

# FloraRoute - Sistema de Optimización de Rutas de Entrega

---

## Descripción General

**FloraRoute** es una aplicación web desarrollada en Python que optimiza rutas de entrega de flores en Lima Metropolitana. El sistema utiliza algoritmos avanzados de teoría de grafos y programación dinámica para calcular la ruta más eficiente entre múltiples puntos de entrega, considerando factores como tráfico vehicular, disponibilidad de inventario y restricciones geográficas.

## Objetivo del Proyecto

Resolver el problema del **Vendedor Viajero (TSP - Traveling Salesman Problem)** en un contexto real de logística urbana, minimizando la distancia total recorrida y el tiempo de entrega mientras se gestionan múltiples restricciones operativas.

---

## Algoritmos Implementados

El proyecto implementa algoritmos divididos en tres categorías principales:

### 1 Algoritmos de Grafos (6 Algoritmos)

#### 1.1 Dijkstra - Camino Más Corto

- **Propósito:** Calcular la ruta más corta entre dos nodos del grafo de calles de Lima
- **Uso en el Sistema:**
  - Encontrar distancias entre cada par de puntos (origen y destinos)
  - Generar caminos completos nodo por nodo para visualización
  - Precálculo de matriz de distancias para TSP
- **Complejidad:**  $O((V + E) \log V)$  con heap binario
- **Implementación:** [src/controllers/calculador\\_rutas.py](#) - método `dijkstra()`
- **Factor de Tráfico:** Los pesos de las aristas se multiplican por un factor de tráfico (1.0 - 2.5x) según la hora del día

```
# Ejemplo de uso interno:  
distancia, camino = self.dijkstra(nodo_origen, nodo_destino)  
# Aplica factor de tráfico: peso_real = peso_base * factor_tráfico
```

#### 1.2 Held-Karp (TSP Exacto) - Programación Dinámica en Grafos

- **Propósito:** Resolver el problema del Vendedor Viajero de forma óptima
- **Uso en el Sistema:**
  - Determinar el orden óptimo de visita a los destinos

- Minimizar la distancia total del recorrido
- Soportar ciclos cerrados (retorno al origen) o abiertos
- **Complejidad:**  $O(n^2 \times 2^n)$  donde  $n$  = número de destinos
- **Factibilidad:** Eficiente para  $n \leq 20$  destinos
- **Implementación:** [src/controllers/calculador\\_rutas.py](#) - método `held_karp()`
- **Técnica:** Programación dinámica con bitmask para representar subconjuntos

```
# Estado DP: dp[subconjunto_visitado][ultimo_nodo] = (distancia_minima,
nodo_previo)
# Reconstruye la secuencia óptima al final
```

### 1.3 Precálculo de Matriz de Distancias

- **Propósito:** Optimizar el algoritmo Held-Karp evitando cálculos redundantes
- **Uso en el Sistema:**
  - Ejecutar Dijkstra una sola vez para cada par de nodos de interés
  - Almacenar resultados en diccionario para consulta  $O(1)$
- **Implementación:** [src/controllers/calculador\\_rutas.py](#) - método `precalcular_matriz_distancias()`

```
# Matriz: {(nodo_i, nodo_j): distancia_minima}
# Solo calcula distancias entre origen y destinos, no todo el grafo
```

### 1.4 Búsqueda de Nodo Cercano (Espacial)

- **Propósito:** Mapear coordenadas GPS a nodos del grafo vial
- **Uso en el Sistema:**
  - Convertir latitud/longitud de destinos a nodos del grafo
  - Validar que los puntos de entrega estén accesibles
- **Método:** Distancia Euclíadiana en coordenadas geográficas
- **Implementación:** [src/utils/cargador\\_datos.py](#) - función `encontrar_nodo_cercano()`

```
# Distancia:  $\sqrt{((lat1 - lat2)^2 + (lon1 - lon2)^2)}$ 
# Retorna el nodo con menor distancia
```

### 1.5 Construcción de Grafo desde CSV

- **Propósito:** Cargar la red vial de Lima desde archivos de datos
- **Uso en el Sistema:**
  - Crear estructura de grafo no dirigido desde `lima_nodes.csv` y `lima_edges.csv`
  - Almacenar como diccionario de adyacencia: `{nodo: {vecino: distancia}}`
- **Implementación:** [src/utils/cargador\\_datos.py](#) - función `cargar_grafo_lima()`

```
# Estructura: grafo[nodo1][nodo2] = distancia_metros  
# No dirigido: grafo[A][B] = grafo[B][A]
```

## 1.6 Cálculo de Camino Completo

- **Propósito:** Generar la secuencia completa de nodos intermedios para visualización
- **Uso en el Sistema:**
  - Dibujar polylines en el mapa siguiendo las calles reales
  - Mostrar la ruta exacta que debe seguir el conductor
- **Implementación:** [src/controllers/calculador\\_rutas.py](#) - método [calcular\\_camino\\_completo\(\)](#)

```
# Input: [origen, destino1, destino2, origen]  
# Output: [origen, nodo_intermedio_1, ..., destino1, nodo_intermedio_2, ..., destino2, ...]
```

---

## 2 Algoritmos de Programación Dinámica

### 2.1 Held-Karp (TSP) - Programación Dinámica General ★

- **Propósito:** Ya descrito en sección de Grafos
- **Naturaleza Dual:** Es tanto un algoritmo de grafos como de programación dinámica
- **Técnica DP:**
  - Subestructura óptima: La mejor ruta a un subconjunto incluye la mejor ruta a subconjuntos menores
  - Memorización: Almacena soluciones parciales para evitar recálculos
  - Reconstrucción de solución mediante backtracking

---

## 3 Algoritmos Adicionales (1 Algoritmo)

### 3.1 DFS/BFS - Exploración y Validación

- **Propósito:**
  - Validar conectividad del grafo
  - Verificar que todos los destinos sean alcanzables desde el origen
- **Uso en el Sistema:**
  - Validación de integridad del grafo al inicio
  - Detección de componentes desconectadas
- **Estado Actual:** Implementado como validación básica
- **Implementación:** Validaciones en [src/controllers/validador.py](#)

## Arquitectura del Sistema



## Instalación y Configuración

### Requisitos Previos

- **Python:** 3.8 o superior
- **Sistema Operativo:** Windows, Linux o macOS
- **Memoria RAM:** 4 GB mínimo (8 GB recomendado para grafos grandes)

### Instalación

#### 1. Clonar el repositorio

```
git clone https://github.com/GilbertHuarcaya/FloraRouteComplejidad  
cd FloraRouteComplejidad
```

## 2. Crear entorno virtual

```
# Windows  
python -m venv venv  
venv\Scripts\activate  
  
# Linux/Mac  
python3 -m venv venv  
source venv/bin/activate
```

## 3. Instalar dependencias

```
pip install -r requirements.txt
```

### Dependencias Principales

#### Librerías Core del Proyecto

Librería	Versión	Propósito en FloraRoute
<b>streamlit</b>	1.29.0	Framework de interfaz web. Crea la aplicación interactiva sin necesidad de HTML/CSS/JS
<b>folium</b>	0.14.0	Generación de mapas interactivos. Visualiza el grafo de calles y las rutas calculadas
<b>streamlit-folium</b>	0.16.0	Integración de Folium con Streamlit. Permite interacción con el mapa (clicks, zoom)
<b>pandas</b>	2.3.3	Manipulación de datos tabulares. Carga CSVs de nodos, aristas y viveros

## Datos del Sistema

### Archivos de Datos

El sistema utiliza 4 archivos CSV principales:

1. **docs/lima\_nodes.csv** (Nodos del grafo)

- Columnas: **node\_id, lat, lon**
- Contiene: ~10,000 nodos de intersecciones viales de Lima

## 2. `docs/lima_edges.csv` (Aristas del grafo)

- Columnas: `node1`, `node2`, `distance`
- Contiene: ~20,000 conexiones entre nodos con distancias en metros

## 3. `src/data/viveros.csv` (Viveros y stock)

- Columnas: `vivero_id`, `nombre`, `nodo_id`, `lat`, `lon`, `stock_rosas`, `stock_claveles`, etc.
- Contiene: Información de viveros con inventario de flores

## 4. Factor de Tráfico (`cargador_datos.py`)

- Factores por hora del día (1.0x - 2.5x)
- Simula congestión vehicular en Lima

---

# 🎮 Uso de la Aplicación

## Ejecución

```
streamlit run src/views/app.py
```

La aplicación se abrirá en <http://localhost:8501>

## Flujo de Uso

### Paso 1: Seleccionar Vivero de Origen

1. Usar el selector en la barra lateral
2. Confirmar selección
3. El vivero aparece como marcador verde en el mapa

### Paso 2: Agregar Destinos (1-20)

1. Hacer clic en el mapa para seleccionar ubicación
2. Las coordenadas se cargan automáticamente
3. Especificar cantidades de flores requeridas
4. Agregar destino (aparece como marcador rojo)
5. Repetir para más destinos

### Paso 3: Calcular Ruta Óptima

1. Elegir si retornar al origen (ciclo cerrado)
2. Presionar "Calcular Ruta"
3. El sistema ejecuta Held-Karp + Dijkstra

### Paso 4: Visualizar Resultados

1. **Mapa:** Polyline azul muestra la ruta siguiendo calles reales
  2. **Métricas:** Distancia total, tiempo estimado, número de paradas
  3. **Tabla de Visitas:** Orden de visitas con distancias y tiempos por segmento
- 

## 📝 Factores Adicionales Implementados

### 1. Factor de Tráfico Dinámico

- **Implementación:** Multiplicador de pesos de aristas según hora del día
- **Rango:** 1.0x (madrugada) a 2.5x (horas pico: 8-9am, 6-7pm)
- **Impacto:** Las rutas calculadas varían según la hora de consulta
- **Código:** `src/utils/cargador_datos.py` - función `obtener_factor_trafico_actual()`

### 2. Validación de Stock de Flores

- **Implementación:** Verificación de inventario antes de agregar destinos
- **Restricción:** No se pueden agregar destinos si el vivero no tiene stock suficiente
- **Código:** `src/controllers/validador.py` - método `validar_stock_flores()`

### 3. Gestión de Inventario

- **Implementación:** Sistema de reducción de stock virtual al crear pedidos
- **Modelos:** `Inventario` y `Vivero` en `src/models/vivero.py`
- **Funcionalidad:**
  - Verificación con `tiene_stock()`
  - Reducción con `reducir_stock()`

### 4. Validaciones Geográficas

- **Implementación:** Verificación de coordenadas dentro de límites de Lima
- **Rangos Válidos:**
  - Latitud: -12.3 a -11.7
  - Longitud: -77.2 a -76.8
- **Código:** `src/controllers/validador.py` - método `validar_rango_geografico_lima()`

### 5. Limitación de Destinos

- **Implementación:** Restricción de 1-20 destinos por ruta
- **Justificación:**
  - Held-Karp factible hasta n=20 (complejidad exponencial)
  - Realismo operativo (capacidad de vehículo)
- **Código:** `src/controllers/validador.py` - método `validar_cantidad_destinos()`

### 6. Mapeo Automático GPS → Nodo del Grafo

- **Implementación:** Búsqueda del nodo más cercano por distancia
- **Propósito:** Convertir coordenadas GPS arbitrarias a nodos válidos del grafo
- **Código:** `src/utils/cargador_datos.py` - función `encontrar_nodo_cercano()`

## 7. Métricas de Rendimiento

- **Implementación:** Medición del tiempo de cómputo de algoritmos
  - **Datos Recolectados:**
    - Tiempo de ejecución de Held-Karp
    - Tiempo de precálculo de matriz de distancias
    - Tiempo de cálculo de camino completo
  - **Código:** [src/controllers/gestor\\_rutas.py](#) - método `calcular_ruta_optima()`
- 

## Complejidad Computacional

Algoritmo	Complejidad Temporal	Complejidad Espacial	Límite Práctico
Dijkstra	$O((V + E) \log V)$	$O(V)$	$V \sim 10,000$ nodos
Held-Karp (TSP)	$O(n^2 \times 2^n)$	$O(n \times 2^n)$	$n \leq 20$ destinos
Precálculo Matriz	$O(n^2 \times (V + E) \log V)$	$O(n^2)$	$n = 21$ (origen + 20 destinos)
Búsqueda Nodo Cercano	$O(V)$	$O(1)$	$V \sim 10,000$ nodos
Camino Completo	$O(n \times (V + E) \log V)$	$O(V \times n)$	$n \leq 20$ segmentos

**Tiempo Total Estimado:** 0.5 - 3 segundos para 20 destinos en hardware moderno

---

## Estructura de Carpetas

```
TF-version-limpia/
├── src/
│   ├── controllers/          # Lógica de negocio y algoritmos
│   │   ├── calculador_rutas.py    # Dijkstra, Held-Karp, TSP
│   │   ├── gestor_rutas.py      # Orquestador principal
│   │   └── validador.py        # Validaciones de negocio
│   ├── models/               # Modelos de datos
│   │   ├── pedido.py           # Pedido, Destino
│   │   ├── ruta.py             # Ruta, Segmento, Métricas
│   │   └── vivero.py           # Vivero, Inventario
│   ├── utils/                # Utilidades
│   │   ├── cargador_datos.py    # Carga CSV, factor tráfico
│   │   └── exportador.py       # Exportación de resultados
│   ├── views/                # Interfaz Streamlit
│   │   └── app.py              # Aplicación principal
│   └── pruebas/              # Tests unitarios
└── docs/
    ├── lima_nodes.csv         # Nodos del grafo de Lima
    └── lima_edges.csv         # Aristas del grafo
```

└ *.csv	# Otros datos
README.md	# Este archivo

---

## 👥 Autores

- **Equipo FloraRoute**
  - **Curso:** Complejidad Algorítmica
  - **Ciclo:** 5
- 

## 📄 Licencia

Este proyecto es de uso académico.

---

## 🔗 Referencias

### Algoritmos Implementados

- Dijkstra's Algorithm: E. W. Dijkstra (1959)
- Held-Karp Algorithm: Held & Karp (1962) - Dynamic Programming solution for TSP
- Traveling Salesman Problem: Theoretical foundation

### Datos Geográficos

- OpenStreetMap (OSM): Datos de red vial de Lima
- Folium: Visualización de mapas interactivos

### Frameworks

- Streamlit: Desarrollo rápido de aplicaciones web
- Pandas: Análisis de datos