Determining Optimal 3D Pole Placement using Minimum Steiner Trees

JR-Bam

January 16, 2026

Abstract

This document details an algorithm that determines optimal pole placements within a real world terrain by connecting terminal nodes representing substations in an optimal network using a Minimum Steiner Tree. In determining weights for potential connections, a node's 3D position which includes the X, Y, and DEM will take into account 50% of the calculation, with the SLOPE and LOAD DENSITY each taking into account 25%.

Contents

List of Figures

1 About Steiner Trees

The Steiner Tree problem states that given a graph G and a subset of vertices $\{v_a, v_b, ..., v_z\}$ or terminal nodes within the graph, what is the network that connects all terminal nodes in a way that enjoys minimum size (sum of edge weights) [Gee25]. The result of this is called a **Minimum Steiner Tree**, making it best for finding a way to connect a list of nodes that minimizes the cost of the final network as shown in Figure 1.

This problem can be seen as the generalization of two other problems, the *Shortest Path Problem* and the *Minimum Spanning Tree Problem*. If the terminal nodes are all of the nodes in a graph, then the problem becomes equivalent to the minimum spanning tree problem. This is in contrast to when there are two terminal nodes, wherein it becomes equivalent to the shortest path problem [Gee25].

The problem within this document of optimizing pole positions includes terminal nodes that serve as substations of the network. These terminal nodes do not necessarily include all of the potential pole placements, as much as the number of substations not being limited to 2. As such, **optimizing pole positions is neither a problem equivalent to the shortest path problem nor the minimum spanning tree problem**.

This case is better suited as the general case of the Steiner tree problem. However, while optimized algorithms for the two specific cases already exist, none exist for the general case. This is considered as NP-hard, which means that it is unlikely that there is an efficient algorithm that can solve the problem for large instances. For the sake of practicality, approximations are made that produces minimum steiner trees that are *good enough* while still optimizing for performance. An example is one made by Kou [KMB81], but another which is an explicit improvement upon Kou is the one developed by Mehlhorn [Meh88]. For the purposes of this document, the latter one will be used to aid in the algorithm.

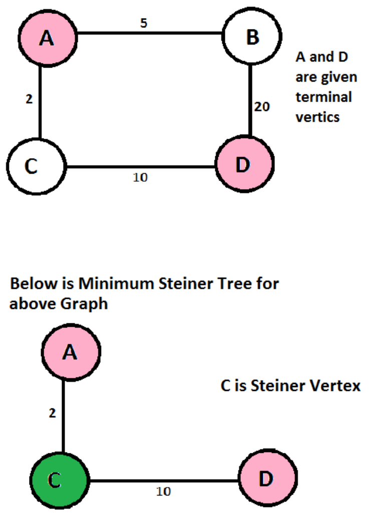
A
5
B
A and D are given terminal vertics
2
20
C
10
D
Below is Minimum Steiner Tree for above Graph
A
C is Steiner Vertex
2
C
10
D

Figure 1: Example of a Minimum Steiner Tree, from [Gee25]

2 Algorithm Design

This section details the algorithm developed to determine the optimal placement of electrical poles. The method systematically integrates elevation data, terrain characteristics, and electrical load profiles to generate a network plan that minimizes infrastructure cost while adhering to engineering constraints.

A flowchart is provided on Figure 2, in which it details the steps required within this algorithm. The result of which is a list of nodes and edges that form an optimal electrical network which spans the specified area.

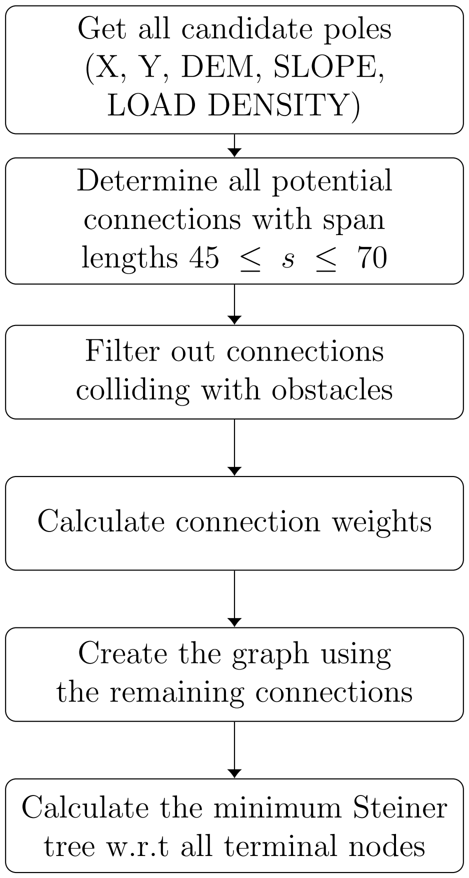
Get all candidate poles (X, Y, DEM, SLOPE, LOAD DENSITY)
Determine all potential connections with span lengths $45 \leq s \leq 70$
Filter out connections colliding with obstacles
Calculate connection weights
Create the graph using the remaining connections
Calculate the minimum Steiner tree w.r.t all terminal nodes

Figure 2: Flowchart for Determining Optimal Pole Placement

2.1 Input Data

Candidate Poles This is a list of the data of all nodes that will be part of the calculations. This includes the 3D positions of the poles, the slope at that given point, and its load density. More specifically, the following data will be accounted:

ID The unique ID to differentiate the given pole.

X The X coordinate.

Y The Y coordinate.

- **DEM** Digital Elevation Model. Treated as the Z coordinate.
- **Slope** The slope at the area of the given pole.
- **Load Density** The load density of the given pole.

Terminal Node IDs This is a list of node ids that represents all the nodes where the resulting network **must** include. A terminal node may represent a start node, or an end node that the start node must connect to.

Obstacle Data This is data that correspond to the positions and edges of obstacles that potential connections ideally need to avoid to ensure proper connections. This is in the form of a .geojson file and includes a list of MultiPolygon objects that represent the obstacles.

2.2 Implementation Steps

The following subsection will enumerate each step within this algorithm. This also includes code listings, writen in Python, for the implementations of each step.

2.2.1 Load the Data

The original data is within a .csv file. The implementation utilizes the pandas library to read the file and store its contents within a DataFrame object for future operations. See Appendix A.1.

2.2.2 Initialize Potential Connections

After loading the data, the algorithm then lists out all possible edges or connections between any two nodes or poles. It will only includes edges that have a span length of $45-70$ meters. In order to determine the span length between two poles, the X and Y of both poles will be used, and in using the Euclidean formula:

$$span = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

the span will be checked if it is within $45 - 70$, otherwise this potential connection will not be included in the forthcoming steps. See Appendix A.2.

2.2.3 Remove Obstructing Connections

Potential connections are further filtered by determining whether or not it collides with obstacles. These obstacles are represented as polygons that is stored in a .geojson file and loaded into the script. For each edge, the program will draw a line between both nodes and will check if it intersects with any of the polygons. If it does, then it will be removed from the list of potential connections. See Appendix A.3.

2.2.4 Weight Calculation

In this step, the weights for each remaining edge is calculated. For each edge, the factors that will be used for calculating the final weight is the **3D distance**, **Slope**, and **Load Density**. See Appendix A.4.

$$D^{3D}_{raw} = \sqrt{(X_1 - X_2)^2 + (Y_1 - Y_2)^2 + (DEM_1 - DEM_2)^2}$$

$$Slope_{raw} = \frac{slope_1 + slope_2}{2}$$

$$LoadDensity_{raw} = \frac{loaddensity_1 + loaddensity_2}{2}$$

After calculating the raw values of the factors, these are normalized in order to fit it within the range $[0, 1]$.

$$D^{3D} = \frac{D^{3D}_{raw}}{\max(D^{3D}_{raw})}$$

$$Slope = \frac{Slope_{raw}}{\max(Slope_{raw})}$$

$$LoadDensity = \frac{LoadDensity_{raw}}{\max(LoadDensity_{raw})}$$

Finally, the weight of the given edge is calculated. The distance has the most influence in the final weight, with the slope and load density both sharing a lesser influence overall.

$$weight = 0.5 \times D^{3D} + 0.25 \times Slope + 0.25 \times LoadDensity$$

2.2.5 Creating the Graph

Using the data for each remaining node, and the weights calculated for each remaining edge, A weighted graph $G = (V, E)$ is created. See Appendix A.5.

2.2.6 Calculating the Minimum Steiner Tree

Given the graph G, and the terminal nodes provided as input, the minimum steiner tree is created using the approximation algorithm provided by Mehlhorn [Meh88]. The data of the resulting tree is then saved in a .csv file. This includes the ids of the poles of all connections involved in the tree, as well as the weight and raw data for the edge slope, 3d distance, and load density. See Appendix A.6.

3 Results and (Mini) Discussion

After applying this algorithm within the data provided, the network graph in Figure 3 is created. The data for this graph is also exported as a .csv file. Given the figure, the node positions are relatively spaced out, which is indicative of the data, in which the distances are within the acceptable range.

For future studies that aim to improve upon this algorithm, an area worth tackling is the calculations for the weights. Perhaps a more accurate way of combining the factors such as 3D distance, slope, and load density can be developed to provide a more realistic depiction of optimal electrical pole placements. Furthemore, other factors can also be taken into account when designing pole placements and feeder routing.

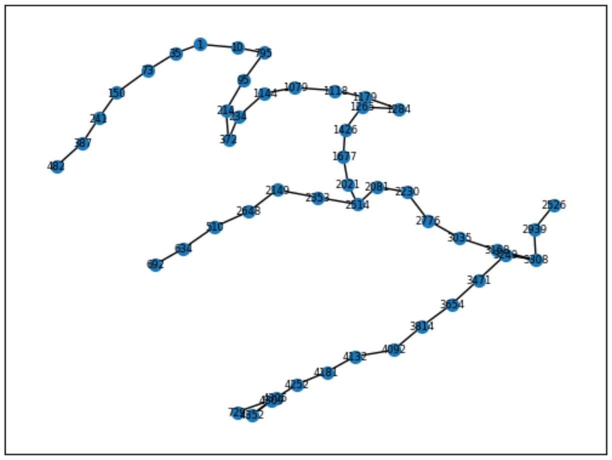

Figure 3: Resulting Network Graph from the Algorithm

A Code Listings

A.1 Load input data and preprocess

```python
import pandas as pd

# Load the Data
data = pd.read_csv("DATA-SHEET.csv").rename(columns={
    "USTP_Alubijid_DEM": "DEM",
    "USTP_Alubijid_Slope": "SLOPE",
    "X_coordinate": "X",
    "Y_coordinate": "Y"
}).fillna(0)
```

A.2 Filtering connections with optimal span length

```python
from scipy.spatial import KDTree

max_neighbors = len(data)

d_max = 70
d_min = 45

def create_potential_edges_pandas(data, k, d_max=65):
    tree = KDTree(data[["X", "Y"]].values)
    fids = data["fid"].values

    distances, indices = tree.query(data[["X", "Y"]].values, k=k)

    edges = [
        {
            "nodeid1": fids[i],
            "nodeid2": fids[neighbor_idx],
            "distance": distance
        }
        for i in range(len(data))
        for neighbor_idx, distance in zip(indices[i][1:], distances[i
    ][1:])
        if d_min <= distance <= d_max
    ]

    return pd.DataFrame(edges)

# Create edges using fids directly
edges = create_potential_edges_pandas(data, max_neighbors, d_max)

# Include both node data within the dataframe
edges = (edges
    .merge(data, left_on='nodeid1', right_on='fid', suffixes=('', '_1'))
    .merge(data, left_on='nodeid2', right_on='fid', suffixes=('_1', '_2'))
    .drop(['fid_1'], axis=1)  # Remove duplicate fid columns
)
```

A.3 Filtering Edges that collide with obstacles

```python
import geojson
from shapely.geometry import shape, LineString
from shapely.strtree import STRtree

obstacle_data = geojson.load(open("BUILDING-FOOTPRINTS.geojson", "r"))
obstacles = [shape(feature["geometry"]) for feature in obstacle_data["
    features"]]
spatial_index = STRtree(obstacles)

def edge_collides_with_obstacle(row):
    line = LineString([
        (row['X_1'], row['Y_1']),
        (row['X_2'], row['Y_2'])
    ])
    potential_obstacles_indices = spatial_index.query(line)
    for idx in potential_obstacles_indices:
        if line.intersects(obstacles[idx]):
            return True
    return False

# Apply the function to each row
collision_mask = edges.apply(edge_collides_with_obstacle, axis=1)
filtered_edges = edges[~collision_mask]  # Keep edges that DON'T collide
```

A.4 Calculating weights

```python
import numpy as np

computed_edges = filtered_edges.assign(
    raw_d3d = np.sqrt((filtered_edges['DEM_1'] - filtered_edges['DEM_2'])
    **2
                    + (filtered_edges['X_1'] - filtered_edges['X_2'])**2
                    + (filtered_edges['Y_1'] - filtered_edges['Y_2'])**2),
    raw_slope = (filtered_edges['SLOPE_1'] + filtered_edges['SLOPE_2']) /
    2,
    raw_load_density = (filtered_edges['LOAD-DENSITY_1'] + filtered_edges[
    'LOAD-DENSITY_2']) / 2
)[["nodeid1", "nodeid2", "raw_d3d", "raw_slope", "raw_load_density", "X_1"
    , "Y_1", "X_2", "Y_2", "DEM_1", "DEM_2"]]

# Get max values
max_d3d = computed_edges['raw_d3d'].max()
max_slope = computed_edges['raw_slope'].max()
max_load_density = computed_edges['raw_load_density'].max()

# Normalize
computed_edges['d3d'] = computed_edges['raw_d3d'] / max_d3d
computed_edges['slope'] = computed_edges['raw_slope'] / max_slope
computed_edges['load_density'] = computed_edges['raw_load_density'] /
    max_load_density
```

```python

# Final cost calculations
weight_d3d = 0.5
weight_slope = 0.25
weight_load_density = 0.25

edges_with_costs = computed_edges.assign(
    weight = weight_d3d * computed_edges["d3d"] + weight_slope *
    computed_edges["slope"] + weight_load_density * (1 - computed_edges["
    load_density"])
)
```

A.5 Creating Graph

```python
import networkx as nx

nodes = edges_with_costs[["nodeid1", "X_1", "Y_1", "DEM_1"]].
    drop_duplicates(subset=["nodeid1"]).rename(columns={
    "nodeid1": "fid",
    "X_1": "x",
    "Y_1": "y",
    "DEM_1": "dem"
})

node_attrs = nodes.set_index('fid').to_dict('index')

G = nx.from_pandas_edgelist(edges_with_costs, source="nodeid1", target="
    nodeid2", edge_attr=[
    "weight", "raw_d3d", "raw_slope", "raw_load_density"])

nx.set_node_attributes(G, node_attrs)
```

A.6 Finding Optimized Network using Minimum Steiner Tree

```python
from networkx.algorithms import approximation

optimized_network = approximation.steiner_tree(G, end_nodes + [2514])
nx.to_pandas_edgelist(optimized_network, "fid1", "fid2", nodelist=G.nodes)
    .to_csv("optimized.csv")
nx.draw_networkx(
    optimized_network,
    pos={ key: (item["x"], item["y"]) for key, item in node_attrs.items()
    },
    node_size=50,
    font_size=6
)
```

References

[Gee25] GeeksforGeeks. Steiner tree problem, Jul 2025. `https://www.geeksforgeeks.org/dsa/steiner-tree/`.

[KMB81] L Kou, G Markowsky, and L Berman. A fast algorithm for steiner trees. *Acta Inform.*, 15(2):141–145, 1981. `https://doi.org/10.1007/BF00288961`.

[Meh88] Kurt Mehlhorn. A faster approximation algorithm for the steiner problem in graphs. *Information Processing Letters*, 27(3):125–128, 1988. `https://www.sciencedirect.com/science/article/pii/002001908890066X`.