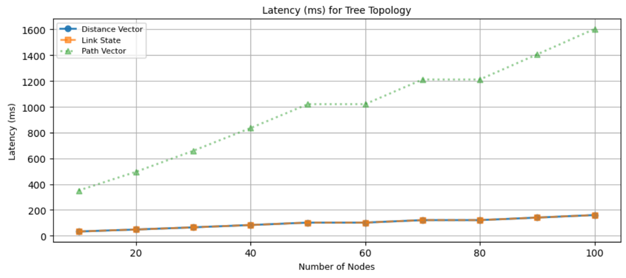


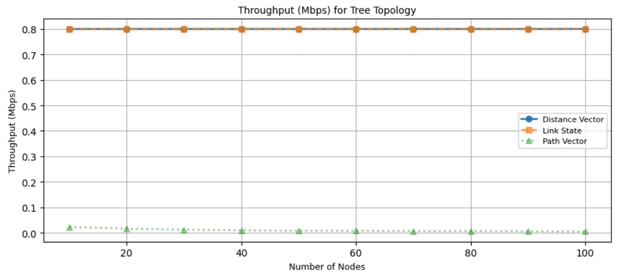


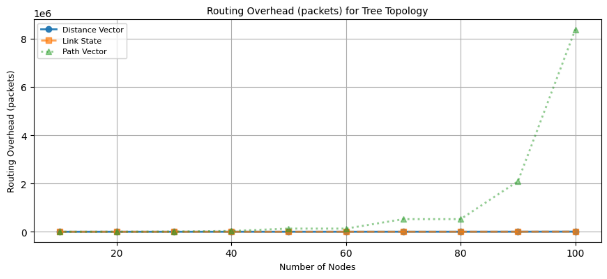
For a grid topology, **Link State Routing is the most efficient algorithm**. In a grid, nodes are arranged in a structured, interconnected pattern with multiple paths between sources and destinations.

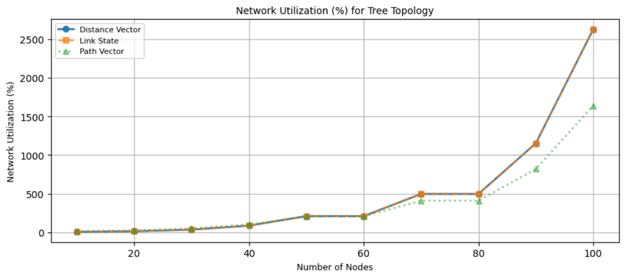
Distance Vector struggles with **slow convergence** and propagating updates across the multiple interconnected paths, while Path Vector incurs **significant overhead due to the transmission of full paths.**

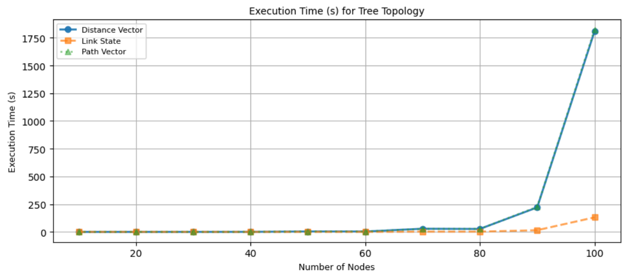
Tree Topology











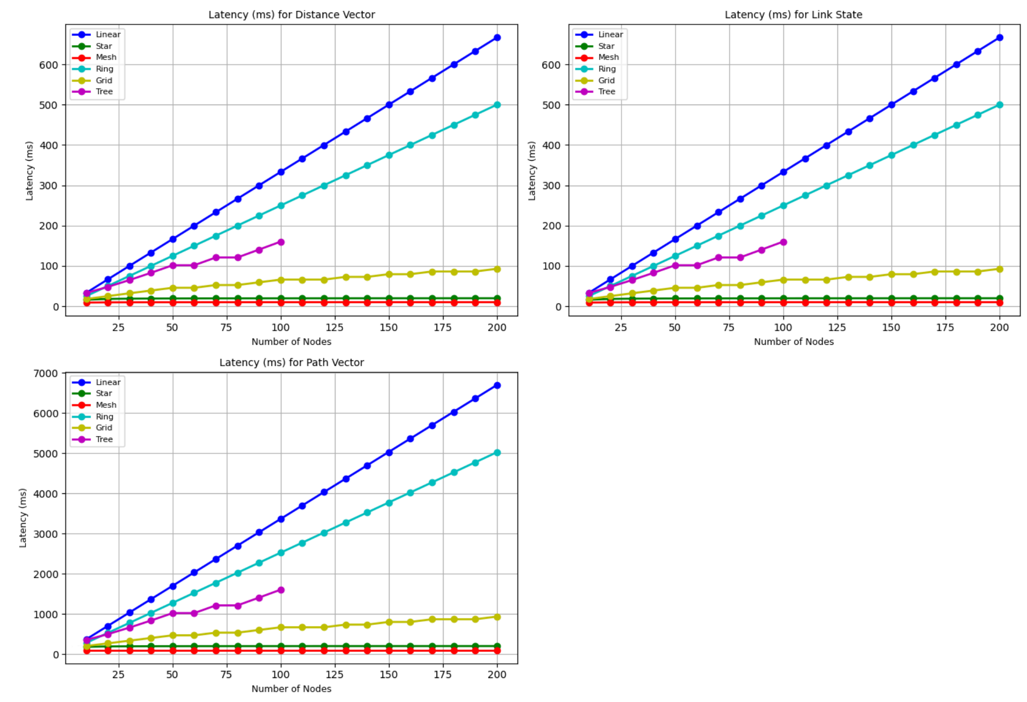
For a tree topology, Link State Routing is the most suitable algorithm. In a tree, nodes are hierarchically connected with a clear parent-child relationship, and Link State's global view of the network ensures efficient route computation using Dijkstra's algorithm. It converges quickly and avoids routing loops, which are critical in tree structures where redundant paths are absent.

Distance Vector struggles in trees due to slower convergence and potential routing loops when link failures occur. Path Vector, while robust for loop prevention, introduces unnecessary overhead

Other Visualizations Grouped by Metric

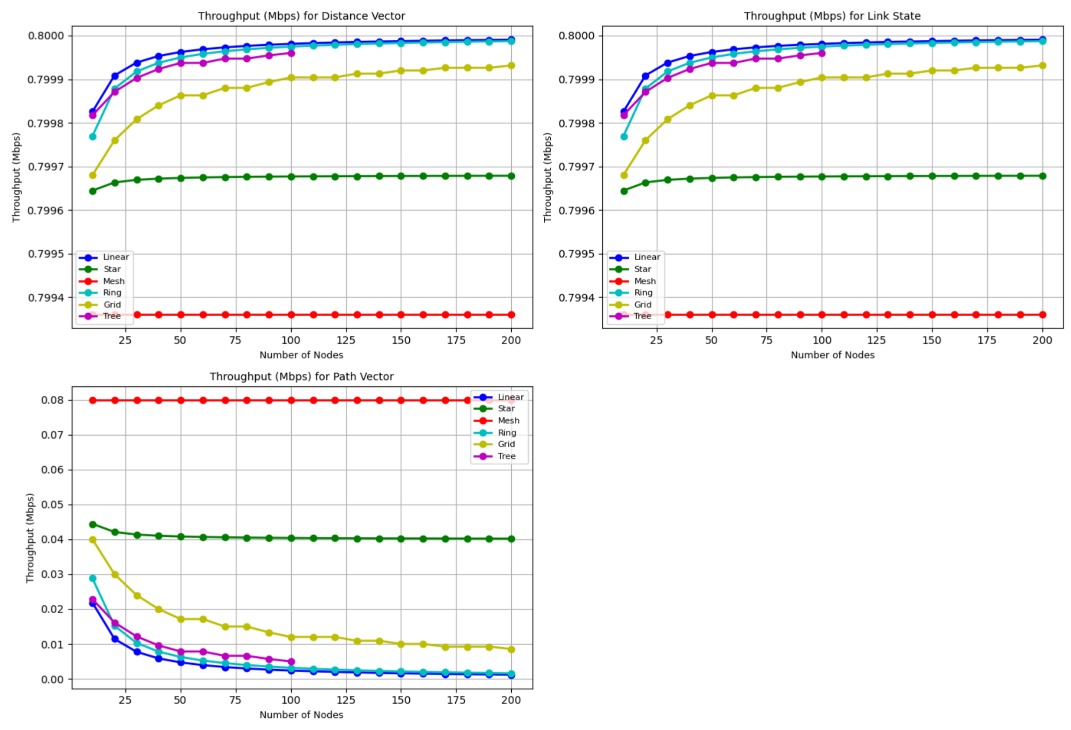
Latency

Latency (ms) Comparison Across Algorithms



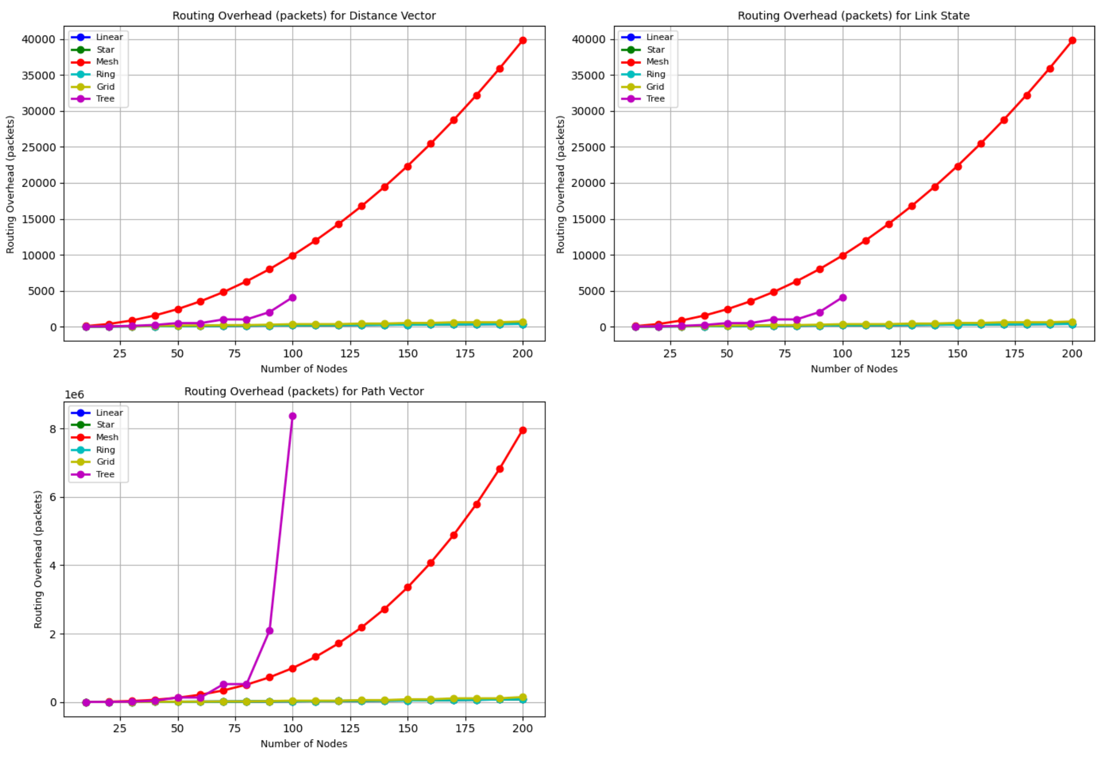
Throughput

Throughput (Mbps) Comparison Across Algorithms



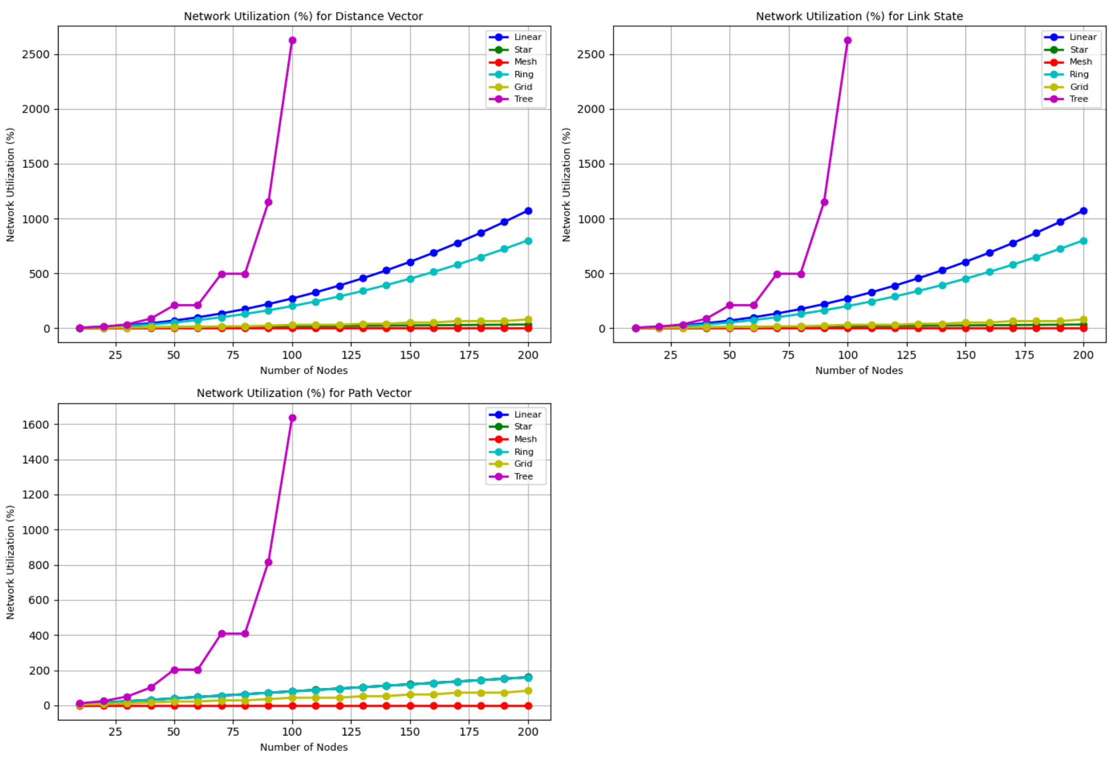
Routing Overhead

Routing Overhead (packets) Comparison Across Algorithms

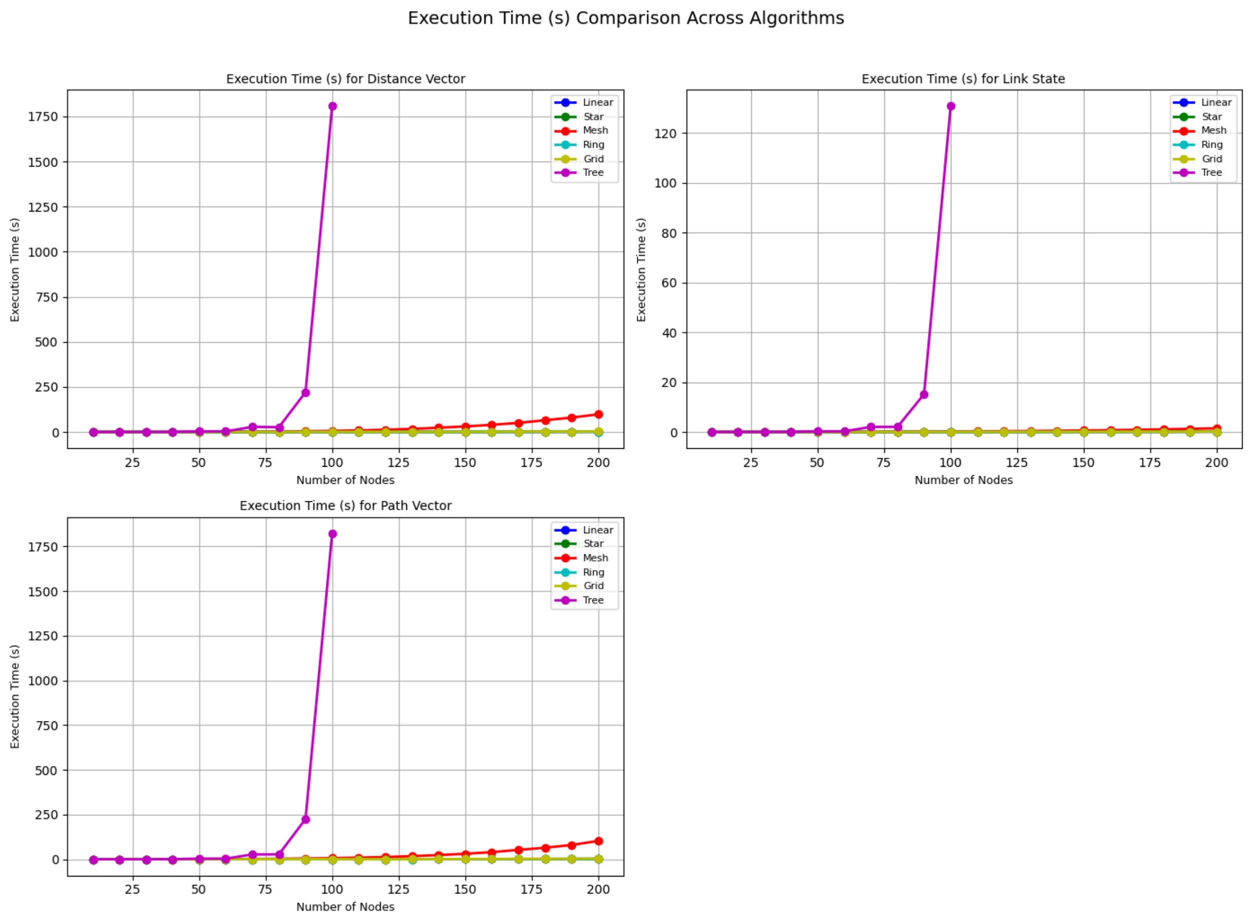


Network Utilization

Network Utilization (%) Comparison Across Algorithms

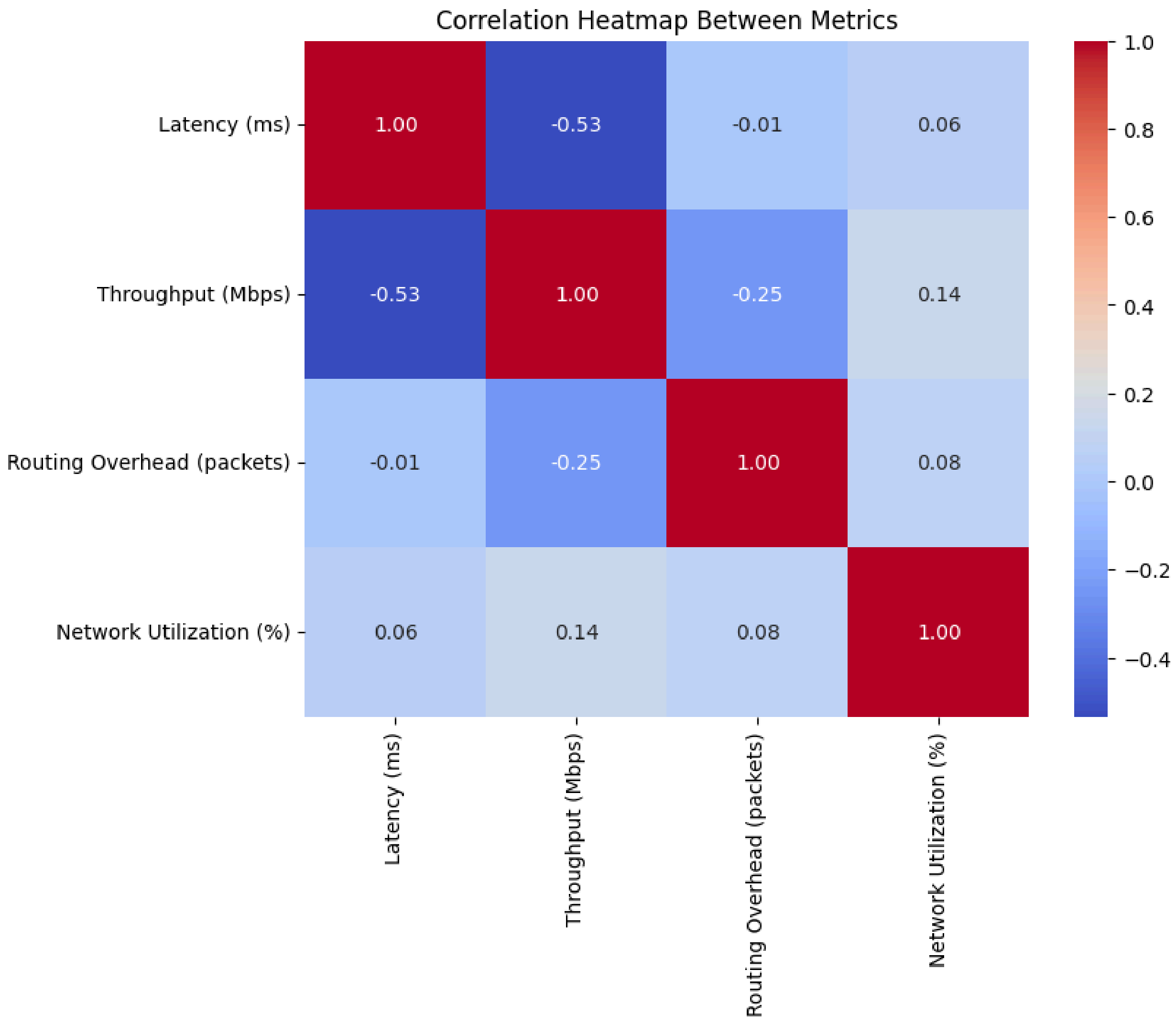


Execution time



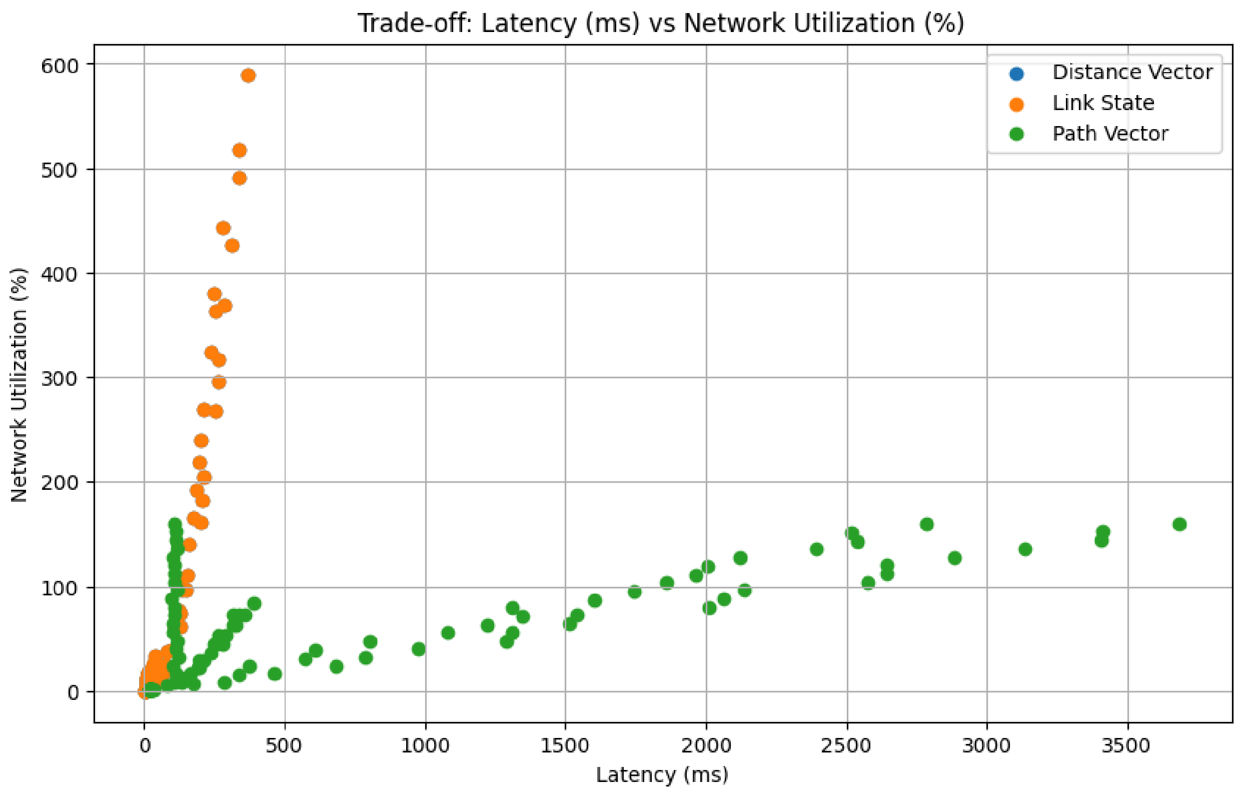
Note that more analysis of networks initialized with random weights have been simulated and provided in the Jupyter notebook in the code base that I have attached, You can also find them in the Github repository: https://github.com/rpp5524/CSE_514_Project_rpp5524

Observations



The correlation heatmap suggests that there is minimal to slight correlation between the factors network utilization and throughput.

Latency and Throughput show the strongest relationship, which aligns with expectations in network performance analysis. The weak correlations involving Routing Overhead suggest that, while it is an important metric, its direct effect on other performance metrics is minimal in this scenario.



This trade-off highlights that **Link State prioritizes speed** but **consumes significant network resources**, while **Path Vector focuses on robustness with higher latency.**

Discussions

A tree topology grows exponentially with the number of levels. For n nodes: A binary tree has a height of approximately $\log_2(n)$. The number of edges increases linearly as n-1, but nodes at lower levels may have more paths to traverse.

Distance Vector and Path Vector algorithms, which rely on repeated table updates or path propagation, can struggle due to the hierarchical nature of trees.

Below is a summary of the Topology choice for the algorithms

Topology	Best Algorithm	Time complexity
Linear	Link State	$O(N^2)$
Star	Distance Vector	$O(N)$
Mesh	Link State	$O(N^2 + E)$
Ring	Link State	$O(N^2)$
Grid	Path Vector	$O(N^2 + E)$
Tree	Link State	$O(N \log N)$

Summary

Algorithm	**Type**	**Routing information**	**Key Technique**	**Pros**	**Cons**
Distance Vector	Decentralized	Distance to destinations	Bellman-Ford Update Rule	Simple, low memory usage	Slow convergence, count-to-infinity problem
Link State	Centralized	Global Topology Map	Dijkstra's Algorithm	Fast convergence, no loops	High memory and overhead
Path Vector	Decentralized	Full path to destinations	Path Advertisement	Prevents loops, scalable	High overhead, slower updates

Accomplishments - Key Findings

Key Insights:

- Link State does well in latency but at the cost of higher network utilization. This is of course affected by topology and doesn't work well in a network where there is a central node. For example: networks with DHCP protocol having only 1 DHCP server are not ideally meant for star topologies.
- Path Vector sacrifices latency to handle more complex routing, keeping network utilization moderate. This is particularly useful, when latency is not a requirement of the application, but the number of users expected is large. For example: video game players waiting in a lobby of the game or people waiting in the queue of high demand concert ticket sales
- Distance Vector shows low resource utilization but appears to scale poorly, potentially failing in larger topologies.

This trade-off highlights that Link State prioritizes speed but consumes significant network resources, while Path Vector focuses on robustness with higher latency.

Bibliography

[1] Ma, X. (2024). A summary of the routing algorithm and their optimization, performance. arXiv preprint arXiv:2402.15749

[2] W. Xia, Y. Wen, C. H. Foh, D. Niyato and H. Xie, "A Survey on Software-Defined Networking," in *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 27-51, Firstquarter 2015

[3] Shishira SR, A. Kandasamy and K. Chandrasekaran, "RSim: Routing simulator for analyzing the performance of routing algorithms in a network," *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, Bangalore, India, 2016, pp. 355-358

[4] Network Topologies in Computer Networks
https://www.geeksforgeeks.org/types-of-network-topology/