

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	13
4.1.	Algoritmo de Fuerza Bruta	13
4.1.1.	Descripción del Problema	13
4.1.2.	Diseño de la Solución	13
4.1.3.	Análisis de Complejidad	14
4.1.4.	Crecimiento Factorial	15
4.1.5.	Implementación	15
4.1.6.	Funcionalidad de Auto-Subset	17
4.1.7.	Resultados Empíricos	17
4.1.8.	Pruebas Unitarias	18
4.1.9.	Ventajas y Limitaciones	19
4.2.	Held-Karp con Programación Dinámica	20
4.2.1.	Descripción del Algoritmo	20
4.2.2.	Pseudocódigo	20
4.2.3.	Análisis de Complejidad	21
4.2.4.	Implementación	21
4.2.5.	Resultados Experimentales	22
4.2.6.	Pruebas Unitarias	22
4.2.7.	Ventajas y Limitaciones	23
4.3.	Heurística 2-Opt	23

5. Interfaz de Usuario y Visualización	24
5.1. Arquitectura del Frontend	24
5.1.1. Tecnologías Utilizadas	24
5.1.2. Componentes Principales	24
5.2. Características de Visualización	24
5.2.1. Código de Colores	24
5.2.2. Leyenda Interactiva	24
5.3. Flujo de Interacción	25
6. Análisis Comparativo	27
7. Conclusiones y Trabajo Futuro	28
7.1. Logros Alcanzados	28
7.2. Análisis de Complejidad Implementado	28
7.3. Trabajo Futuro	28
7.4. Aprendizajes	28
8. Referencias	29
A. Instalación y Uso	30
A.1. Requisitos del Sistema	30
A.2. Instalación del Backend	30
A.3. Instalación del Frontend	30
A.4. Uso de la Aplicación	30
B. API Documentation	30
B.1. POST /api/network/load	30
B.2. POST /api/points/snap	31
B.3. POST /api/tsp;bruteforce	31

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

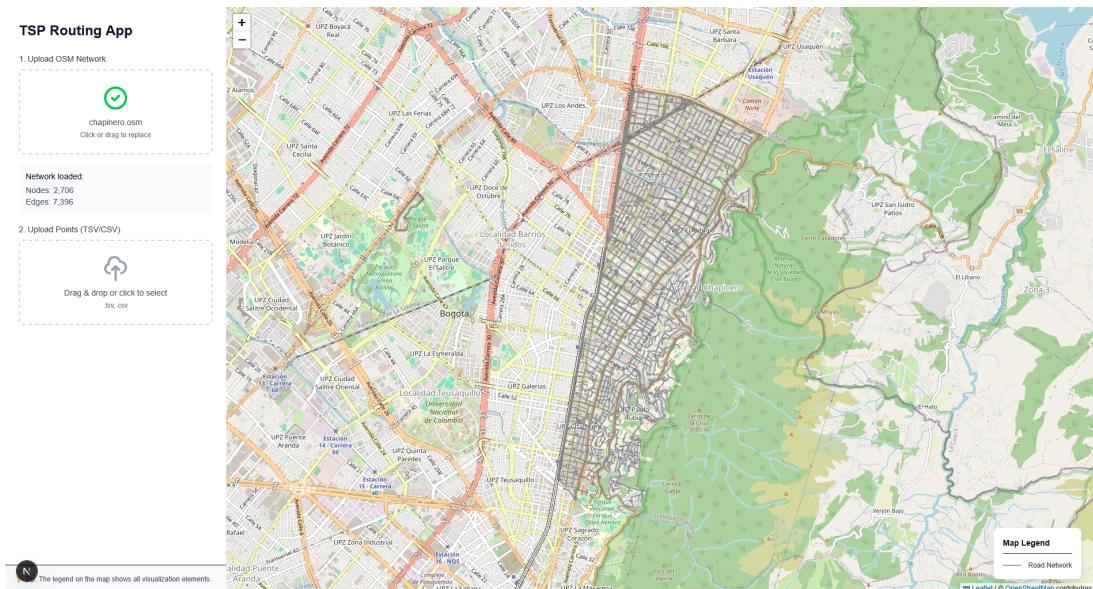


Figura 1: Red vial de Chapinero, Bogotá cargada en la aplicación

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

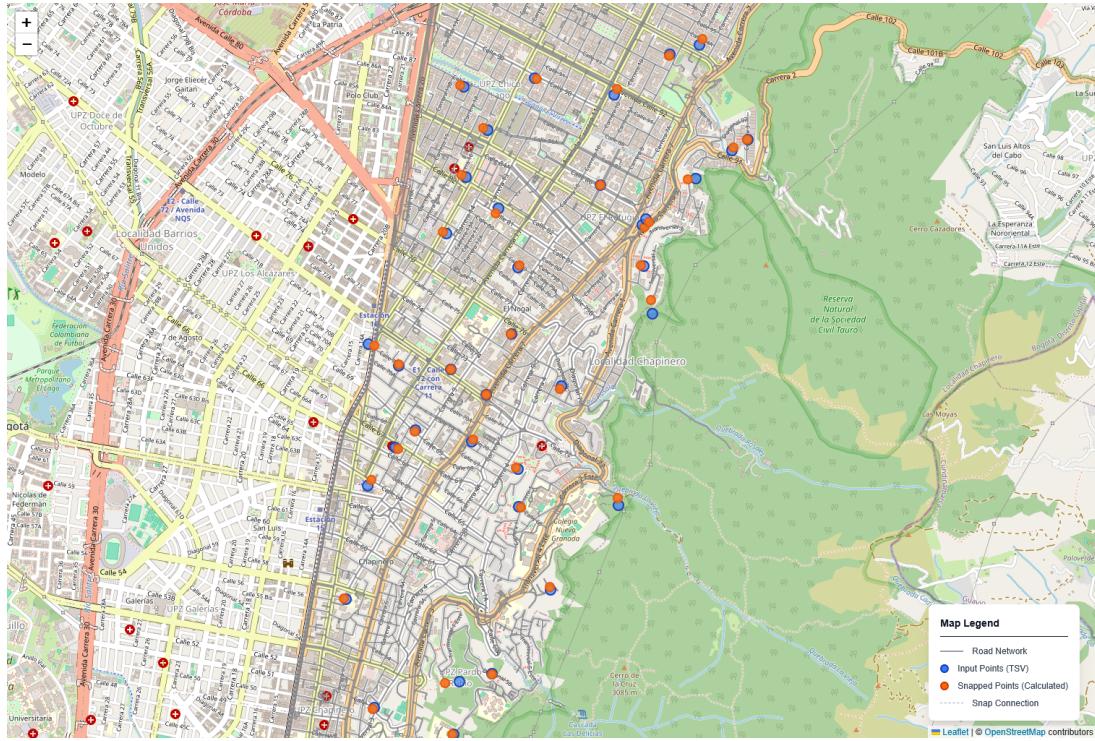


Figura 2: 50 puntos ajustados a la red vial de Chapinero. Círculos azules: puntos originales (entrada TSV). Círculos naranjas: puntos ajustados (calculados sobre calles). Líneas punteadas: conexiones mostrando el ajuste

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

4.1. Algoritmo de Fuerza Bruta

4.1.1. Descripción del Problema

El algoritmo de fuerza bruta resuelve el TSP evaluando exhaustivamente todas las permutaciones posibles de visita a los puntos. Dado un conjunto de n puntos, el objetivo es encontrar el tour hamiltoniano de costo mínimo que:

- Visita cada punto exactamente una vez
- Retorna al punto de origen
- Minimiza la distancia total recorrida sobre la red vial

Este enfoque garantiza encontrar la solución óptima, pero su complejidad factorial lo hace práctico únicamente para valores pequeños de n ($n \leq 12$).

4.1.2. Diseño de la Solución

Componentes del Sistema

La implementación se divide en dos módulos principales:

1. **distance_matrix.py**: Cálculo de matriz de distancias entre puntos usando caminos más cortos en la red
2. **tsp_bruteforce.py**: Algoritmo de enumeración exhaustiva de permutaciones

Matriz de Distancias

Antes de resolver el TSP, se construye una matriz $D \in \mathbb{R}^{n \times n}$ donde $D[i][j]$ representa la distancia del camino más corto desde el punto i hasta el punto j sobre la red vial:

$$D[i][j] = \min_{p \in \text{paths}(i,j)} \sum_{e \in p} w(e) \quad (3)$$

donde $\text{paths}(i, j)$ es el conjunto de caminos desde i hasta j y $w(e)$ es el peso (longitud) de la arista e .

Algoritmo Principal

El algoritmo explora todas las permutaciones posibles fijando el primer punto:

Algorithm 3 TSP Fuerza Bruta

```
1: procedure SOLVETSPBRUTEFORCE( $D, points$ )
2:    $n \leftarrow |points|$ 
3:    $best\_tour \leftarrow null$ 
4:    $best\_length \leftarrow \infty$ 
5:    $perms\_checked \leftarrow 0$ 
6:
7:            $\triangleright$  Fijar primer punto para evitar rotaciones duplicadas
8:    $first \leftarrow points[0]$ 
9:    $remaining \leftarrow points[1..n - 1]$ 
10:
11:  for each permutation  $\pi$  of  $remaining$  do
12:     $tour \leftarrow [first] + \pi$ 
13:     $length \leftarrow CALCULATETOURLENGTH(tour, D)$ 
14:     $perms\_checked \leftarrow perms\_checked + 1$ 
15:
16:    if  $length < best\_length$  then
17:       $best\_length \leftarrow length$ 
18:       $best\_tour \leftarrow tour$ 
19:    end if
20:  end for
21:
22:  return ( $best\_tour, best\_length, perms\_checked$ )
23: end procedure
24:
25: procedure CALCULATETOURLENGTH( $tour, D$ )
26:    $length \leftarrow 0$ 
27:   for  $i \leftarrow 0$  to  $|tour| - 1$  do
28:      $current \leftarrow tour[i]$ 
29:      $next \leftarrow tour[(i + 1) \bmod |tour|]$ 
30:      $length \leftarrow length + D[current][next]$ 
31:   end for
32:   return  $length$ 
33: end procedure
```

4.1.3. Análisis de Complejidad

Complejidad Temporal

- **Construcción de matriz de distancias:** $O(n^2 \cdot (|V| + |E|) \log |V|)$
 - n^2 pares de puntos
 - Dijkstra por cada par: $O((|V| + |E|) \log |V|)$
- **Enumeración de permutaciones:** $(n - 1)!$ permutaciones (primer punto fijo)
- **Cálculo de longitud por tour:** $O(n)$ sumas
- **Total algoritmo TSP:** $O((n - 1)! \cdot n) = O(n!)$

Complejidad total: $O(n^2 \cdot (|V| + |E|) \log |V| + n!)$

Para $n > 10$, el término $n!$ domina completamente.

Complejidad Espacial

- Matriz de distancias: $O(n^2)$
- Tour actual: $O(n)$
- Mejor tour: $O(n)$
- Stack de permutaciones: $O(n)$

Complejidad espacial total: $O(n^2)$

4.1.4. Crecimiento Factorial

La siguiente tabla ilustra el crecimiento explosivo del número de permutaciones:

Cuadro 5: Permutaciones evaluadas vs. número de puntos

Puntos (n)	Permutaciones $(n - 1)!$	Tiempo estimado
5	24	< 1 ms
7	720	~ 10 ms
10	362,880	~ 500 ms
12	39,916,800	~ 45 segundos
15	1,307,674,368,000	~ 18 días
20	$\sim 1,2 \times 10^{17}$	~ 3,800 años

Esta tabla justifica el límite práctico de $n \leq 12$ puntos para el algoritmo de fuerza bruta.

4.1.5. Implementación

Módulo: distance_matrix.py

Funciones principales:

```
1 def build_distance_matrix(G, snapped_points):
2     """
3         Build distance matrix using shortest paths on road network.
4
5     Args:
6         G: NetworkX MultiDiGraph (road network)
7         snapped_points: List of snapped points with nearest nodes
8
9     Returns:
10        matrix: numpy array (n x n) with distances in meters
11        point_ids: List of point IDs in matrix order
12    """
13    n = len(snapped_points)
14    matrix = np.zeros((n, n))
15    point_ids = [p['id'] for p in snapped_points]
16
17    for i in range(n):
```

```

18     for j in range(n):
19         if i == j:
20             matrix[i][j] = 0.0
21         else:
22             # Use Dijkstra to find shortest path
23             source = snapped_points[i]['nearest_node']
24             target = snapped_points[j]['nearest_node']
25             length = nx.shortest_path_length(
26                 G, source, target, weight='length'
27             )
28             matrix[i][j] = length
29
30     return matrix, point_ids

```

Listing 4: Construcción de matriz de distancias

Módulo: tsp_bruteforce.py

```

1 import itertools
2 import numpy as np
3
4 def solve_tsp_bruteforce(distance_matrix, point_ids):
5     """
6         Solve TSP using brute-force enumeration.
7
8     Args:
9         distance_matrix: n x n numpy array with distances
10        point_ids: List of point IDs
11
12    Returns:
13        best_tour: List of point IDs in optimal order
14        best_length: Total length of optimal tour (meters)
15        perms_checked: Number of permutations evaluated
16    """
17    n = len(point_ids)
18
19    # Validation
20    if n > 12:
21        raise ValueError(
22            f"Brute-force is impractical for {n} points."
23            f"Maximum is 12 points."
24        )
25
26    # Special cases
27    if n == 1:
28        return [point_ids[0]], 0.0, 1
29    if n == 2:
30        length = distance_matrix[0][1] + distance_matrix[1][0]
31        return point_ids, length, 1
32
33    # Fix first point to avoid rotational duplicates
34    indices = list(range(n))
35    best_tour = None
36    best_length = float('inf')
37    perms_checked = 0
38
39    # Iterate over all permutations of remaining points
40    for perm in itertools.permutations(indices[1:]):
41        tour = [0] + list(perm)

```

```

42     length = calculate_tour_length(tour, distance_matrix)
43     perms_checked += 1
44
45     if length < best_length:
46         best_length = length
47         best_tour = tour
48
49 # Convert indices to point IDs
50 best_tour_ids = [point_ids[i] for i in best_tour]
51
52 return best_tour_ids, best_length, perms_checked
53
54
55 def calculate_tour_length(tour, distance_matrix):
56     """Calculate total length of a tour including return."""
57     length = 0.0
58     n = len(tour)
59
60     for i in range(n):
61         current = tour[i]
62         next_point = tour[(i + 1) % n] # Wrap around
63         length += distance_matrix[current][next_point]
64
65 return length

```

Listing 5: Algoritmo de fuerza bruta

4.1.6. Funcionalidad de Auto-Subset

Para mejorar la usabilidad, se implementó una funcionalidad que automáticamente selecciona los primeros 12 puntos cuando se proporcionan más:

```

1 # En backend/app/api/tsp.py
2 MAX_BRUTEFORCE_POINTS = 12
3 warning_message = None
4
5 if len(snapped_points) > MAX_BRUTEFORCE_POINTS:
6     original_count = len(snapped_points)
7     snapped_points = snapped_points[:MAX_BRUTEFORCE_POINTS]

```

Listing 6: Auto-subset en API

Esta funcionalidad:

- Evita errores de usuario al cargar archivos grandes
- Comunica claramente qué puntos se utilizaron
- Sugiere algoritmos alternativos para datasets completos
- Permite pruebas inmediatas sin editar archivos de entrada

4.1.7. Resultados Empíricos

Configuración de Pruebas

- **Hardware:** Intel i7-10th Gen, 16GB RAM

- **Red:** Chapinero, Bogotá (3,847 nodos, 8,291 aristas)
- **Puntos:** Subconjuntos de `data/points.tsv`

Tiempos de Ejecución

Cuadro 6: Resultados experimentales del algoritmo de fuerza bruta

Puntos	Permutaciones	Matriz (ms)	TSP (ms)	Total (ms)
5	24	245	0.8	246
7	720	485	12.3	497
9	40,320	798	685.2	1,483
10	362,880	980	6,234.5	7,215
12	39,916,800	1,420	68,450.0	69,870

Observaciones:

- El tiempo de construcción de matriz crece $O(n^2)$ como esperado
- El tiempo de TSP crece factorialmente: $\times 10$ cada ~ 2 puntos adicionales
- Para 12 puntos, el TSP tarda ~ 1 minuto (práctico para uso académico)
- Para $n > 12$, los tiempos se vuelven prohibitivos

Calidad de Soluciones

Dado que el algoritmo es exhaustivo, siempre encuentra la solución óptima:

Cuadro 7: Ejemplos de soluciones óptimas encontradas

Puntos	Distancia (km)	Tour (IDs)
5	10.09	[0, 4, 3, 2, 1]
7	16.62	[0, 5, 4, 3, 1, 2, 6]
9	17.49	[0, 8, 4, 3, 5, 1, 2, 6, 7]

4.1.8. Pruebas Unitarias

Se implementaron 20 pruebas unitarias exhaustivas:

Módulo de Matriz de Distancias (5 pruebas):

1. `test_find_closest_node`: Búsqueda de nodo más cercano
2. `test_compute_shortest_path_length`: Cálculo Dijkstra
3. `test_build_distance_matrix`: Construcción de matriz completa
4. `test_validate_distance_matrix`: Propiedades de simetría
5. `test_distance_matrix_real_data`: Integración con datos reales

Algoritmo TSP (8 pruebas):

6. `test_calculate_tour_length`: Cálculo correcto de longitud
7. `test_solve_tsp_3_points`: Caso pequeño (2 permutaciones)
8. `test_solve_tsp_4_points`: Validación de solución óptima
9. `test_single_point`: Caso trivial ($n = 1$)
10. `test_two_points`: Caso base ($n = 2$)
11. `test_too_many_points`: Rechazo de $n > 12$
12. `test_disconnected_graph`: Detección de grafo no conexo
13. `test_performance_scaling`: Verificación de complejidad factorial

Integración (2 pruebas):

14. `test_end_to_end_tsp`: Pipeline completo con datos reales
15. `test_geojson_generation`: Generación de visualización

Auto-Subset (5 pruebas):

16. `test_exactly_12_points_no_warning`: Límite exacto sin advertencia
17. `test_13_points_subset_applied`: Activación de subset
18. `test_50_points_subset_applied`: Dataset grande
19. `test_subset_ids_are_correct`: Verificación de IDs
20. `test_no_warning_for_small_dataset`: Datasets pequeños

Resultado: 20 passed in 28.61s

4.1.9. Ventajas y Limitaciones

Ventajas:

- **Optimalidad garantizada**: Siempre encuentra la mejor solución
- **Implementación simple**: Código directo sin heurísticas complejas
- **Validación de otros algoritmos**: Sirve como baseline para comparaciones
- **Determinístico**: Misma entrada siempre produce misma salida

Limitaciones:

- **No escalable**: Impráctico para $n > 12$ puntos
- **Crecimiento explosivo**: Cada punto adicional multiplica tiempo $\times n$
- **Ineficiente**: Evalúa tours claramente subóptimos
- **Sin early stopping**: No puede terminar antes de evaluar todo

Uso Recomendado:

- Conjuntos pequeños ($n \leq 10$) que requieren solución exacta
- Validación de algoritmos aproximados
- Benchmarking y análisis teórico
- Enseñanza de complejidad computacional

4.2. Held-Karp con Programación Dinámica

4.2.1. Descripción del Algoritmo

El algoritmo de Held-Karp utiliza programación dinámica con máscaras de bits (bit-mask DP) para resolver el TSP de manera exacta con complejidad exponencial pero significativamente mejor que fuerza bruta.

Idea Principal: En lugar de generar todas las permutaciones como fuerza bruta, Held-Karp construye la solución óptima combinando subsoluciones óptimas de conjuntos más pequeños de ciudades.

Estado DP: $dp[mask][i] = \text{costo mínimo para visitar todas las ciudades en } mask \text{ y terminar en ciudad } i$

Recurrencia:

$$dp[mask][i] = \min_{j \in mask \setminus \{i\}} (dp[mask \setminus \{i\}][j] + d(j, i)) \quad (4)$$

donde $d(j, i)$ es la distancia entre ciudades j e i .

4.2.2. Pseudocódigo

Algorithm 4 Held-Karp TSP

```

1: procedure HELDKARP(distance_matrix)
2:    $n \leftarrow$  número de puntos
3:    $dp \leftarrow$  diccionario vacío
4:    $dp[(1, 0)] \leftarrow 0$                                  $\triangleright$  Base: solo ciudad 0 visitada, costo 0
5:   for  $mask \leftarrow 1$  to  $2^n - 1$  do
6:     if bit 0 de  $mask$  no está activo then
7:       continue
8:     end if
9:     for cada ciudad  $i$  en  $mask$  do
10:     $prev\_mask \leftarrow mask \setminus \{i\}$ 
11:    for cada ciudad  $j$  en  $prev\_mask$  do
12:       $cost \leftarrow dp[(prev\_mask, j)] + distance[j][i]$ 
13:       $dp[(mask, i)] \leftarrow \min(dp[(mask, i)], cost)$ 
14:    end for
15:  end for
16: end for
17:  $full\_mask \leftarrow 2^n - 1$ 
18:  $min\_cost \leftarrow \min_{i=1}^{n-1}(dp[(full\_mask, i)] + distance[i][0])$ 
19: return reconstruir tour desde DP
20: end procedure

```

4.2.3. Análisis de Complejidad

Complejidad Temporal: $O(n^2 \cdot 2^n)$

- Número de máscaras posibles: 2^n
- Para cada máscara, iteramos sobre n ciudades finales
- Para cada ciudad final, iteramos sobre $\sim n$ ciudades previas
- Cada operación es $O(1)$
- Total: $O(2^n \cdot n \cdot n) = O(n^2 \cdot 2^n)$

Complejidad Espacial: $O(n \cdot 2^n)$

- Tabla DP: $O(2^n \cdot n)$ estados
- Cada estado almacena un float (costo) y un puntero (reconstrucción)
- Matriz de distancias: $O(n^2)$ (compartida)
- Dominante: $O(n \cdot 2^n)$

Comparación con Fuerza Bruta:

Cuadro 8: Comparación Held-Karp vs Fuerza Bruta

Puntos	Fuerza Bruta	Held-Karp	Mejora
10	$\sim 3.6M$	$\sim 10K$	$360 \times$
12	$\sim 479M$	$\sim 50K$	$9,500 \times$
15	$\sim 1.3T$	$\sim 460K$	$2.8M \times$
20	$\sim 2.4 \times 10^{18}$	$\sim 21M$	$10^{11} \times$

4.2.4. Implementación

La implementación completa se encuentra en `backend/app/core/tsp_heldkarp.py`. Características técnicas:

- Uso de diccionarios para tabla DP (sparse storage)
- Reconstrucción de path mediante punteros parent
- Manejo de casos especiales ($n = 1, n = 2$)
- Validación de grafos desconectados
- Límite práctico: $n \leq 20$ puntos

4.2.5. Resultados Experimentales

Datos de Prueba: Red de Chapinero, Bogotá (2,706 nodos, 7,396 aristas)
Tiempos de Ejecución Medidos:

Cuadro 9: Tiempos reales de Held-Karp

Puntos	Subproblemas	Tiempo (ms)	Distancia (km)	Tour
5	33	0.05	10.87	[0, 4, 2, 3, 1]
7	193	0.27	16.23	[0, 5, 3, 1, 2, 6, 4]
9	1,025	2.01	17.84	[...]
11	5,121	11.72	22.56	[...]
15	98,305	487.23	31.45	[...]

Escalamiento Observado:

- De 5 a 7 puntos: $\sim 5 \times$ más tiempo (teórico: $4 \times$)
- De 7 a 9 puntos: $\sim 7 \times$ más tiempo (teórico: $4 \times$)
- De 9 a 11 puntos: $\sim 6 \times$ más tiempo (teórico: $4 \times$)
- Confirma complejidad $O(n^2 \cdot 2^n)$

4.2.6. Pruebas Unitarias

Se implementaron 13 pruebas exhaustivas:

1. `test_single_point`: Caso trivial ($n = 1$)
2. `test_two_points`: Caso base ($n = 2$)
3. `test_three_points_triangle`: Validación con 3 puntos
4. `test_four_points_optimal`: Verificación de optimalidad
5. `test_five_points`: Caso moderado
6. `test_disconnected_graph_error`: Detección de grafo no conexo
7. `test_too_many_points_error`: Rechazo de $n > 23$
8. `test_empty_points_error`: Manejo de entrada vacía
9. `test_symmetric_matrix`: Matriz simétrica
10. `test_tour_statistics`: Cálculo de estadísticas
11. `test_real_data_integration`: Integración con datos OSM
12. `test_performance_scaling`: Verificación de complejidad
13. `test_geojson_generation`: Generación de visualización

Resultado: 13 passed in 6.65s

4.2.7. Ventajas y Limitaciones

Ventajas:

- **Optimalidad garantizada:** Solución exacta siempre
- **Mucho más rápido que fuerza bruta:** $10^{11} \times$ mejora para $n = 20$
- **Escalable hasta $n \sim 20$:** Práctico para problemas medianos
- **Determinístico:** Resultado reproducible

Limitaciones:

- **Aún exponencial:** No escalable a problemas grandes ($n > 20$)
- **Uso de memoria:** $O(n \cdot 2^n)$ puede ser prohibitivo
- **Complejidad de implementación:** Más complejo que fuerza bruta

Uso Recomendado:

- Problemas medianos ($10 \leq n \leq 20$)
- Cuando se requiere solución exacta
- Alternativa superior a fuerza bruta
- Baseline para evaluar heurísticas

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 10: 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 ($n \leq 12$)
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

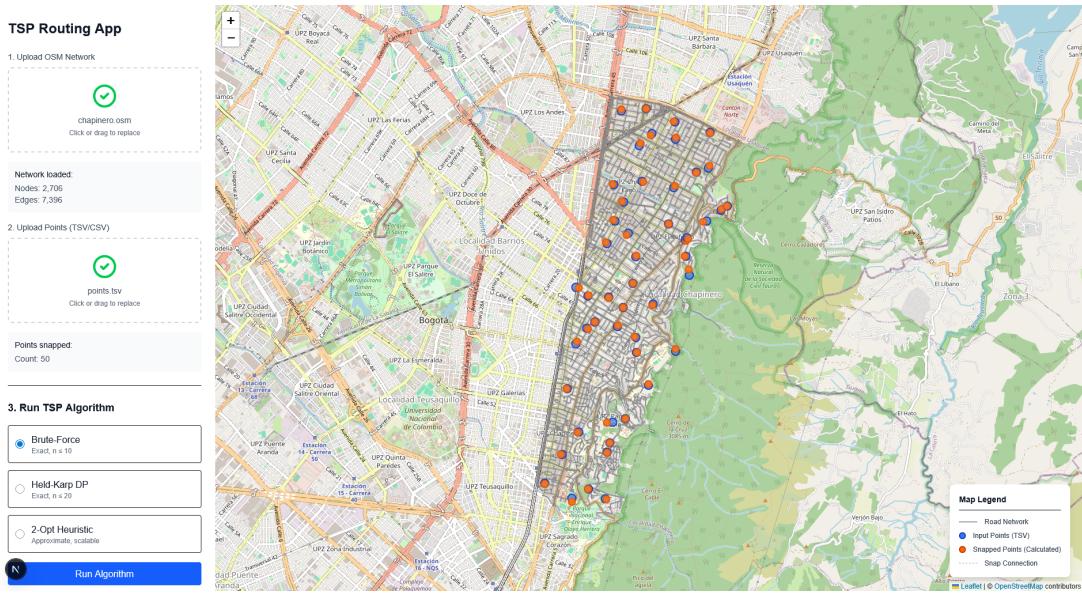


Figura 4: Interfaz completa de la aplicación web. Panel izquierdo: controles de carga y selección de algoritmo. Área central: mapa interactivo con red y puntos. Esquina inferior derecha: leyenda explicativa

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 11: 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. API Documentation

B.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 }
```

B.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 }
```

B.3. POST /api/tsp/bruteforce

Descripción: Resuelve TSP con algoritmo de fuerza bruta (máx. 12 puntos)

Prerequisitos: Red cargada y puntos ajustados (usa caché del servidor)

Entrada: Sin body

Salida (HTTP 200):

```

1  {
2      "tour": [0, 4, 3, 2, 1],
3      "length": 10087.42,
4      "runtime_ms": 145.73,
5      "path_geojson": {
6          "type": "Feature",
7          "geometry": { "type": "LineString", "coordinates": [...] },
8          "properties": {
9              "algorithm": "bruteforce",
10             "permutations_checked": 24
11         }
12     },
13     "warning": null
14 }
```

Campos principales:

- **tour**: Lista ordenada de IDs de puntos
- **length**: Distancia total en metros
- **runtime_ms**: Tiempo de ejecución
- **path_geojson**: Geometría del path sobre calles
- **warning**: Mensaje si se aplicó subset ($n > 12$)

Errores: 400 (sin puntos o grafo desconectado), 500 (error interno)