1. Introducción

A lo largo de la historia, la movilidad ha sido una necesidad inherente a los seres humanos y conforme hemos ido avanzando como civilización, hemos acumulado conocimientos y desarrollado tecnologías que nos han permitido idear nuevas formas para transportarnos, cada vez más sofisticadas y adaptadas a nuestras necesidades. Cada uno de estos avances nos ha permitido explorar y conectarnos de manera más eficiente con el mundo que nos rodea. En el actual contexto, el transporte urbano se ha convertido en una pieza fundamental. La planificación adecuada de las rutas y paraderos no solo es esencial para el desplazamiento eficiente de personas (y objetos), sino también para fomentar la movilidad sostenible y reducir la congestión y la contaminación.

En este trabajo, se aborda el desafío de, por medio de los algoritmos de Prim y Kruskal, construir árboles de expansión que nos den una idea sobre la planificación de rutas y paraderos en el transporte urbano, específicamente dentro de la comuna de Talca. Se describe en detalle la implementación del sistema, incluyendo los algoritmos utilizados y las funcionalidades desarrolladas. Así mismo, se presentarán los resultados obtenidos.

2. Aplicación

2.1. Introducción

Comenzando con un archivo GeoJSON que contiene información sobre los paraderos, en particular nos interesan las coordenadas geográficas. A partir de allí, se genera la construcción de un grafo completamente conectado. Para ello debemos realizar un proceso combinatorio, aquello es completado con un doble bucle for, restringiendo el alcance del segundo loop, para no repetir innecesariamente la inserción de nodos en el grafo. Este proceso es crucial para garantizar que el grafo esté completamente conectado y listo para ser utilizado en los algoritmos de Kruskal y Prim. Una vez que completado el grafo, aplicamos los algoritmos de Kruskal y Prim, que nos permiten encontrar las conexiones óptimas entre los paraderos, generando los árboles de expansión correspondientes. Finalmente, utilizamos las bibliotecas Streamlit, folium y streamlit_folium, para mostrar visualmente los resultados obtenidos. A continuación, se detallarán los puntos principales y su correspondiente aplicación dentro del código desarrollado.

2.2. Detalle de implementación

2.2.1. Lectura y creación de grafo

Dada la naturaleza del sistema usado para representar la información, donde el código es ejecutado una y otra vez, no podemos permitirnos realizar la lectura del archivo y creación del grafo en cada ejecución. Por ello, dentro de la función read_data(), se hace uso de la librería pickle.

Se realiza la lectura del archivo GeoJSON y se consulta previamente si existe un registro anterior del grafo almacenado en el directorio.

```
geojson = gpd.read_file(DATA_PATH, encoding='utf-8')
    # Clean data
    geojson = geojson[geojson["id"].notna()] # Avoid NaN values on ids

    G = nx.Graph() # G[x] -> ( (p1,p2), dist )
    try:
        G = pickle.load(open('graph.pickle', 'rb'))
        return G, geojson.geometry.values
    except:
        pass
```

En caso de no existir registro del grafo en sistema, se procede con su creación, dentro de la misma función read_data().

```
for i in range(len(geojson.geometry.values)):
        for j in range(i + 1, len(geojson.geometry.values)):
            G.add_edge(
                (geojson.geometry.values[i].y, geojson.geometry.values[i].x),
                (geojson.geometry.values[j].y, geojson.geometry.values[j].x),
                weight = haversine(
                    (geojson.geometry.values[i].y, geojson.geometry.values[i].x),
                    (geojson.geometry.values[j].y, geojson.geometry.values[j].x),
                    Unit.KILOMETERS
                )
            )
    pickle.dump(G, open('graph.pickle', 'wb'))
    return G, geojson.geometry.values
```

Su funcionamiento es bastante simple, realiza una combinatoria usando doble loop for y por cada combinación de coordenadas geográficas de paraderos, calcula la distancia y almacena una tupla dentro del grafo con su respectivo peso representado por la distancia entre los dos puntos.

Formato: ((p1,p2), weight = distancia)

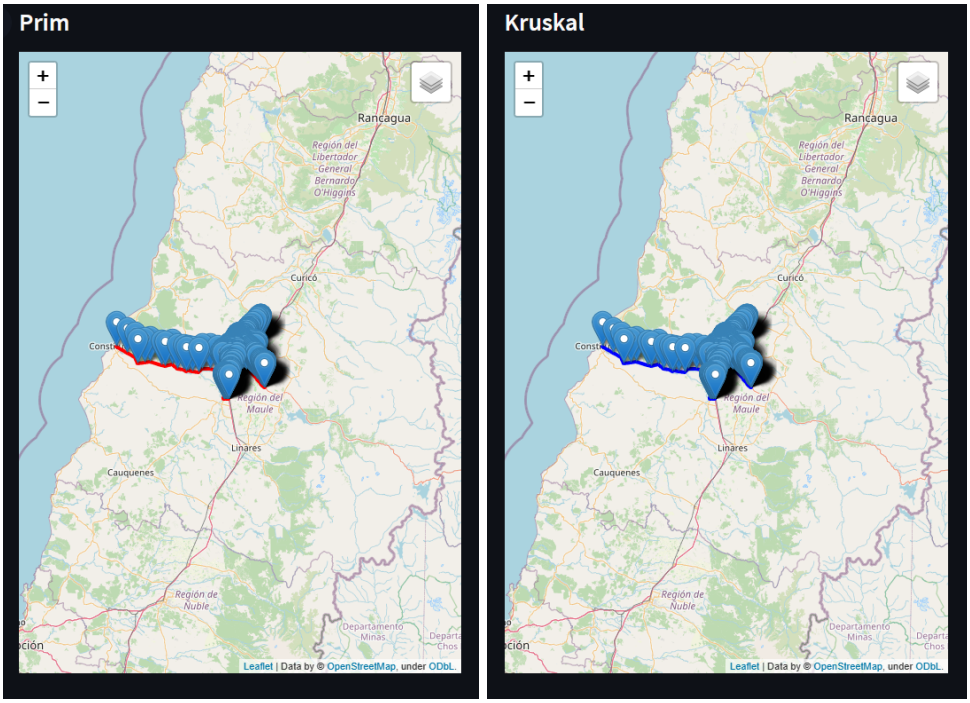
La tupla p1 y p2, hace referencia a cada arista, donde tanto p1 como p2 son los nodos y cada nodo corresponde a una tupla, que almacena coordenadas geográficas en la forma (longitud, latitud). Finalmente, la distancia es calculada usando la fórmula de Haversine, que utiliza la superficie esférica de la tierra para realizar sus cálculos, en este caso se ha usado la implementación realizada por Julien Deniau en el paquete Python haversine 2.8.0.

2.2.2. Aplicación

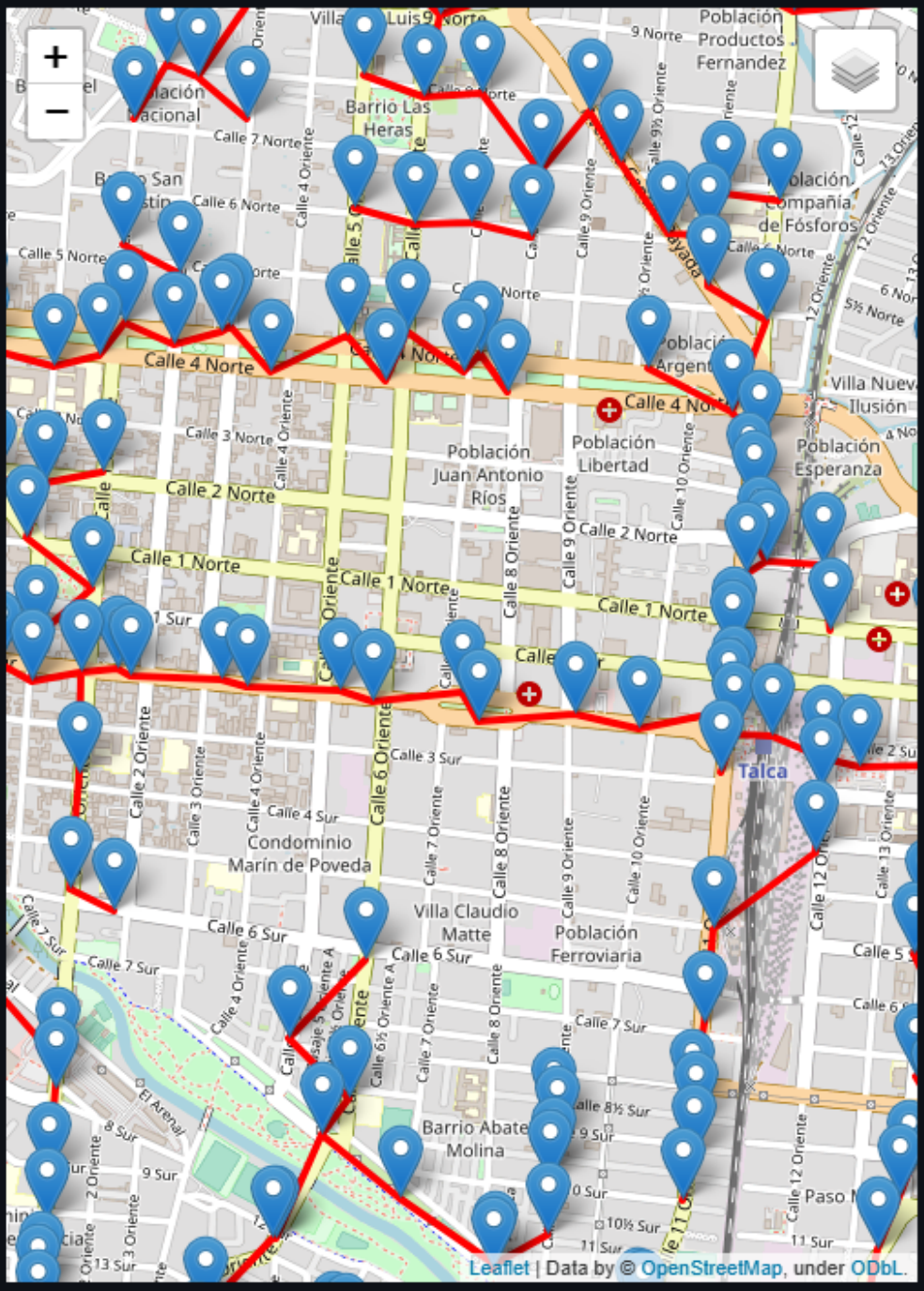
La implementación del algoritmo es realizada utilizando la biblioteca Python NetworkX, a su vez y para efectos de representación de resultados, se calcula el tiempo de ejecución.

```
# Prim
start_time_prim = time.time_ns()
t_prim = nx.minimum_spanning_tree(G, algorithm='prim')
end_time_prim = time.time_ns()
# Kruskal
start_time_kruskal = time.time_ns()
t_kruskal = nx.minimum_spanning_tree(G, algorithm='kruskal')
end_time_kruskal = time.time_ns()
```

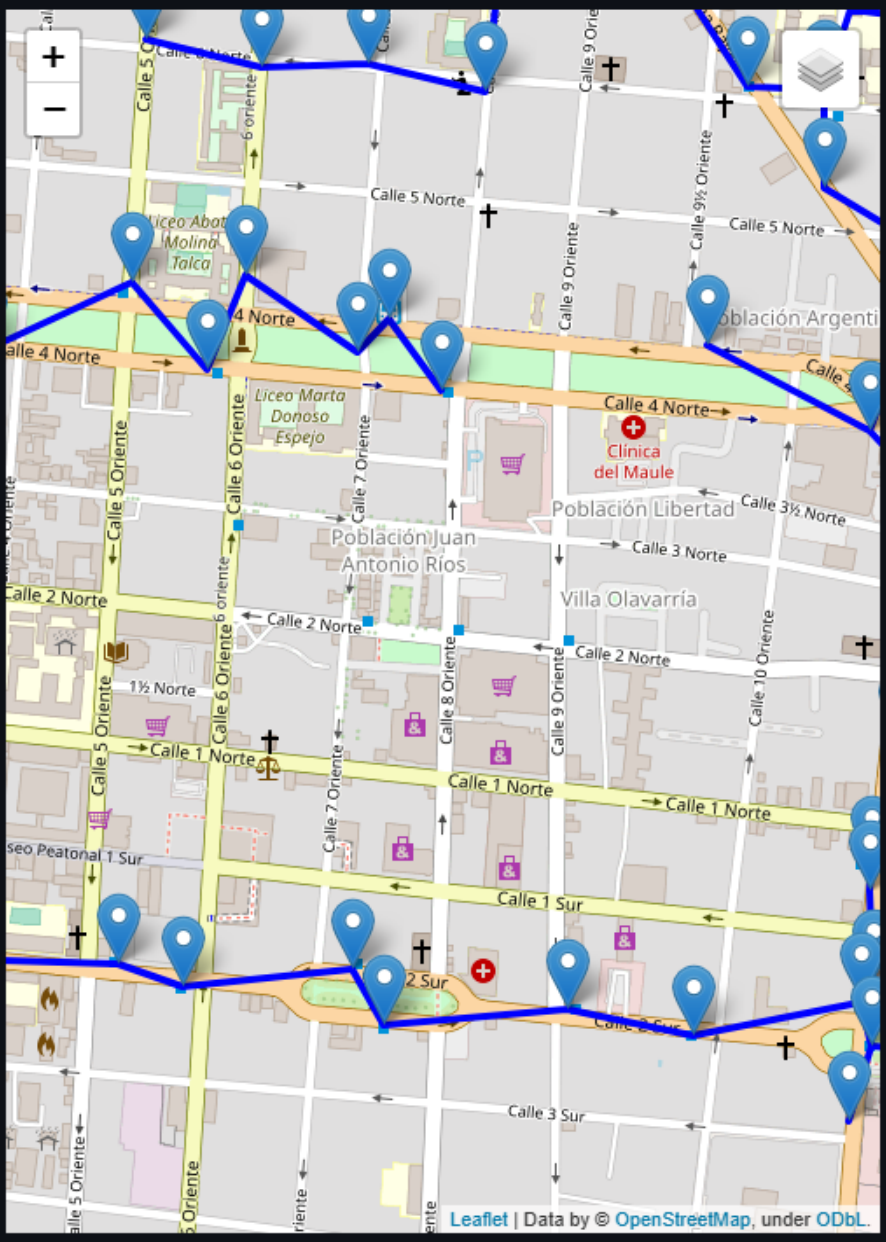
2.3. Pruebas de funcionamiento



Prim

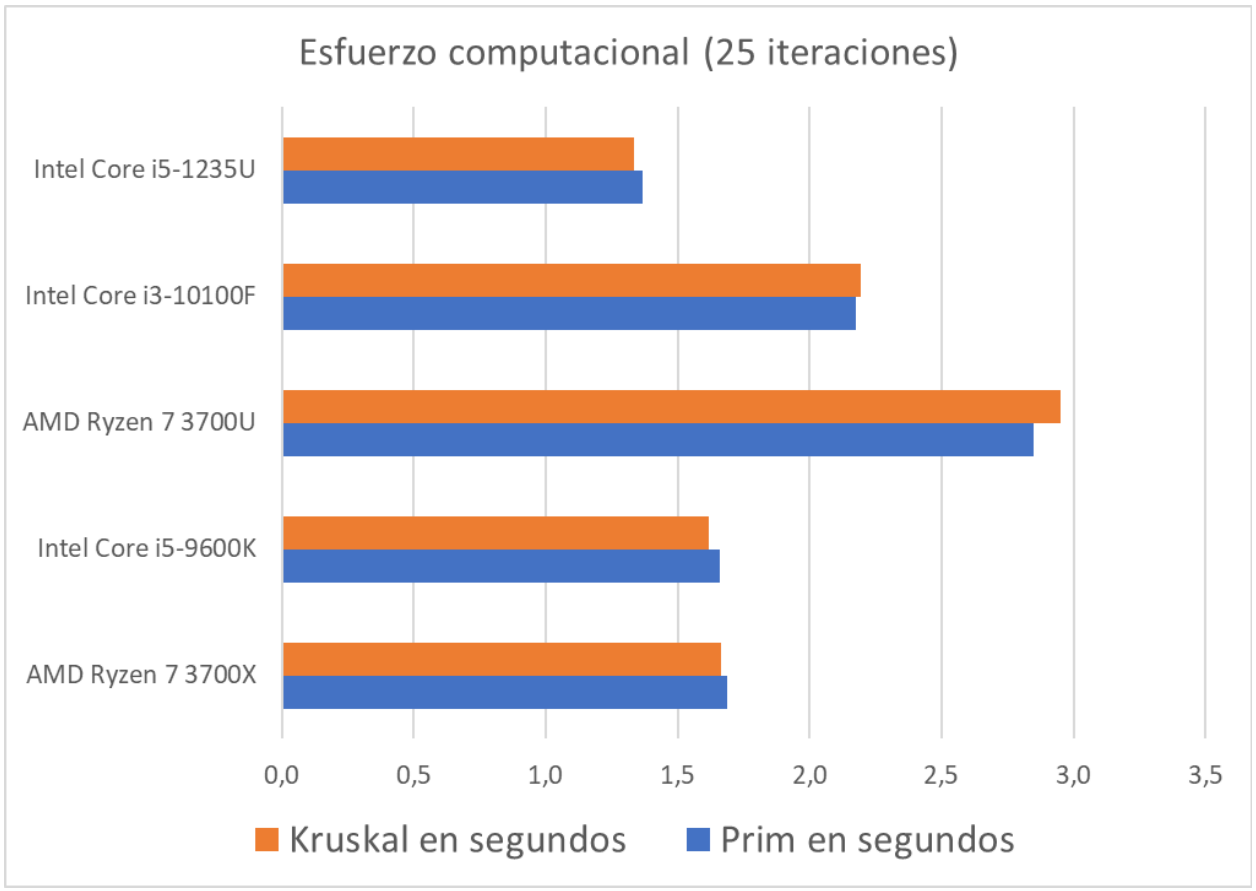

Población Productos Fernandez
Barrio Las Heras
Población Compañía de Fósforos
Villa Nueva Ilusión
Población Esperanza
Población Libertad
Población Juan Antonio Ríos
Calle 4 Norte
Calle 3 Norte
Calle 2 Norte
Calle 1 Norte
Calle 1 Sur
Calle 2 Oriente
Calle 3 Oriente
Calle 4 Oriente
Calle 6 Oriente
Calle 7 Oriente
Calle 8 Oriente
Calle 9 Oriente
Calle 10 Oriente
Calle 12 Oriente
Calle 13 Oriente
Talca
Calle 3 Sur
Calle 4 Sur
Condominio Marín de Poveda
Villa Claudio Matte
Población Ferroviaria
Calle 6 Sur
Calle 7 Sur
Barrio Abate Molina
Paso
Leaflet | Data by © OpenStreetMap, under ODbL.

Kruskal


Calle 5 Norte
Calle 4 Norte
Calle 3 Norte
Calle 2 Norte
Calle 1 Norte
Calle 1 Sur
Calle 3 Sur
Calle 5 Oriente
Calle 6 Oriente
Calle 7 Oriente
Calle 8 Oriente
Calle 9 Oriente
Calle 10 Oriente
Liceo Marta Donoso Espejo
Clínica del Maule
Población Libertad
Población Juan Antonio Ríos
Villa Olavarría
Leaflet | Data by © OpenStreetMap, under ODbL.

3. Resultados y conclusiones

A partir del análisis que se puede realizar a los dos grafos generados, al aplicar los algoritmos Prim y Kruskal para la creación de árbol de expansión mínima. Se obtuvieron ciertos parámetros, de los cuales la mayoría coinciden para ambos casos. Principalmente se observa que ambos grafos poseen 918 vértices y 917 aristas, coincidiendo a su vez el ser acíclicos. Además, ambos grafos comparten el mismo peso total, 229,22898 km. Si bien ambos grafos consiguen el mismo resultado, estos se diferencian en su tiempo de procesamiento, teniendo el algoritmo Prim un tiempo promedio de 1.943728648 s, mientras que el algoritmo Kruskal un tiempo promedio de 1.944805869 s.



La elección del algoritmo depende del contexto y los requisitos específicos de la aplicación. En términos generales, si el grafo es denso (con muchas aristas), el algoritmo de Prim suele ser más eficiente. Por otro lado, si el grafo es disperso (con pocas aristas), el algoritmo de Kruskal puede ser más adecuado. En aplicaciones reales, se recomienda evaluar las características del grafo, como su densidad, el tamaño del grafo y los requisitos de rendimiento, para tomar una decisión.

4. Referencias

Streamlit. (s.f.). Streamlit Documentation. Recuperado de https://docs.streamlit.io/

NetworkX. (s.f.). NetworkX Documentation. Recuperado de https://networkx.org/documentation/stable/index.html