TÉCNICO LISBOA

Analysis of an Optical Transport Backbone Network

Group 8

Raj Maharjan (ist1115593)
Ahmad Ashraf Zargar (ist1115594)

Submitted to: Prof. João José de Oliveira Pires

January 2026

Contents

Abstract

Transport networks form the backbone of modern communication infrastructures, enabling transparent and high-capacity interconnection between service networks. In this project, we study an optical transport backbone based on the Germany Network topology. The analysis covers topological characterization, shortest-path and connectivity properties, and traffic engineering under uncapacitated and capacitated routing scenarios. Graph-theoretic metrics and routing strategies are evaluated using both weighted and unweighted network models. The results provide insights into network robustness and efficiency, highlighting the impact of routing metrics and sorting strategies on overall network performance.

1 Introduction

High-speed optical transport networks are essential for supporting the increasing demand for data traffic generated by modern applications and services. These networks are typically deployed as backbone infrastructures, interconnecting multiple metropolitan and access networks in a transparent manner. Their performance and robustness depend strongly on the underlying physical topology, routing strategies, and traffic characteristics.

The objective of this report is to analyze a real-world-inspired optical transport network, namely the Germany Network, using graph-based modelling and traffic engineering techniques. The study is structured into three main phases. First, a topological analysis is performed to characterize node degrees and their distribution. Second, shortest-path, connectivity, and survivability properties are evaluated using both weighted and unweighted graph models. Finally, traffic routing is analyzed under uncapacitated and capacitated scenarios using different metrics and demand sorting strategies.

2 Network Modelling and Methodology

The physical topology used in this project is derived from the Germany Network. This work corresponds to Group 8, for which the node corresponding to Mannheim was removed from the original topology and replaced by a simple direct link between Frankfurt and Karlsruhe, in accordance with the project specifications, preserving network connectivity, as shown in Fig. 1. Each city is represented as a node, and each optical link is represented as an edge in the graph.

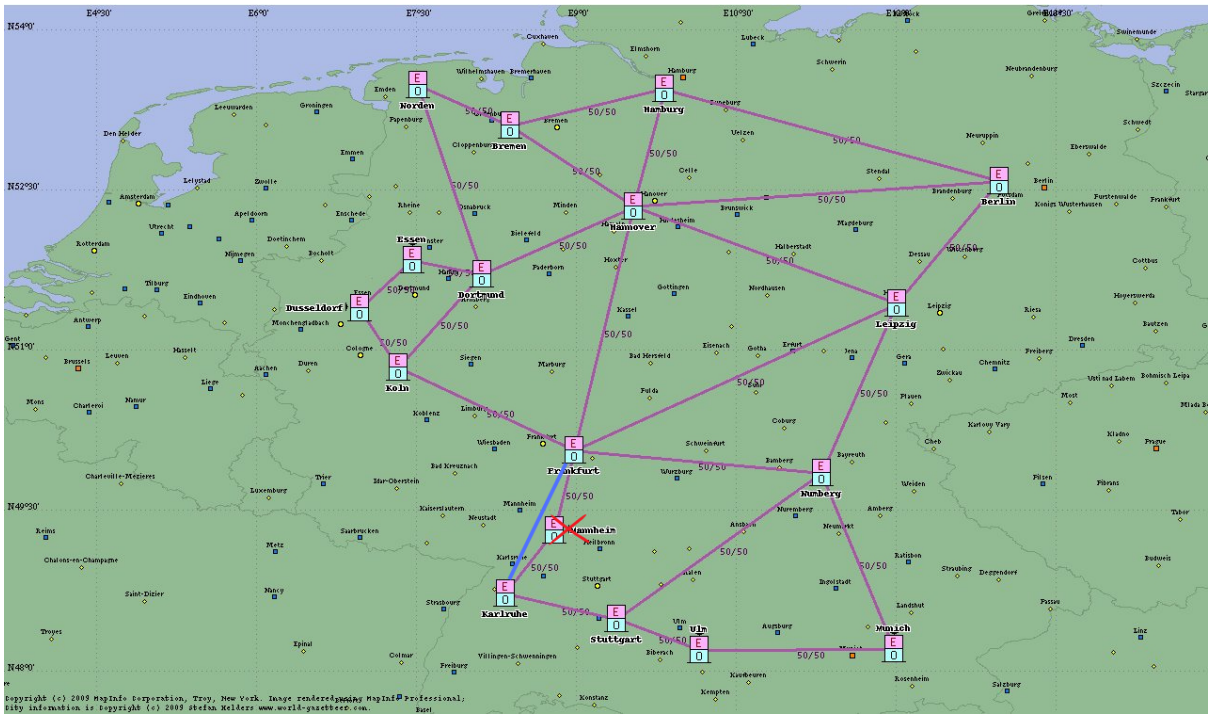


Figure 1: Network Topology with Mannheim removed

Two graph models are considered:

- Unweighted graph: used for hop-count-based analysis.
- Weighted graph: where edge weights correspond to physical link lengths, obtained from geographical distances multiplied by a group-specific distance factor.

All analyses and simulations were implemented in Python using the provided Jupyter Notebook. The Python library NetworkX was used for graph modelling and analysis, leveraging its built-in functions and standard graph algorithms to compute shortest paths, connectivity metrics, and routing solutions.

3 Topological Analysis (Phase 1)

In the first phase, the network is modelled as an unweighted graph to study node degree properties.

3.1 Node Degree Metrics

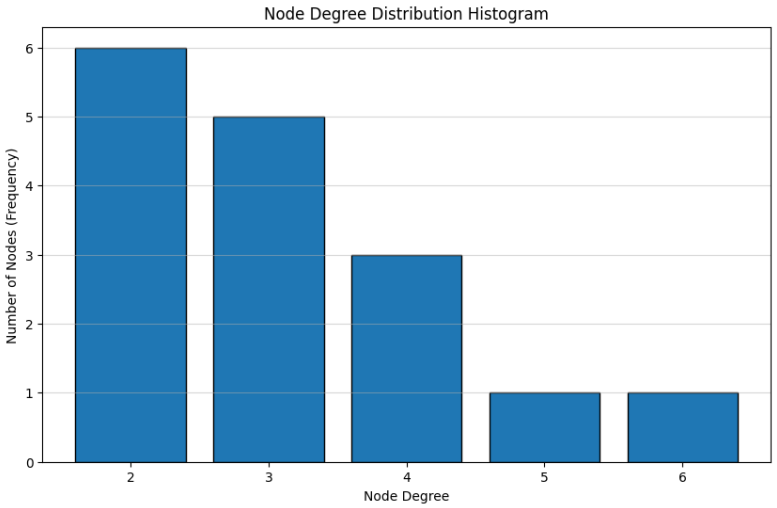
The following parameters were computed: Minimum node degree, Maximum node degree, Average node degree (δ) and Degree variance (δ^2).

For this, we first generated the adjacency matrix, stored in variable `df_topology`. To get the node degrees for each the nodes, we simply summed each of the columns

corresponding to that node. Table 2a shows the node names and their corresponding node degrees. A histogram of node degrees was generated to visualize the degree distribution. The results show how connectivity is distributed across the network and provide insights into the presence of highly connected nodes (hubs) and potential bottlenecks. Fig 2b shows the histogram of the node degrees.

City	Degree
Norden	2
Bremen	3
Hamburg	3
Hannover	6
Berlin	3
Leipzig	4
Nurnberg	4
Munich	2
Ulm	2
Stuttgart	3
Karlsruhe	2
Frankfurt	5
Koln	3
Dusseldorf	2
Essen	2
Dortmund	4

(a) Node degrees



(b) Histogram of node degrees

Figure 2: Node degree analysis: table and histogram side by side

We can see that the `minimum node degree` is equal to 2.0 for nodes Norden, Munich, Ulm, Karlsruhe, Dusseldorf and Essen. Furthermore, we can also see that the `maximum node degree` is equal to 6.0 for node Hannover.

In the Jupyter notebook, we calculated the `minimum node degree`, `maximum node degree`, `average node degree` and `degree variance` using the inbuilt python methods: `.min(), .max(), .avg() and .var()`. The `average node degree` = 3.12 and `degree variance` = 1.36.

4 Shortest Path & Connectivity Analysis (Phase 2)

This phase considers both weighted and unweighted graph models. For the weighted graph, edge weights correspond to physical link lengths. These distances were manually obtained using Google Earth Pro by placing landmarks corresponding to the nodes on the map and measuring the distances between them. To account for group-specific scaling, the measured distances were then multiplied by a factor of 1.8, as specified for Group 8. Fig. 8 shows the nodes and the links with distances between them as link weights.

4.1 Shortest Paths

For every source destination pair, shortest path was computed for both unweighted and weighted graph models using the Python module routing_v2_proj.py provided in the project description.

For the Unweighted graph, the edge weights are equal to 1, so the shortest paths minimized the number of hops between nodes. The result is stored in the variable `unweighted_distances`, which contains, for each source–destination pair, the shortest path as a node sequence and its corresponding hop count.

For the Weighted graph, as already mentioned, the edge weights represent physical link distances, measured using Google Earth Pro and scaled by a factor of 1.8 for Group 8. Shortest paths minimize the total physical distance, taking weights into account. The result is stored in `weighted_distances`, which contains, for each source–destination pair, the shortest path node sequence and its total weighted distance. An example in the case of weighted graph, of the distance and the shortest path between Hamburg and Nurnberg is shown in Fig. 3

```
print(f"Sample weighted path: {weighted_distances[38]}")
```

```
Sample weighted path: {'source': 3, 'destination': 7, 'distance': 1038.7440000000001, 'path': [3, 4, 6, 7]}
```

Figure 3: Output of distance between and shortest path between Hamburg and Nurnberg

4.2 Hop and Distance Histograms

To analyze the distribution of shortest paths in the network, two histograms were generated. Fig. 4 shows number of hops per path, based on the unweighted graph and fig. 5 shows physical path distances, based on the weighted graph. For the distance-based histogram, the distances are continuous, so the number of bins was set to 25 to provide a detailed but readable distribution.

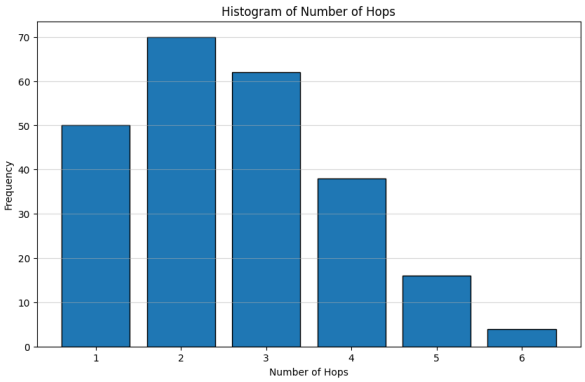


Figure 4: Number of hops per path

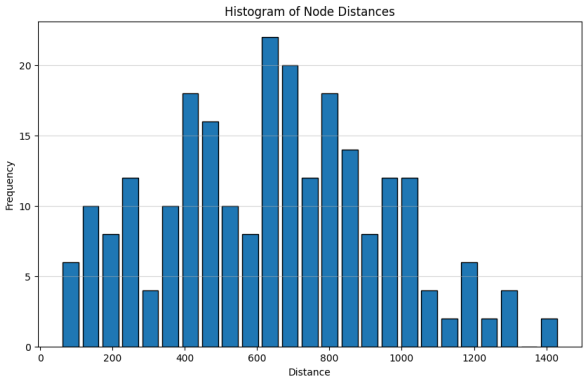


Figure 5: Physical Path lengths

We observe in hop based histogram, most shortest paths require only a few hops, indicating that the network is well-connected and that nodes are easily reachable from one another. Only a small number of paths span a larger number of hops. However, in distance based histogram, distances vary continuously, with most paths around 600-800 km, while a few paths are significantly longer. The 25-bin histogram allows us to visualize this spread clearly. This shows that although the network is efficiently connected in terms of hops, the physical distances can differ significantly. Hence, we conclude that distance-aware routing is important in optical transport networks, as hop based routing does not always show the greater picture of the topology.

4.3 Average Number of Hops per Demand and Network Diameter

The average number of hops per demand and the network diameter are key metrics to characterize the efficiency and reachability of the network.

To compute the average number of hops per demand, it is necessary to consider a conceptual full-mesh logical topology. In this model, every node is connected to every other node conceptually, representing all possible source–destination pairs. This is not the physical network, but a tool to ensure that the average is calculated over all possible demands. Although the physical topology is sparse, hop-based metrics are evaluated assuming a full-mesh logical demand model, where traffic exists between every node pair.

Fig. 9 shows the full-mesh logical topology of the 16-node network. Each link represents a possible hop between two nodes and contributes to the total hop count. and Fig. 10 shows the logical topology from the perspective of Frankfurt, illustrating the hop distance from Frankfurt to all other nodes in the network. This helps visualize how individual node connectivity contributes to the overall average.

4.3.1 Average Number of Hops per Demand

The unweighted hop matrix; variable `hop_matrix_unweighted`; contains the number of hops for the shortest path between every pair of nodes. The contents of the variable can be seen below, in table 1. Each entry h_{ij} represents the number of hops from ith node to jth node.

Step 1: Sum the upper triangle:

Since the network is undirected, the hop count from node i to node j is the same as from j to i. Therefore, only the upper triangle of the matrix (excluding the diagonal) is summed to avoid double counting. This sum is given by the formula:

$$\sum_{i=1}^{N-1} \sum_{j=i+1}^{N} h_{ij} = 315$$

Step 2: Calculate the total number of bidirectional links:

For a fully connected logical topology, the number of bidirectional links N_{bi} is given by:

$$N_{bi} = \frac{N(N-1)}{2} = \frac{16(16-1)}{2} = 120$$

Step 3: Calculate the average number of hops per demand:

To calculate average number of hops per demand $\langle h \rangle$, we simply divide the sum of the upper triangle by the number of bidirectional links.

$$\langle h \rangle = \frac{1}{N_{bi}} \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} h_{ij} = \frac{315}{120} = 2.625$$

4.3.2 Network Diameter

The network diameter is the maximum shortest-path distance (in hops) between any pair of nodes. From the hop matrix, it is simply

$$Diameter = max(hop\_matrix\_unweighted) = 6$$

This shows that the farthest nodes in the network are connected via 6 hops, confirming that the network is efficiently connected.

4.3.3 Comparison with Estimation formulas

Semi-empirical formulas can provide a quick estimation of the average number of hops without computing all paths:

- Formula 1:
$$\langle h \rangle \approx \sqrt{\frac{N-2}{\langle \delta \rangle - 1}} = \sqrt{\frac{16-2}{3.12-1}} = 2.56$$

- Formula 2:
$$\langle h \rangle \approx 1.12\sqrt{\frac{N}{\langle \delta \rangle}} = 1.12\sqrt{\frac{16}{3.12}} = 2.53$$

From these calculations, we observe that the computed average number of hops from the hop matrix, 2.625, is very close to the estimates provided by these formulas. This confirms that the network is well-connected and that the formulas provide a reliable approximation of average path length.

4.4 Connectivity Metrics

To evaluate the robustness and survivability of the network, several connectivity metrics were computed using the unweighted graph model. For the calculation of these metrics, we first defined our network as a `Graph` object G provided by the `NetworkX` library, and called the `Graph` object's methods like `node_connectivity(G)`, `edge_connectivity(G)` and `algebraic_connectivity(G)`.

4.4.1 Minimum Node Degree

The minimum node degree, denoted by δ(G), is defined as the smallest number of links incident to any node in the network. It represents the weakest-connected node.

$$\delta(G) = \min_{v \in V} \deg(v)$$

In our network, δ(G) = 2, which means that every node is connected to at least 2 other nodes.

4.4.2 Node Connectivity

The node connectivity, denoted by κ(G), is the minimum number of nodes that must be removed (along with their incident links) to disconnect the graph. In our network, κ(G) = 2. This indicates that the network remains connected even after the failure of any single node, but may become disconnected if two specific nodes fail simultaneously.

4.4.3 Edge connectivity

The edge connectivity, denoted by λ(G), is the minimum number of edges whose removal disconnects the graph. In our network, λ(G) = 2. This shows that at least two link failures are required to partition the network.

4.4.4 Relationship between these Metrics

For any undirected graph, the following inequality holds: $\kappa(G) \leq \lambda(G) \leq \delta(G)$. In our network, $\kappa(G) = \lambda(G) = \delta(G) = 2$. This equality indicates that the the network's survivability is constrained by its minimum degree.

4.4.5 Algebraic Connectivity

The algebraic connectivity, denoted by μ_2, is the second-smallest eigenvalue of the Laplacian matrix (L) of the graph. A Laplacian matrix is the matrix defined as the difference between the Degree matrix (D) and the Adjacency matrix (A) i.e. $L = D - A$ It provides

a spectral measure of how well-connected the network is. A higher value of μ_2 implies stronger overall connectivity and better resilience to failures. We obtained $\mu_2 = 0.3393$, this low value indicates limited robustness and vulnerability to node or link removals, which is consistent with the relatively sparse physical topology.

This result is expected as the network contains only 25 physical links, which is significantly fewer than the maximum possible number of links in a fully connected topology ($N_{bi} = 120$). Therefore, while the network is connected and resilient to single failures, its limited link redundancy results in a lower algebraic connectivity value.

4.5 Minimum Cut Sets

To evaluate the network's survivability between two specific nodes, we computed the minimum x–y cut sets for both nodes and edges. For our group, the source and destination nodes are: $X = Hamburg$ and $Y = Stuttgart$.

4.5.1 Minimum x-y Edge Cut Set

The minimum edge cut set is the smallest set of edges whose removal would disconnect node X from node Y. Using `NetworkX: edge_cut = nx.minimum_edge_cut(G, X, Y)`, we get the resulting edge cut set as {(Nurnberg, Stuttgart), (Karlsruhe, Stuttgart), (Ulm, Stuttgart)} with cut size = 3. This indicates that at least 3 link failures are required to disconnect *Hamburg* from *Stuttgart*. For visual representation, view Fig. 15.

4.5.2 Minimum x-y Node Cut Set

The minimum node cut set is the smallest set of nodes whose removal would disconnect X from Y, excluding X and Y themselves. Using `NetworkX: node_cut = nx.minimum_node_cut(G, X, Y)`, we get the resulting node cut set as {Karlsruhe, Nurnberg} with cut size = 2. This indicates that removal of just two intermediate nodes is sufficient to disconnect *Hamburg* from *Stuttgart*. This demonstrates the network's vulnerability along certain critical paths and highlights potential points for redundancy improvement. For visual representation, view Fig. 16.

These metrics are useful for planning backup paths: by ensuring that service paths avoid shared nodes and links in the cut sets, we can improve the network's resilience against failures.

4.6 Service Path and Backup Paths

To ensure resilient communication between nodes $X = Hamburg$ and $Y = Stuttgart$, we determined a primary service path and multiple backup paths. Two strategies were

used: A) **Edge disjoint paths** where no two paths share the same link and B) **Node disjoint paths** where no two paths share the same intermediate node.

4.6.1 Unweighted Graph

The service path is the shortest path in terms of number of hops. The Edge-disjoint backup path are all simple shortest first paths that avoid any edges already used by the service path and the Node-disjoint backup paths are all simple shortest first paths that avoid intermediate nodes used by the service path.

- Service Path:

  From *Hamburg* to *Stuttgart*, the service path is `['Hamburg', 'Hannover', 'Frankfurt', 'Karlsruhe', 'Stuttgart'], hops: 4`

- Edge-disjoint backup paths:

  We have 2 edge-disjoint backup paths:

  1. `['Hamburg', 'Berlin', 'Leipzig', 'Nurnberg', 'Stuttgart'], hops: 4`
  2. `['Hamburg', 'Bremen', 'Hannover', 'Leipzig', 'Frankfurt', 'Nurnberg', 'Munich', 'Ulm', 'Stuttgart'], hops: 8`

  For a detailed visual analysis refer to Fig. 11.

- Node-disjoint backup paths:

  We only have 1 node-disjoint backup path: `['Hamburg', 'Berlin', 'Leipzig', 'Nurnberg', 'Stuttgart'], hops: 4`

  For a detailed visual analysis refer to Fig. 12.

4.6.2 Weighted Graph

Here, the service path is the path that has the minimum shortest distance. Here as well, the Edge-disjoint backup paths are all simple shortest first paths that avoid any edges already used by the service path and the Node-disjoint backup paths are all simple shortest first paths that avoid intermediate nodes used by the service path.

- Service Path:

  From *Hamburg* to *Stuttgart*, the service path is `['Hamburg', 'Hannover', 'Frankfurt', 'Karlsruhe', 'Stuttgart'], distance: 1045.44 km`

- Edge-disjoint backup paths:

  We have 2 edge-disjoint backup paths:

1. ['Hamburg', 'Berlin', 'Leipzig', 'Nurnberg', 'Stuttgart'], hops: 1429.45 km
2. ['Hamburg', 'Bremen', 'Hannover', 'Dortmund', 'Koln', 'Frankfurt', 'Nurnberg', 'Munich', 'Ulm', 'Stuttgart'], distance: 1919.09 km

For a detailed visual analysis refer to Fig. 13.

- Node-disjoint backup paths:

  We only have 1 node-disjoint backup path: ['Hamburg', 'Berlin', 'Leipzig', 'Nurnberg', 'Stuttgart'], distance: 1429.45 km

  For a detailed visual analysis refer to Fig. 14.

We observe that the primary service path between *Hamburg* and *Stuttgart* is identical in both the unweighted and weighted graphs. Edge-disjoint backup paths provide multiple independent alternatives by utilizing different links, while node-disjoint backups ensure redundancy by avoiding shared intermediate nodes, protecting against failures of critical nodes. In the weighted graph, some backup paths are significantly longer than their hop-based counterparts, highlighting the trade-off between minimizing the number of hops and optimizing for total physical distance in routing decisions.

5 Traffic Engineering & Routing Analysis (Phase 3)

Phase III focuses on analyzing the routing of traffic demands across the network without considering link capacity constraints and later considering an environment which is constrained. In this stage, the `network topology` and `traffic matrix` are used to determine the optimal paths for all demands based on specific metrics, such as hop count and physical distance.

The phase begins with examining both the `traffic matrix`, which quantifies the volume of data to be transported between nodes, and the `demand matrix`, which identifies which node pairs require routing, enabling effective planning and optimization of high-speed optical networks. These matrices are instrumental in determining where network resources should be provisioned, and how to efficiently route traffic to `maximize throughput`, `minimize congestion`, and ensure `robust network performance` under peak demands.

5.1 Traffic Matrix

The `traffic matrix` represents the volume of data (in Gb/s) that needs to be transported between every pair of nodes in the network. Each element T_{ij} of the matrix indicates the traffic demand from node i to node j.

Key observation from the traffic matrix, presented in Table 2 are followed:

- **Symmetric traffic:** The matrix is symmetric if the network is bidirectional, i.e., $T_{ij} = T_{ji}$. This is common in optical networks where links carry traffic in both directions, which are also present in research works [6].
- **Zero nodes:** Nodes such as "Essen" and "Ulm" are disconnected (traffic set to 0). This could represent nodes under maintenance or nodes without traffic demand.

Traffic matrix have high utility in optical networks, some of them are presented as follows:

- Traffic matrices allow capacity planning: determining how much bandwidth each optical fiber must support [5].
- They enable traffic engineering, i.e., routing traffic to avoid congestion while maximizing network utilization [2].
- They help simulate realistic network behavior under peak and off-peak conditions, critical for high-speed optical backbone networks [4].

5.2 Demand Matrix

The `demand matrix` is derived from the traffic matrix as a binary indicator of presence or absence of traffic:

$$\text{demand matrix}_{ij} = \begin{cases} 1 & \text{if } T_{ij} > 0 \\ 0 & \text{if } T_{ij} = 0 \end{cases}$$

Table 4 presents the derived demand matrix from the traffic matrix, we can interpret it as follows:

- **1:** There is a demand between node i and j.
- **0:** No demand exists, so routing can ignore that pair.

Demand matrix also plays a crucial role in optical networks, it could provide the utility as follow:

- Demand matrices provide a simplified abstraction for routing by focusing on relevant source–destination pairs rather than the full network [3].
- They reduce computational complexity in routing and wavelength assignment by narrowing the search for feasible paths [7].
- Demand matrices are central to blocking analysis and capacity planning, as performance depends directly on traffic demands and network topology [1].

5.3 Uncapacitated Routing Problem

The uncapacitated routing problem is a foundational step in network planning and traffic engineering. It determines routing paths for a given set of traffic demands on a fixed topology while ignoring link capacity constraints, enabling the identification of baseline routing patterns and performance trade-offs before capacity limitations are introduced.

Formally, the uncapacitated routing problem can be defined as follows:

Given:

- A network topology represented as a graph $G = (V, E)$ where V is the set of nodes (network devices, cities, or endpoints) and E is the set of edges (links, connections).
- A demand matrix D where d_{ij} represents the traffic demand from node i to node j.

Determine:

- A set of routing paths P_{ij} for each demand pair (i, j) such that traffic flows from source i to destination j.

Characteristics:

- Uncapacitated assumption: All edges $e \in E$ have infinite capacity ($c_e = \infty$), implying no capacity constraint link exists in this setup.

Objective Metrics: Different optimization objectives can be pursued depending on network requirements

- `Number of hops:` Minimizing the number of intermediate nodes (shortest hop paths).
- `Total distance:` Minimizing physical distance traveled by traffic (important for latency-sensitive optical networks).
- `Load balancing:` Evenly distributing traffic across links to avoid congestion in subsequent capacitated analyses.

5.4 Why Solve the Uncapacitated Routing Problem First?

Although real optical networks operate under strict capacity constraints, solving the uncapacitated routing problem is a crucial preliminary step in network design. It isolates the effects of topology and traffic demands, providing insight into intrinsic routing behavior before capacity limits are imposed. The main motivations are:

- **Baseline characterization:**

  - Identifies the `maximum link load` $L_{\max}$ under ideal conditions.
  - Serves as a reference for defining link capacities in subsequent capacitated simulations ($U = \alpha \cdot L_{\max}$).

- **Routing strategy evaluation:**
  - `Shortest-First:` Prioritizes short paths, generally promoting efficient resource usage.
  - `Longest-First:` Routes long paths first, stressing multiple links early.
  - `Largest-First:` Prioritizes high-volume demands, strongly influencing load concentration.

  Comparing link loads across these strategies reveals which approach yields more balanced resource utilization.

- **Network design insights:**
  - Identifies critical links and potential bottlenecks.
  - Supports informed capacity planning and robustness analysis.

5.5 Solution Procedure

This section describes the methodology used to evaluate uncapacitated routing under different demand-ordering strategies and routing metrics. The objective is to analyze link load distribution and identify critical links in the network.

Step 1: Network and Traffic Representation

- Given the graph $G = (V, E)$.
- `Adjacency matrix A`: indicates physical connectivity between cities.
- `Traffic matrix T`: specifies traffic demand (in Gb/s) between every source–destination pair.
- In the uncapacitated scenario, all links are assigned a very large capacity: $C_e =$ `MAX_LINK_CAP` $\gg \max(T_{ij})$, ensuring that no demand is blocked due to capacity constraints.

Step 2: Path Computation

Two routing metrics are considered:

- `Hop-Based Shortest Paths:` The `adjacency matrix` is treated as an unweighted graph. For each source node, the shortest paths (minimum hop count) to all destinations are computed. A `hop-count matrix` is derived, where each entry represents the number of hops along the shortest path.

- `Distance-Based Shortest Paths:` A weighted adjacency matrix is constructed using physical link distances. Shortest paths minimizing total distance are computed. For each source–destination pair, the total path length is calculated by summing link distances along the selected path.

These steps generate a set of candidate paths for routing and a corresponding metric matrix (hop count or distance) used for demand ordering.

Step 3: Demand Ordering Strategy

To study the impact of routing order, traffic demands are sorted according to three strategies: `Shortest-First, Longest-First, Largest-First`. The function `orderPaths()` produces an ordered list of source–destination demand IDs based on the chosen strategy and metric.

Step 4: Routing Procedure (Shortest-First Example) The routing process iteratively assigns each demand to a path following the chosen ordering strategy. For clarity, the Shortest-First routing procedure is described in detail below.

- **Demand Selection:**
  - Select the next demand (s, d) from the ordered list.
  - The demand corresponds to the smallest metric value (fewest hops or shortest distance) among all unrouted demands.
- **Path Feasibility Check:**
  - If the demand has a single shortest path, that path is selected directly.
  - If multiple shortest paths exist:
    - For each candidate path, compute the current load on all links along the path.
    - Evaluate the maximum or total path load.
- **Path Selection Rule:**
  - Among all feasible shortest paths, select the path that: min (path load), i.e., the path whose links currently carry the least amount of traffic. This minimizes early congestion and promotes load balancing.
- **Load Update:**

– Route the full traffic demand along the selected path.

– Increment the load of every link on the path by the demand volume.

– Since the network is uncapacitated, no blocking occurs.

This procedure is repeated until all demands are routed.

Step 5: Link Load Extraction After all demands are routed:

- The `final link load matrix` is obtained.
- For each physical link (u,v), the total bidirectional load is calculated as: $L_{uv} = L_{u\to v} + L_{v\to u}$. These values represent the cumulative traffic carried out by each optical link.

Step 6: Performance Evaluation and Visualization

The link load analysis across different routing metrics and demand ordering strategies highlights the impact of sorting on network load distribution. Two routing metrics were considered: `Hops` and `Distance`, each combined with three demand ordering strategies: `shortest-first, longest-first, and largest-load-first`.

Hops-Based Routing From the link load results (Table 3), the `shortest-first` strategy concentrates traffic on a few links, leading to uneven utilization. In particular, the Leipzig–Nürnberg link carries the highest load (980 Gb/s bi-directional), resulting in a large `maximum link load` $L_{\max}$ and high load variability. The `largest-load-first` strategy distributes traffic more evenly, reducing $L_{\max}$ to 930 Gb/s and the standard deviation to approximately 255.8 Gb/s. The `longest-first` strategy also improves fairness compared to `shortest-first`, but to a lesser extent.

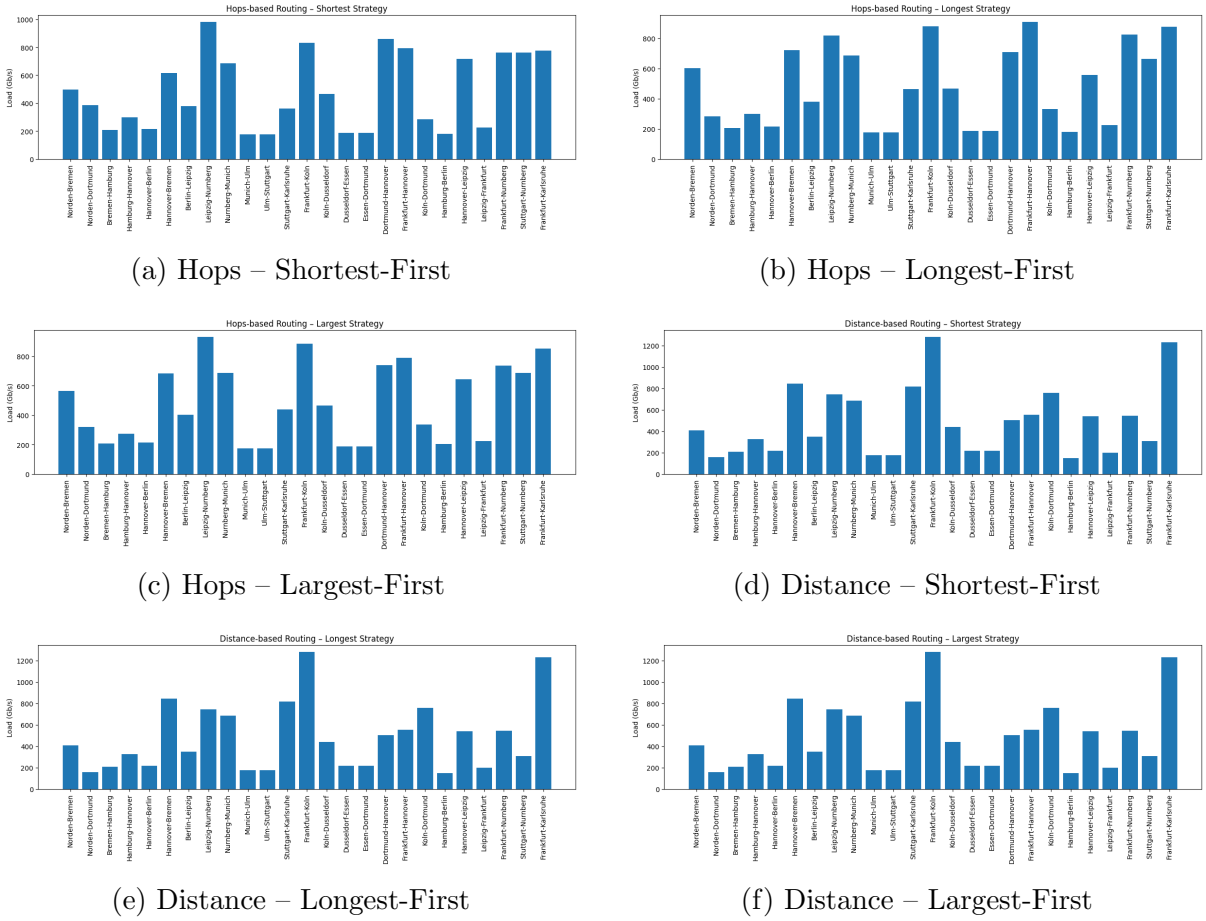
Distance-Based Routing For `distance-based routing`, all sorting strategies yield similar outcomes, with $L_{\max} = 1279$ Gb/s and a standard deviation of about 315.5 Gb/s. This indicates that minimizing distance alone leads to persistent traffic aggregation on shortest paths, limiting load balancing regardless of demand ordering.

Table 5 summarizes the key load metrics for all routing strategies and demand orderings, providing a detailed basis for evaluating network balance and link utilization.

Considering both routing metrics and all sorting strategies:

- **Largest-load-first with Hops-based routing** is the most balanced solution, achieving the highest fairness and lowest load concentration among the evaluated options.
- **Shortest-first**, while minimizing traversal hops, results in highly utilized links and potential bottlenecks, reducing overall fairness.
- **Distance-based** strategies are less effective at balancing load, as they do not significantly reduce maximum link utilization or variance.

The **largest-load-first sorting strategy**, when combined with the **Hops routing metric**, provides the most equitable and balanced network utilization. **This strategy reduces the risk of congestion on critical links, minimizes the variance of link loads, and ensures a more uniform distribution of traffic across the network, making it the preferred approach for efficient and reliable traffic engineering.**



(a) Hops – Shortest-First

(b) Hops – Longest-First

(c) Hops – Largest-First

(d) Distance – Shortest-First

(e) Distance – Longest-First

(f) Distance – Largest-First

Figure 6: Link load distributions under uncapacitated routing for different metrics and demand ordering strategies

5.6 Capacitated Routing Analysis: Longest-First Strategy

In this section, we analyze the network performance under link capacity constraints using the **longest-first** demand ordering strategy. The analysis considers two metrics: the **number of hops** and the **total path length (distance)**. The longest-first strategy prioritizes routing demands with the longest paths first, which strongly influences how link capacities are consumed and how congestion propagates across the network.

5.6.1 Methodology

The analysis proceeds as follows:

1. **Baseline Run:** All links are initially assumed to have infinite capacity (`MAX_LINK_CAP` = 999,999 Gb/s). The maximum link load $L_{\max}$ is measured for each metric under the longest-first strategy. This serves as a reference for assigning actual link capacities in the subsequent simulations.

2. **Capacity Assignment:** For a given link (i, j), the capacity is defined as $u(i, j) = \alpha \times L_{\max}$, where $\alpha \in (0, 1]$ is a scaling factor. Simulations start with $\alpha = 1$ and decrease in steps of 0.05 until the *blocking ratio* exceeds 0.5.

3. **Performance Metrics:** At each α, the following are recorded:

   - **Blocking Ratio:** Fraction of demands that could not be routed due to insufficient capacity.
   - **Average Path Length:** Mean of the lengths of all successfully routed paths.
   - **Maximum Path Length:** Length of the longest successfully routed path.

5.6.2 Hops Metric Results

Figure 7a shows the blocking ratio and path lengths as a function of link capacity for the Hops metric. Initially, with $\alpha = 1$, all demands are routed successfully, resulting in a blocking ratio of zero. As α decreases, the blocking ratio rises gradually, reaching approximately 0.5 when link capacity falls below 150 Gb/s.

The maximum path length remains nearly constant across all capacities. This is a direct consequence of the longest-first strategy: the longest demands are routed first, securing their ideal shortest paths. Subsequent demands may need to reroute via alternate paths, slightly increasing the average path length, but never exceeding the established maximum.

5.6.3 Distance Metric Results

For the Distance metric (Figure 7b), similar trends are observed. Initially, the network accommodates all traffic, but as α decreases, the blocking ratio rises sharply, reaching the 0.5 threshold at $\alpha \approx 0.2$. Average path length remains fairly stable, while the maximum path length shows minor fluctuations, reflecting rerouting for shorter demands.

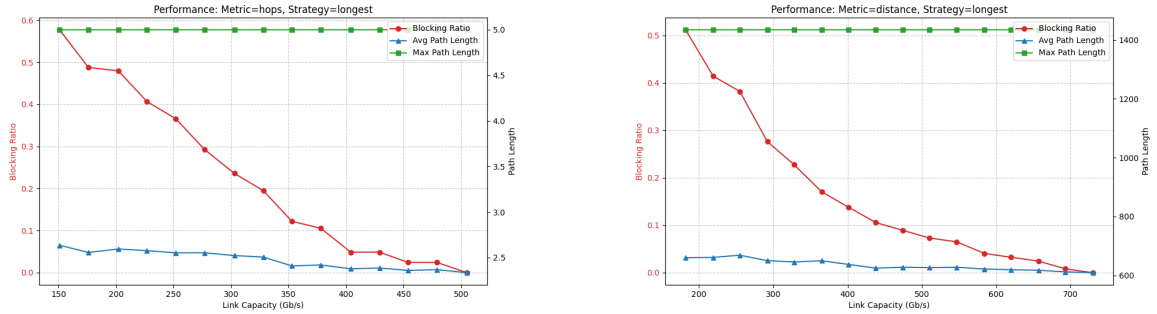
5.6.4 Discussion

The longest-first strategy exhibits the following behaviors:

- **Blocking Ratio:** Longest-first routing increases blocking under reduced capacity, as long demands saturate multiple links and block shorter requests.

- **Maximum Path Length:** Remains nearly constant across α, since the longest demands are routed first before congestion occurs.
- **Average Path Length:** Increases slightly as α decreases due to rerouting around congested links.
- **Hops vs. Distance:** Hops-based routing reaches blocking at higher capacities, indicating stronger topological bottlenecks than distance-related constraints.

The capacitated routing analysis shows that the longest-first strategy leads to high blocking ratios as link capacities decrease. While long demands retain optimal paths, shorter demands are increasingly blocked or rerouted. Distance-based routing is slightly more resilient than hops-based routing under severe capacity constraints, highlighting the importance of both demand ordering and metric selection in capacity-aware routing design.



(a) Hops metric with Longest-First.

(b) Distance metric with Longest-First.

Figure 7: Blocking ratio and path lengths versus link capacity for Hops and Distance metrics using the Longest-First strategy.

6 Conclusion

This report analyzes an optical transport backbone based on the Germany network topology, combining graph-theoretic characterization with uncapacitated and capacitated routing analyses. While the topology is well connected and resilient to single failures, limited redundancy reduces robustness under multiple failures and high traffic loads. Shortest-path routing minimizes hop count and distance but causes load concentration, whereas demand-ordering strategies such as largest-load-first improve fairness and reduce congestion. Capacitated routing results further demonstrate the sensitivity of blocking performance to routing metrics and demand ordering, highlighting the need for traffic-aware routing strategies that balance efficiency, robustness, and load distribution.

References

[1] Luiz H Bonani and Iguatemi E Fonseca. Estimating the blocking probability in wavelength-routed optical networks. *Optical switching and networking*, 10(4):430–438, 2013.

[2] VK Chaubey and Kiran Divakar. Modeling and simulation of wdm optical networks under traffic control protocols. *Optical Fiber Technology*, 15(2):95–99, 2009.

[3] Carlos Jorge da Cruz Rodrigues. Optical network planning for static applications. Master's thesis, ISCTE-Instituto Universitario de Lisboa (Portugal), 2018.

[4] Ugo Fiore, Paolo Zanetti, Francesco Palmieri, and Francesca Perla. Traffic matrix estimation with software-defined nfv: Challenges and opportunities. *Journal of computational science*, 22:162–170, 2017.

[5] João JO Pires. On the capacity of optical backbone networks. *Network*, 4(1):114–132, 2024.

[6] J Ribeiro. *Machine Learning Techniques for Designing Optical Networks to Face Future Challenges*. PhD thesis, University of Lisboa Lisboa, Portugal, 2023.

[7] Ayşegül Yayımlı. A practical layered model for flexible-grid optical networks to reduce the routing complexity. In *2016 18th International Conference on Transparent Optical Networks (ICTON)*, pages 1–4. IEEE, 2016.

7 Annex

7.1 Figures

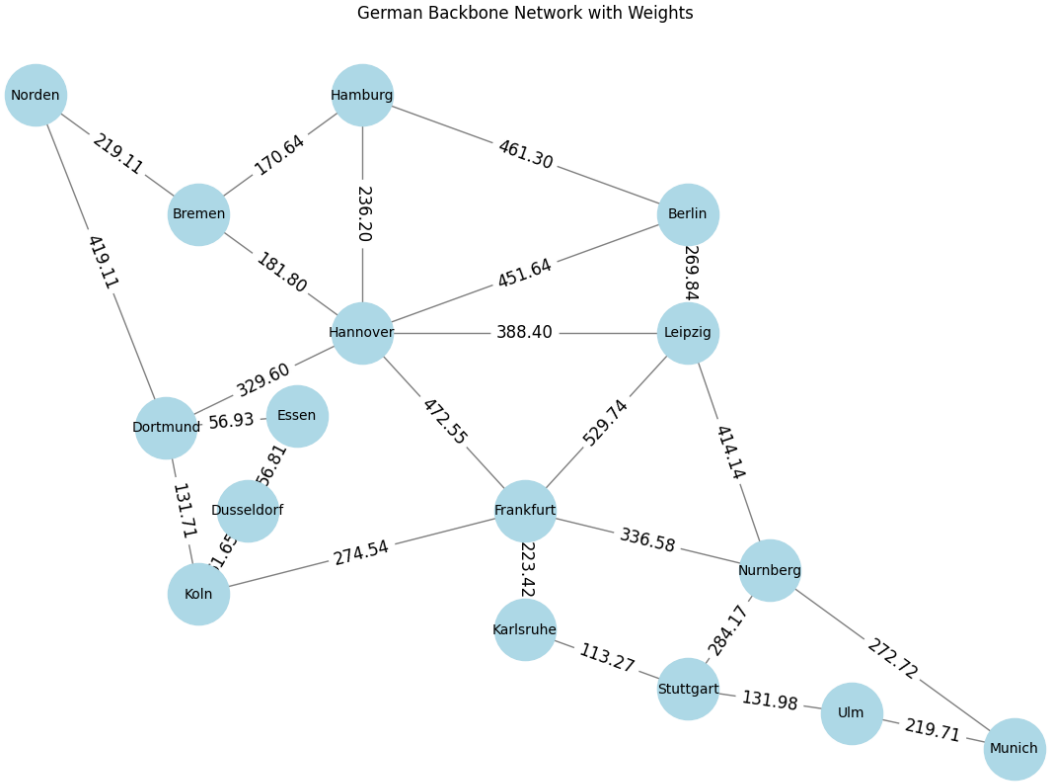


Figure 8: Weighted graph of the network

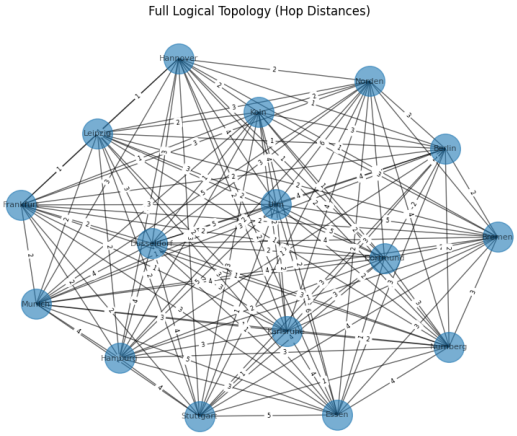


Figure 9: Full mesh logical topology

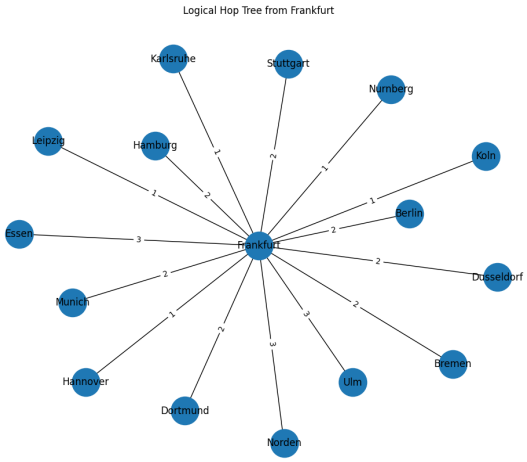


Figure 10: Full mesh logical topology wrt Frankfurt

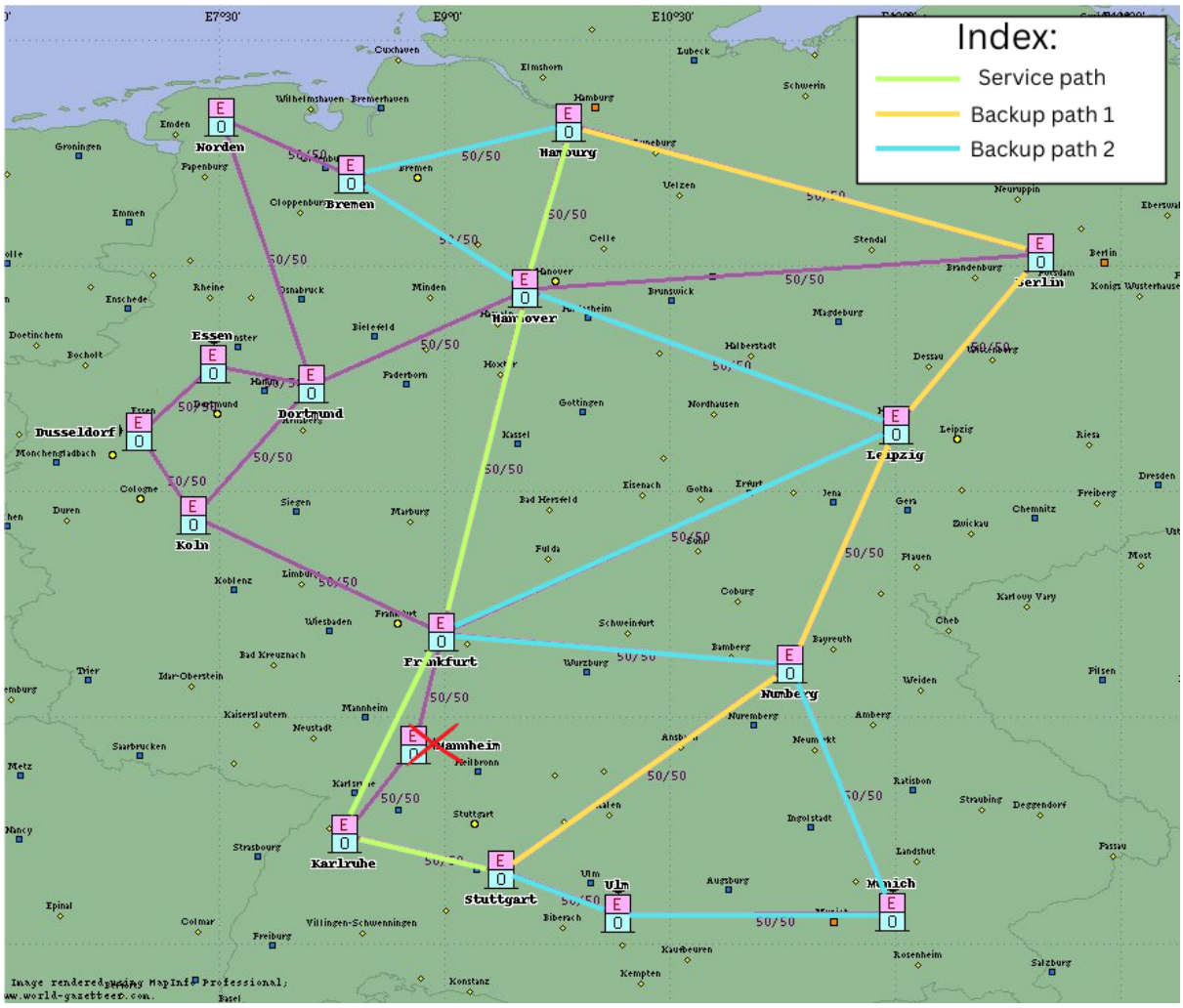


Figure 11: Service path (4 hops) and edge disjoint backup paths 1 and 2 (4 and 8 hops) for unweighted graph

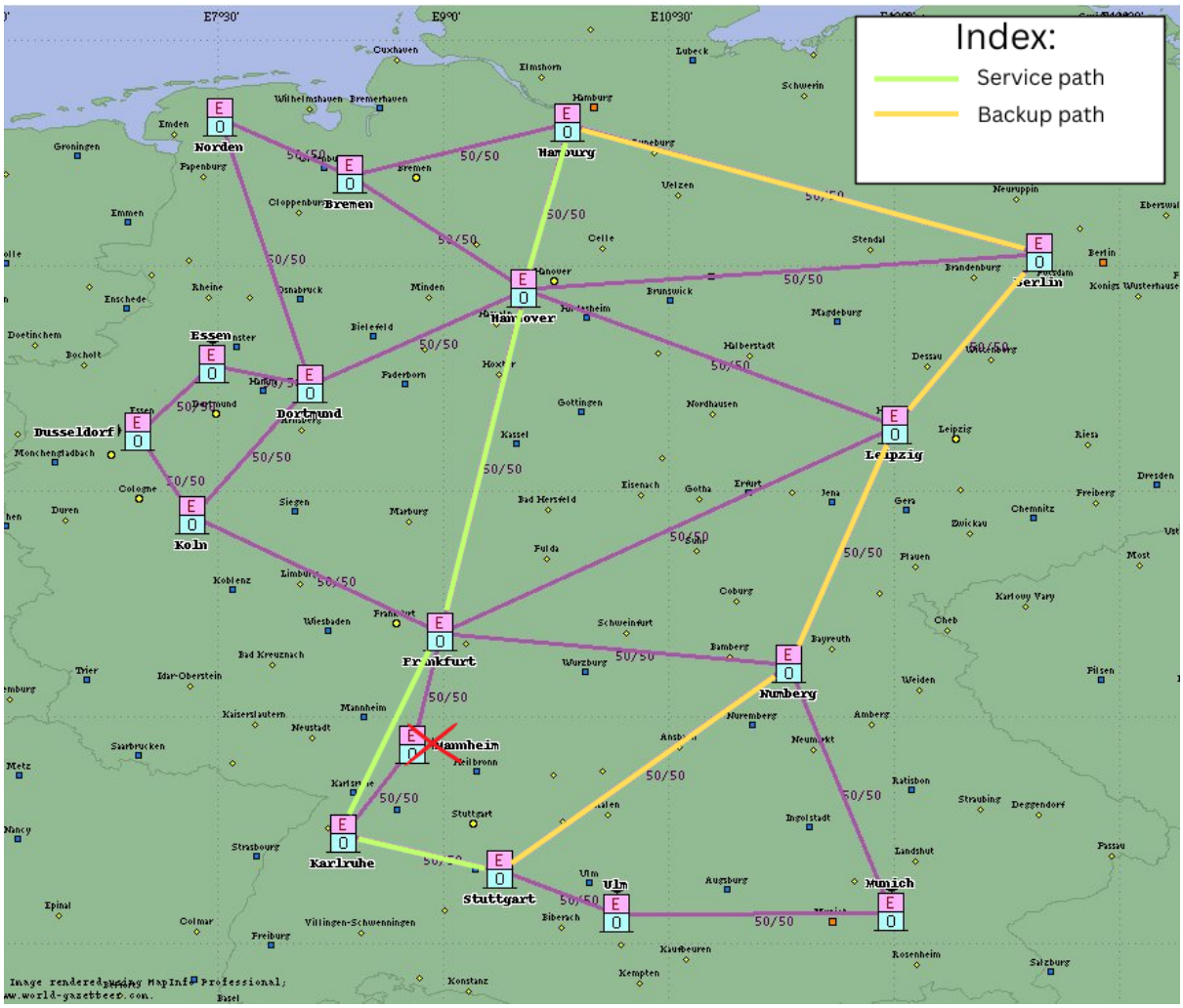


Figure 12: Service path (4 hops) and node disjoint backup path (4 hops) for unweighted graph

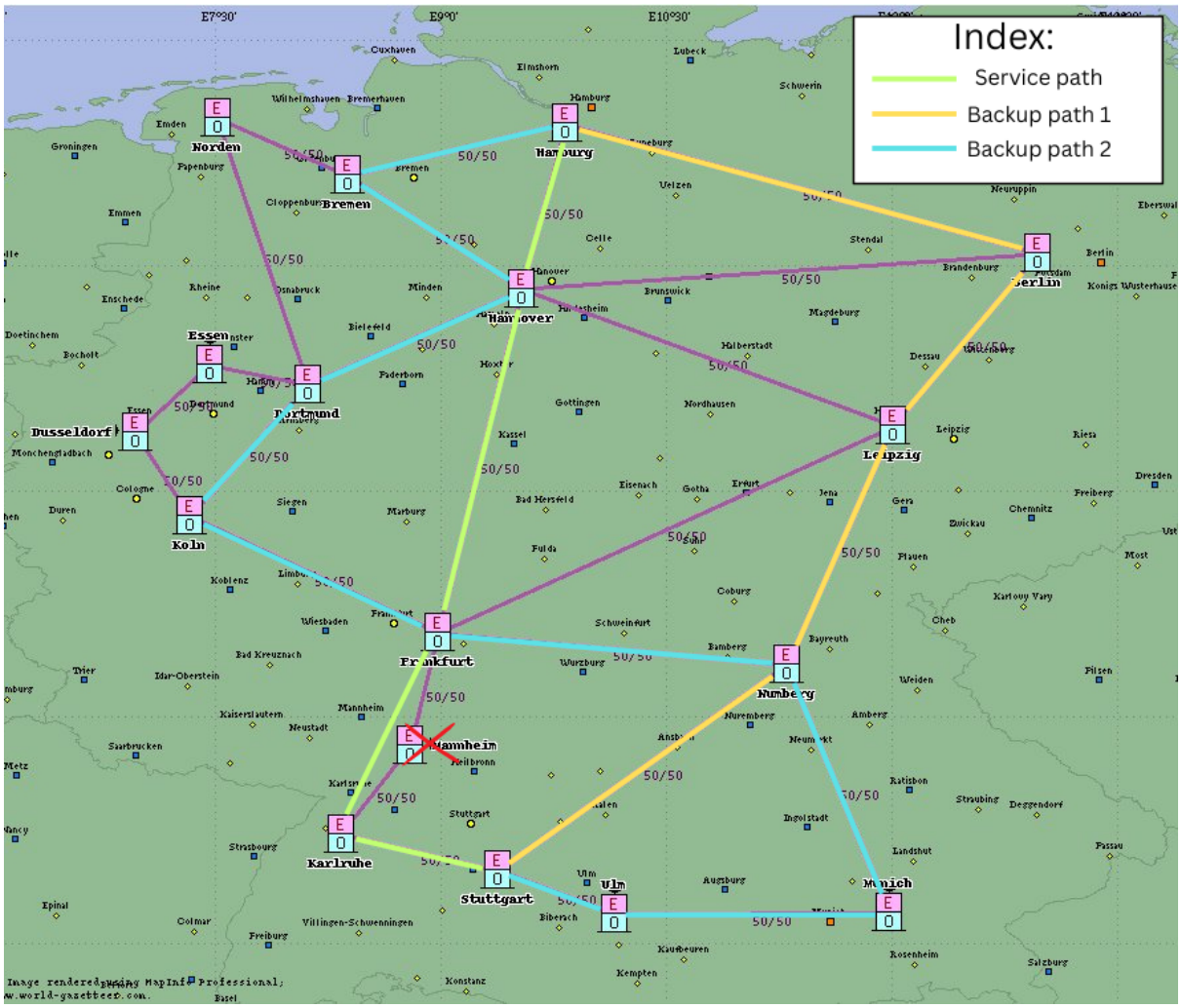


Figure 13: Service path (1045.44 km) and edge disjoint backup paths 1 and 2 (1429.45 km and 1919.08 km) for weighted graph

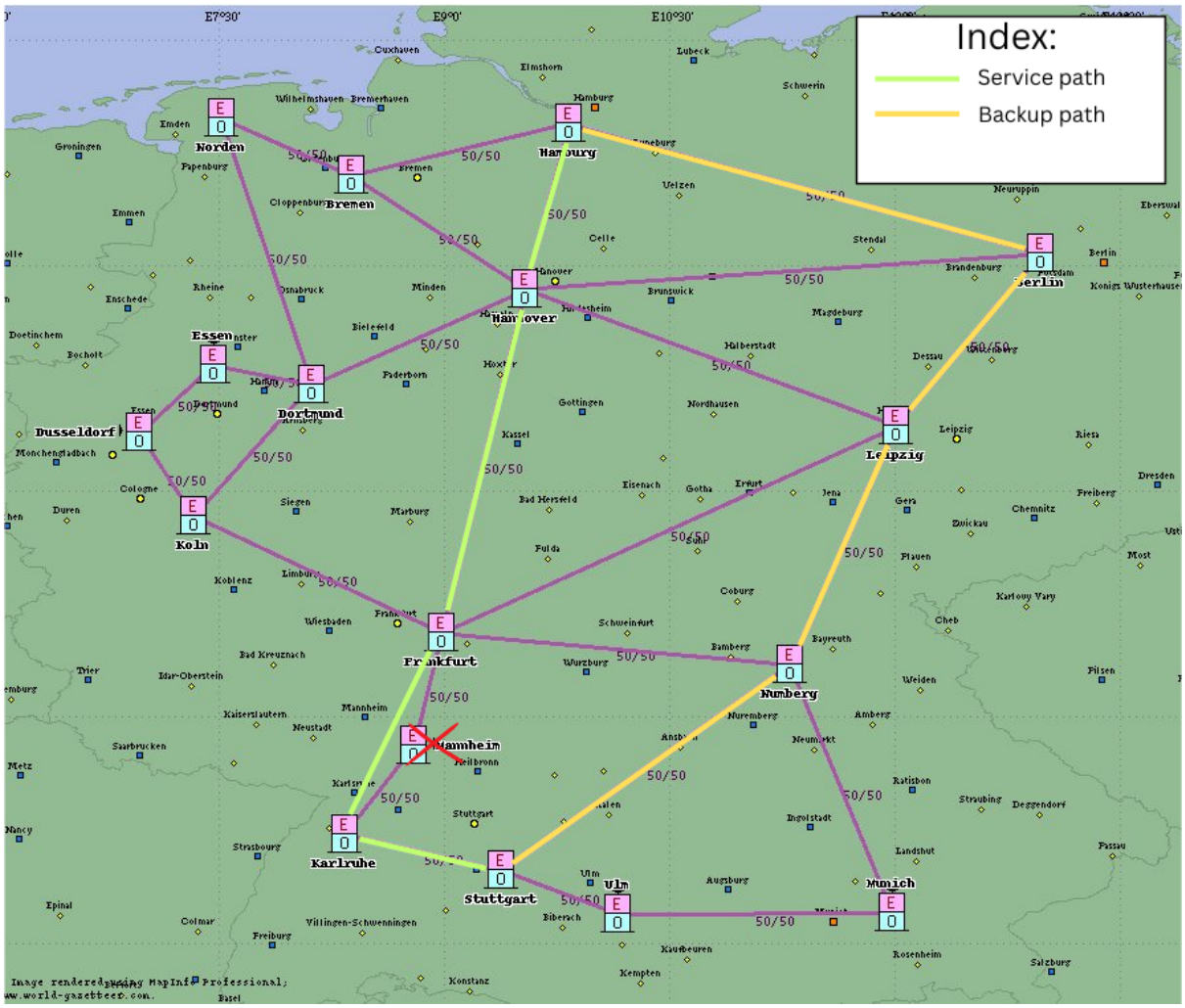


Figure 14: Service path (1045.44 km) and node disjoint backup path (1429.45 km) for weighted graph

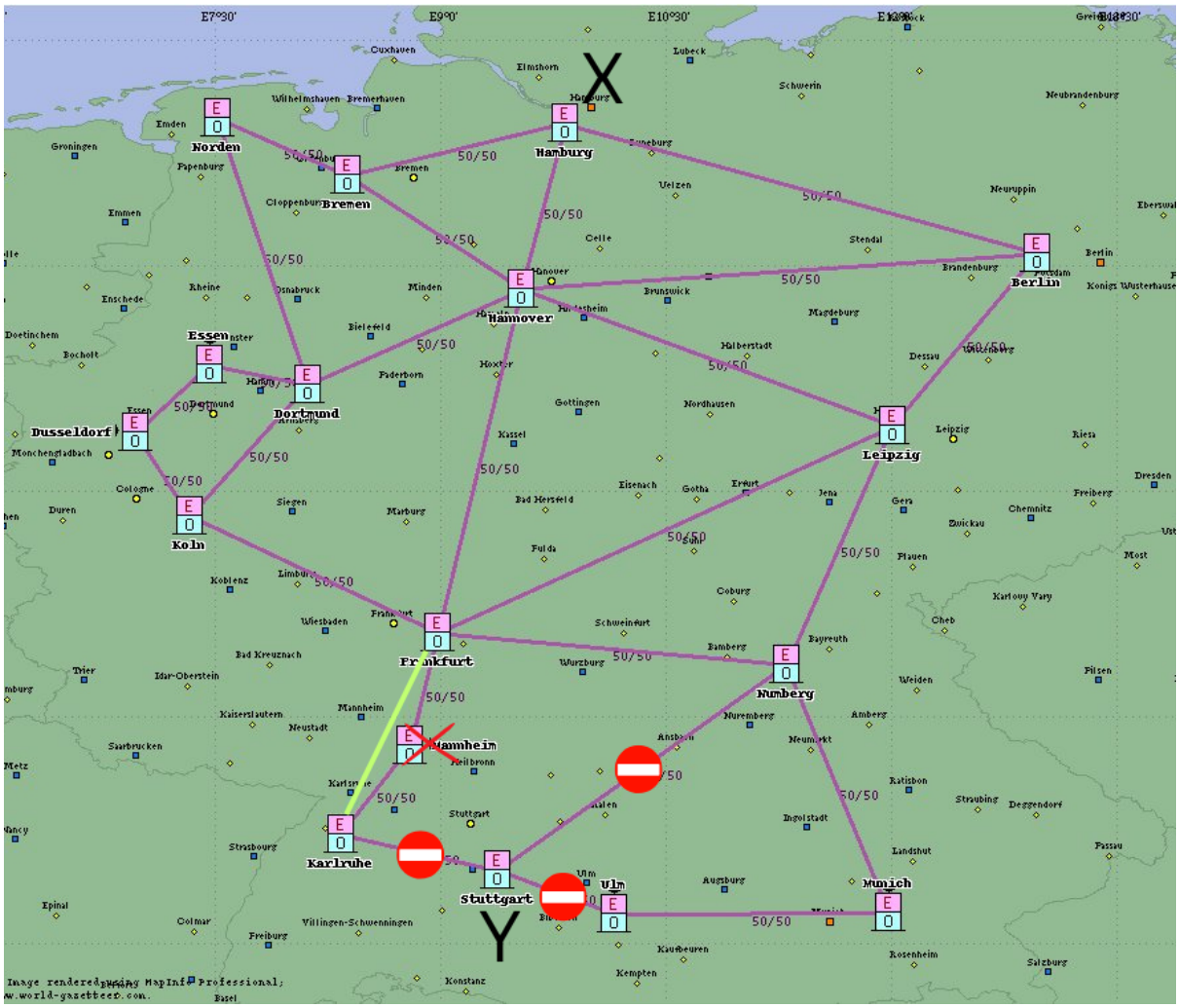


Figure 15: Visualization of minimum edge cut set

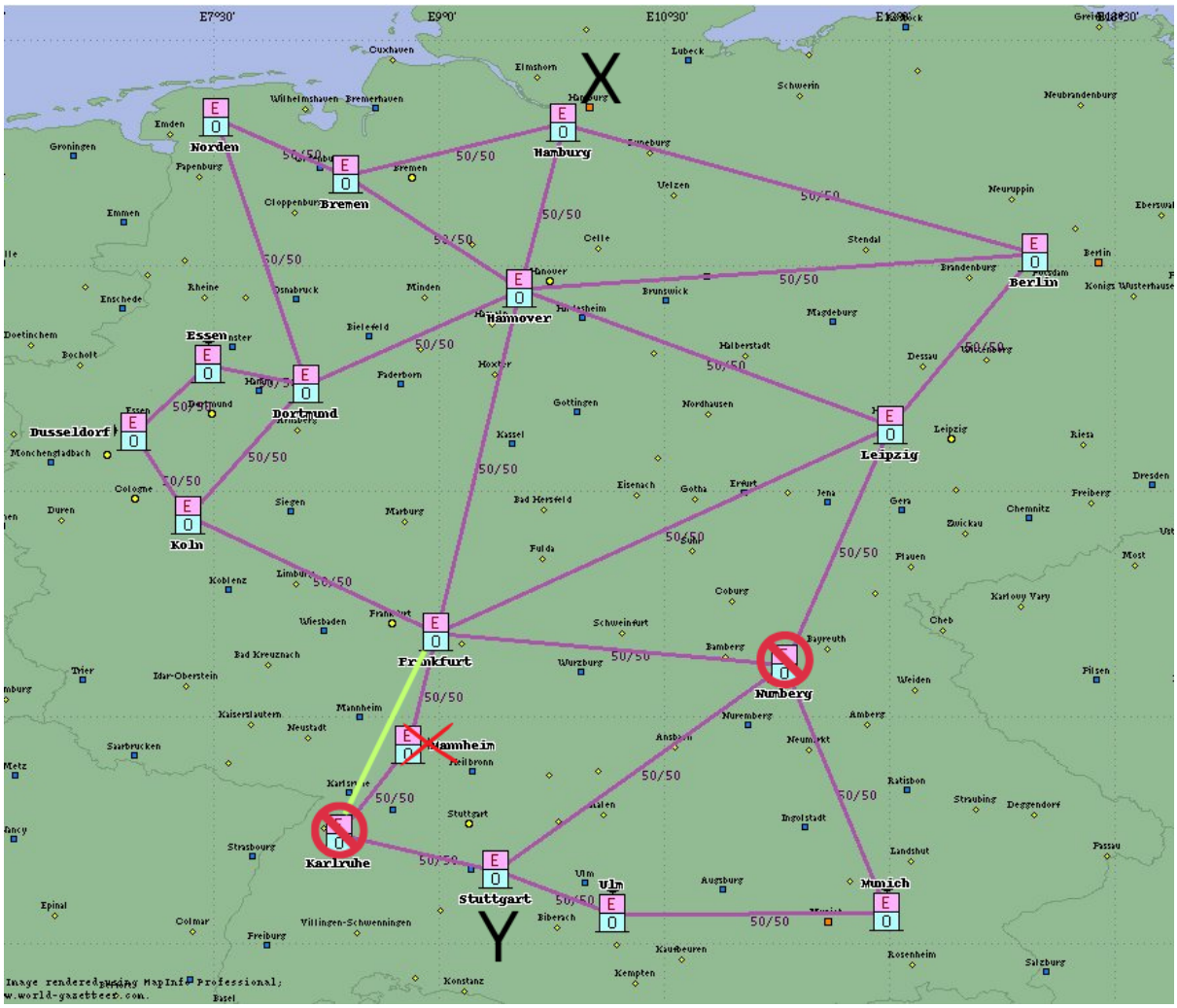


Figure 16: Visualization of minimum node cut set

7.2 Tables

Node #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	1	2	2	3	3	4	5	6	5	4	3	2	3	2	1
2	1	0	1	1	2	2	3	4	5	4	3	2	3	4	3	2
3	2	1	0	1	1	2	3	4	5	4	3	2	3	4	3	2
4	2	1	1	0	1	1	2	3	4	3	2	1	2	3	2	1
5	3	2	1	1	0	1	2	3	4	3	3	2	3	4	3	2
6	3	2	2	1	1	0	1	2	3	2	2	1	2	3	3	2
7	4	3	3	2	2	1	0	1	2	1	2	1	2	3	4	3
8	5	4	4	3	3	2	1	0	1	2	3	2	3	4	5	4
9	6	5	5	4	4	3	2	1	0	1	2	3	4	5	6	5
10	5	4	4	3	3	2	1	2	1	0	1	2	3	4	5	4
11	4	3	3	2	3	2	2	3	2	1	0	1	2	3	4	3
12	3	2	2	1	2	1	1	2	3	2	1	0	1	2	3	2
13	2	3	3	2	3	2	2	3	4	3	2	1	0	1	2	1
14	3	4	4	3	4	3	3	4	5	4	3	2	1	0	1	2
15	2	3	3	2	3	3	4	5	6	5	4	3	2	1	0	1
16	1	2	2	1	2	2	3	4	5	4	3	2	1	2	1	0

Table 1: Unweighted Hop Matrix

Table 2: Traffic Matrix (Gb/s) between cities

City	Norden	Bremen	Hamburg	Hannover	Berlin	Leipzig	Nurnberg	Munich	Ulm	Stuttgart	Karlsruhe	Frankfurt	Koln	Dusseldorf	Essen	Dortmund
Norden	0	28	47	53	28	47	0	28	0	0	28	0	0	0	0	0
Bremen	28	0	28	47	0	28	0	53	0	47	53	0	0	53	0	53
Hamburg	47	28	0	28	47	0	28	0	0	28	0	53	0	0	0	0
Hannover	53	47	28	0	28	47	53	0	0	53	28	0	53	0	0	0
Berlin	28	0	47	28	0	28	47	53	0	47	0	0	0	53	0	53
Leipzig	47	28	0	47	28	0	28	47	0	28	47	0	28	0	0	0
Nurnberg	0	0	28	53	47	28	0	28	0	53	28	0	53	28	0	28
Munich	28	53	0	0	53	47	28	0	0	47	53	28	0	53	0	53
Ulm	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Stuttgart	47	28	53	0	0	53	47	28	0	28	47	53	28	47	0	47
Karlsruhe	0	47	28	53	47	28	53	47	0	0	28	47	53	28	0	28
Frankfurt	28	53	0	28	0	47	28	53	0	28	0	28	47	53	0	53
Koln	0	0	53	0	0	0	0	28	0	47	28	0	28	47	0	47
Dusseldorf	0	0	0	53	0	28	53	0	0	53	47	28	0	28	0	28
Essen	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Dortmund	0	53	0	0	53	0	28	53	0	28	53	47	28	0	0	0

Table 3: Final Link Load Matrix (Gb/s) — Hops-Based Routing, Shortest-First Strategy

City	**N**	**B**	**Ha**	**Ho**	**Be**	**L**	**Nu**	**M**	**U**	**S**	**K**	**F**	**Ko**	**D**	**E**	**Do**
Norden	0	212	0	0	0	0	0	0	0	0	0	0	0	0	0	206
Bremen	284	0	103	284	0	0	0	0	0	0	0	0	0	0	0	0
Hamburg	0	103	0	109	103	0	0	0	0	0	0	0	0	0	0	0
Hannover	0	331	190	0	81	312	0	0	0	0	0	368	0	0	0	468
Berlin	0	0	75	134	0	203	0	0	0	0	0	0	0	0	0	0
Leipzig	0	0	0	406	175	0	468	0	0	0	0	75	0	0	0	0
Nurnberg	0	0	0	0	0	512	0	343	0	359	0	371	0	0	0	0
Munich	0	0	0	0	0	0	343	0	100	0	0	0	0	0	0	0
Ulm	0	0	0	0	0	0	0	75	0	100	0	0	0	0	0	0
Stuttgart	0	0	0	0	0	0	403	0	75	0	181	0	0	0	0	0
Karlsruhe	0	0	0	0	0	0	0	0	0	181	0	412	0	0	0	0
Frankfurt	0	0	0	424	0	150	390	0	0	0	365	0	446	0	0	0
Koln	0	0	0	0	0	0	0	0	0	0	0	387	0	256	0	128
Dusseldorf	0	0	0	0	0	0	0	0	0	0	0	0	209	0	81	0
Essen	0	0	0	0	0	0	0	0	0	0	0	0	0	106	0	81
Dortmund	181	0	0	393	0	0	0	0	0	0	0	0	156	0	106	0

Table 4: Demand Matrix (Binary) between cities

City	N	B	Ha	Ho	Be	L	Nu	M	U	S	K	F	Ko	D	E	Do
Norden	0	1	1	1	1	1	0	1	0	0	1	0	0	0	0	0
Bremen	1	0	1	1	0	1	0	1	0	1	1	0	0	1	0	1
Hamburg	1	1	0	1	1	0	1	0	0	1	0	1	0	0	0	0
Hannover	1	1	1	0	1	1	1	0	0	1	1	0	1	0	0	0
Berlin	1	0	1	1	0	1	1	1	0	1	0	0	0	1	0	1
Leipzig	1	1	0	1	1	0	1	1	0	1	1	0	1	0	0	0
Nurnberg	0	0	1	1	1	1	0	1	0	1	1	0	1	1	0	1
Munich	1	1	0	0	1	1	1	0	0	1	1	1	0	1	0	1
Ulm	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Stuttgart	1	1	1	0	0	1	1	1	0	1	1	1	1	1	0	1
Karlsruhe	0	1	1	1	1	1	1	1	0	0	1	1	1	1	0	1
Frankfurt	1	1	0	1	0	1	1	1	0	1	0	1	1	1	0	1
Koln	0	0	1	0	0	0	0	1	0	1	1	0	1	1	0	1
Dusseldorf	0	0	0	1	0	1	1	0	0	1	1	1	0	1	0	1
Essen	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Dortmund	0	1	0	0	1	0	1	1	0	1	1	1	1	0	0	0

Table 5: Balance Comparison Summary Across Routing Metrics and Strategies

Metric	Strategy	$L_{\max}$ (Gb/s)	Mean (Gb/s)	Std (Gb/s)	Blocked Traffic
Hops	shortest	980	480.16	264.20	0
Hops	longest	907	480.16	260.99	0
Hops	largest	930	480.16	255.85	0
Distance	shortest	1279	481.28	315.55	0
Distance	longest	1279	481.28	315.55	0
Distance	largest	1279	481.28	315.55	0

7.3 Code

Listing 1: Python Jupyter Notebook

```
import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import scipy as sp
import routing_v2_proj as routing


city = "Norden, Bremen, Hamburg, Hannover, Berlin, Leipzig,
   Nurnberg, Munich, Ulm, Stuttgart, Karlsruhe, Frankfurt, Koln,
   Dusseldorf, Essen, Dortmund"

N = 16

city_list = [c.strip() for c in city.split(',')]
G = nx.Graph()
G.add_nodes_from(city_list)

# Define physical links (edges) based on a typical German
   backbone topology
edges = [
    ("Norden", "Bremen"), ("Norden","Dortmund"), ("Bremen", "
       Hamburg"), ("Hamburg", "Hannover"),
    ("Hannover", "Berlin"), ("Hannover", "Bremen"), ("Berlin", "
       Leipzig"), ("Leipzig", "Nurnberg"), ("Nurnberg", "Munich")
       ,
    ("Munich", "Ulm"), ("Ulm", "Stuttgart"), ("Stuttgart", "
       Karlsruhe"), ("Frankfurt", "Koln"),
    ("Koln", "Dusseldorf"), ("Dusseldorf", "Essen"), ("Essen", "
       Dortmund"), ("Dortmund", "Hannover"),
    ("Frankfurt", "Hannover"), ("Koln", "Dortmund"), ("Hamburg",
       "Berlin"), ("Hannover", "Leipzig"),
    ("Leipzig", "Frankfurt"), ("Frankfurt", "Nurnberg"), ("
       Stuttgart","Nurnberg"), ("Frankfurt", "Karlsruhe")
]
G.add_edges_from(edges)

# Create the adjacency matrix
adj_matrix = nx.to_numpy_array(G)
```

```
df_topology = pd.DataFrame(adj_matrix, index=G.nodes(), columns=G
    .nodes())


plt.figure(figsize=(12, 8))
nx.draw(
    G,
    with_labels=True,
    node_size=2000,
    node_color="lightblue",
    font_size=10,
    font_weight="bold",
    edge_color="gray"
)
plt.title("German Backbone Network Topology")
plt.show()

pos = {
    "Norden": (0, 6),
    "Hamburg": (2, 6),
    "Bremen": (1, 5),
    "Hannover": (2, 4),
    "Berlin": (4, 5),
    "Leipzig": (4, 4),
    "Dortmund": (0.8, 3.2),       # moved slightly left and up
    "Essen": (1.6, 3.3),          # moved slightly up
    "Dusseldorf": (1.3, 2.5),     # moved slightly right
    "Koln": (1, 1.8),             # moved slightly down
    "Frankfurt": (3, 2.5),
    "Karlsruhe": (3, 1.5),
    "Stuttgart": (4, 1),
    "Ulm": (5, 0.8),
    "Nurnberg": (4.5, 2),
    "Munich": (6, 0.5),
}

plt.figure(figsize=(12, 8))
nx.draw(
    G,
    pos,
    with_labels=True,
```

```
    node_size=2000,
    node_color="lightblue",
    edge_color="gray",
    font_size=10
)
plt.title("German Backbone Network (Geographic Layout)")
plt.show()

degrees = df_topology.sum(axis=1)
print("Degrees of each node:")
print(degrees)

min_degree = degrees.min()
max_degree = degrees.max()
avg_degree = degrees.mean()

print(f"Minimum degree: {min_degree}")
print(f"Maximum degree: {max_degree}")
print(f"Average degree: {avg_degree:.2f}")

var_degree = degrees.var(ddof=0)  # population variance
# If you prefer sample variance, use ddof=1

print(f"Variance of degree: {var_degree:.2f}")

bins = np.arange(min_degree - 0.5, max_degree + 1.5, 1)

degrees.plot(kind='hist',
             bins=bins,
             edgecolor='black',
             rwidth=0.8,
             figsize=(10, 6))
plt.xlabel('Node Degree')
plt.ylabel('Number of Nodes (Frequency)')
plt.title('Node Degree Distribution Histogram')
plt.grid(axis='y', alpha=0.5)
plt.show()

weighted_matrix = df_topology.copy()

weighted_matrix.loc['Norden', 'Bremen'] = 121.73
```

```
weighted_matrix.loc['Bremen', 'Norden'] = 121.73

weighted_matrix.loc['Norden', 'Dortmund'] = 232.84
weighted_matrix.loc['Dortmund', 'Norden'] = 232.84

weighted_matrix.loc['Bremen', 'Hamburg'] = 94.8
weighted_matrix.loc['Hamburg', 'Bremen'] = 94.8

weighted_matrix.loc['Hamburg', 'Hannover'] = 131.22
weighted_matrix.loc['Hannover', 'Hamburg'] = 131.22

weighted_matrix.loc['Hamburg', 'Berlin'] = 256.28
weighted_matrix.loc['Berlin', 'Hamburg'] = 256.28

weighted_matrix.loc['Hannover', 'Berlin'] = 250.91
weighted_matrix.loc['Berlin', 'Hannover'] = 250.91

weighted_matrix.loc['Hannover', 'Bremen'] = 101
weighted_matrix.loc['Bremen', 'Hannover'] = 101

weighted_matrix.loc['Hannover', 'Leipzig'] = 215.78
weighted_matrix.loc['Leipzig', 'Hannover'] = 215.78

weighted_matrix.loc['Hannover', 'Frankfurt'] = 262.53
weighted_matrix.loc['Frankfurt', 'Hannover'] = 262.53

weighted_matrix.loc['Hannover', 'Dortmund'] = 183.11
weighted_matrix.loc['Dortmund', 'Hannover'] = 183.11

weighted_matrix.loc['Berlin', 'Leipzig'] = 149.91
weighted_matrix.loc['Leipzig', 'Berlin'] = 149.91

weighted_matrix.loc['Leipzig', 'Nurnberg'] = 230.08
weighted_matrix.loc['Nurnberg', 'Leipzig'] = 230.08

weighted_matrix.loc['Leipzig', 'Frankfurt'] = 294.3
weighted_matrix.loc['Frankfurt', 'Leipzig'] = 294.3

weighted_matrix.loc['Nurnberg', 'Munich'] = 151.51
weighted_matrix.loc['Munich', 'Nurnberg'] = 151.51
```

```
weighted_matrix.loc['Nurnberg', 'Stuttgart'] = 157.87
weighted_matrix.loc['Stuttgart', 'Nurnberg'] = 157.87

weighted_matrix.loc['Nurnberg', 'Frankfurt'] = 186.99
weighted_matrix.loc['Frankfurt', 'Nurnberg'] = 186.99

weighted_matrix.loc['Munich', 'Ulm'] = 122.06
weighted_matrix.loc['Ulm', 'Munich'] = 122.06

weighted_matrix.loc['Stuttgart', 'Karlsruhe'] = 62.93
weighted_matrix.loc['Karlsruhe', 'Stuttgart'] = 62.93

weighted_matrix.loc['Karlsruhe', 'Frankfurt'] = 124.12
weighted_matrix.loc['Frankfurt', 'Karlsruhe'] = 124.12

weighted_matrix.loc['Frankfurt', 'Koln'] = 152.52
weighted_matrix.loc['Koln', 'Frankfurt'] = 152.52

weighted_matrix.loc['Koln', 'Dusseldorf'] = 34.25
weighted_matrix.loc['Dusseldorf', 'Koln'] = 34.25

weighted_matrix.loc['Koln', 'Dortmund'] = 73.17
weighted_matrix.loc['Dortmund', 'Koln'] = 73.17

weighted_matrix.loc['Dusseldorf', 'Essen'] = 31.56
weighted_matrix.loc['Essen', 'Dusseldorf'] = 31.56

weighted_matrix.loc['Essen', 'Dortmund'] = 31.63
weighted_matrix.loc['Dortmund', 'Essen'] = 31.63

weighted_matrix.loc['Stuttgart', 'Ulm'] = 73.32
weighted_matrix.loc['Ulm', 'Stuttgart'] = 73.32


weighted_matrix = weighted_matrix*1.8

# Create an empty graph
G_weighted = nx.Graph()

# Add nodes (optional, but you have positions)
for node in pos:
```

```
    G_weighted.add_node(node)

# Add weighted edges from the weighted_matrix
for i, node1 in enumerate(weighted_matrix.index):
    for j, node2 in enumerate(weighted_matrix.columns):
        weight = weighted_matrix.iloc[i, j]
        if weight != 0:  # only add edges that exist
            G_weighted.add_edge(node1, node2, weight=weight)

plt.figure(figsize=(12, 8))

# Draw nodes and edges
nx.draw(
    G_weighted,
    pos,
    with_labels=True,
    node_size=2000,
    node_color="lightblue",
    edge_color="gray",
    font_size=10
)

# Format edge labels to 2 decimal places and increase font size
edge_labels = { (u, v): f"{d['weight']:.2f}" for u, v, d in
   G_weighted.edges(data=True) }
nx.draw_networkx_edge_labels(
    G_weighted,
    pos,
    edge_labels=edge_labels,
    font_size=12  # increased from 8 to 12
)

plt.title("German Backbone Network with Weights")
plt.show()

weighted_list = weighted_matrix.values.tolist()
unweighted_list = df_topology.values.tolist()
print(f"Weighted matrix shape: {len(weighted_list)}x{len(
   weighted_list[0])}")
print(f"Unweighted matrix shape: {len(unweighted_list)}x{len(
   unweighted_list[0])}")
```

```
# For UNWEIGHTED Graph

import routing_v2_proj as routing

# Create graph object
graph = routing.Graph()

# 1. Get shortest paths for UNWEIGHTED graph
print("=== UNWEIGHTED GRAPH ===")
unweighted_paths = routing.shortestPaths(graph, unweighted_list)

# 2. Count hops
hop_matrix_unweighted = routing.countHops(unweighted_paths)

print(f"Number of nodes: {len(unweighted_list)}")
print(f"Hop matrix (sample): {hop_matrix_unweighted[:3]}")

# For WEIGHTED Graph

print("\n=== WEIGHTED GRAPH ===")
weighted_paths = routing.shortestPaths(graph, weighted_list)

# Count hops for weighted (but uses distances)
hop_matrix_weighted = routing.countHops(weighted_paths)

# To get distances, you need to extract them from paths
def extract_distances_and_paths(paths_list):
    """Extract distances and paths for all node pairs"""
    all_distances = []
    all_paths = []

    for src_paths in paths_list:
        for p in src_paths:
            all_distances.append({
                'source': p['source'],
                'destination': p['destination'],
                'distance': p['distance'],
                'path': p['path'][0] if p['path'] else []  # Take
                    first shortest path
            })
```

```
    return all_distances

weighted_distances = extract_distances_and_paths(weighted_paths)
unweighted_distances = extract_distances_and_paths(
   unweighted_paths)

# Extract hops and distances
hops_data = []
distances_data = []

for item in weighted_distances:
    if item["source"] != item["destination"]:
        # Number of hops = number of nodes in path - 1
        hops = len(item["path"]) - 1
        hops_data.append(hops)
        distances_data.append(item["distance"])

# ------------------------------
# Histogram: Number of Hops
# ------------------------------
plt.figure(figsize=(10, 6))

# Discrete bins centered on integers
bins_hops = np.arange(min(hops_data) - 0.5, max(hops_data) + 1.5,
    1)
plt.hist(hops_data, bins=bins_hops, edgecolor="black", rwidth
   =0.8)

plt.xlabel("Number of Hops")
plt.ylabel("Frequency")
plt.title("Histogram of Number of Hops")
plt.grid(axis="y", alpha=0.5)
plt.show()

# ------------------------------
# Histogram: Distances
# ------------------------------
plt.figure(figsize=(10, 6))

plt.hist(distances_data, bins=25, edgecolor="black", rwidth=0.8)
```

```
plt.xlabel("Distance")
plt.ylabel("Frequency")
plt.title("Histogram of Node Distances")
plt.grid(axis="y", alpha=0.5)
plt.show()

N_bi = N*(N-1)/2


# for the hop matrix, doing the same thing.
# Total 1s in upper triangle (excluding diagonal)
sum_h = int(np.triu(np.array(hop_matrix_unweighted), k=1).sum())

sum_h

avg_no_hops_per_demand = sum_h/N_bi
avg_no_hops_per_demand

semi_emp_avg_no_hops_per_demand_1 = np.sqrt((N-2)/(avg_degree-1))
semi_emp_avg_no_hops_per_demand_1

semi_emp_avg_no_hops_per_demand_2 = 1.12*np.sqrt((N)/(avg_degree)
   )
semi_emp_avg_no_hops_per_demand_2

H = np.array(hop_matrix_unweighted)
N = len(city_list)

G_logical = nx.Graph()
G_logical.add_nodes_from(city_list)

for i in range(N):
    for j in range(i + 1, N):
        G_logical.add_edge(
            city_list[i],
            city_list[j],
            hops=int(H[i, j])
        )

        plt.figure(figsize=(8,6))
```

```
pos = nx.spring_layout(G_logical, seed=42)

nx.draw(
    G_logical, pos,
    with_labels=True,
    node_size=900,
    font_size=8,
    alpha=0.6
)

edge_labels = nx.get_edge_attributes(G_logical, "hops")
nx.draw_networkx_edge_labels(
    G_logical, pos,
    edge_labels=edge_labels,
    font_size=6
)

plt.title("Full Logical Topology (Hop Distances)")
plt.show()

root = "Frankfurt"

tree = nx.single_source_shortest_path_length(G, root)

G_tree = nx.Graph()
for dst, hops in tree.items():
    if dst != root:
        G_tree.add_edge(root, dst, hops=hops)

plt.figure(figsize=(10, 8))
pos = nx.spring_layout(G_tree, seed=42)

nx.draw(G_tree, pos, with_labels=True, node_size=1200)
nx.draw_networkx_edge_labels(
    G_tree, pos,
    edge_labels=nx.get_edge_attributes(G_tree, "hops")
)

plt.title(f"Logical Hop Tree from {root}")
plt.show()
```

```python
# Minimum node degree
min_degree = min(dict(G.degree()).values())

# Node connectivity
node_connectivity = nx.node_connectivity(G)

# Edge connectivity
edge_connectivity = nx.edge_connectivity(G)

# Algebraic connectivity
algebraic_connectivity = nx.algebraic_connectivity(G)



plt.figure(figsize=(8,6))
pos = nx.spring_layout(G_logical, seed=42)

nx.draw(
    G_logical, pos,
    with_labels=True,
    node_size=900,
    font_size=8,
    alpha=0.6
)

edge_labels = nx.get_edge_attributes(G_logical, "hops")
nx.draw_networkx_edge_labels(
    G_logical, pos,
    edge_labels=edge_labels,
    font_size=6
)

plt.title("Full Logical Topology (Hop Distances)")
plt.show()

print(f"Minimum node degree   (G): {min_degree}")
print(f"Node connectivity   (G): {node_connectivity}")
print(f"Edge connectivity   (G): {edge_connectivity}")
print(f"Algebraic connectivity   : {algebraic_connectivity:.4f}")
```

```
x = "Hamburg"
y = "Stuttgart"

# Minimum  x y  edge cut
edge_cut = nx.minimum_edge_cut(G, x, y)

print("Edge cutset:")
print(edge_cut)
print("Cut size:", len(edge_cut))

# Minimum  x y  node cut
node_cut = nx.minimum_node_cut(G, x, y)

print("Node cutset:")
print(node_cut)
print("Cut size:", len(node_cut))

# for unweighted graph


# Service (primary) path: minimum-hop path
service_path = nx.shortest_path(G, x, y)

print("Service path:")
print(service_path)
print("Hops:", len(service_path) - 1)

# All simple paths between x and y
all_paths = list(nx.all_simple_paths(G, x, y))

print(f"Total simple paths: {len(all_paths)}")

# 1) Select service path (shortest path in hops)
service_path = nx.shortest_path(G, x, y)
print("\nService path:")
print(service_path, "hops:", len(service_path) - 1)

# 2) Store USED edges (service path first)
used_edges = set(
    frozenset((u, v)) for u, v in zip(service_path[:-1],
       service_path[1:])
```

```
)

# 3) Sort all paths by hop count
all_paths_sorted = sorted(all_paths, key=lambda p: len(p) - 1)

# 4) Select mutually edge-disjoint backup paths
edge_disjoint_backups = []

for path in all_paths_sorted:
    path_edges = set(
        frozenset((u, v)) for u, v in zip(path[:-1], path[1:])
    )
    if path_edges.isdisjoint(used_edges):
        edge_disjoint_backups.append(path)
        used_edges.update(path_edges)  # reserve edges

    if len(edge_disjoint_backups) == 3:  # limit number of
       backups
        break

# 5) Print results
print("\nMutually edge-disjoint backup paths:")
for p in edge_disjoint_backups:
    print(p, "hops:", len(p) - 1)


# 1) Select the service path (shortest path in hops)
service_path = nx.shortest_path(G, x, y)
print("\nService path:")
print(service_path, "hops:", len(service_path) - 1)

# 2) Store USED nodes (exclude source and destination)
used_nodes = set(service_path[1:-1])

# 3) Sort all paths by hop count
all_paths_sorted = sorted(all_paths, key=lambda p: len(p) - 1)

# 4) Select mutually node-disjoint backup paths
node_disjoint_backups = []

for path in all_paths_sorted:
```

```
    path_nodes = set(path[1:-1])  # intermediate nodes only

    if path_nodes.isdisjoint(used_nodes):
        node_disjoint_backups.append(path)
        used_nodes.update(path_nodes)  # reserve nodes

    if len(node_disjoint_backups) == 3:  # limit backups
        break

# 5) Print results
print("\nMutually node-disjoint backup paths:")
for p in node_disjoint_backups:
    print(p, "hops:", len(p) - 1)


# 1) Service path (shortest distance)
service_path = nx.shortest_path(Gw, x, y, weight="weight")
service_dist = nx.shortest_path_length(Gw, x, y, weight="weight")

print("\nWeighted service path:")
print(service_path, "distance:", service_dist)

# 2) Store used edges (unordered)
used_edges = set(
    frozenset((u, v)) for u, v in zip(service_path[:-1],
       service_path[1:])
)

# 3) Generate candidate paths ordered by total distance
candidate_paths = list(nx.shortest_simple_paths(Gw, x, y, weight=
   "weight"))

# 4) Select mutually edge-disjoint backup paths
edge_disjoint_backups = []

for path in candidate_paths:
    path_edges = set(
        frozenset((u, v)) for u, v in zip(path[:-1], path[1:])
    )

    if path_edges.isdisjoint(used_edges):
```

```
            edge_disjoint_backups.append(path)
            used_edges.update(path_edges)

        if len(edge_disjoint_backups) == 3:
            break

# 5) Print results
print("\nWeighted mutually edge-disjoint backup paths:")
for p in edge_disjoint_backups:
    dist = nx.path_weight(Gw, p, weight="weight")
    print(p, "distance:", dist)


# 1) Service path (shortest distance)
service_path = nx.shortest_path(Gw, x, y, weight="weight")
service_dist = nx.shortest_path_length(Gw, x, y, weight="weight")

print("\nWeighted service path:")
print(service_path, "distance:", service_dist)

# 2) Store used intermediate nodes
used_nodes = set(service_path[1:-1])

# 3) Generate candidate paths ordered by total distance
candidate_paths = list(nx.shortest_simple_paths(Gw, x, y, weight=
    "weight"))

# 4) Select mutually node-disjoint backup paths
node_disjoint_backups = []

for path in candidate_paths:
    path_nodes = set(path[1:-1])  # intermediate nodes only

    if path_nodes.isdisjoint(used_nodes):
        node_disjoint_backups.append(path)
        used_nodes.update(path_nodes)

    if len(node_disjoint_backups) == 3:
        break

# 5) Print results
```

```
print("\nWeighted mutually node-disjoint backup paths:")
for p in node_disjoint_backups:
    dist = nx.path_weight(Gw, p, weight="weight")
    print(p, "distance:", dist)


X = 28
Y = 47
Z = 53

symbol_map = {
    "X": X,
    "Y": Y,
    "Z": Z,
    "0": 0
}

traffic_matrix = np.array([
    [symbol_map[v] for v in row]
    for row in traffic_symbols
])

df_traffic = pd.DataFrame(
    traffic_matrix,
    index=city_list,
    columns=city_list
)

print(df_traffic)

zero_nodes = ["Essen", "Ulm"]
zero_indices = [city_list.index(n) for n in zero_nodes]
print(zero_indices)

for idx in zero_indices:
    traffic_matrix[idx, :] = 0
    traffic_matrix[:, idx] = 0

demand_matrix = (traffic_matrix > 0).astype(int)

strategies = ["shortest", "longest", "largest"]
```

```
routing.MAX_LINK_CAP = 999999  # uncapacitated

adj_list = df_topology.values.tolist()
traffic_list = traffic_matrix.tolist()

# Hop based routing

graph_obj = routing.Graph()
paths_hops = routing.shortestPaths(graph_obj, adj_list)
hop_matrix = routing.countHops(paths_hops)

# distance based routing

distance_adj_list = weighted_matrix.values.tolist()

# shortest path using distances
paths_dist = routing.shortestPaths(graph_obj, distance_adj_list)

def compute_distance_matrix(paths_dist, distance_matrix, N):
    dist_mat = [[0]*N for _ in range(N)]

    for i, row in enumerate(paths_dist):  # i = source index
        for path_info in row:
            j = path_info['destination'] - 1  # convert to 0-
               index
            path = path_info['path'][0]       # inner list of
               nodes
            if len(path) > 1:
                d = 0
                for k in range(len(path)-1):
                    u = path[k]-1  # convert node to 0-index
                    v = path[k+1]-1
                    d += distance_matrix[u][v]
                dist_mat[i][j] = d
            else:
                dist_mat[i][j] = 0  # same node
    return dist_mat


distance_matrix_paths = compute_distance_matrix(
    paths_dist,
```

```
    weighted_matrix.values.tolist(),
    len(city_list)
)


metrics = {
    "Hops": (paths_hops, hop_matrix),
    "Distance": (paths_dist, distance_matrix_paths)
}

for metric_name, (paths, metric_matrix) in metrics.items():
    print(f"\n===== {metric_name.upper()}-BASED ROUTING =====")

    for strategy in strategies:
        print(f"\n--- Strategy: {strategy} ---")

        # 1. Order paths
        ordered_ids = routing.orderPaths(
            paths,
            traffic_list,
            metric_matrix,
            order=strategy
        )

        # 2. Route traffic
        current_load_matrix = routing.create_load_matrix(adj_list
           )

        final_load_matrix, _, _, blocked_traffic, _ = routing.
           route(
            ordered_ids,
            current_load_matrix,
            [row[:] for row in adj_list]
        )

        print(f"Total Load: {np.sum(final_load_matrix)}")
        print(f"Blocked Traffic: {blocked_traffic}")

        # 3. Extract link loads
        loads = []
        link_names = []
```

```
        for u, v in edges:
            u_idx = city_list.index(u)
            v_idx = city_list.index(v)

            load_uv = final_load_matrix[u_idx][v_idx]
            load_vu = final_load_matrix[v_idx][u_idx]

            total_load = load_uv + load_vu
            loads.append(total_load)
            link_names.append(f"{u}-{v}")

        print("Final Load Matrix:")
        final_load_matrix = np.array(final_load_matrix)
        print(final_load_matrix)
        # 4. Plot (ONE plot per strategy)
        plt.figure(figsize=(12, 5))
        plt.bar(link_names, loads)
        plt.xticks(rotation=90)
        plt.ylabel("Load (Gb/s)")
        plt.title(f"{metric_name}-based Routing     {strategy.
            capitalize()} Strategy")
        plt.tight_layout()
        plt.show()

        #
            ------------------------------------------------------------


# CAPACITATED ROUTING ANALYSIS: STRATEGY COMPARISON
# ------------------------------------------------------------
print("\n=== CAPACITATED ROUTING ANALYSIS: STRATEGY COMPARISON
    ===")
import copy

strategies = ["shortest", "longest", "largest"]
metrics = ["hops", "distance"]
routing.graph = routing.Graph()

# Data structure to store results: results[metric][strategy] = {
    alphas, capacities, brs, avg_lens, max_lens}
comp_results = {m: {s: {} for s in strategies} for m in metrics}
```

```
for metric in metrics:
    print(f"\n--- Metric: {metric} ---")
    if metric == "hops":
        current_adj_list = df_topology.values.tolist()
    else:
        current_adj_list = weighted_matrix.values.tolist()

    for strategy in strategies:
        print(f"  > Simulating Strategy: {strategy}")

        # 1. Baseline Run (Uncapacitated)
        routing.MAX_LINK_CAP = 999999
        graph_obj = routing.graph
        initial_paths = routing.shortestPaths(graph_obj,
           current_adj_list)
        hop_matrix_initial = routing.countHops(initial_paths)
        ordered_demands = routing.orderPaths(initial_paths,
           traffic_list, hop_matrix_initial, order=strategy)

        initial_load_matrix = routing.create_load_matrix(
           current_adj_list)
        final_load_m, _, _, _, _ = routing.route(
            copy.deepcopy(ordered_demands),
            initial_load_matrix,
            [row[:] for row in current_adj_list]
        )
        load_values = [final_load_m[i][j] for i in range(N) for j
            in range(N) if final_load_m[i][j] > 0]
        L_max = max(load_values) if load_values else 0

        # 2. Alpha Titration
        alphas, capacities, brs, avg_lens, max_lens = [], [], [],
            [], []
        alpha = 1.0
        while alpha > 0.04:
            cap = int(alpha * L_max)
            if cap == 0: cap = 1

            routing.MAX_LINK_CAP = cap
            try:
```

```
            l_mat, p_mat, d_mat, b_traffic, b_paths = routing
                .route(
                 copy.deepcopy(ordered_demands),
                 routing.create_load_matrix(current_adj_list),
                 [row[:] for row in current_adj_list]
            )
            b_ratio = len(b_paths) / len(ordered_demands)
            d_mat_np = np.array(d_mat, dtype=float)
            d_mat_np[d_mat_np <= 0] = np.nan
            d_mat_np[d_mat_np >= 99999] = np.nan

            m_len = np.nanmax(d_mat_np) if not np.isnan(
                d_mat_np).all() else 0
            a_len = np.nanmean(d_mat_np) if not np.isnan(
                d_mat_np).all() else 0

            alphas.append(alpha)
            capacities.append(cap)
            brs.append(b_ratio)
            avg_lens.append(a_len)
            max_lens.append(m_len)

            if b_ratio > 0.5: break
        except IndexError:
            brs.append(1.0); avg_lens.append(0); max_lens.
                append(0)
            alphas.append(alpha); capacities.append(cap)
            break
        alpha = round(alpha - 0.05, 2)

    comp_results[metric][strategy] = {
        'alphas': alphas, 'capacities': capacities, 'brs':
            brs,
        'avg_lens': avg_lens, 'max_lens': max_lens
    }

    # --- Individual Performance Plot for this Strategy ---
    fig, ax1 = plt.subplots(figsize=(10, 6))
    color = 'tab:red'
    ax1.set_xlabel('Link Capacity (Gb/s)')
    ax1.set_ylabel('Blocking Ratio', color=color)
```

```
        ax1.plot(capacities, brs, marker='o', color=color, label=
            'Blocking Ratio')
        ax1.tick_params(axis='y', labelcolor=color)
        ax1.grid(True, linestyle='--', alpha=0.7)

        ax2 = ax1.twinx()
        ax2.set_ylabel('Path Length', color='black')
        ax2.plot(capacities, avg_lens, marker='^', color='tab:
            blue', label='Avg Path Length')
        ax2.plot(capacities, max_lens, marker='s', color='tab:
            green', label='Max Path Length')
        ax2.tick_params(axis='y')

        plt.title(f'Performance: Metric={metric}, Strategy={
            strategy}')
        fig.legend(loc='upper right', bbox_to_anchor=(1,1),
            bbox_transform=ax1.transAxes)
        plt.savefig(f'performance_{metric}_{strategy}.png')
        plt.close(fig) # Close to save memory

    # 3. Plotting Comparisons for this Metric
    # Plot A: Blocking Ratio Comparison
    plt.figure(figsize=(10, 6))
    for strategy in strategies:
        data = comp_results[metric][strategy]
        plt.plot(data['capacities'], data['brs'], marker='o',
            label=f'Strategy: {strategy}')
    plt.xlabel('Link Capacity (Gb/s)')
    plt.ylabel('Blocking Ratio')
    plt.title(f'Blocking Ratio Comparison (Metric: {metric})')
    plt.legend()
    plt.grid(True, linestyle='--', alpha=0.7)
    plt.savefig(f'comparison_blocking_{metric}.png')
    plt.show()

    # Plot B: Avg Path Length Comparison
    plt.figure(figsize=(10, 6))
    for strategy in strategies:
        data = comp_results[metric][strategy]
        plt.plot(data['capacities'], data['avg_lens'], marker='^'
            , label=f'Strategy: {strategy}')
```

```
    plt.xlabel('Link Capacity (Gb/s)')
    plt.ylabel('Average Path Length')
    plt.title(f'Average Path Length Comparison (Metric: {metric})
        ')
    plt.legend()
    plt.grid(True, linestyle='--', alpha=0.7)
    plt.savefig(f'comparison_avglen_{metric}.png')
    plt.show()

print("\n=== CAPACITATED ANALYSIS COMPLETE ===")
```