

# Load Balancing Optimization in Content Delivery Networks Using Minimum Spanning Tree Algorithms

Athilla Zaidan Zidna Fann – 13524068

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia

[athillazaidanstudy@gmail.com](mailto:athillazaidanstudy@gmail.com), [13524068@std.stei.itb.ac.id](mailto:13524068@std.stei.itb.ac.id)

**Abstract**— This paper proposes a Load-Aware Modified Minimum Spanning Tree (LAM-MST) algorithm to enhance the efficiency of Content Delivery Networks (CDNs). Unlike classical MST approaches that focus solely on minimizing distance, LAM-MST dynamically adjusts edge weights based on node degree to promote fairer load distribution. Through experiments on intercity graphs of major Indonesian cities, the proposed method demonstrates better traffic balancing with minimal cost overhead, highlighting its potential to reduce bottlenecks in real-world CDN infrastructures.

**Index Terms**—LAM-MST, graph, minimum spanning tree, load balancing, content delivery network

## I. INTRODUCTION

Large-scale digital environments have established Content Delivery Networks (CDNs) to become a foundational infrastructure for supporting rapid, reliable, and geographically scalable distribution of web content. Modern CDNs function by strategically deploying edge servers in diverse geographic locations, thereby enhancing to higher availability and reliability of content in distributed networks. This architectural design plays a vital role in meeting the increasing demands of applications like video streaming, online gaming, distributed cloud infrastructures, and large-scale web services.

Increasing user demand presents a major challenge for CDNs, particularly in developing a load balancing mechanism that distributes traffic evenly across servers, thus reducing the risk of bottlenecks and improving system resilience under heavy loads. Load imbalance, particularly in edge-heavy locations, can significantly damage Quality of Service (QoS) and lead to suboptimal user experiences.

However, a critical constraint of standards Minimum Spanning Tree (MST) Algorithms that are often used for maximizing the effectiveness of CDNs is their lack of awareness regarding server-load constraints. Approaches such as Prim's and Kruskal's focus exclusively on minimizing overall link costs, with no regard for balancing traffic among nodes. Consequently, topologies may emerge where strategically placed servers become overloaded due to uneven traffic flow.

To specifically address this limitation, this paper proposes a solution that is **Load-Aware Modified Minimum Spanning Tree (LAM-MST)** approach. In this method, each edge's

weight is adjusted dynamically based on the current load or degree of its connected nodes. By integrating load sensitivity directly into the MST construction process, the resulting network topology aims to maintain low overall cost while achieving better load balancing.

## II. THEORETICAL FOUNDATION

### A. Graph Theory Fundamentals

Graph theory is a fundamental mathematical framework for representing relationships between objects.

#### 1. Definition of Graph

A graph is formally defined as an ordered pair  $G = (V, E)$ , where  $V$  is a nonempty set of vertices, and  $E$  is a set (can be empty) of edges connecting pairs of vertices. If  $\{u, v\} \in E$ , then the vertices  $u$  and  $v$  are said to be adjacent, and the edge is incident to both [9].

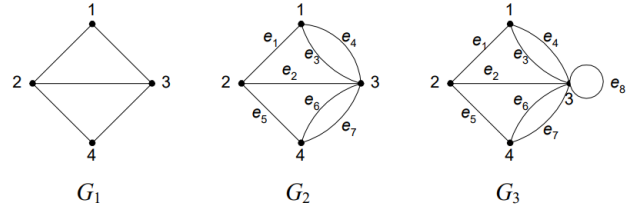


Fig. 1. Graph.

Source: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf>.

The degree of a vertex, denoted as  $\deg(v)$ , is the number of edges that are incident to that vertex. In an *undirected graph*, this is simply the count of all edges connected to  $v$ . In a *directed graph*, each vertex has an *in-degree* (the number of incoming edges) and an *out-degree* (the number of outgoing edges).

#### 2. Types of Graph

Graphs can be classified into several fundamental types based on the structure of their edges and additional attributes

assigned to those edges.

- **Undirected Graph**

In Undirected Graph, each edge is represented in  $\{u, v\}$ . This indicates a mutual relationship between  $u$  and  $v$ . That is the traversal from  $u$  to  $v$  is equivalent to traversal from  $v$  to  $u$ .

- **Directed Graph**

In a directed graph, each edge is represented by an ordered pair  $(u, v)$ , indicating a direction from vertex  $u$  to vertex  $v$ .

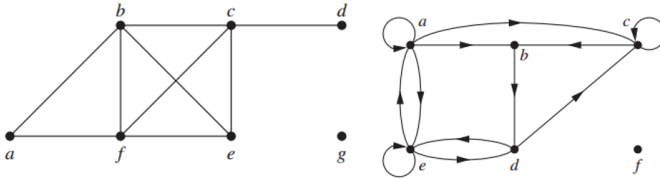


Fig. 2. Undirected and Directed Graph.  
Source: [https://www.academia.edu/15209290/Discrete\\_Maths](https://www.academia.edu/15209290/Discrete_Maths).

- **Unweighted Graph**

An unweighted graph is a graph in which all edges are considered equal in terms of cost, distance, or capacity. In other words, edges do not carry numerical weights, and the existence of an edge simply indicates a connection between two vertices.

- **Weighted Graph**

A weighted graph is a graph in which each edge is assigned a numerical value, known as a weight. These weights may represent various factors such as distance, latency, bandwidth, or cost, depending on the application. Weighted graphs are essential in optimization problems, particularly in algorithms such as Dijkstra's shortest path and Prim's minimum spanning tree, where the weights influence decision-making in traversing the graph [9].

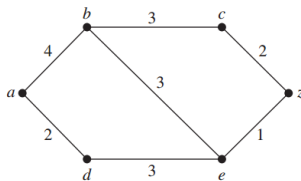


Fig. 3. Weighted Graph.  
Source: [https://www.academia.edu/15209290/Discrete\\_Maths](https://www.academia.edu/15209290/Discrete_Maths).

### 3. Graph Representation

A graph can be represented in several different ways depending on the required computational efficiency and the nature of the graph itself. The two most common representations are the adjacency matrix and the adjacency list.

- **Adjacency Matrix**

An adjacency matrix is a square matrix the size of  $|V| \times |V|$ ,

where  $|V|$  is the number of vertices in the graph. The entry row  $i$  and column  $j$  in the matrix, denoted as  $A[i][j]$  that indicate the edge between vertices  $v_i$  and  $v_j$

- **Adjacency List**

An adjacency list represents a graph as an array or list of lists. Each vertex stores a list of its adjacent vertices along with the weights of the corresponding edges. This representation is memory-efficient for sparse graphs, where most vertex pairs are not connected. It also supports efficient traversal and iteration over neighbors.

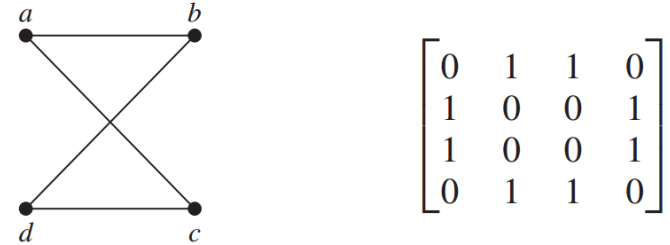


Fig. 4. Adjacency Matrix.  
Source: [https://www.academia.edu/15209290/Discrete\\_Maths](https://www.academia.edu/15209290/Discrete_Maths).

### 4. Connectivity and Paths

A fundamental property of a graph is its *connectivity*, which describes whether or not all vertices in the graph are reachable from one another. A graph is said to be *connected* if, for every pair of vertices  $u$  and  $v$ , there exists a path between them. If even one pair of vertices is not connected by any path, the graph is considered *disconnected*.

A *path* in a graph is defined as a sequence of vertices  $v_1, v_2, \dots, v_k$  such that for each consecutive pair  $(v_i, v_{i+1})$ , there exists an edge  $\{v_i, v_{i+1}\} \in E$ . In a directed graph, the direction of each edge must also be followed. The *length* of a path refers to the number of edges it contains, or the sum of edge weights if the graph is weighted.

- **Simple Path**

A simple path is a path in which all vertices are distinct, except possibly the first and last vertex if the path forms a cycle.

- **Cycle**

A cycle is a path that starts and ends at the same vertex, with no repetition of edges or vertices except the starting/ending point. A graph with no cycles is called an *acyclic graph*, and an undirected acyclic connected graph is specifically known as a *tree* [9].

### 5. Subgraphs

A subgraph is a graph formed by selecting a subset of vertices and edges from a larger graph  $G = (V, E)$ . Formally, a subgraph  $G' = (V', E')$  satisfies  $V' \subseteq V$  and  $E' \subseteq E$ . Subgraphs are useful for simplifying analysis, focusing on specific regions of a graph, or extracting meaningful structures from large networks [9].

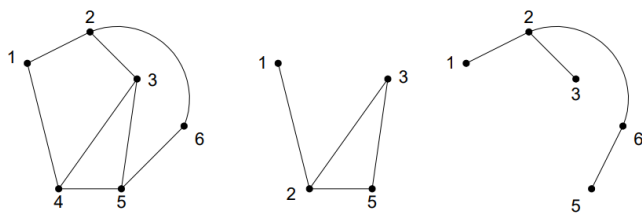


Fig. 5. Subgraph.

Source: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf>.

## B. Tree

A tree is a special type of graph that plays a crucial role in modeling hierarchical and minimal structures in networks, algorithms, and data systems. In the context of this paper, trees serve as the foundation for minimum-cost connectivity models such as Minimum Spanning Trees (MSTs).

### 1. Definition of Tree

A tree is a connected undirected graph with no cycles. That is, for any pair of vertices in the graph, there exists exactly one simple path connecting them. This ensures that a tree is both connected and acyclic [9].

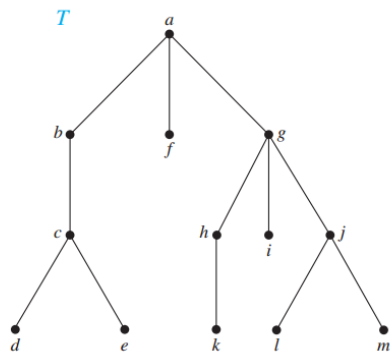


Fig. 6. Tree. Source: [https://www.academia.edu/15209290/Discrete\\_Maths](https://www.academia.edu/15209290/Discrete_Maths).

### 2. Tree Properties

A tree with  $n$  vertices always contains exactly  $n - 1$  edges. This fundamental property ensures that the structure remains minimally connected without forming cycles. Adding any new edge to a tree will inevitably create a single cycle, violating its acyclic nature. Conversely, removing any existing edge will disconnect the graph, as each edge is essential to maintaining connectivity. Moreover, between every pair of vertices in a tree, there exists a unique simple path, emphasizing the efficiency and non-redundancy of the tree structure.

These properties make trees highly efficient structures for establishing minimal connections without redundancy, ensuring optimal use of resources while maintaining full network connectivity. Because of these characteristics, the concept of trees is widely applicable in various domains,

particularly in network design and optimization. In the context of Content Delivery Networks (CDNs), tree-based structures can be used to model efficient content distribution paths that reduce overall transmission cost, minimize latency, and ensure that content reaches end users through the shortest or least expensive routes available.

### 3. Spanning Tree

Given a connected undirected graph  $G = (V, E)$ , a spanning tree is a subgraph  $T = (V, E')$ , where  $E' \subseteq E$ , that includes all vertices of  $G$  and forms a tree. A spanning tree maintains connectivity of the original graph with exactly  $|V| - 1$  edges, and by definition, does not contain cycles [9].

### 4. Minimum Spanning Tree (MST)

When each edge in a graph is assigned a numerical weight representing cost, distance, or latency, the goal is often to find a spanning tree with the minimum total weight. Such a tree is called a Minimum Spanning Tree (MST).

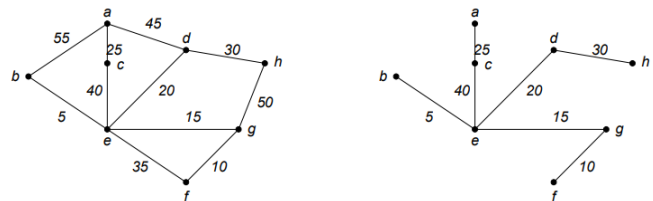


Fig. 7. Minimum Spanning Tree.

Source: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/23-Pohon-Bag1-2024.pdf>.

Two classical greedy algorithms are widely used to compute MSTs:

#### Prim's Algorithm [9]:

1. Start with any vertex.
2. At each step, add the minimum-weight edge that connects a visited vertex to an unvisited vertex.
3. Repeat until all vertices are included in the tree.

#### Kruskal's Algorithm [9]:

1. Sort all edges in increasing order of weight.
2. Add edges one by one, skipping those that would form a cycle.
3. Stop when  $|V| - 1$  edges have been added.

While MSTs are effective at minimizing total connection cost, traditional algorithms like Prim's and Kruskal's do not account for load distribution or node degree. In large-scale network systems such as Content Delivery Networks (CDNs), this can result in bottlenecks, where high-degree nodes are overused.

To address this limitation, the next section introduces an approach that modifies MST to become load-aware: the Load-Aware Modified Minimum Spanning Tree (LAM-MST).

### C. Content Delivery Networks and Load Balancing Challenges

#### 1. Definition of CDN

A Content Delivery Network (CDN) is a geographically distributed network of proxy servers designed to deliver content to end-users with high availability and performance. CDNs replicate content across multiple servers located at various locations, allowing users to access data from the nearest edge server rather than the origin server. This design reduces latency and bandwidth usage, and improves the overall responsiveness of services.

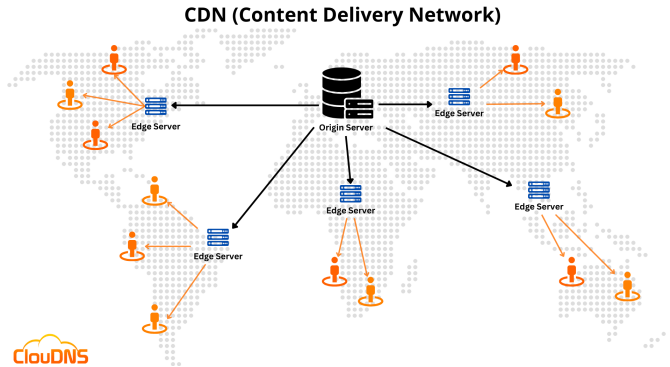


Fig. 8. Content Delivery Network.

Source: <https://www.cloudns.net/blog/wp-content/uploads/2023/04/CDN.png>

#### 2. CDN Architecture

The architecture of a CDN typically consists of:

- **Origin Servers:** The central repository of web content.
- **Edge Servers:** Distributed servers that cache content closer to users.
- **Routing Logic:** Algorithms that determine which edge server should serve which user.

Clients are redirected to the nearest or most optimal edge server based on various criteria such as geographic location, current server load, or network conditions. This hierarchical and distributed architecture enables scalable delivery of content to millions of users simultaneously.

#### 3. Load Balancing Issues in CDNs

As user demands increase, load balancing becomes a critical factor in maintaining Quality of Service (QoS). Ideally, traffic should be distributed evenly across all available servers. However, in practice, some servers—especially those at strategic network positions—may experience disproportionately high traffic, leading to:

- Bottlenecks and increased latency
- Server overloading and potential downtime
- Inefficient use of network resources

These issues degrade user experience and reduce the reliability of the entire network.

#### 4. Limitations of MST in CDN Topology Design

Minimum Spanning Tree (MST) algorithms such as Prim's and Kruskal's are often applied to construct efficient network topologies that minimize total connection cost. However, traditional MST algorithms do not consider node-level metrics such as server capacity or real-time traffic load.

As a result, MST-generated topologies may create structures where high-degree nodes are overloaded with connections and traffic, while others are underutilized. This leads to:

- Imbalanced traffic distribution
- Centralization of network pressure on a few nodes
- Increased risk of congestion and performance degradation

#### 5. Toward a Load-Aware Topology

To overcome these challenges, this paper propose a modified version of MST that incorporates load-awareness into the edge selection process. This approach, called the Load-Aware Modified Minimum Spanning Tree (LAM-MST), aims to produce more balanced network topologies by penalizing high-degree nodes during edge selection.

The next section will introduce the formulation and rationale behind the LAM-MST method.

#### D. Load-Aware Modified Minimum Spanning Tree (LAM-MST)

The LAM-MST is an upgrade of the classical Minimum Spanning Tree algorithm that specifically designed to overcome the problem of uneven load distribution in network topologies, particularly in Content Delivery Network (CDNs). While classical MST algorithm minimize the total edge weights, they often result in structure that overload certain strategic nodes. LAM-MST incorporates node degree awareness into the weight calculation to make a more balanced trees.

##### 1. Edge Weight Modification

The core idea of LAM-MST is to adjust the original edge weights dynamically based on the nodes that being involved. The modified weight  $W'(u, v)$  can be computed as:

$$W'(u, v) = w(u, v) + \alpha \cdot (\deg(u) + \deg(v))$$

In this formula  $w(u, v)$  is the original weight of the edge and  $W'(u, v)$  is the adjusted weight.  $\deg(u)$  and  $\deg(v)$  denote the degrees of  $u$  and  $v$  respectively at the time of evaluation, and lastly  $\alpha$  is the changeable penalty factor that determines how strong node degree should influence edge selection.

This LAM-MST method involve a penalty strategy that favors lower-degree nodes and discourages the selection of edges that connected to a heavy degree nodes, making the network traffic balanced.

##### 2. LAM integration into MST Algorithm

The LAM-MST strategy can be integrated into already existing MST algorithms, such as Prim's and Kruskal's Algorithm by replacing the original weight  $w$  to the new weights  $W'$ .

While both Prim's as well as Kruskal's are theoretically compatible with LAM-MST, Kruskal's algorithm is superior in this case. Kruskal's edge focus approach allows a really straightforward incorporation of the modified weights, since

all the edges can be processed and sorted based on  $W'$  value prior to the tree construction phase. Kruskal's Algorithm also support dynamic updates of node degrees during edge selection, this really aligns well with the LAM strategy.

On the other hand, Prim's Algorithm builds MST incrementally from a single starting vertex, only relying based on a priority queue to select the next minimum weight edge. Integrating LAM strategy to Prim's Algorithm is quite challenging, as it would require frequent updates to the priority queue and recalculations of edge weights during the process. With this in mind, for the purpose of integrating LAM-MST strategy, this paper will be using Kruskal's Algorithm from this point forward.

### 3. Advantages of LAM-MST

The Load-Aware Modified Minimum Spanning Tree (LAM-MST) offers several notable advantages over the traditional MST algorithms, especially when applied in environments that are really sensitive to load distribution such as Content Delivery Network.

- **Balance Load Distribution**

This strategy mainly operate by penalizing edges connected to high-degree nodes, LAM-MST discourages overloading certain nodes and promotes a more uniform spread of connection across the network.

- **Reduced Bottlenecks**

The algorithm avoids the formation of centralized structures where a single node handles a disproportionate share of traffic, thereby reducing the risk of congestion and service latency.

- **Cost Efficiency with Flexibility**

Although LAM-MST may slightly increase the total cost compared to the minimum spanning tree, the penalty factor  $\alpha$  offers flexibility in adjusting the balance between cost and load distribution.

### E. Geographical Distance and the Haversine Formula

In graph-based modeling of real-world systems such as Content Delivery Networks (CDNs), it is important for edge weights to reflect the physical cost of connecting different locations. This study uses geographical distance as a proxy for transmission cost or latency.

To model this realistically, the Haversine formula is used, which computes the great-circle distance between two points on the Earth's surface using their latitude and longitude. This allows the network graph to reflect real-world spatial relationships between cities.

The detailed formula and its application in this study are described in the Methodology section.

## III. METHODOLOGY

### A. System Model

In this research, the underlying infrastructure of a Content Delivery Network (CDN) is modeled as an undirected weighted graph  $G = (V, E)$ , where:

- Each vertex  $v \in V$  represents a server, data center, or edge node in the network.

- Each edge  $e = (u, v) \in E$  represents a potential communication link between two nodes, and is associated with a weight  $w(u, v)$  that reflects the transmission cost.

The weight of each edge may correspond to physical distance, network latency, or bandwidth cost. In this study, a geographical approximation is used, where weights are based on estimated latency between cities or nodes derived from real-world spatial distribution.

The resulting graph is assumed to be connected, sparse, and scalable—characteristics typically found in real CDN topologies. Nodes located in highly populated or strategically significant areas are assigned more edges to simulate dense interconnection, resembling regional content hubs.

This network model provides the basis for evaluating the performance of the Load-Aware Modified Minimum Spanning Tree (LAM-MST) algorithm, in terms of both cost optimization and balanced traffic distribution across the network.

### B. Parameter Configuration

The core parameter in the LAM-MST algorithm is the penalty factor  $\alpha$ , which controls the influence of node degree in modifying edge weights. The modified weight formula:

$$W'(u, v) = w(u, v) + \alpha \cdot (\deg(u) + \deg(v))$$

introduces a trade-off between minimizing total connection cost and balancing node utilization. A higher value of  $\alpha$  increases the penalty for connecting to high-degree nodes, thereby promoting a more even distribution of traffic load.

In this study, two configurations are compared:

- $\alpha = 0.0$   
baseline using the standard MST algorithm without load awareness.
- $\alpha = 100.0$   
moderately penalizes high-degree nodes to encourage load balancing.

These values are chosen to highlight the structural and behavioral differences between conventional MST and the LAM-MST approach under realistic CDN scenarios.

### C. Graph Construction Strategy

To simulate a realistic CDN deployment in Indonesia while keeping the topology manageable, this study focuses on major cities located on the islands of Java and Sumatra. These cities are selected based on their population size, digital infrastructure relevance, and geographical distribution, ensuring both coverage and diversity in connectivity scenarios.

Each city is modeled as a node in the graph, and the geographical distance between them is used as the base edge weight. To calculate these distances accurately over the Earth's curvature, the Haversine formula is applied:

$$d(u, v) = 2r \cdot \arcsin \left( \sqrt{\sin^2 \left( \frac{\Delta\phi}{2} \right) + \cos(\phi_u) \cos(\phi_v) \sin^2 \left( \frac{\Delta\lambda}{2} \right)} \right)$$

where:

- $\phi_u, \phi_v$ : the latitudes of cities  $u$  and  $v$  in radians,



- $\Delta\phi = \phi_v - \phi_u$ ,  $\Delta\lambda = \lambda_v - \lambda_u$ ,
- $r$ : the radius of Earth, approximated as 6371 km.

To maintain sparsity and realism in the graph, a distance threshold of 1000 kilometers is used. An edge is created between two cities only if the calculated distance is less than or equal to this threshold. This reflects practical constraints in physical inter-city routing and ensures that connections are feasible from an infrastructure standpoint.

This results in a sparse, yet connected, graph structure consisting of 10 cities, five from Java and five from Sumatra. The resulting topology shows natural clustering behavior, with dense intra-island connections and limited inter-island bridging.

The adjacency matrix generated from this construction is exported to CSV format and used as input for both MST and LAM-MST simulations.

#### D. Evaluation Metrics

To evaluate the impact of incorporating load-awareness into the minimum spanning tree construction, we compare two configurations of the algorithm: the standard MST with  $\alpha = 0$ , and the Load-Aware Modified MST (LAM-MST) with  $\alpha = 100$ . The following metrics are used for comparison:

- **Total Tree Cost**  
The sum of all edge weights in the final spanning tree. This metric reflects the overall cost of connecting all nodes. Lower values are desirable for cost minimization.
- **Maximum Node Degree**  
The highest degree among all nodes in the spanning tree. A lower maximum degree indicates better load distribution and reduced risk of bottlenecks.
- **Degree Standard Deviation**  
The standard deviation of node degrees across the network. This metric captures the fairness of load distribution—lower values suggest more balanced utilization of nodes.

These metrics allow us to quantitatively compare the structural behavior and cost-effectiveness of LAM-MST against the baseline MST. In particular, the goal is to observe whether the increase in total cost from applying load penalties ( $\alpha = 100$ ) results in better balance in the degree distribution.

### IV. IMPLEMENTATION

#### A. Graph Construction

To construct a realistic model of a Content Delivery Network (CDN), this study selects ten major cities in Indonesia as nodes in the graph. These cities are strategically chosen to represent urban centers across two major islands that is Java and Sumatra where population density, network traffic, and infrastructure development are significantly high. The selected cities are:

- **Java Island**  
Jakarta (JKT), Bandung (BDG), Semarang (SMG), Yogyakarta (YGY), Surabaya (SBY)

- **Sumatra Island**  
Medan (MDN), Padang (PDG), Palembang (PLB), Pekanbaru (PKU), Jambi (JBI)

Each city is associated with a pair of geographical coordinates (latitude and longitude), which were retrieved from the [LatLong.net](https://www.latlong.net) geolocation database and verified using [Google Maps](https://www.google.com/maps/). These coordinates serve as the basis for estimating inter-city distances using the Haversine formula, which computes great-circle distances between points on the Earth's surface. The full mathematical expression of this formula is given in the Methodology section.

To maintain graph sparsity and ensure realistic regional communication, a distance threshold of 1000 km is applied. Only city pairs within this threshold are connected by an edge in the graph. The resulting adjacency matrix and adjacency list are used for further analysis and implementation.

To support the construction process and data handling, several Python libraries are utilized in the implementation:

- **math**  
Used to implement the Haversine formula, particularly for trigonometric operations such as radians, sin, cos, and atan2.
- **numpy**  
Utilized for efficiently creating and managing the adjacency matrix through its multi-dimensional array operations.
- **pandas**  
Employed to construct and export the adjacency matrix and adjacency list into CSV format for further analysis and visualization.

These libraries are chosen due to their efficiency, ease of use, and widespread adoption in scientific computing and data processing within the Python ecosystem.

#### 1. City Coordinates and Code Mapping

```

Graph Construction
# cities name 3 letter code
city_codes = {
    "Jakarta": "JKT",
    "Bandung": "BDG",
    "Semarang": "SMG",
    "Yogyakarta": "YGY",
    "Surabaya": "SBY",
    "Medan": "MDN",
    "Padang": "PDG",
    "Palembang": "PLB",
    "Pekanbaru": "PKU",
    "Jambi": "JBI"
}

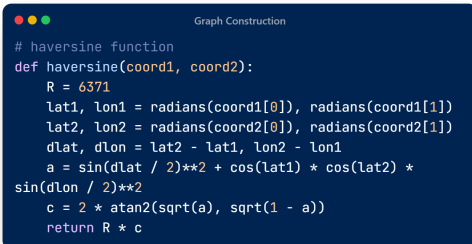
# cities latitude and longitude data
cities = {
    "Jakarta": (-6.175110, 106.865036),
    "Bandung": (-6.917464, 107.619125),
    "Semarang": (-7.005145, 110.438126),
    "Yogyakarta": (-7.795580, 110.369492),
    "Surabaya": (-7.257472, 112.752090),
    "Medan": (3.595196, 98.672226),
    "Padang": (-0.947083, 100.417183),
    "Palembang": (-2.976074, 104.775429),
    "Pekanbaru": (0.507068, 101.447777),
    "Jambi": (-1.610123, 103.613121)
}

```

Fig. 9. Python code for city latitude and longitude data

The code snippet shown defines a Python dictionary containing the names, coordinates (latitude and longitude), and corresponding 3-letter codes for each selected city. This mapping facilitates consistent referencing of city nodes in the graph construction and output files. The use of structured city codes ensures clarity and compactness in matrix representations and algorithm outputs.

## 2. Haversine Distance Calculation

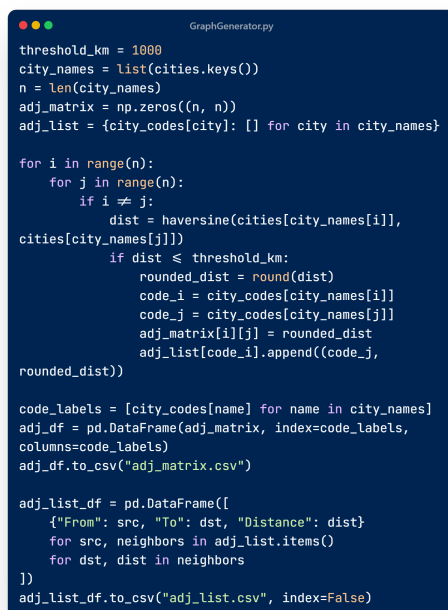


```
# haversine function
def haversine(coord1, coord2):
    R = 6371
    lat1, lon1 = radians(coord1[0]), radians(coord1[1])
    lat2, lon2 = radians(coord2[0]), radians(coord2[1])
    dlat, dlon = lat2 - lat1, lon2 - lon1
    a = sin(dlat / 2)**2 + cos(lat1) * cos(lat2) *
    sin(dlon / 2)**2
    c = 2 * atan2(sqrt(a), sqrt(1 - a))
    return R * c
```

Fig. 10. Python code for Haversine function

The code snippet shown presents the implementation of the Haversine function, which is used to calculate the great-circle distance between two geographic coordinates on the Earth's surface. This function forms the core of edge weight computation, converting real-world latitude and longitude values into approximate distances in kilometers. The result of this function determines whether two cities should be connected based on the specified distance threshold.

## 3. Building the Adjacency Matrix and List



```
threshold_km = 1000
city_names = list(cities.keys())
n = len(city_names)
adj_matrix = np.zeros((n, n))
adj_list = {city_codes[city]: [] for city in city_names}

for i in range(n):
    for j in range(n):
        if i != j:
            dist = haversine(cities[city_names[i]],
            cities[city_names[j]])
            if dist <= threshold_km:
                rounded_dist = round(dist)
                code_i = city_codes[city_names[i]]
                code_j = city_codes[city_names[j]]
                adj_matrix[i][j] = rounded_dist
                adj_list[code_i].append((code_j,
                rounded_dist))

code_labels = [city_codes[name] for name in city_names]
adj_df = pd.DataFrame(adj_matrix, index=code_labels,
columns=code_labels)
adj_df.to_csv("adj_matrix.csv")

adj_list_df = pd.DataFrame([
    {"From": src, "To": dst, "Distance": dist}
    for src, neighbors in adj_list.items()
    for dst, dist in neighbors
])
adj_list_df.to_csv("adj_list.csv", index=False)
```

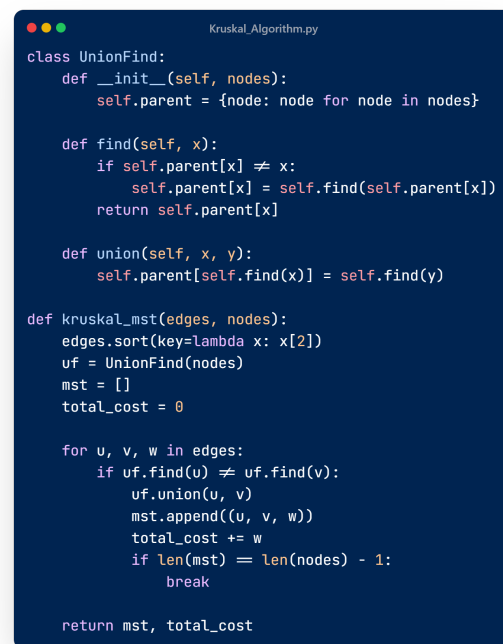
Fig. 11. Python code for constructing adjacent matrix and adjacent list

The code snippet shown utilizes the Haversine function to compute pairwise distances between all cities. If the distance between two cities is less than or equal to the 1000 km threshold, an edge is created in both the adjacency matrix and adjacency list. The matrix format supports efficient distance lookup for algorithms, while the list format is used in the MST construction phase. The adjacency data is also exported to CSV for reproducibility and further analysis.

## B. Kruskal's Minimum Spanning Tree Algorithm

Kruskal's algorithm is used as the baseline implementation of the Load-Aware Modified Minimum Spanning Tree (LAM-MST) with the penalty factor set to  $\alpha = 0$ . In this configuration, edge weights are not modified by node degrees, resulting in a standard minimum spanning tree construction based purely on physical distances.

The implementation is done in Python using a simple edge list and Union-Find (Disjoint Set Union) structure. All edges are sorted by weight, and the algorithm iteratively selects edges that do not form a cycle, resulting in an acyclic, fully connected subgraph with minimum total cost.



```
class UnionFind:
    def __init__(self, nodes):
        self.parent = {node: node for node in nodes}

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        self.parent[self.find(x)] = self.find(y)

def kruskal_mst(edges, nodes):
    edges.sort(key=lambda x: x[2])
    uf = UnionFind(nodes)
    mst = []
    total_cost = 0

    for u, v, w in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            mst.append((u, v, w))
            total_cost += w
            if len(mst) == len(nodes) - 1:
                break

    return mst, total_cost
```

Fig. 12. Kruskal's Algorithm in Python

This version of Kruskal's algorithm serves as a reference point to evaluate the effectiveness of the load-aware version discussed in the next subsection.

## C. Load-Aware Modified Minimum Spanning Tree Algorithm

To address the limitations of traditional MST algorithms, this section presents an enhanced implementation called Load-Aware Modified Minimum Spanning Tree (LAM-MST), based

on the Kruskal algorithm with a penalty factor  $\alpha = 100$ . In this configuration, the edge weights are dynamically adjusted according to the degree of the nodes involved, following the modified formula:

$$W'(u, v) = w(u, v) + \alpha \cdot (\deg(u) + \deg(v))$$

This adjustment discourages over-connection to highly connected nodes and leads to a more balanced load distribution across the network.

The following Python implementation extends the Kruskal's Algorithm baseline by integrating real-time degree tracking during MST construction. The degrees are updated every time an edge is added to the tree.

```
LAM-MST_Algorithm.py

class UnionFind:
    def __init__(self, nodes):
        self.parent = {node: node for node in nodes}

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        self.parent[self.find(x)] = self.find(y)

def lam_mst(edges, nodes, alpha=1):
    deg = {node: 0 for node in nodes}
    mst = []
    total_cost = 0
    uf = UnionFind(nodes)

    while len(mst) < len(nodes) - 1:
        modified_edges = sorted(
            edges, key=lambda x: x[2] + alpha *
            (deg[x[0]] + deg[x[1]])
        )

        for u, v, w in modified_edges:
            if uf.find(u) != uf.find(v):
                uf.union(u, v)
                deg[u] += 1
                deg[v] += 1
                mst.append((u, v, w))
                total_cost += w
                break

    return mst, total_cost
```

Fig. 13. LAM-MST Algorithm in Python

This implementation prioritizes load balancing over purely minimizing distance, and the effect of this trade-off will be analyzed in the next section through structural comparison and cost evaluation.

## V. TESTS AND RESULTS

This section presents the comparative evaluation between the classical Minimum Spanning Tree (MST) and the Load-Aware Modified Minimum Spanning Tree (LAM-MST) algorithm. The goal is to analyze their performance in optimizing connection cost and balancing traffic load across network nodes.

### A. Experimental Setup

All experiments were conducted on a dataset consisting of 10 major cities in Indonesia, spread across Java and Sumatra. The connectivity threshold was set to 1000 km, enabling a reasonably dense network with sufficient connection options.

TABLE I  
ADJACENCY MATRIX OF SELECTED CITIES USED IN EXPERIMENTS  
(IN KM)

	JKT	BDG	SMG	YGY	SBY	MDN	PDG	PLB	PKU	JB
JKT	0	117	405	427	661	0	922	424	956	623
BDG	117	0	311	319	568	0	0	540	0	739
SMG	405	311	0	88	257	0	0	771	0	965
YGY	427	319	88	0	269	0	0	819	0	0
SBY	661	568	257	269	0	0	0	0	0	0
MDN	0	0	0	0	0	0	541	997	462	798
PDG	922	0	0	0	0	541	0	534	198	363
PLB	424	540	771	819	0	997	534	0	536	199
PKU	956	0	0	0	0	462	198	536	0	337
JB	623	739	965	0	0	798	363	199	337	0

TABLE II  
ADJACENCY LIST OF SELECTED CITIES USED IN EXPERIMENTS

From	To	Distance (km)	From	To	Distance (km)
JKT	BDG	117	MDN	PDG	541
JKT	SMG	405	MDN	PLB	997
JKT	YGY	427	MDN	PKU	462
JKT	SBY	661	MDN	JB	798
JKT	PDG	922	PDG	JKT	922
JKT	PLB	424	PDG	MDN	541
JKT	PKU	956	PDG	PLB	534
JKT	JB	623	PDG	PKU	198
BDG	JKT	117	PDG	JB	363
BDG	SMG	311	PLB	JKT	424
BDG	YGY	319	PLB	BDG	540
BDG	SBY	568	PLB	SMG	771
BDG	PLB	540	PLB	YGY	819
BDG	JB	739	PLB	MDN	997
SMG	JKT	405	PLB	PDG	534
SMG	BDG	311	PLB	PKU	536
SMG	YGY	88	PLB	JB	199
SMG	SBY	257	PKU	JKT	956
SMG	PLB	771	PKU	MDN	462
SMG	JB	965	PKU	PDG	198
YGY	JKT	427	PKU	PLB	536
YGY	BDG	319	PKU	JB	337
YGY	SMG	88	JB	JKT	623
YGY	SBY	269	JB	BDG	739
YGY	PLB	819	JB	SMG	965
SBY	JKT	661	JB	MDN	798
SBY	BDG	568	JB	PDG	363
SBY	SMG	257	JB	PLB	199
SBY	YGY	269	JB	PKU	337

Two algorithmic configurations were evaluated:

- Kruskal's Standard MST ( $\alpha = 0$ )  
Edge weights are purely based on geographic distance using the Haversine formula.
- LAM-MST ( $\alpha = 100$ )  
Edge weights are adjusted by adding penalties based on node degree, promoting more balanced connectivity.

The implementation and evaluation were performed using Python. Three primary metrics were used for comparison:

- 1) Total Tree Cost  
sum of all edge weights in the final spanning tree.



- 2) Maximum Node Degree  
the highest degree (number of connections) among all nodes.
- 3) Degree Standard Deviation  
variance in degree distribution across all nodes.

For full reproducibility, all source code and datasets used in this study are publicly available at:

<https://github.com/AthillaZaidan/Load-Balancing-Optimization-in-Content-Delivery-Networks-Using-Minimum-Spanning-Tree-Algorithms>

## B. Results and Analysis

With the that 10 cities data, we get these results for the two approaches:

```

=== Kruskal's MST ( $\alpha = 0$ ) ===
SMG - YGY : 88 km
JKT - BDG : 117 km
PDG - PKU : 198 km
PLB - JBI : 199 km
SMG - SBY : 257 km
BDG - SMG : 311 km
PKU - JBI : 337 km
JKT - PLB : 424 km
MDN - PKU : 462 km

Metrics:
Total Cost: 2393 km
Max Degree: 3
Degree Std Dev: 0.75
Degree per node:
SMG: 3      PKU: 3
YGY: 1      PLB: 2
JKT: 2      JBI: 2
BDG: 2      SBY: 1
PDG: 1      MDN: 1

```

Fig. 14. Kruskal's Algorithm ( $\alpha = 0$ ) result

```

=== LAM-MST ( $\alpha = 100$ ) ===
SMG - YGY : 88 km
JKT - BDG : 117 km
PDG - PKU : 198 km
PLB - JBI : 199 km
SMG - SBY : 257 km
BDG - YGY : 319 km
PKU - JBI : 337 km
JKT - PLB : 424 km
MDN - PDG : 541 km

Metrics:
Total Cost: 2480 km
Max Degree: 2
Degree Std Dev: 0.4
Degree per node:
SMG: 2      PKU: 2
YGY: 2      PLB: 2
JKT: 2      JBI: 2
BDG: 2      SBY: 1
PDG: 2      MDN: 1

```

Fig. 15. LAM-MST ( $\alpha = 100$ ) result

The table below summarizes the result of the two approaches:

TABLE III  
COMPARISON BETWEEN MST AND LAM-MST

Metric	MST ( $\alpha = 0$ )	LAM-MST ( $\alpha = 100$ )
Total Cost (km)	2393	2480
Maximum Node Degree	3	2
Degree Std Deviation	0.75	0.4

To complement the numerical analysis, the following figures visualize the resulting network topologies generated by each algorithm. Each node represents a city, and each edge corresponds to a selected connection in the spanning tree, annotated with the associated distance.

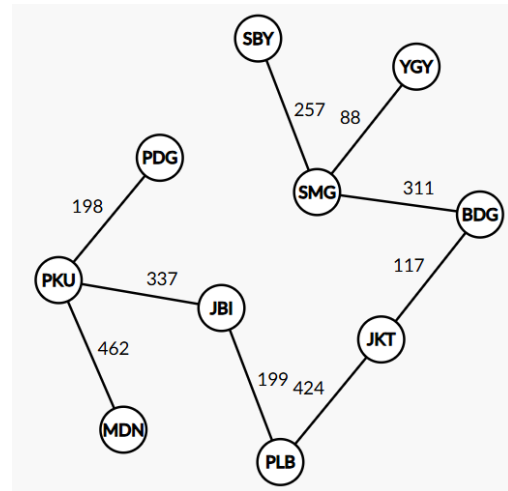


Fig. 16. Visualization of graph that constructed using Kruskal's Algorithm

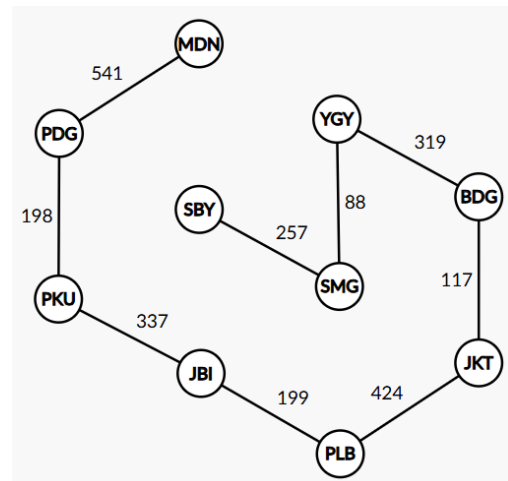


Fig. 17. Visualization of graph that constructed using LAM-MST

The LAM-MST algorithm produces a more evenly distributed network structure, as evidenced by a lower maximum degree and smaller standard deviation in node degrees.

While the total cost increases slightly, this trade-off yields a better load balancing outcome, which is crucial for avoiding bottlenecks in real-world Content Delivery Network (CDN) scenarios.

Overall, LAM-MST demonstrates its effectiveness in distributing load more fairly while maintaining acceptable network efficiency.

Visual inspection of the graphs above supports the quantitative metrics discussed earlier, reinforcing the advantage of incorporating load-awareness into the tree construction process.

## VI. CONCLUSION

This study evaluates the effectiveness of the Load-Aware Modified Minimum Spanning Tree (LAM-MST) approach in optimizing graph-based Content Delivery Networks (CDNs). By integrating node degree penalties into the edge weight calculation, LAM-MST addresses the limitations of traditional Minimum Spanning Tree (MST) algorithms that only focus on minimizing total connection cost.

The implementation and testing on a graph consisting of ten major cities in Indonesia show that while LAM-MST slightly increases the total cost (from 2393 km to 2480 km), it achieves a more balanced structure with lower maximum node degree (from 3 to 2) and reduced degree standard deviation (from 0.75 to 0.4). This result indicates better traffic distribution and load balancing, which are essential in maintaining efficient and scalable CDN performance.

Overall, the LAM-MST algorithm provides a meaningful trade-off between connection efficiency and fairness in node utilization. These findings highlight the importance of considering both cost and load distribution when designing communication networks.

## VII. APPENDIX

### A. Github Repository for this project

<https://github.com/AthillaZaidan/Load-Balancing-Optimization-in-Content-Delivery-Networks-Using-Minimum-Spanning-Tree-Algorithms>

### B. Video Explanation

## VIII. ACKNOWLEDGMENT

The author would like to express sincere gratitude to God for His endless blessings and guidance. Through His grace, the author remained in good health and spirit throughout the process of writing this paper.

The deepest thanks are also extended to the author's beloved parents, whose unwavering support and tireless efforts have been a constant source of strength and motivation.

Special appreciation is given to Mr. Arrival Dwi Sentosa, S.Kom., M.T., lecturer of Discrete Mathematics for class K02 IF1220, for his dedication, clarity in teaching, and continued support during the semester.

The author is also deeply thankful to the author's family in Keluarga Mahasiswa Jambi (KMJ), who have provided a

meaningful space for growth, learning, and belonging throughout this academic journey.

Warm thanks go to fellow classmates in K02 and K01 for their valuable insights and friendly discussions, which greatly contributed to the completion of this paper.

Finally, the author expresses heartfelt appreciation to the author's beloved partner, who has stood by the author's side through endless long nights this semester, offering encouragement, warmth, and unwavering support.

## REFERENCES

- [1] Raja Ayyanar Adarsh Nagarajan. Application of minimum spanning tree algorithm for network reduction of distribution systems, 2014. [Online]. Available: [https://www.researchgate.net/publication/286679449\\_Application\\_of\\_Minimum\\_Spanning\\_Tree\\_algorithm\\_for\\_network\\_reduction\\_of\\_distribution\\_systems](https://www.researchgate.net/publication/286679449_Application_of_Minimum_Spanning_Tree_algorithm_for_network_reduction_of_distribution_systems) [Accessed: Jun. 17, 2025].
- [2] Ahmed Khattab Marwa Mamdouh, Khaled Elsayed. Rpl load balancing via minimum degree spanning tree, 2016. [Online]. Available: [http://eece.cu.edu.eg/~akhattab/files/RPL\\_WiMob\\_2016.pdf](http://eece.cu.edu.eg/~akhattab/files/RPL_WiMob_2016.pdf) [Accessed: Jun. 17, 2025].
- [3] Rajkumar Buyya Mukaddim Pathan and Athena Vakali. Content delivery networks: State of the art, insights, and imperatives, 2008. [Online]. Available: <http://www.buyya.com/papers/CDNIntro-2008.pdf> [Accessed: Jun. 17, 2025].
- [4] Rinaldi Munir. *Graf (Bagian 1)*, 2024. [Online]. Available: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf>. [Accessed: Jun. 18, 2025].
- [5] Rinaldi Munir. *Graf (Bagian 2)*, 2024. [Online]. Available: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/21-Graf-Bagian2-2024.pdf>. [Accessed: Jun. 18, 2025].
- [6] Rinaldi Munir. *Graf (Bagian 3)*, 2024. [Online]. Available: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/22-Graf-Bagian3-2024.pdf>. [Accessed: Jun. 18, 2025].
- [7] Rinaldi Munir. *Pohon (Bagian 1)*, 2024. [Online]. Available: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/23-Pohon-Bag1-2024.pdf>. [Accessed: Jun. 18, 2025].
- [8] Rinaldi Munir. *Pohon (Bagian 2)*, 2024. [Online]. Available: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/24-Pohon-Bag2-2024.pdf>. [Accessed: Jun. 18, 2025].
- [9] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Education, New York, 7 edition, 2012. [Online]. Available: [https://www.academia.edu/15209290/Discrete\\_Maths](https://www.academia.edu/15209290/Discrete_Maths). [Accessed: Jun. 18, 2025].

## STATEMENT

I hereby declare that this paper is an original work, written entirely on my own, and does not involve adaptation, translation, or plagiarism of any other individual's work.

Bandung, 20 June 2025

Athilla Zaidan Zidna Fann, 13524068