Analysis and Implementation of Boruvka's Algorithm

James Calnan

University Of Exeter

Introduction

- Boruvka's algorithm, developed in 1926, is fundamental in graph theory for finding a minimum spanning tree (MST).
- The algorithm is notable for its efficient connection of graph components.
- Applicable in modern contexts, especially in distributed and parallel computing.
- Its relevance today extends to network design, clustering in data science, and infrastructure planning, showcasing its adaptability to large-scale and technologically advanced applications.

Principles of Boruvka's Algorithm

- Boruvka's algorithm works by progressively reducing graph components.
- Initially, each vertex is considered a separate component.
- The algorithm merges these components by selecting the smallest edge connecting separate components.
- Highly parallelisable and efficient, particularly for sparse graphs.

Pseudocode

```
function Boruvka(nodes, edges):
    Initialise each node as a separate component in sets
    Initialise an empty dictionary mst for the Minimum Spanning Tree
    while the number of components in sets is greater than 1:
        Initialise an empty dictionary min_edges for tracking minimum edges for each component
        for each edge in edges:
            Determine the components of the nodes connected by the edge
            If the nodes belong to different components:
                Update min_edges for each component if the current edge is smaller
                than the existing minimum
        for each edge in min_edges:
            If the edge connects two different components:
                Add the edge to mst
                Merge the two components into a single component in sets
        Update the number of components in sets
    return mst
```

Complexity Analysis

- Time Complexity: $O(E \log V)$
- Efficient scaling with the number of nodes.
- Space Complexity: Influenced by graph representation (Adjacency List: $O(V + E)$, Matrix: $O(V^2)$).

Graph Visualisation Code

```python
# Need a way to print the graph so that it is easy to visualize using networkx
def print_graph(nodes, edges, layout=nx.shell_layout, save=False, filename='graph.png'):
    G = nx.Graph()
    G.add_nodes_from(nodes)
    for (u, v), weight in edges.items():
        G.add_edge(u, v, weight=weight)

    pos = layout(G)  # Get the position layout for nodes
    nx.draw(G, pos, with_labels=True, node_size=500,
            node_color='lightblue', font_size=8, font_weight='bold')

    # Draw edge labels
    edge_labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='red')

    # If save is true, save the graph
    if save:
        plt.savefig(f'mst graphs/{filename}', dpi=600)

    plt.show()
```

Simple Graph

```
nodes = ['A', 'B', 'C', 'D', 'E', 'F']
edges = {
    ('A', 'B'): 3,
    ('A', 'E'): 1,
    ('A', 'C'): 6,
    ('B', 'C'): 5,
    ('B', 'D'): 9,
    ('C', 'D'): 7,
    ('C', 'F'): 11,
    ('D', 'E'): 2,
    ('E', 'F'): 3,
    ('F', 'B'): 2
}
```

Graph Visualisation

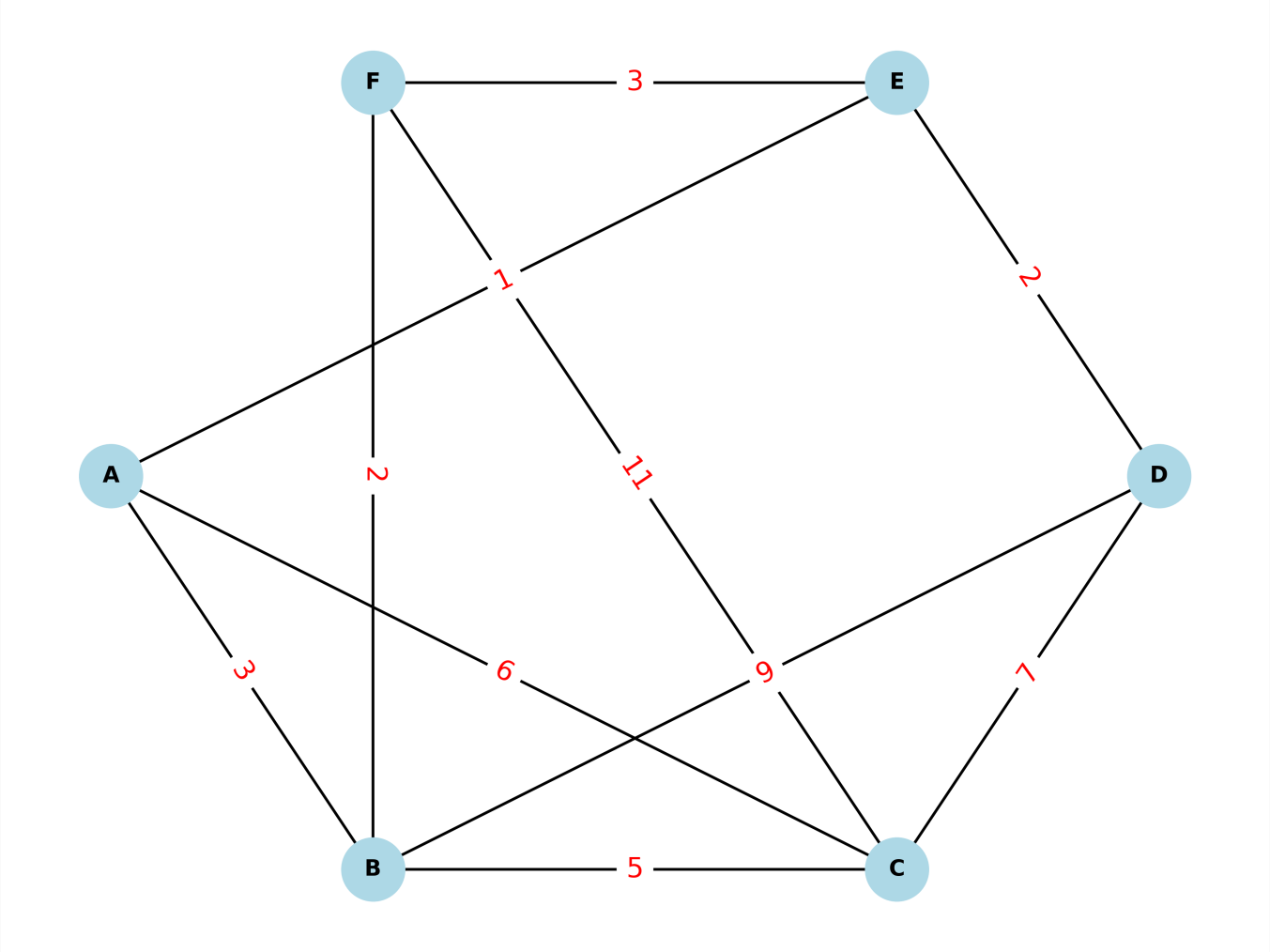

Figure 1: Graph Representation

Boruvkas Implementation

```
def boruvka(nodes, edges):
    sets = {node: node for node in nodes}
    mst = {}  # Dictionary to store the minimum spanning tree

    edge_heap = []
    for edge, weight in edges.items():
        heapq.heappush(edge_heap, (weight, edge))

    while len(mst) < len(nodes) - 1:
        min_edges = {}  # Store minimum edge for each component

        for weight, (node1, node2) in edge_heap:
            set1, set2 = sets[node1], sets[node2]
            if set1 != set2:
                if set1 not in min_edges or weight < min_edges[set1][0]:
                    min_edges[set1] = (weight, (node1, node2))
                if set2 not in min_edges or weight < min_edges[set2][0]:
                    min_edges[set2] = (weight, (node1, node2))

        for weight, (node1, node2) in min_edges.values():
            set1, set2 = sets[node1], sets[node2]
            edge = None
            if set1 != set2:
                edge = (node1, node2)
                mst[edge] = weight
                new_set = min(set1, set2)
                for node in sets:
                    if sets[node] == set1 or sets[node] == set2:
                        sets[node] = new_set

    return mst  # Return the minimum spanning tree
```

MST Visualisation

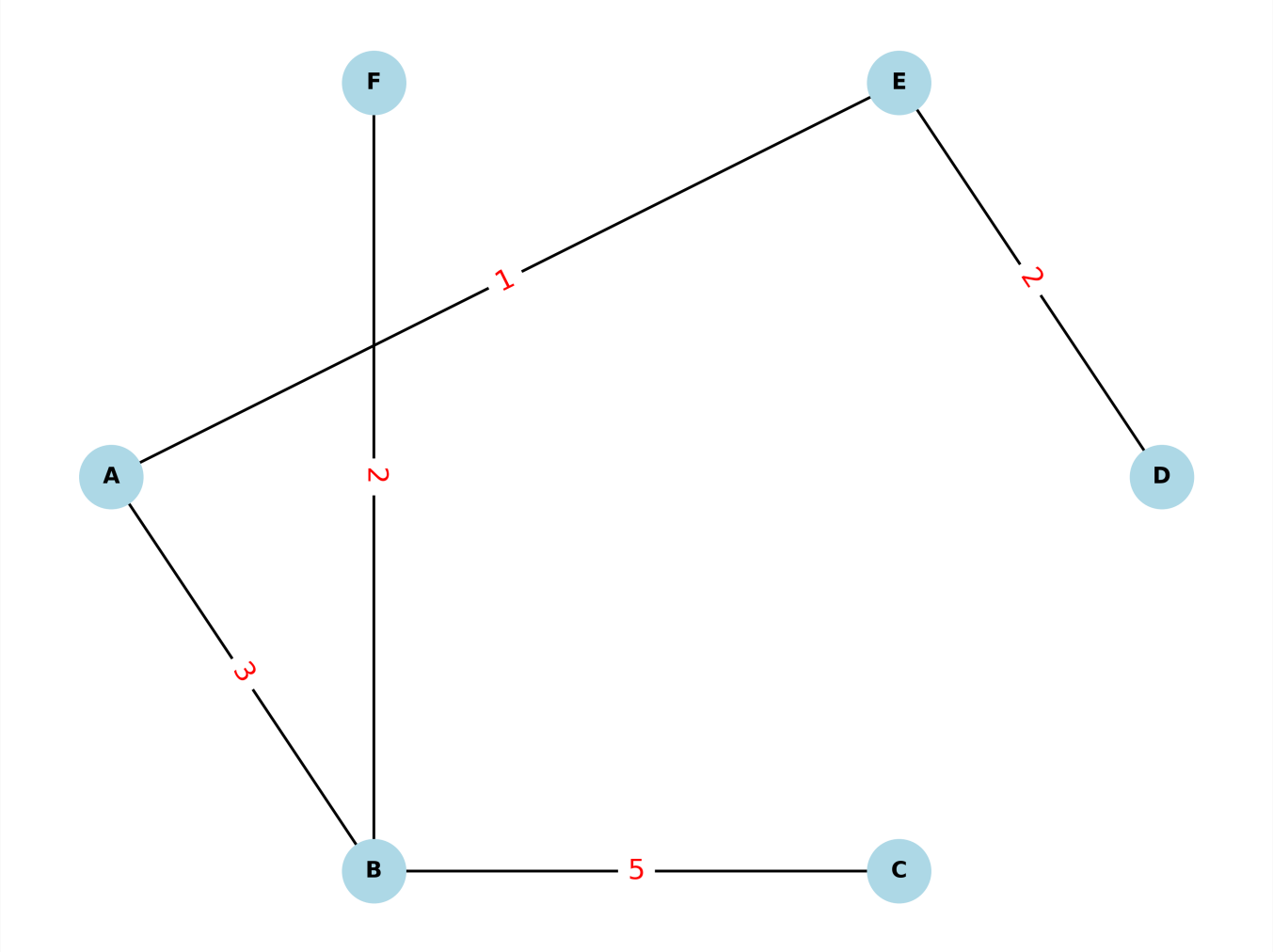

Figure 2: MST Representation

Code to create a random connected graph

```python
# Function to create a random connected graph with n nodes and m edges
def generate_random_connected_graph(n, m):
    # Generate node names (e.g., 'A', 'B', 'C', ..., 'Z', 'AA', 'AB', ...)
    nodes = [chr(65 + i) if i < 26 else chr(64 + i // 26) + chr(65 + i % 26) for i in range(n)]

    # Ensure the graph is connected by creating a path that visits each node once
    # Shuffle the nodes to ensure the path is random
    shuffled_nodes = random.sample(nodes, n)
    # Connect the shuffled nodes with edges
    edges = {(shuffled_nodes[i], shuffled_nodes[i + 1]): random.randint(1, 10) for i in range(n - 1)}

    # Create all possible edges between nodes except the ones already in the path
    all_possible_edges = [(nodes[i], nodes[j]) for i in range(n)
                          for j in range(i+1, n) if (nodes[i], nodes[j]) not in edges]

    # Randomly select additional edges to add, ensuring not to exceed m or
    # the number of remaining possible edges
    additional_edges_count = min(m - (n - 1), len(all_possible_edges))
    if additional_edges_count > 0:
        additional_random_edges = random.sample(all_possible_edges, additional_edges_count)
        # Assign a random weight to each additional edge
        additional_edges = {edge: random.randint(1, 10) for edge in additional_random_edges}
        # Add the additional edges to the graph
        edges.update(additional_edges)

    return nodes, edges

n = 10
m = 25
# Initialize the graph with n nodes and m edges
nodes, edges = generate_random_connected_graph(n, m)
```

Another Example

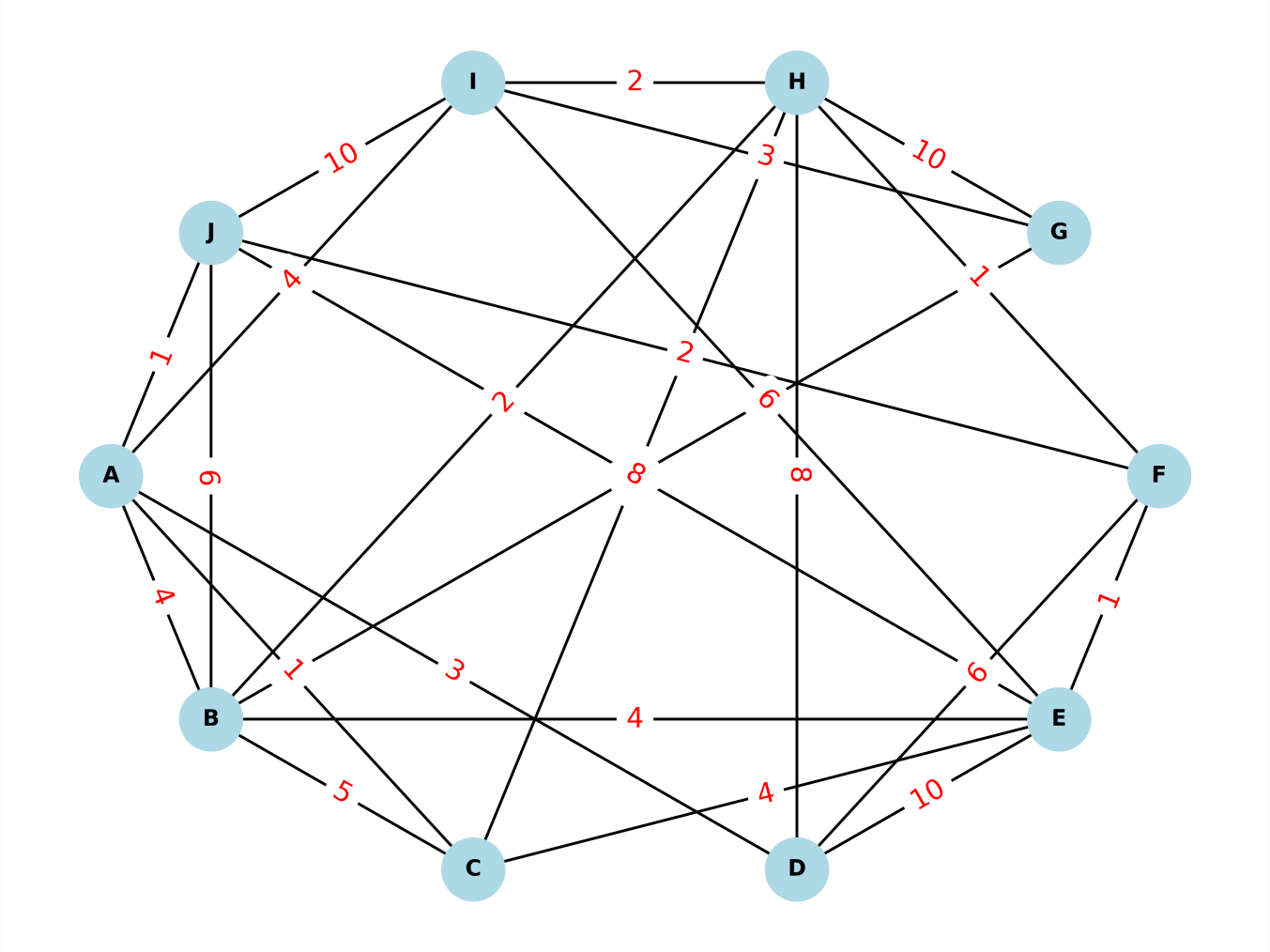

Figure 3: Random Graph

MST

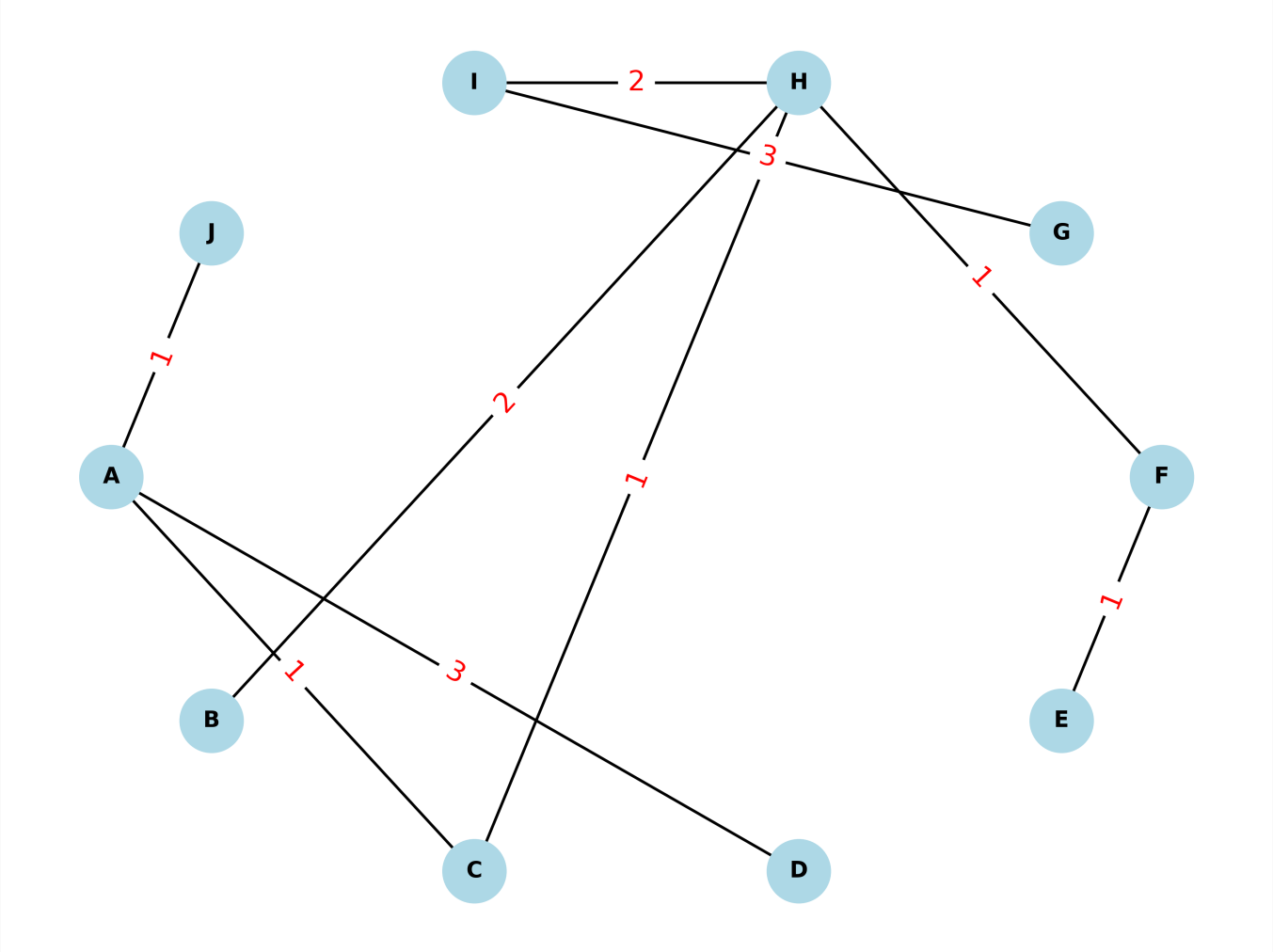

Figure 4: MST

Case Study: San Joaquin County Road Network

- Application of Boruvka's algorithm to real-world road network data.
- Spacial Dataset Repository [1]
- Contains NodeID, Normalised X and Y, Edge Data
- Significantly reduced the total weight of the network.

Original Graph Weight: 831572

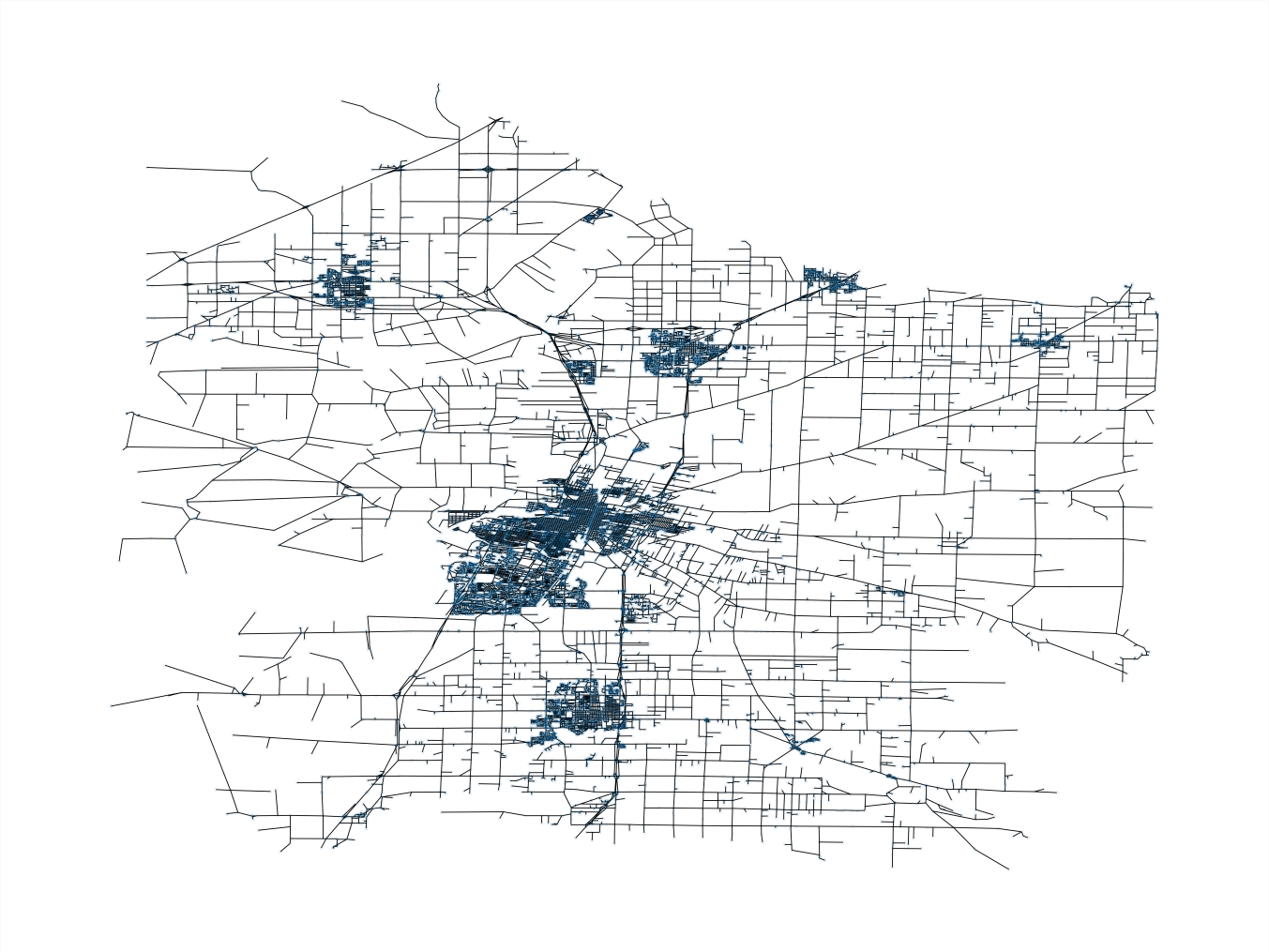

Figure 5: MST

Optimised Graph Weight: 531061

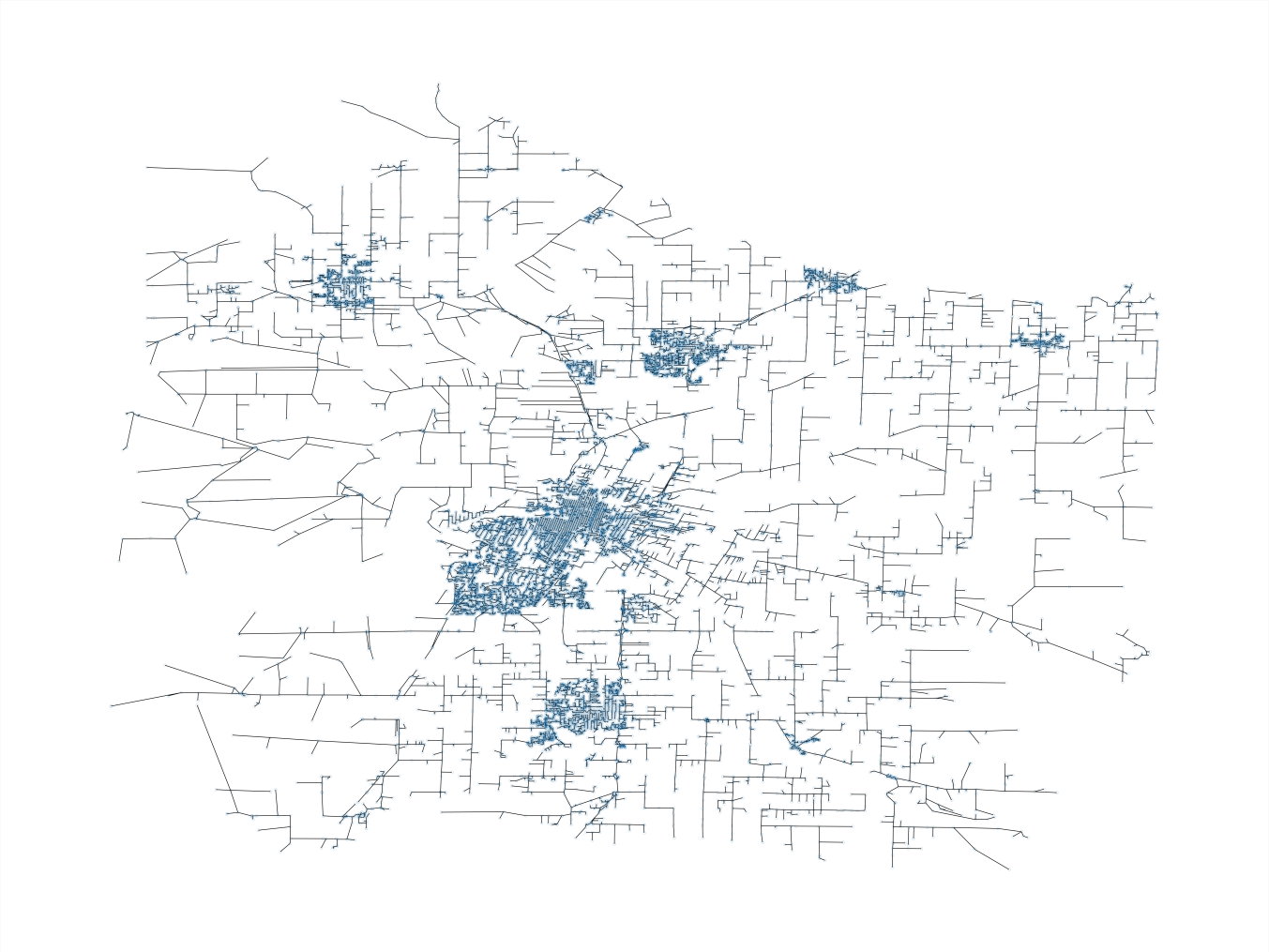

Figure 6: MST

Conclusion

- Boruvka's algorithm remains essential in graph theory and various applications.
- Ongoing enhancements in computing continue to increase its utility.

References i


Spatial dataset repository.
https://users.cs.utah.edu/~lifeifei/SpatialDataset.htm.

Accessed: 10-Dec-2023.

Thank you for watching my presentation