MINIMUM SPANNING TREE

IMPLEMENTATION AND COMPARISON OF
PRIMS AND KRUSKALS ALGORITHMS
ON ROAD-NETWORKS

DESIGN AND ANALYSIS OF ALGORITHMS
Course Code: CSE-5311-009

University of Texas at Arlington
For Master of Science Degree of
Computer Science

By

Tanuj Avinash Palaspagar **ID No: 1002090864**
Venkata Sai Likhita Kurapati **ID No: 1002066295**

Under the Guidance of

Prof: Md Hasanuzzaman(Zaman) Noor

Department of Computer Science

Index

Introduction

The project deals with the implementation of two minimum spanning tree algorithms which are “Prims and Kruskal's” Algorithms.
The project deals with the comparison between those two algorithms taking in real-world data and plotting a graph for the same. We use python as the base programming language to implement our program and all the nodes and edges used and all that data will be retrieved using the network package.

We have an inbuilt time function in python which we’ve used to compare the running times between them. The graphs are plotted using the matplotlib library to analyze and compare the dataset by using the above two algorithms.
We have included different datasets such as manhattan.graphml, newyork.graphml for this project which shows the time difference for each of these algorithms when tested with these various sizes of our real-time datasets. The graphml is used to retrieve the data from the datasets and form graphs and hence its format. The structural properties such as nodes and vertices which are included in the graphs of this dataset can be easily described using the graphml file format. It is a flexible extension to add application-specific data.

Prims Algorithm is significantly faster when we’ve got a dense graph in the limit which has more edges and vertices. This is because it just traverses the adjacent nodes for each node and it doesn’t have to sort the edges and compares only a limited number of edges per loop.

Kruskal’s algorithm works better for sparse graphs because the number of edges is smaller, so sorting them gets easier. Kruskal’s allow both new to new and old to old to get connected which creates the circuit and the algorithm has to check them each time.
.

Implementation

Project Structure:

```
main.py                     : main code file, run this on default

mst.py                      : code containing the algorithms. This file is called by
main.py

dataset/                    : directory with various datasets in .graphml format

        /tokyo.graphml      : huge dataset with 898,430 nodes

        /los_angeles.graphml: large dataset with 297,333 nodes

        /newyork.graphml    : medium dataset with 54,128

        /manhatten.graphml  : small dataset with 4,426

        /maldives.graphml   : tiny dataset with 639 nodes

docs/                       : directory with outputs, results, and report

        /code_algorithms/   : screenshots of implementation of Prim's and Kruskal's
algorithm from mst.py

        /graphs/            : generated graphs including original data, mst generated after
running mst.py

        /results/           : snapshots of program output on different datasets where
algorithm = 'both'

        /PROJECT_REPORT.pdf : report

references.txt              : references for this project

README.md                   : information to run the program
```

The code uses Networkx library to handle the data and graphs. Install the networkx library running the following line in the terminal:

```
pip install networkx
```

main.py

init() method contains variables to be changed to run the code accordingly

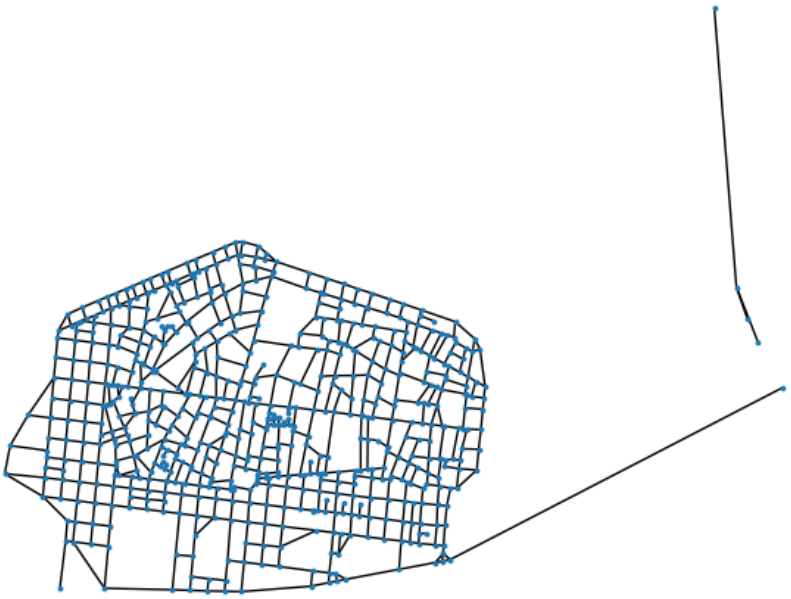
- graphsize : 'tiny', 'small', 'medium', 'large', 'huge'
- algorithm : 'none', 'kruskal', 'prim', 'both'
    - 'none' Generate the graph passed in the dataset.
    - 'kruskal' Implements Kruskal's algorithm to generate and show minimum spanning tree and print running time in console
    - 'prim' Implements Prim's algorithm to generate and show minimum spanning tree and print running time in console
    - 'both' Will NOT generate graphs: only runtime-comparative values printed in console
- Recommended: Use only 'both' for 'large' and 'huge' graphsize values, the program uses a lot of resources and time for these datasets
- do_algo() (referenced by implement_algo()) method does not give good results for 'tiny' 'small' 'medium' datasets
- do_algo1() (referenced by implement_algo()) method is modified do_algo() to work with those datasets
- All other methods run as default, information for each can be found in the comments

mst.py

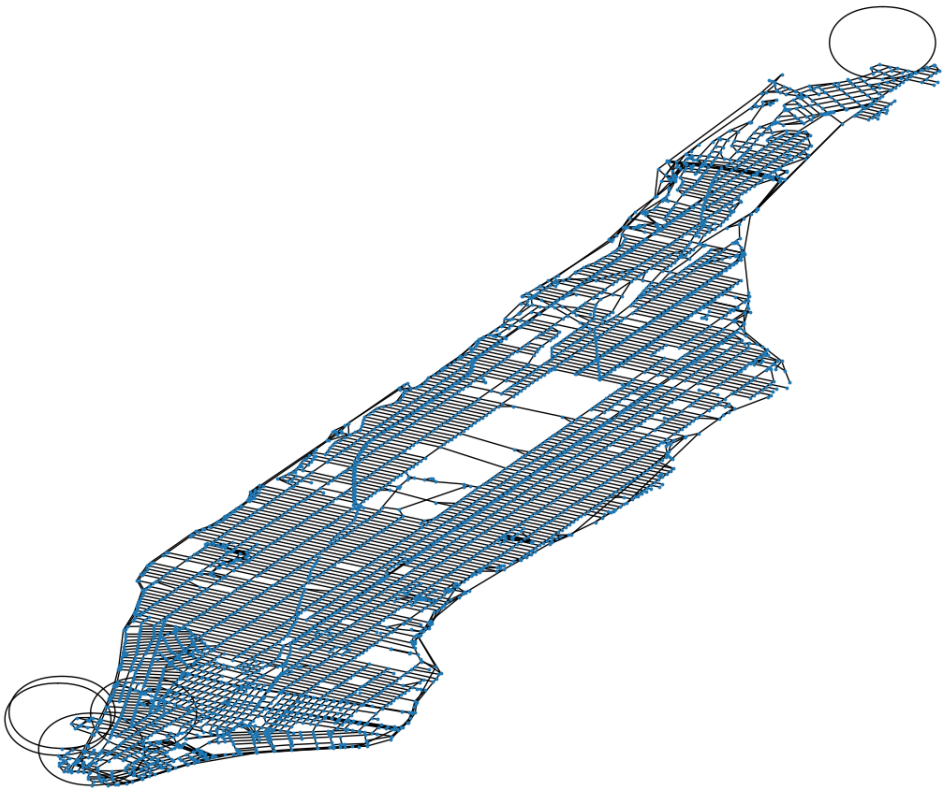
kruskal_mst_edges() implementation of Kruskal's algorithm on the graph of interest
prim_mst_edges() implementation of Prium's algorithm on the graph of interest
minimum_spanning_edges() method deals with calling the other functions

Dataset

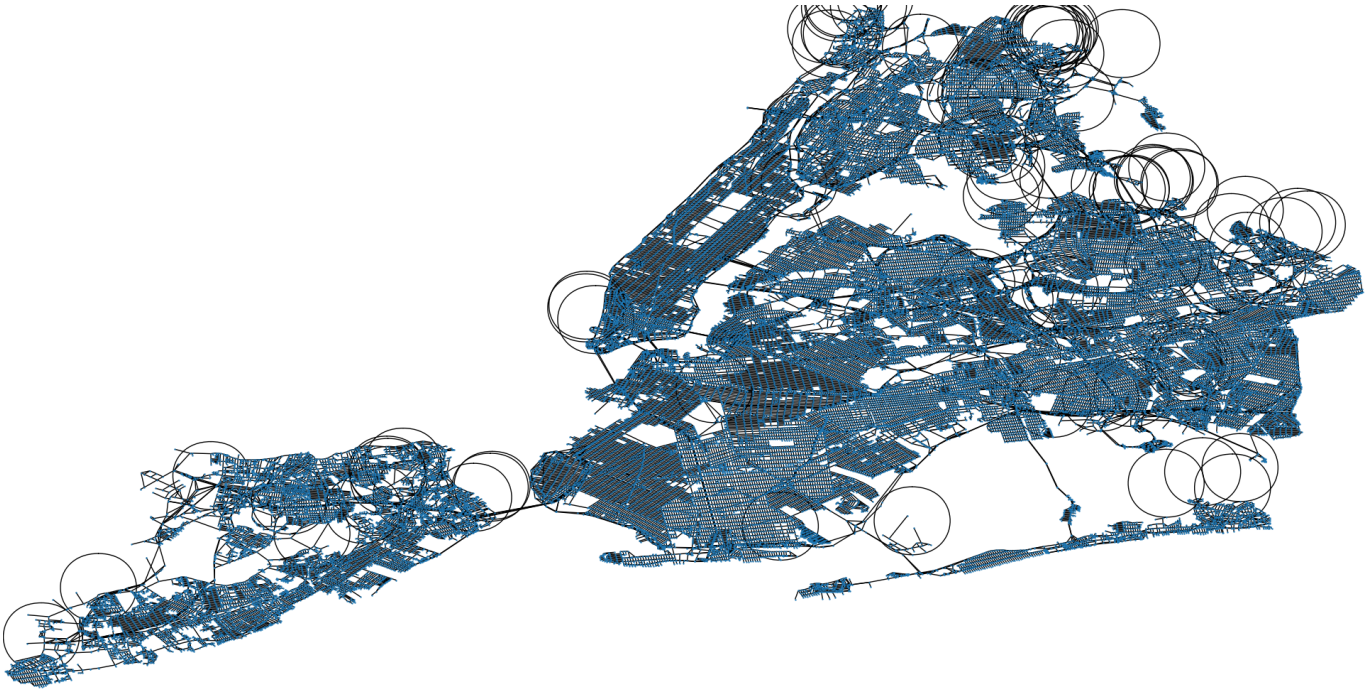
Tiny:
Maldives Road Network with 639 nodes



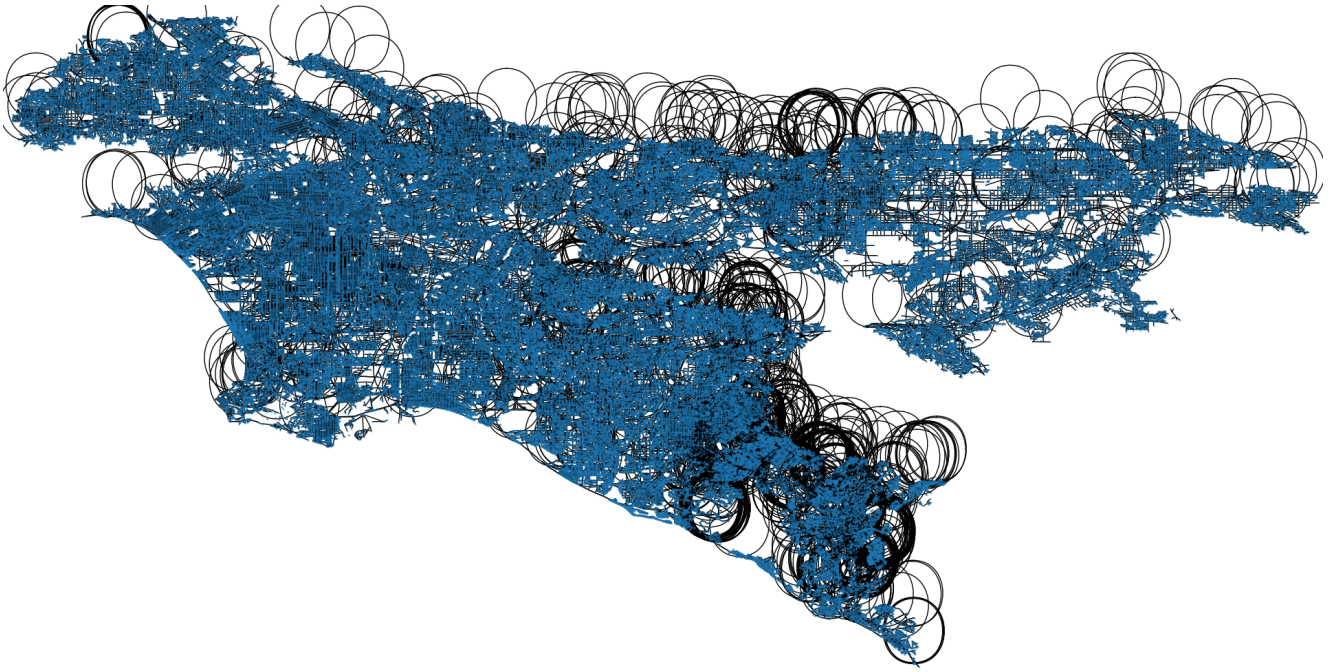
Small:
Manhattan Road network with 4,426 nodes



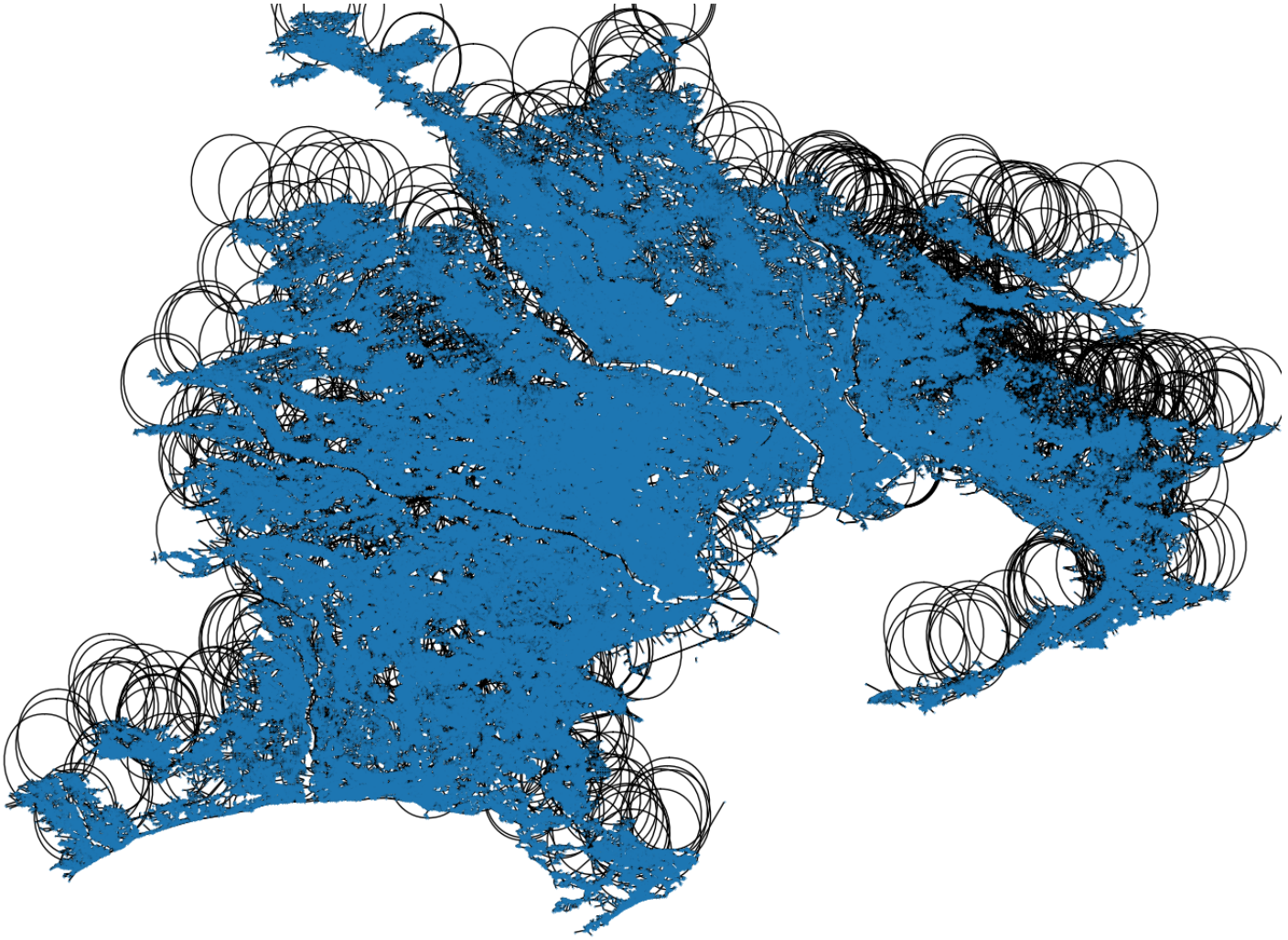
Medium:
New York Road Network with 54,128 nodes



Large:
Los Angeles Road Network with 297,333 nodes



Huge:
Tokyo Road Network with 898,430 nodes



Prim's Minimum Spanning Tree

The idea behind prims algorithm is simple, it finds a subset of the edges that forms a tree that includes every vertex, such that the total weight of the edges in that tree is minimized.
The algorithm was developed by Czech mathematician Vojtech Jarnik in 1930 and later rediscovered and republished by computer scientists Robert C. Prim in 1957 and Edsger W. Dijkstra in 1959, hence its name Jarnik's algorithm, Prims-Jarnik algorithm or Prims-Dijkstra algorithm.
The algorithm operates by building this tree one vertex at a time, from one arbitrary an arbitrary starting vertex at each step adding the cheapest possible connection from the tree to another vertex. However, for the graphs that are sufficiently dense, Prim's algorithm can be made to run in linear time, improving the time bounds simultaneously.

Description of the algorithm

To be precise the starting vertex of the algorithm will be chosen arbitrarily, because the first iteration of the main loop of the algorithm will have a set of vertices in Q set that have equal weights, and the algorithm will start a new tree F(empty forest) when it completes a new spanning tree for each component in the input graph.

The algorithm might start with a new vertex s by setting C[s] to be a number having values smaller than C(for instance 0), and it may be modified to only find a single spanning tree rather than finding the whole spanning forest by stopping whenever it encounters another vertex flagged as having no associated edge.

The Algorithm will be implemented in a way by using a Boolean array mstSet[] to represent the set of vertices included in MST.

If a value mstSet[v] is true, then vertex v is included in MST else not. The key[] is used to store the key values of all the vertices. Another parent[] array is used to store indexes of parent nodes in MST. The parent array is the output array that is used to show the minimum spanning tree.

```
Key[i]= INT_MAX, mstSet[i]=false;
Key[0]=0;
Parent[0]=-1;
```

Firstly we chose an element r and set S={r} and A= phi. (Take r as the root of our spanning tree). Then we find the lightest edge such that one endpoint is in S and the other is in V \ S. We add this edge to A and the other endpoint to S. If V \ S tends to phi then we stop and output the minimum spanning tree (S, A) else we go to start again.

We print the constructed MST by setting mstSet[u]=true and printing the Minimum spanning tree by constructing printMST(parent, graph);

```python
def prim_mst_edges(G, weight="weight"):
    """
    Iterate over edges of Prim's algorithm

    Parameters
    ----------
    G : NetworkX Graph
        The graph holding the tree of interest.

    weight : string (default: 'weight')
        The name of the edge attribute holding the edge weights.
    """
    push = heappush
    pop = heappop

    nodes = set(G)
    c = count()

    while nodes:
        u = nodes.pop()
        frontier = []
        visited = {u}

        for v, d in G.adj[u].items():
            wt = d.get(weight, 1)
            push(frontier, (wt, next(c), u, v, d))
        while nodes and frontier:
            W, _, u, v, d = pop(frontier)
            if v in visited or v not in nodes:
                continue
            yield u, v, d

            # update frontier
            visited.add(v)
            nodes.discard(v)

            for w, d2 in G.adj[v].items():
                if w in visited:
                    continue
                new_weight = d2.get(weight, 1)
                push(frontier, (new_weight, next(c), v, w, d2))

```

Time Complexity

Prims algorithm has a time complexity of O(V^2), V being the number of vertices and this can be improved up to O(E log V) using Fibonacci heaps. Every edge is inserted in the priority queue only once and insertion in the priority queue takes logarithmic time.

Space Complexity

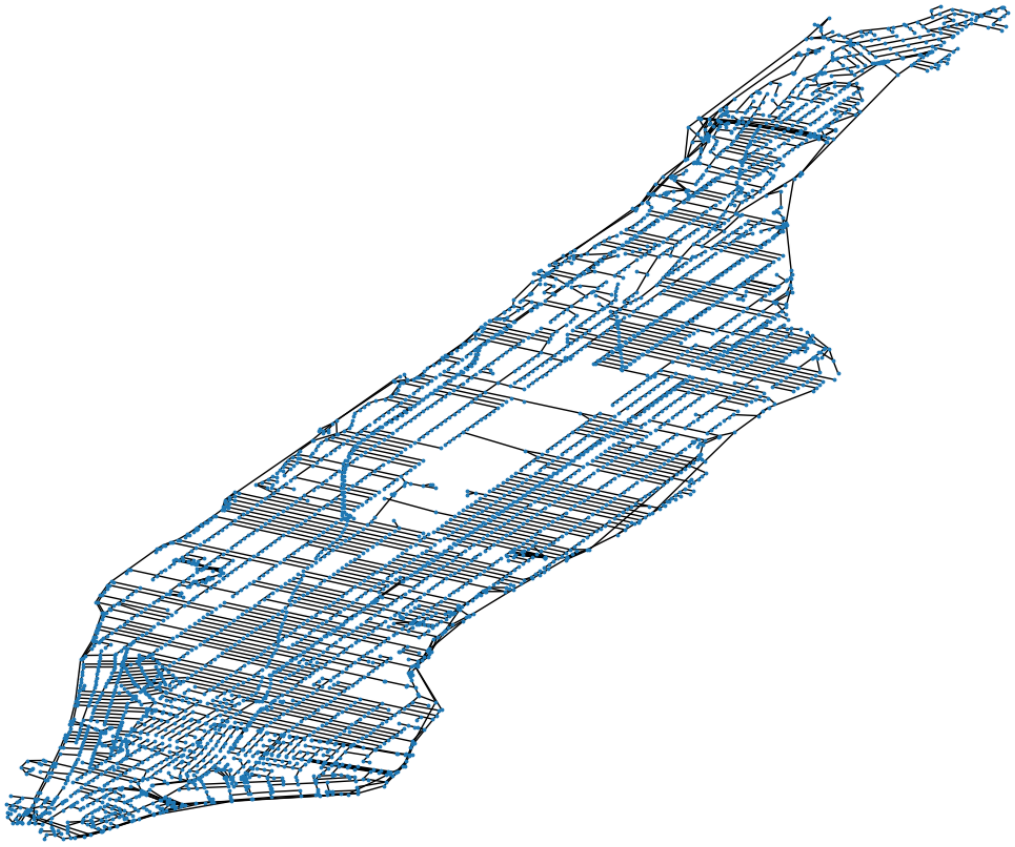
Space Complexity of prims algorithm is O(E+V) where E is the edges and V is the vertices totally present in the graph. Adding both these will give us space complexity.

Negative Edge Cycles

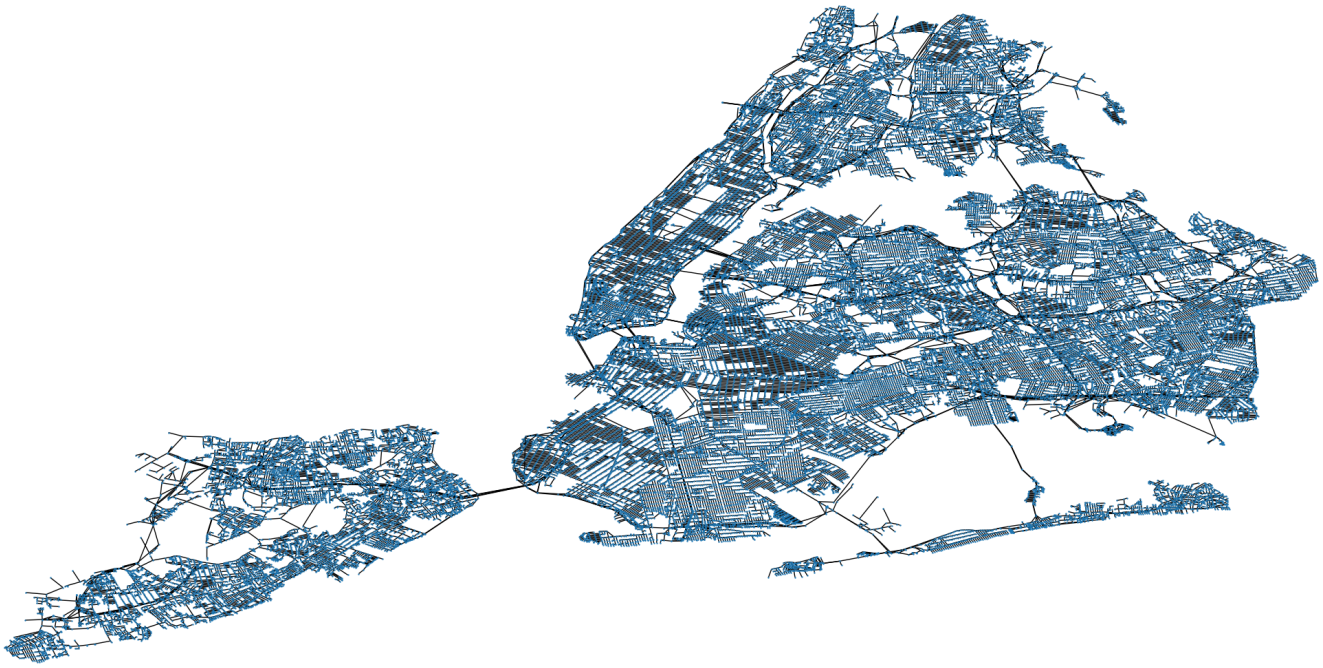
The negative edge weights are no problem to the Prims Algorithm. We always get a spanning tree if there is an unused edge that has a lower weight than any edge in the cycle that would be created by adding it to the spanning tree.

Graphs generated after the implementation of Prim's Minimum Spanning Tree:

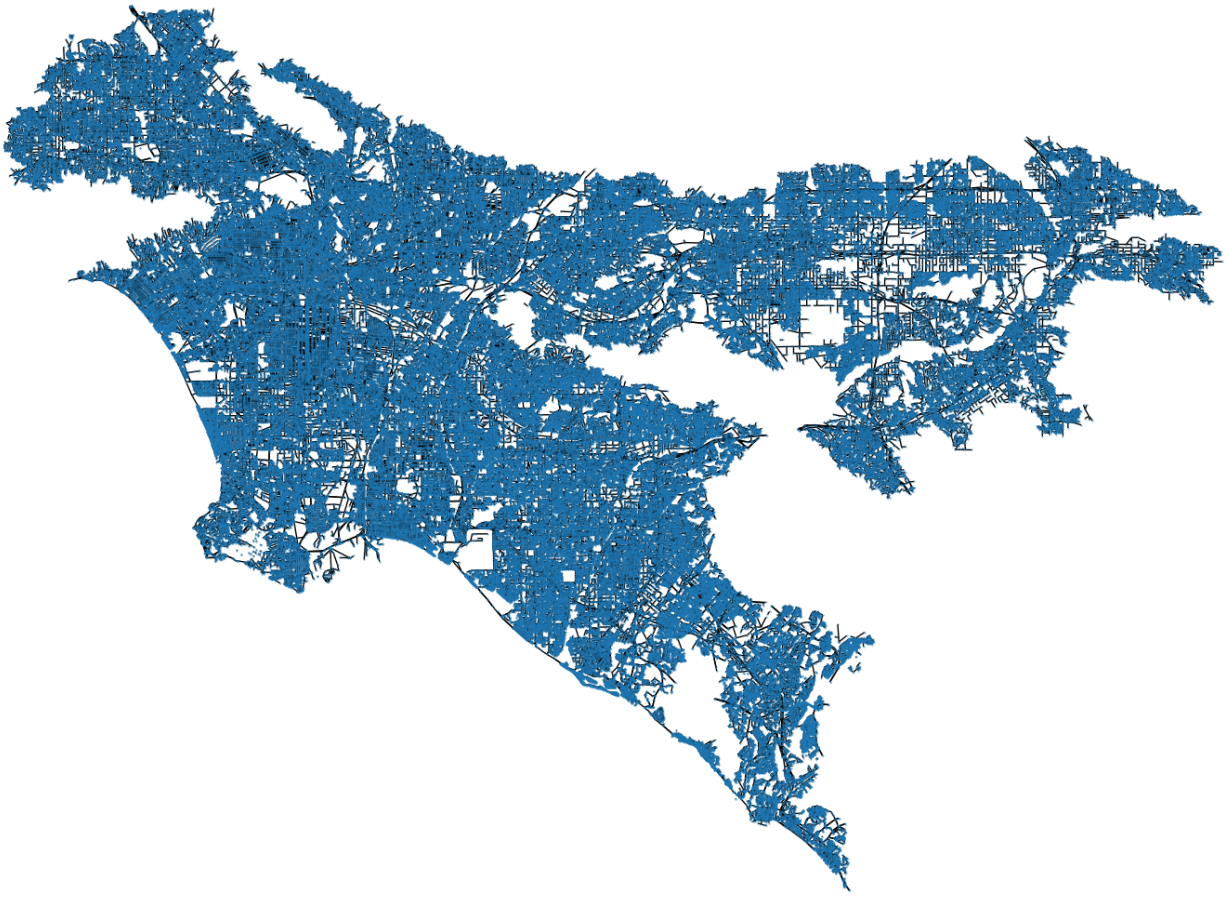
TINY



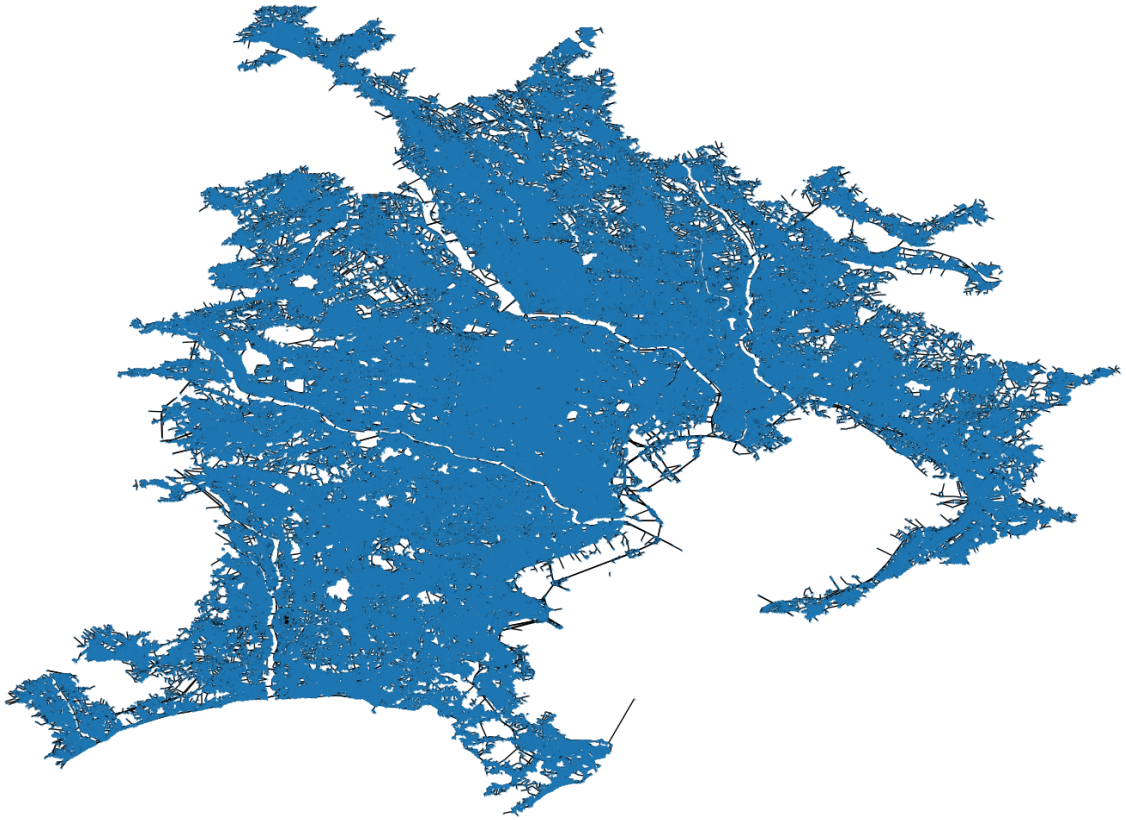
SMALL



MEDIUM



LARGE



HUGE

Kruskal's Minimum Spanning Tree

Kruskal's is one of the greedy algorithms to find the minimum spanning tree of a graph. The main idea of Kruskal's is to pick the minimum weighted edge that we haven't picked yet AND that does not create a cycle if we put it into a spanning tree. In Kruskal's at every stage, we take the globally smallest edge that would never create a cycle. Unique weights will ensure that Kruskal's algorithm will have only one possible permutation of the weights (imagine you have weights (3,2,4,1,2,5), there are 2 possible permutations in which the weights are sorted: 1,2 (the one after 3), 2 (the one after 1), 3,4,5 and you can swap the two 2s and the weights will still be sorted) and therefore there is only one possible minimum spanning tree.

The algorithm first appeared in 'The proceedings of the American Mathematical Society in 1956, written by Joseph Kruskal, and rediscovered by Loberman and Weinberger.

We first sort the edges by weight by using a comparison sort and remove an edge with the minimum weight in S to operate in that constant time. We keep track of all the vertices and chose accordingly. Kruskal's is inherently sequential and hard to parallelize. We extract the minimum weight edge in every iteration. A parallel sorting is possible in time O(n), the runtime of Kruskal's algorithm can be hence reduced as well. Therefore, the basic idea is to partition the edges and connect the vertices of the same tree to reduce the cost of sorting.

Description of the algorithm

Kruskal's algorithm sorts all the edges in the increasing order of their edge weights and keeps adding nodes to the tree only if the chosen edge doesn't form a cycle. It then picks the smallest edge and checks if that edge creates a cycle or loop in the tree, it doesn't form a cycle, then includes that edge in the MST. Otherwise, discard it.
Repeat step 2 until it includes |V|-1 edges in MST. The Union-Find is the most popular algorithm to determine loops in the graph. The algorithm for Kruskal's is terminated when n-1 edges are added to the graph. Let (u, v) be an edge in T–T*.

Let S be the CC containing u at the time (u, v) was added to T.
We claim (u, v) is a least-cost edge crossing cut (S, V – S).
First, (u, v) crosses the cut, since u and v were not connected when Kruskal's algorithm selected (u, v). Next, if there were a lower-cost edge e crossing the cut, e would connect two nodes that were not connected.

Thus, Kruskal's algorithm would have selected e instead of (u, v), a contradiction. Since T* is an MST, there is a path from u to v in T*. The path begins in S and ends in V – S, so it contains an edge (x, y) crossing the cut. Then $T^{*'} = T^* \cup \{(u, v)\} - \{(x, y)\}$ is an spanning tree of G and $c(T^{*'}) = c(T^*) + c(u, v) - c(x, y)$. Since $c(x, y) \geq c(u, v)$, we have $c(T^{*'}) \leq c(T^*)$. Since T* is a minimum spanning tree, $c(T^{*'}) = c(T^*)$. Note that $|T - T^{*'}| = |T - T^*| - 1$.

Therefore, if we repeat this process once for each edge in T – T*, we will have converted T* into T while preserving c(T*). Thus c(T) = c(T*) which proves the tree to be Kruskal's algorithm and the T* to be a minimum spanning tree.

```python
def kruskal_mst_edges(G, weight="weight"):
    """
    Iterate over edge of a Kruskal's algorithm

    Parameters
    ----------
    G : NetworkX Graph
        The graph holding the tree of interest.

    weight : string (default: 'weight')
        The name of the edge attribute holding the edge weights.
    """
    subtrees = UnionFind()
    edges = G.edges(data=True)

    included_edges = []
    open_edges = []
    for e in edges:
        d = e[-1]
        wt = d.get(weight, 1)

        edge = (wt,) + e
        open_edges.append(edge)

    sorted_open_edges = sorted(open_edges, key=itemgetter(0))

    # Condense the lists into one
    included_edges.extend(sorted_open_edges)
    sorted_edges = included_edges
    del open_edges, sorted_open_edges, included_edges

    for wt, u, v, d in sorted_edges:
        if subtrees[u] != subtrees[v]:
            yield u, v, d
            subtrees.union(u, v)
```

Time Complexity

The time complexity for Kruskal's algorithm is O(ElogE) or O(ElogV), Sorting of edges takes O(ELogE) time. Once sorted, we iterate through all edges and the find-union algorithm is used, whose operations can take at most O(LogV) time.

Hence the overall complexity is O(ELogE + ELogV) time. The value of E can be at most O(V2), so O(LogV) is O(LogE) the same. Hence its time complexity is O(ElogE) or O(ElogV).

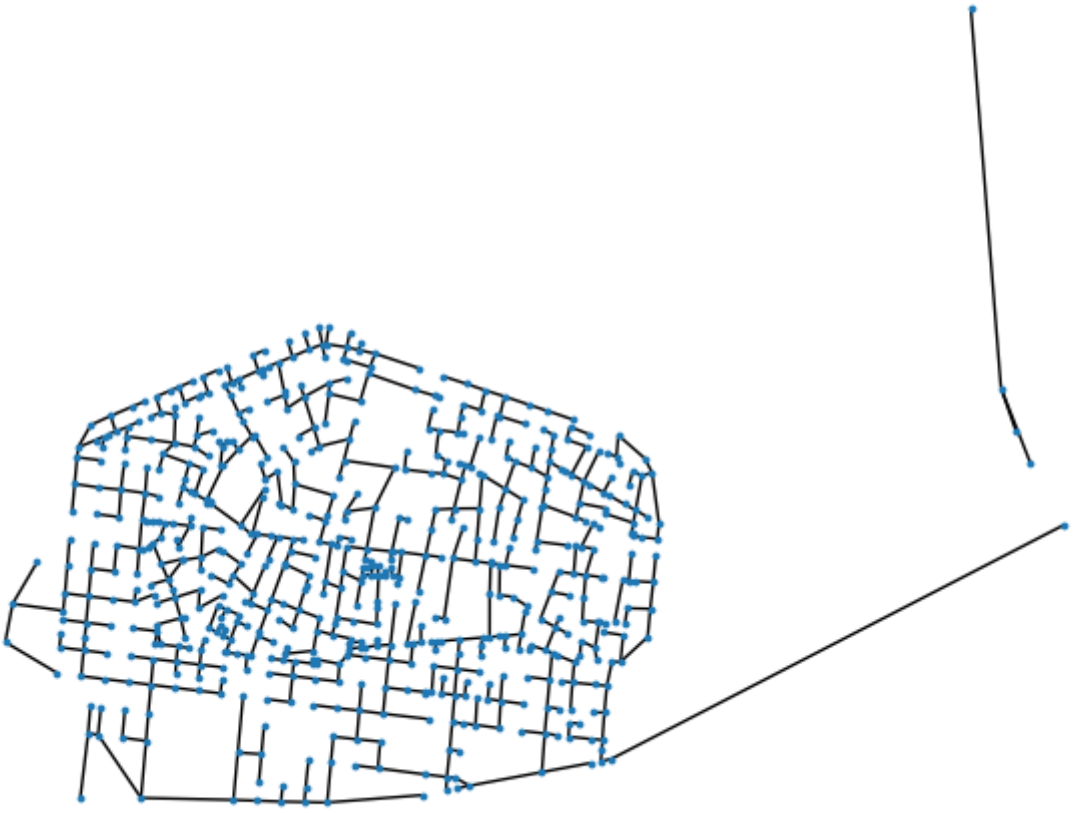
Space Complexity

The space complexity for this algorithm would be O(V+E) where V is the number of vertices and E is the number of edges in the graph.

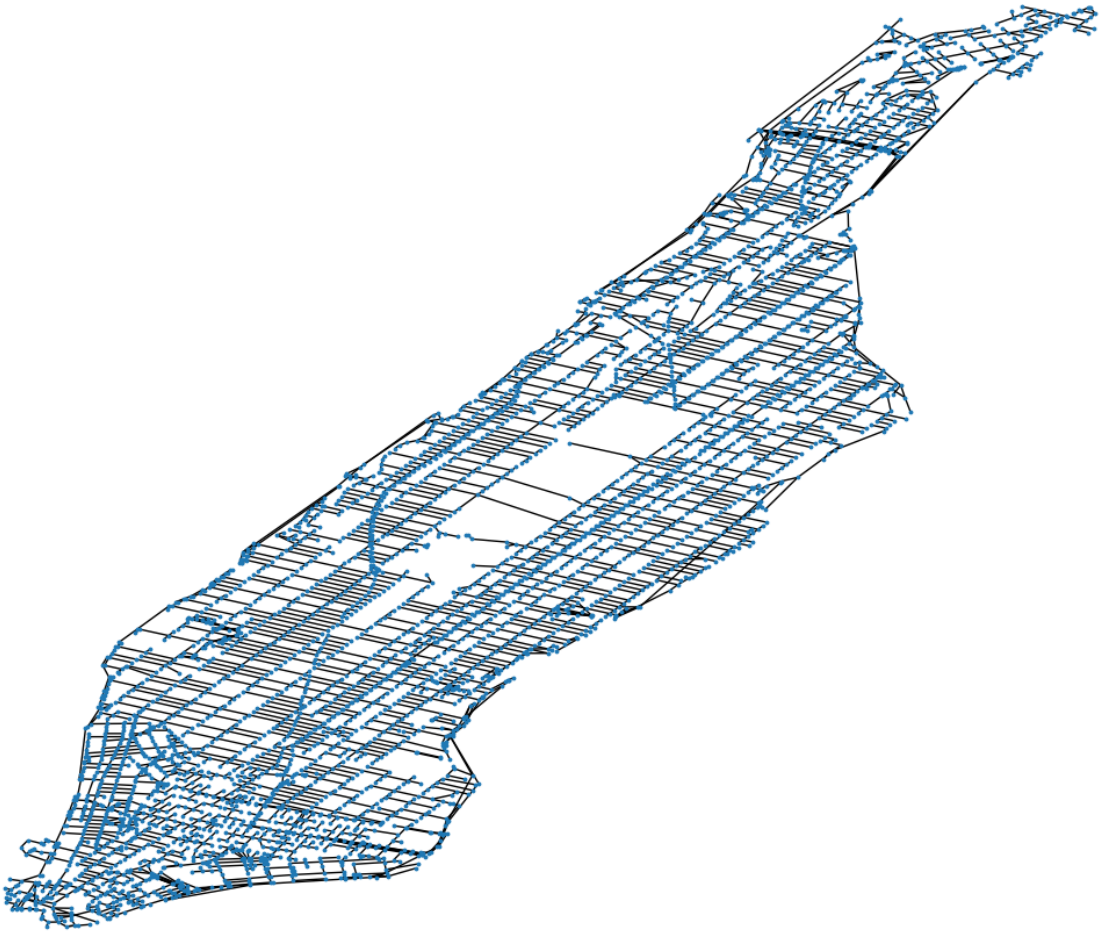
Negative Edge Cycles

The negative edge weights are no problem to Kruskal's Algorithm. The spanning tree gets better if there is an unused edge that has a lower weight than any edge on the cycle that would be created by adding it to the spanning tree. We just add the unused edge and remove one of the higher-weight edges on that cycle.

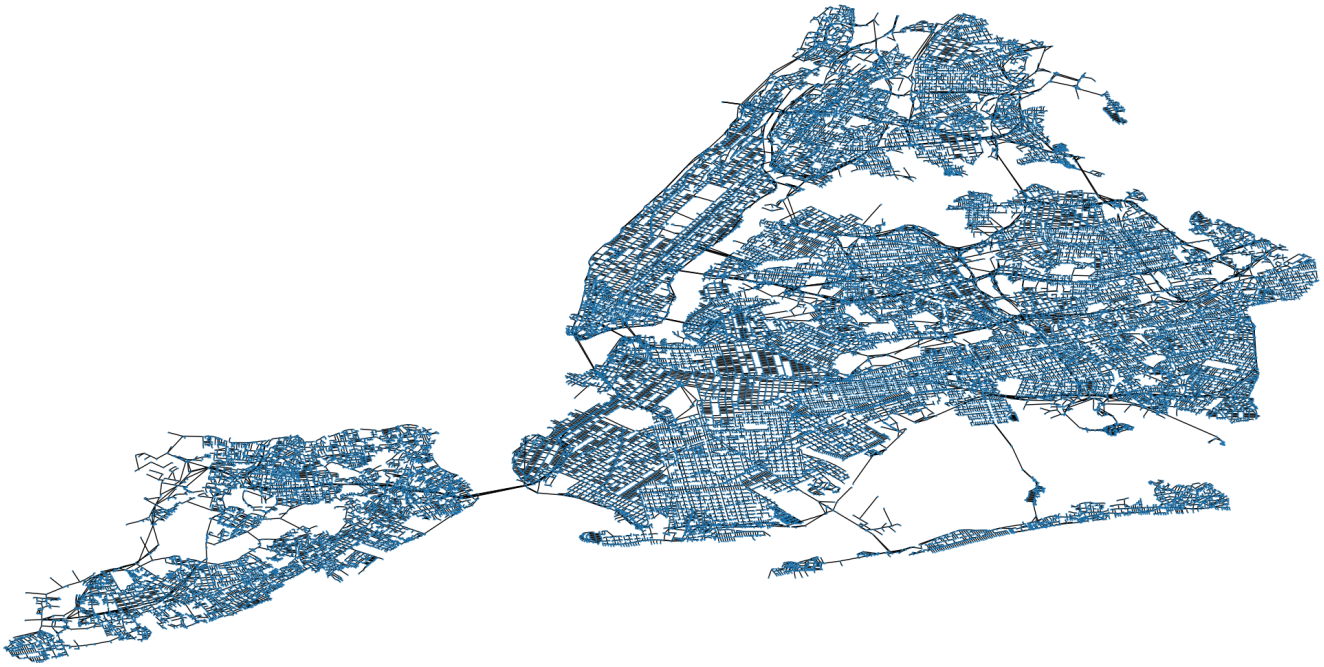
Graphs generated after the implementation of Prim's Minimum Spanning Tree:



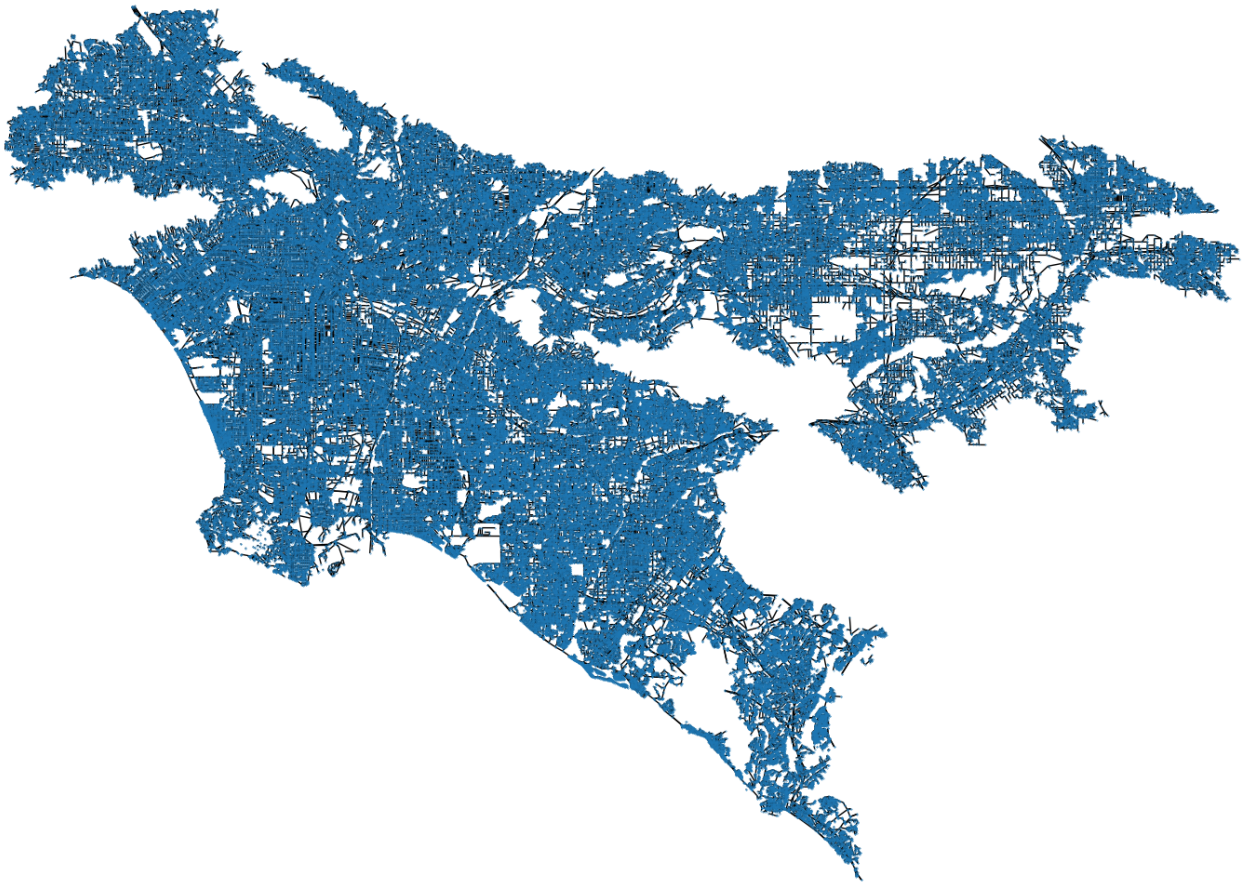
TINY



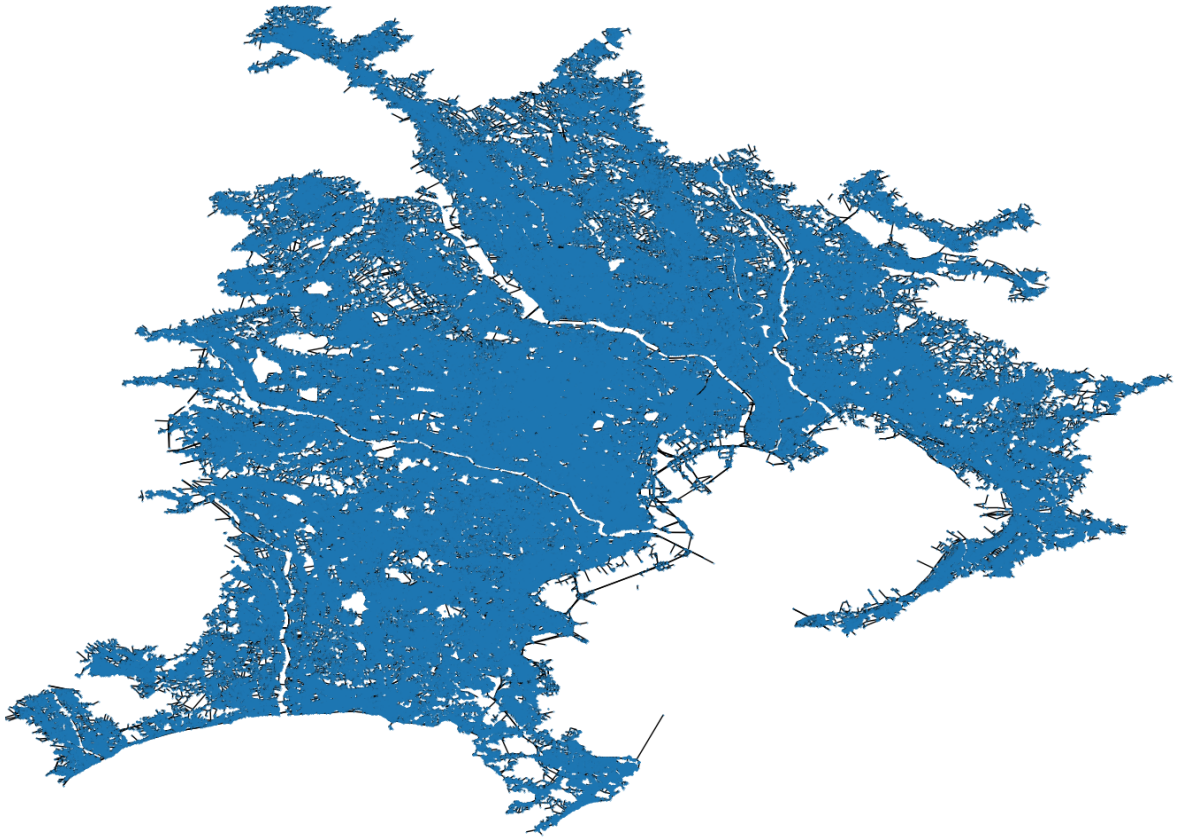
SMALL



MEDIUM



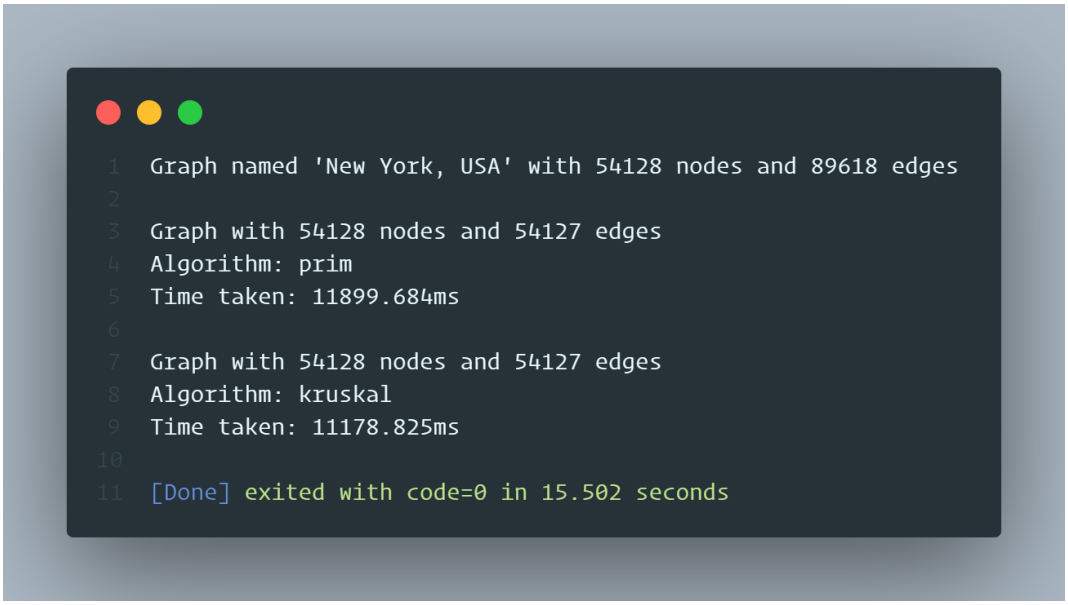
LARGE



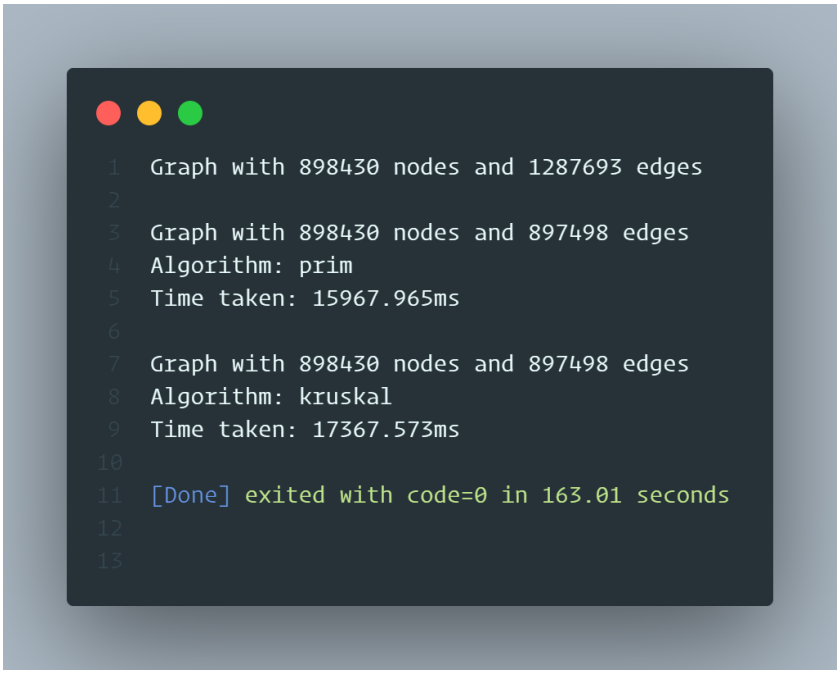
HUGE

Result Analysis

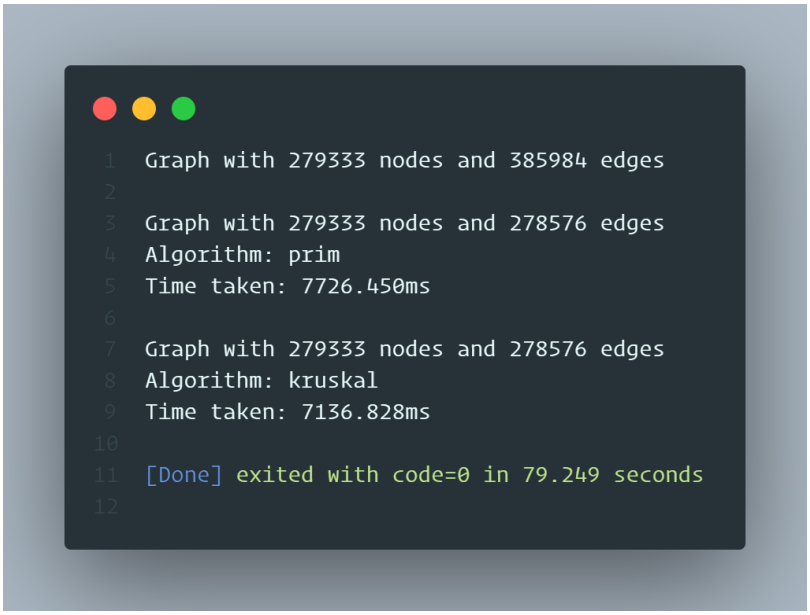
The Analysis between Prims and Kruskal's Algorithm

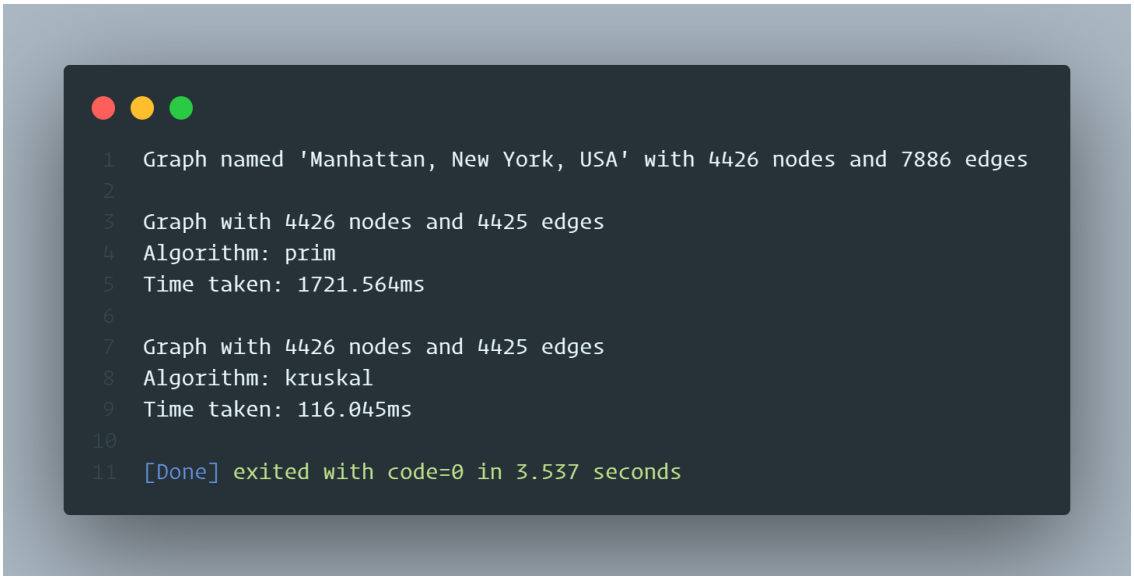


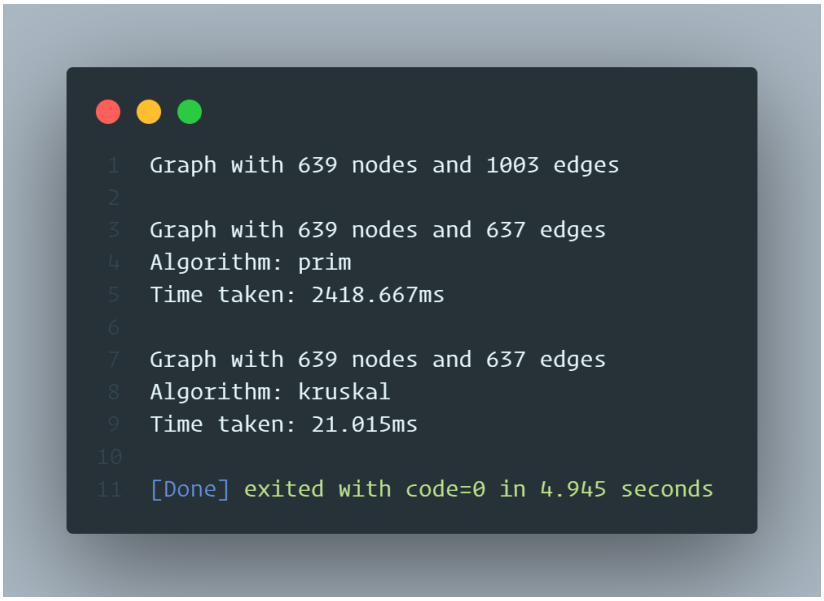
Output: The output for the real-time data of New York with 54128 nodes and 54127 edges gives runtimes for the following algorithms and the difference is seen as shown.



This shows the output for the huge number of nodes that is a dense graph for which the time in seconds is shown as the difference for both Prim's and Kruskal's algorithms. It's shown that the time taken for prims is lesser than the time taken for Kruskal's.







This shows the output for a sparse graph where the time difference is shown and the time taken for Kruskal's is lesser than the time taken for Prims as shown below.

References

1. *https://kylekizirian.github.io/prims-algorithm.html*
   #data structures and graphs
2. *https://www.kaggle.com/datasets/crailtap/street-network-of-new-york-in-graphml*
   #dataset
3. *https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/KA5HJ3*
   #dataset
4. *https://www.kaggle.com/code/usui113yst/basic-network-analysis-tutorial*
   #netorkx on graphml
5. *https://www.softwaretestinghelp.com/minimum-spanning-tree-tutorial/*
   #algorithm
6. *https://networkx.org/documentation/networkx-1.9/reference/index.html*
   #everything networkx
7. *http://www.ics.uci.edu/~eppstein/PADS/*
   #programming, algorithms