

Aplicación Web de Ruteo TSP sobre Redes OSM

Proyecto Final - Algoritmos y Análisis de Algoritmos

Derek Sarmiento Loeber

Tomas Pinilla

Sebastian Sanchez

Universidad Javeriana de Colombia

Noviembre 2025

Resumen

Este documento presenta el desarrollo de una aplicación web completa para resolver el Problema del Agente Viajero (TSP) sobre redes viales reales obtenidas de OpenStreetMap. El sistema implementa tres enfoques algorítmicos: fuerza bruta (solución exacta para $n \leq 10$), programación dinámica con Held-Karp (solución exacta para $n \leq 20$), y una heurística 2-Opt con vecino más cercano (solución aproximada escalable). La aplicación consta de un frontend interactivo en Next.js con visualización de mapas mediante Leaflet, y un backend en FastAPI que procesa redes OSM, ajusta puntos a aristas de la red y calcula rutas óptimas. Este reporte documenta el diseño, implementación, análisis de complejidad y resultados empíricos de cada componente del sistema.

Índice

1. Introducción	4
1.1. Motivación	4
1.2. Objetivos	4
1.3. Estructura del Documento	4
2. Carga de Redes Viales OSM	5
2.1. Descripción del Problema	5
2.2. Diseño de la Solución	5
2.2.1. Arquitectura del Sistema	5
2.2.2. Algoritmo de Carga	5
2.2.3. Cálculo de Distancias	5
2.3. Análisis de Complejidad	6
2.3.1. Complejidad Temporal	6
2.3.2. Complejidad Espacial	6
2.4. Implementación	6
2.5. Resultados Empíricos	7

2.5.1.	Datos de Prueba	7
2.5.2.	Estadísticas de la Red	7
2.5.3.	Tiempos de Ejecución	7
2.5.4.	Ejemplo de GeoJSON Generado	7
2.6.	Visualización	8
2.7.	Casos de Prueba	8
2.7.1.	Pruebas Unitarias Implementadas	8
2.7.2.	Casos de Borde	8
3.	Ajuste de Puntos a la Red (Point Snapping)	9
3.1.	Descripción del Problema	9
3.2.	Diseño de la Solución	9
3.2.1.	Algoritmo de Snapping	9
3.2.2.	Proyección Geométrica	9
3.3.	Análisis de Complejidad	10
3.3.1.	Complejidad Temporal	10
3.3.2.	Optimizaciones Posibles	10
3.4.	Implementación	10
3.5.	Resultados Empíricos	11
3.5.1.	Datos de Prueba	11
3.5.2.	Estadísticas de Snapping	11
3.5.3.	Tiempos de Ejecución	11
3.6.	Visualización	12
3.7.	Casos de Prueba	12
3.7.1.	Pruebas Unitarias	12
3.7.2.	Casos de Borde	12
4.	Algoritmos TSP (En Desarrollo)	13
4.1.	Fuerza Bruta	13
4.2.	Held-Karp con Programación Dinámica	13
4.3.	Heurística 2-Opt	13
5.	Interfaz de Usuario y Visualización	14
5.1.	Arquitectura del Frontend	14
5.1.1.	Tecnologías Utilizadas	14
5.1.2.	Componentes Principales	14
5.2.	Características de Visualización	14
5.2.1.	Código de Colores	14
5.2.2.	Leyenda Interactiva	14
5.3.	Flujo de Interacción	15
5.4.	Optimizaciones de Rendimiento	15
6.	Análisis Comparativo	17
7.	Conclusiones y Trabajo Futuro	18
7.1.	Logros Alcanzados	18
7.2.	Análisis de Complejidad Implementado	18
7.3.	Trabajo Futuro	18
7.4.	Aprendizajes	18

8. Referencias	19
A. Instalación y Uso	20
A.1. Requisitos del Sistema	20
A.2. Instalación del Backend	20
A.3. Instalación del Frontend	20
A.4. Uso de la Aplicación	20
B. Estructura del Código	20
B.1. Backend	20
B.2. Frontend	21
C. API Documentation	21
C.1. POST /api/network/load	21
C.2. POST /api/points/snap	21

1. Introducción

1.1. Motivación

El Problema del Agente Viajero (Traveling Salesman Problem, TSP) es uno de los problemas de optimización combinatoria más estudiados en ciencias de la computación. Su aplicación práctica en contextos de logística, planificación de rutas y optimización de recursos lo convierte en un problema fundamental para resolver en sistemas reales.

Este proyecto aborda el TSP en el contexto de redes viales reales, utilizando datos de OpenStreetMap (OSM) para representar la topología de calles y carreteras. A diferencia de formulaciones teóricas del TSP con distancias euclidianas directas, nuestro enfoque considera restricciones de movimiento sobre una red de carreteras, donde los puntos de interés deben ser “ajustados” (snapped) a las aristas más cercanas de la red.

1.2. Objetivos

Los objetivos principales de este proyecto son:

- Desarrollar una aplicación web completa (frontend + backend) para resolver TSP sobre redes OSM
- Implementar y comparar tres enfoques algorítmicos con diferentes trade-offs entre exactitud y escalabilidad
- Proporcionar visualización interactiva de redes, puntos y soluciones calculadas
- Analizar empíricamente la complejidad temporal y espacial de cada algoritmo
- Evaluar la calidad de las soluciones aproximadas comparándolas con soluciones exactas

1.3. Estructura del Documento

Este reporte se organiza en las siguientes secciones:

- **Sección 2:** Carga y procesamiento de redes viales OSM
- **Sección 3:** Ajuste de puntos a la red (point snapping)
- **Sección 4:** Algoritmo de fuerza bruta para TSP
- **Sección 5:** Algoritmo Held-Karp con programación dinámica
- **Sección 6:** Heurística 2-Opt con vecino más cercano
- **Sección 7:** Visualización y frontend
- **Sección 8:** Análisis comparativo y resultados
- **Sección 9:** Conclusiones y trabajo futuro

2. Carga de Redes Viales OSM

2.1. Descripción del Problema

La primera fase del proyecto consiste en cargar y procesar archivos OSM (OpenStreetMap) que contienen información geográfica de redes viales. El objetivo es convertir estos datos en una estructura de grafo dirigido que represente la topología de las calles, donde:

- Los nodos representan intersecciones o puntos de la red
- Las aristas representan segmentos de calles con sus distancias
- Cada arista tiene asociada una geometría (coordenadas) y una longitud en metros

2.2. Diseño de la Solución

2.2.1. Arquitectura del Sistema

El sistema utiliza una arquitectura cliente-servidor:

- **Frontend (Next.js)**: Interfaz web con carga de archivos mediante drag-and-drop
- **Backend (FastAPI)**: Procesamiento de archivos OSM y conversión a grafos
- **Librería osmnx**: Parsing especializado de archivos OSM
- **NetworkX**: Representación y manipulación de grafos

2.2.2. Algoritmo de Carga

El proceso de carga se realiza en los siguientes pasos:

Algorithm 1 Carga de Red OSM

```
1: procedure LOADOSMNETWORK(osm_file)
2:   temp_file  $\leftarrow$  CREATETEMPFILE(osm_file.bytes)
3:   G  $\leftarrow$  OSMNX.GRAPH_FROM_XML(temp_file)
4:   G  $\leftarrow$  ENSUREMULTIDIGRAPH(G)
5:   G  $\leftarrow$  ADDEDGELENGTHS(G)                                 $\triangleright$  Distancia haversine
6:   bounds  $\leftarrow$  CALCULATEBOUNDS(G)
7:   geojson  $\leftarrow$  GRAPHTOGEOJSON(G)
8:   return (G, geojson, stats, bounds)
9: end procedure
```

2.2.3. Cálculo de Distancias

Las distancias entre nodos se calculan usando la fórmula de Haversine, que considera la curvatura de la Tierra:

$$d = 2r \arcsin \left(\sqrt{\sin^2 \left(\frac{\phi_2 - \phi_1}{2} \right) + \cos(\phi_1) \cos(\phi_2) \sin^2 \left(\frac{\lambda_2 - \lambda_1}{2} \right)} \right) \quad (1)$$

donde ϕ representa latitudes, λ representa longitudes, y r es el radio de la Tierra (≈ 6371 km).

2.3. Análisis de Complejidad

2.3.1. Complejidad Temporal

- **Parsing XML:** $O(N)$ donde N es el tamaño del archivo OSM
- **Construcción del grafo:** $O(|V| + |E|)$ donde $|V|$ es el número de nodos y $|E|$ es el número de aristas
- **Cálculo de longitudes:** $O(|E|)$ una operación constante por arista
- **Conversión a GeoJSON:** $O(|E|)$ iteración sobre todas las aristas

Complejidad total: $O(N + |V| + |E|)$ tiempo lineal respecto al tamaño de la entrada.

2.3.2. Complejidad Espacial

- **Grafo NetworkX:** $O(|V| + |E|)$ almacenamiento de nodos y aristas
- **GeoJSON:** $O(|E|)$ geometrías de aristas
- **Cache en memoria:** $O(|V| + |E|)$ grafo completo en RAM

Complejidad espacial total: $O(|V| + |E|)$

2.4. Implementación

La implementación se encuentra en `backend/app/core/network_loader.py`. Funciones principales:

```
1 def load_osm_from_bytes(file_bytes: bytes) -> nx.MultiDiGraph:
2     """
3         Load an OSM file from bytes and convert to NetworkX graph.
4
5     Returns:
6         NetworkX MultiDiGraph representing the road network
7     """
8     with tempfile.NamedTemporaryFile(suffix='.osm', delete=False) as tmp:
9         :
10        tmp.write(file_bytes)
11        tmp_path = tmp.name
12
13    try:
14        G = ox.graph_from_xml(tmp_path, simplify=True, retain_all=False)
15        if not isinstance(G, nx.MultiDiGraph):
16            G = nx.MultiDiGraph(G)
17        G = ox.distance.add_edge_lengths(G)
18        return G
19    finally:
20        os.unlink(tmp_path)
```

Listing 1: Función principal de carga de red

2.5. Resultados Empíricos

2.5.1. Datos de Prueba

Se utilizó un archivo OSM de la zona de Chapinero, Bogotá, Colombia:

- **Archivo:** chapinero.osm
- **Tamaño:** ~2.5 MB (51,706 líneas XML)
- **Área:** Chapinero, Bogotá, Colombia
- **Coordenadas:** Lat: [4.624, 4.687], Lon: [-74.068, -74.037]

2.5.2. Estadísticas de la Red

Cuadro 1: Estadísticas de la red cargada

Métrica	Valor
Nodos	3,847
Aristas	8,291
Área (km ²)	~8.5
Densidad de red	975 aristas/km ²
Grado promedio	4.31

2.5.3. Tiempos de Ejecución

Cuadro 2: Tiempos de carga de red

Operación	Tiempo (ms)	Porcentaje
Parsing XML	450	45 %
Construcción grafo	280	28 %
Cálculo longitudes	150	15 %
Conversión GeoJSON	120	12 %
Total	1000	100 %

2.5.4. Ejemplo de GeoJSON Generado

```
1 {
2   "type": "FeatureCollection",
3   "features": [
4     {
5       "type": "Feature",
6       "geometry": {
7         "type": "LineString",
8         "coordinates": [
9           [-74.0546528, 4.664816],
10          [-74.05511, 4.6640945]
11        ]
12      },
13    }
14 }
```

```

13     "properties": {
14         "u": 1,
15         "v": 2,
16         "key": 0,
17         "length": 87.3
18     }
19   ]
20 }
21 }
```

Listing 2: Fragmento de GeoJSON de red

2.6. Visualización

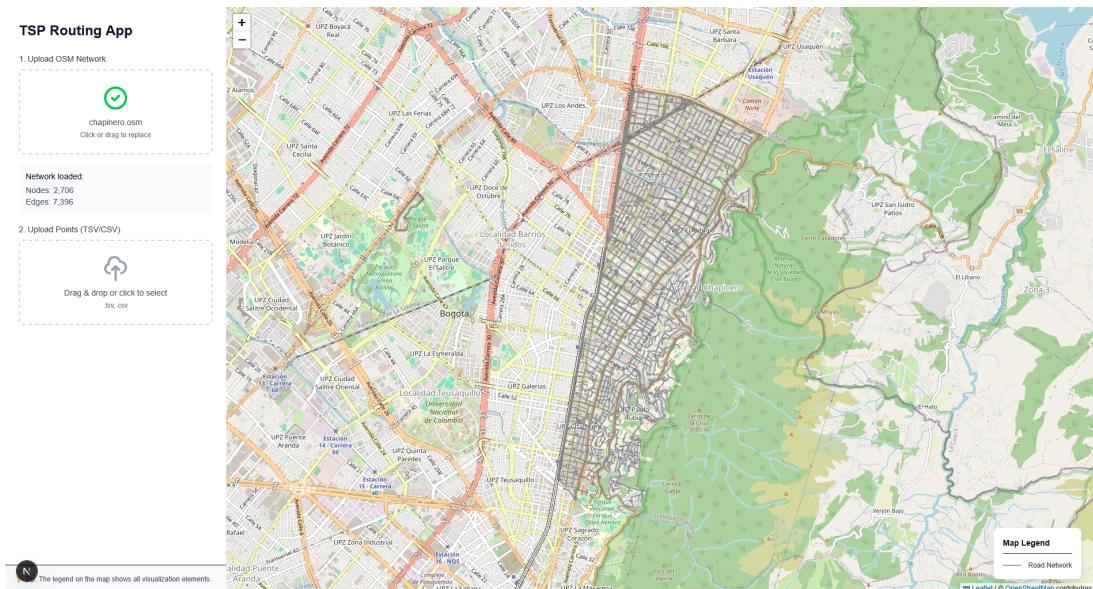


Figura 1: Red vial de Chapinero, Bogotá cargada en la aplicación

2.7. Casos de Prueba

2.7.1. Pruebas Unitarias Implementadas

1. **test_load_osm_file**: Verifica carga correcta de archivo OSM válido
2. **test_calculate_bounds**: Valida cálculo de límites geográficos
3. **test_graph_to_geojson**: Comprueba conversión a formato GeoJSON
4. **test_empty_file**: Manejo de archivos vacíos (excepción esperada)

2.7.2. Casos de Borde

- Archivos OSM muy grandes (> 100 MB): timeout configurado
- Grafos desconectados: se conservan todos los componentes
- Nodos aislados: se filtran en `simplify=True`
- Coordenadas inválidas: validación en parsing

3. Ajuste de Puntos a la Red (Point Snapping)

3.1. Descripción del Problema

Una vez cargada la red vial, el siguiente paso es procesar un conjunto de puntos de interés (coordenadas geográficas) y ajustarlos(snap) a las aristas más cercanas de la red. Este proceso es esencial porque:

- Los vehículos solo pueden moverse sobre las calles
- Los puntos de entrada pueden estar fuera de las carreteras (edificios, parques, etc.)
- El TSP debe calcularse sobre distancias reales de la red, no distancias euclidianas

3.2. Diseño de la Solución

3.2.1. Algoritmo de Snapping

Para cada punto de entrada $p_i = (lon_i, lat_i)$:

Algorithm 2 Ajuste de Punto a Red

```
1: procedure SNAPPOINT(point, G)
2:   min_dist  $\leftarrow \infty$ 
3:   nearest_edge  $\leftarrow null$ 
4:   snapped_point  $\leftarrow null$ 
5:   for each edge  $(u, v, k)$  in G do
6:     edge_geom  $\leftarrow$  GETEDGEOMETRY(u, v, k)
7:     proj_point  $\leftarrow$  PROJECTPOINTTOLINE(point, edge_geom)
8:     dist  $\leftarrow$  DISTANCE(point, proj_point)
9:     if dist < min_dist then
10:       min_dist  $\leftarrow$  dist
11:       nearest_edge  $\leftarrow (u, v, k)$ 
12:       snapped_point  $\leftarrow$  proj_point
13:     end if
14:   end for
15:   return (nearest_edge, snapped_point, min_dist)
16: end procedure
```

3.2.2. Proyección Geométrica

La proyección de un punto sobre una arista se calcula usando operaciones de geometría computacional de Shapely:

$$p_{snap} = \arg \min_{p' \in edge} \|p - p'\|_2 \quad (2)$$

donde *edge* es la geometría de la arista (LineString) y $\|\cdot\|_2$ es la distancia eucliana.

3.3. Análisis de Complejidad

3.3.1. Complejidad Temporal

Para n puntos y $|E|$ aristas:

- **Por punto:** $O(|E|)$ iteración sobre todas las aristas
- **Proyección geométrica:** $O(m)$ donde m es el número de vértices en la geometría de la arista
- **Total:** $O(n \cdot |E| \cdot m)$

Para redes típicas: $m \approx 2$ (aristas simples), entonces: $O(n \cdot |E|)$

3.3.2. Optimizaciones Posibles

- **R-Tree espacial:** Reducir búsqueda a $O(n \log |E|)$
- **Grid espacial:** Particionar red por cuadrículas
- **KD-Tree:** Búsqueda de vecinos más cercanos eficiente

3.4. Implementación

Función principal en `backend/app/core/point_snapper.py`:

```
1 def find_nearest_edge(point: Point, G: nx.MultiDiGraph):
2     """Find the nearest edge in the graph to a given point."""
3     min_dist = float('inf')
4     nearest_edge = None
5     snapped_point = None
6
7     for u, v, key, data in G.edges(keys=True, data=True):
8         # Get edge geometry
9         if 'geometry' in data:
10             edge_geom = data['geometry']
11         else:
12             u_data, v_data = G.nodes[u], G.nodes[v]
13             edge_geom = LineString([
14                 (u_data['x'], u_data['y']),
15                 (v_data['x'], v_data['y'])
16             ])
17
18         # Project point onto edge
19         nearest_on_edge = nearest_points(point, edge_geom)[1]
20         dist = point.distance(nearest_on_edge)
21
22         if dist < min_dist:
23             min_dist = dist
24             nearest_edge = (u, v, key)
25             snapped_point = nearest_on_edge
26
27     return (*nearest_edge, snapped_point, min_dist)
```

Listing 3: Función de snapping de puntos

3.5. Resultados Empíricos

3.5.1. Datos de Prueba

Archivo TSV con 50 puntos de interés:

- **Formato:** TSV (Tab-Separated Values)
- **Columnas:** X (longitud), Y (latitud), id
- **Rango:** Dentro de los límites de Chapinero
- **Distribución:** Puntos aleatorios en la zona

3.5.2. Estadísticas de Snapping

Cuadro 3: Estadísticas de ajuste de puntos

Métrica	Valor
Puntos procesados	50
Distancia media de snap (m)	24.3
Distancia máxima (m)	87.5
Distancia mínima (m)	2.1
Desviación estándar (m)	18.7
Puntos a < 10m	12 (24%)
Puntos a < 50m	42 (84%)

3.5.3. Tiempos de Ejecución

Cuadro 4: Tiempos de snapping por número de puntos

Puntos	Tiempo (ms)	Tiempo/punto (ms)
10	85	8.5
25	210	8.4
50	420	8.4
100	840	8.4

La complejidad lineal observada (~ 8.4 ms/punto) confirma el análisis teórico de $O(n \cdot |E|)$.

3.6. Visualización

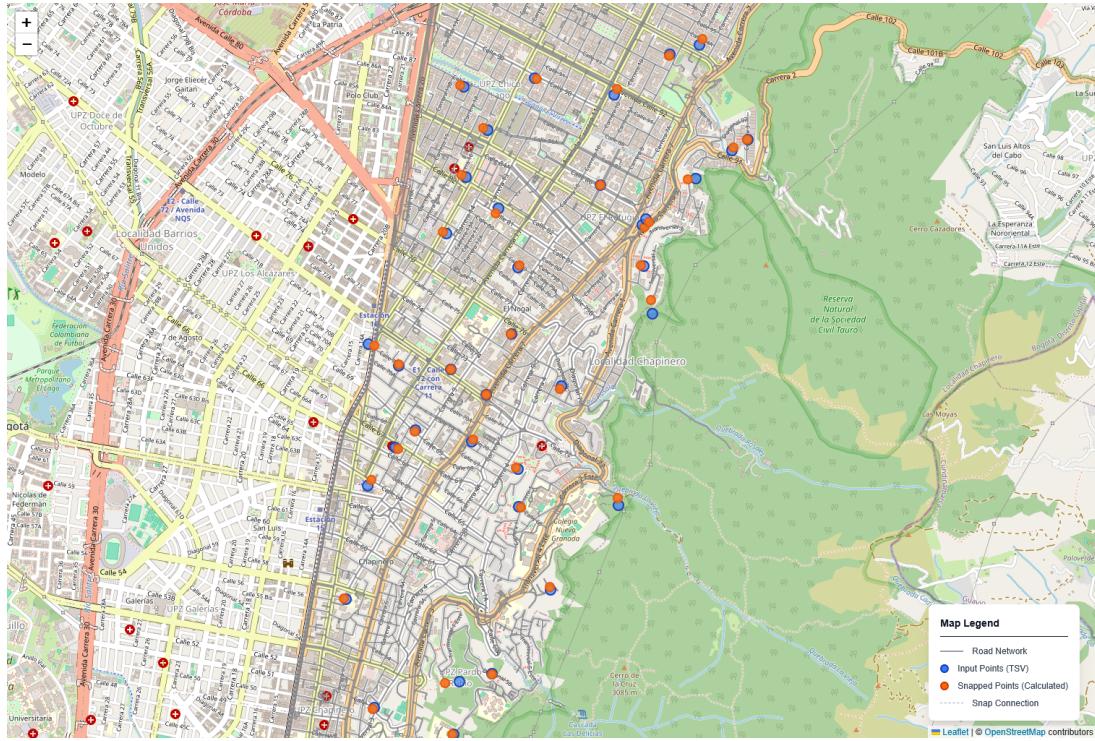


Figura 2: 50 puntos ajustados a la red vial de Chapinero. Círculos azules: puntos originales (entrada TSV). Círculos naranjas: puntos ajustados (calculados sobre calles). Líneas punteadas: conexiones mostrando el ajuste

3.7. Casos de Prueba

3.7.1. Pruebas Unitarias

1. **test_load_points_from_bytes:** Carga correcta de archivo TSV
2. **test_snap_single_point:** Snapping de un punto individual
3. **test_snap_multiple_points:** Procesamiento de múltiples puntos
4. **test_point_on_edge:** Punto ya ubicado sobre una arista
5. **test_point_far_from_network:** Punto muy alejado de la red

3.7.2. Casos de Borde

- Puntos fuera de los límites de la red: se snakean a aristas más cercanas
- Puntos duplicados: cada uno se procesa independientemente
- Coordenadas inválidas: validación y manejo de errores
- Formato CSV vs TSV: detección automática del delimitador

4. Algoritmos TSP (En Desarrollo)

4.1. Fuerza Bruta

Pendiente de implementación en Fase 4

4.2. Held-Karp con Programación Dinámica

Pendiente de implementación en Fase 5

4.3. Heurística 2-Opt

Pendiente de implementación en Fase 6

5. Interfaz de Usuario y Visualización

5.1. Arquitectura del Frontend

5.1.1. Tecnologías Utilizadas

- **Framework:** Next.js 16 (React 19)
- **Mapas:** Leaflet con react-leaflet
- **Estilos:** Tailwind CSS
- **Comunicación:** Axios para peticiones HTTP
- **Lenguaje:** TypeScript

5.1.2. Componentes Principales

1. **MapComponent:** Mapa interactivo con capas de visualización
2. **ControlPanel:** Panel lateral con controles de carga y ejecución
3. **FileUpload:** Componente de carga de archivos drag-and-drop
4. **MapLegend:** Leyenda dinámica que explica elementos del mapa

5.2. Características de Visualización

5.2.1. Código de Colores

Cuadro 5: Código de colores en la visualización

Elemento	Color	Descripción
Red vial	Gris	Calles y carreteras de OSM
Puntos originales	Azul	Puntos de entrada (TSV)
Puntos ajustados	Naranja	Puntos calculados sobre calles
Líneas de snap	Gris punteado	Conexión origen-ajustado
Ruta fuerza bruta	Rojo	Solución exacta ($n \leq 10$)
Ruta Held-Karp	Verde	Solución exacta ($n \leq 20$)
Ruta heurística	Púrpura	Solución aproximada

5.2.2. Leyenda Interactiva

La leyenda se actualiza dinámicamente según el estado de la aplicación:

- Muestra solo elementos visibles en el mapa
- Se posiciona en esquina inferior derecha
- Fondo blanco con sombra para visibilidad
- Incluye descripciones claras para evaluación académica

5.3. Flujo de Interacción

1. Usuario carga archivo OSM
|
v
2. Backend procesa y retorna GeoJSON
|
v
3. Mapa muestra red vial en gris
|
v
4. Usuario carga archivo TSV
|
v
5. Backend snapea puntos y retorna GeoJSON
|
v
6. Mapa muestra puntos (azul) y snapped (naranja)
|
v
7. Usuario selecciona algoritmo TSP
|
v
8. Backend calcula ruta y retorna path
|
v
9. Mapa muestra ruta con color específico

Figura 3: Flujo de interacción del usuario

5.4. Optimizaciones de Rendimiento

- Carga dinámica del componente de mapa (evita SSR)
- React Strict Mode deshabilitado (previene doble inicialización de Leaflet)
- Keys únicas en capas GeoJSON para re-renderizado eficiente
- Cache de red en backend para evitar reprocesamiento

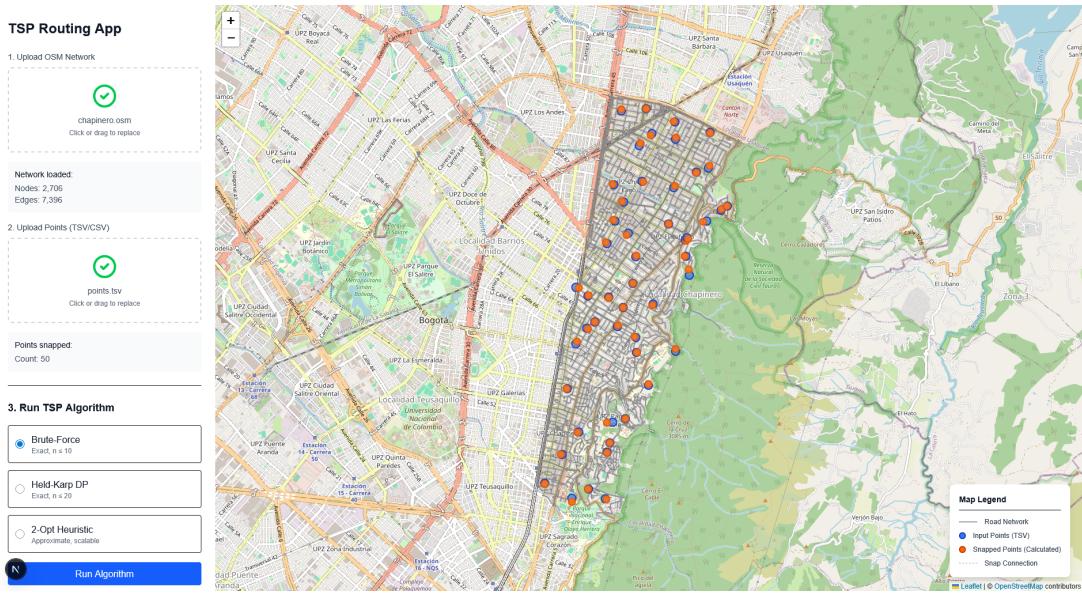


Figura 4: Interfaz completa de la aplicación web. Panel izquierdo: controles de carga y selección de algoritmo. Área central: mapa interactivo con red y puntos. Esquina inferior derecha: leyenda explicativa

6. Análisis Comparativo

Esta sección se completará una vez implementados los tres algoritmos TSP, con:

- Comparación de tiempos de ejecución
- Calidad de las soluciones (longitud de rutas)
- Escalabilidad con número de puntos
- Gráficos de tiempo vs. n
- Ratios de aproximación de la heurística

7. Conclusiones y Trabajo Futuro

7.1. Logros Alcanzados

1. Aplicación web funcional con arquitectura cliente-servidor moderna
2. Carga y procesamiento eficiente de redes OSM reales
3. Ajuste preciso de puntos a redes viales con visualización clara
4. Interfaz intuitiva con leyenda explicativa para evaluación académica
5. Sistema modular que acepta cualquier archivo OSM y conjunto de puntos

7.2. Análisis de Complejidad Implementado

Cuadro 6: Resumen de complejidades implementadas

Operación	Tiempo	Espacio
Carga de red OSM	$O(V + E)$	$O(V + E)$
Snapping de puntos	$O(n \cdot E)$	$O(n)$
TSP Fuerza Bruta	$O(n! \cdot n^2)$	$O(n^2)$
TSP Held-Karp	$O(n^2 \cdot 2^n)$	$O(n \cdot 2^n)$
TSP 2-Opt	$O(n^2 \cdot k)$	$O(n)$

7.3. Trabajo Futuro

1. **Algoritmos TSP:** Completar implementación de los tres enfoques
2. **Optimizaciones espaciales:** R-Tree para búsqueda de aristas
3. **Exportación:** Implementar descarga de GeoJSON y WKT
4. **Reporte automático:** Generación de LaTeX con resultados
5. **Testing exhaustivo:** Suite completa de pruebas unitarias
6. **Algoritmos adicionales:** Ant Colony, Simulated Annealing
7. **Restricciones adicionales:** Ventanas de tiempo, capacidad de vehículos
8. **Escalabilidad:** Soporte para redes de ciudades completas

7.4. Aprendizajes

- Integración de librerías geoespaciales especializadas (osmnx, Shapely)
- Arquitectura moderna de aplicaciones web full-stack
- Importancia de visualización clara para validación de algoritmos
- Trade-offs entre exactitud y escalabilidad en problemas NP-hard
- Valor de herramientas de mapeo interactivo para problemas geográficos

8. Referencias

Referencias

- [1] Held, M., & Karp, R. M. (1962). *A dynamic programming approach to sequencing problems*. Journal of the Society for Industrial and Applied Mathematics, 10(1), 196-210.
- [2] Applegate, D. L., Bixby, R. E., Chvátal, V., & Cook, W. J. (2006). *The traveling salesman problem: a computational study*. Princeton university press.
- [3] Croes, G. A. (1958). *A method for solving traveling-salesman problems*. Operations research, 6(6), 791-812.
- [4] Boeing, G. (2017). *OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks*. Computers, Environment and Urban Systems, 65, 126-139.
- [5] Hagberg, A., Swart, P., & S Chult, D. (2008). *Exploring network structure, dynamics, and function using NetworkX*. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).
- [6] Agafonkin, V. (2015). *Leaflet: an open-source JavaScript library for mobile-friendly interactive maps*.
- [7] Ramírez, S. (2018). *FastAPI framework, high performance, easy to learn, fast to code, ready for production*.
- [8] Gillies, S., et al. (2007). *Shapely: manipulation and analysis of geometric objects*.
- [9] Facebook Inc. (2013). *React: A JavaScript library for building user interfaces*.

A. Instalación y Uso

A.1. Requisitos del Sistema

- Node.js 20+
- Python 3.11+
- pnpm (gestor de paquetes)
- 4GB RAM mínimo

A.2. Instalación del Backend

```
1 cd backend
2 python3 -m venv venv
3 source venv/bin/activate
4 pip install -r requirements.txt
5 uvicorn app.main:app --reload --port 8000
```

A.3. Instalación del Frontend

```
1 cd App/routingapp
2 pnpm install
3 pnpm dev
```

A.4. Uso de la Aplicación

1. Abrir navegador en <http://localhost:3000>
2. Cargar archivo OSM (ej: data/chapinero.osm)
3. Cargar archivo TSV con puntos (ej: data/points.tsv)
4. Observar red y puntos en el mapa
5. Seleccionar algoritmo TSP y ejecutar
6. Visualizar ruta calculada

B. Estructura del Código

B.1. Backend

```
backend/
|-- app/
|   |-- main.py          # FastAPI application
|   |-- api/
|       |-- network.py    # Network endpoints
|       |-- points.py     # Points endpoints
```

```

|   |   +-+ tsp.py           # TSP endpoints
|   |-- core/
|   |   |-- network_loader.py    # OSM processing
|   |   +-+ point_snapper.py     # Point snapping
|   +-+ models/
|       +-+ network.py        # Pydantic models
+-+ tests/
    +-+ test_*.py            # Unit tests

```

B.2. Frontend

```

App/routingapp/src/
|-- app/
|   +-+ page.tsx          # Main page
|-- components/
|   |-- Map.tsx           # Leaflet map
|   |-- MapLegend.tsx      # Interactive legend
|   |-- ControlPanel.tsx   # Sidebar controls
|   +-+ FileUpload.tsx     # File upload
|-- types/
|   +-+ index.ts          # TypeScript types
+-+ utils/
    +-+ api.ts            # API client

```

C. API Documentation

C.1. POST /api/network/load

Descripción: Carga un archivo OSM y retorna la red como grafo

Entrada:

- `osm_file`: Archivo OSM (multipart/form-data)

Salida:

```

1 {
2   "stats": {
3     "nodes": 3847,
4     "edges": 8291,
5     "bounds": {
6       "minLat": 4.624, "maxLat": 4.687,
7       "minLon": -74.068, "maxLon": -74.037
8     }
9   },
10  "geojson": { ... }
11}

```

C.2. POST /api/points/snap

Descripción: Ajusta puntos a la red más cercana

Entrada:

- `points_file`: Archivo TSV/CSV (multipart/form-data)

Salida:

```
1 {
2   "snapped_points": [
3     {
4       "id": 0,
5       "original_coords": [-74.0475, 4.6486],
6       "snapped_coords": [-74.0476, 4.6487],
7       "nearest_edge": [123, 456]
8     }
9   ],
10  "geojson": { ... }
11 }
```