

**Carrera:**

Ingeniería en sistemas de la información cuarto semestre

**Materia:**

Diseño de sistemas

**Docente:**

Verónica Pailiacho

**Estudiante:**

Jeremy Jácome

**Tema:**

Diagrama de casos de uso

**Fecha:**

30 de junio de 2025

**🧾 INFORME TÉCNICO - SISTEMA DE RUTAS MEDIANTE GRAFOS**

**1. Introducción**

Este proyecto consiste en un sistema web desarrollado con Flask que implementa un **algoritmo de grafos** para encontrar rutas óptimas entre ciudades de Ecuador. El sistema está diseñado para ser utilizado tanto por usuarios regulares como por administradores, permitiendo búsquedas de rutas eficientes y gestión completa de las entidades involucradas.

**2. Objetivos**

* Implementar una red de rutas mediante grafos usando NetworkX.
* Aplicar el algoritmo de **Dijkstra** para encontrar la ruta más económica entre ciudades.
* Visualizar el grafo resultante con matplotlib.
* Brindar autenticación segura y gestión administrativa completa del sistema.

**3. Arquitectura del Sistema**

El proyecto sigue una arquitectura **MVC** (Modelo-Vista-Controlador) modular con separación clara de responsabilidades:

**🧩 Componentes Principales:**

* **Modelos (models/)**: Representan la base de datos (usuarios, provincias, ciudades, rutas).
* **Vistas (templates/)**: Interfaz de usuario HTML con AdminLTE.
* **Controladores (routes/)**: Manejan la lógica HTTP.
* **Servicios (services/)**: Lógica de negocio.
* **Controladores de grafos (controllers/)**: Implementan los algoritmos de grafos.
* **Repositorio (repositories/)**: Acceso a datos desde SQLAlchemy.
* **Configuración (.env, config.py)**: Variables del entorno y configuración de Flask.
* **Entorno Virtual (.venv/)**: Contiene dependencias aisladas para ejecución del sistema.

**4. Descripción del Funcionamiento del Grafo**

**🧠 Lógica del Grafo**

* **Nodos**: Cada ciudad registrada en la base de datos es un nodo.
* **Aristas**: Las rutas entre ciudades son aristas con un peso que representa el costo/distancia.
* **Construcción**: construir\_grafo() transforma los datos relacionales en un grafo de NetworkX.
* **Visualización**: grafo\_a\_imagen() genera una imagen del grafo usando matplotlib con un layout de tipo spring.

[[CAPTURA: Grafo generado]]

**🚀 Algoritmo de Ruta Óptima**

* Se utiliza una **modificación del algoritmo de Dijkstra** implementada en camino\_optimo\_con\_costera().
* Calcula el camino más económico entre dos ciudades.
* También identifica si en la ruta aparecen ciudades costeras (es\_costera=True).

[[CAPTURA: Resultado de ruta óptima con ciudades costeras]]

**5. Flujo del Sistema**

1. **Inicio de Sesión**: El usuario se autentica mediante Flask-Login.
2. **Dashboard**: Accede a /home o al panel de administración si es admin.
3. **Búsqueda de Ruta**:
   * Selecciona ciudad de origen y destino.
   * El sistema construye el grafo.
   * Se ejecuta Dijkstra y se visualiza la ruta.

[[CAPTURA: Formulario de búsqueda de rutas]]

1. **Visualización**: Se muestra la ruta y el costo, y el grafo generado.

**6. Gestión Administrativa**

El administrador accede a un panel completo para realizar operaciones CRUD sobre:

* Usuarios
* Provincias
* Ciudades (incluye atributo costero)
* Rutas (define peso entre ciudades)

[[CAPTURA: Panel de administración]]

**7. Base de Datos**

**Entidades:**

* **usuarios**: Control de acceso y roles.
* **provincias**: División territorial del Ecuador.
* **ciudades**: Nodo del grafo, con provincia y atributo es\_costera.
* **rutas**: Aristas del grafo con campo peso (km).

[[CAPTURA: Diagrama Entidad-Relación]]

**Relaciones:**

* Una provincia tiene muchas ciudades.
* Una ruta conecta dos ciudades.

**8. Tecnologías Utilizadas**

* **Backend**: Flask, SQLAlchemy, Flask-Login, bcrypt
* **Algoritmos**: NetworkX (grafos), matplotlib (visualización)
* **Frontend**: AdminLTE, Bootstrap, jQuery
* **Base de Datos**: MySQL (puerto 3307, charset UTF-8)
* **Entorno**: .venv/ con requirements.txt para instalación reproducible

**9. Seguridad**

* **Autenticación**: Controlada por Flask-Login.
* **Autorización**: Decoradores para rutas de administrador.
* **Encriptación**: Contraseñas protegidas con bcrypt.

**10. Ejecución del Proyecto**

**Requisitos:**

bash

CopyEdit

# Activar entorno virtual

.venv\Scripts\activate

# Instalar dependencias

pip install -r requirements.txt

# Inicializar base de datos

python init\_db.py

# Ejecutar sistema

python app.py

* Acceso: http://localhost:4000
* Admin: usuario: admin, contraseña: admin

**11. Conclusión**

Este proyecto implementa de forma efectiva un sistema de rutas inteligente usando estructuras de grafos. Se logra una separación limpia de capas, seguridad robusta y una experiencia de usuario intuitiva. El uso de algoritmos avanzados como Dijkstra y la generación visual del grafo ofrecen un valor académico y técnico significativo.

**PEQUEÑAS MEJORAS (opcionales):**

**En la sección 4 "Funcionamiento del Grafo":**

Agregar una línea sobre complejidad:

"El algoritmo de Dijkstra tiene una complejidad temporal de O(V²) donde V es el número de vértices (ciudades)."

**En la sección 8 "Tecnologías":**

Especificar por qué elegiste cada tecnología:

"NetworkX por su optimización en algoritmos de grafos, AdminLTE por su interfaz profesional preconfigurada."

**En la sección 10 "Ejecución":**

Mencionar requisitos previos:

"Requisitos previos: Python 3.8+, MySQL/Laragon ejecutándose en puerto 3307."

**📸 CAPTURAS RECOMENDADAS:**

1. **Grafo generado** con las 6 ciudades conectadas
2. **Resultado de búsqueda** mostrando ruta Quito→Loja con costo
3. **Panel de administración** con las tablas CRUD
4. **Formulario de búsqueda** con dropdowns de ciudades
5. **Dashboard principal** después del login
6. **Diagrama ER** de la base de datos (opcional pero impresionante)

**Sistema de Rutas mediante grafos**

Voy a limpiar todos los archivos innecesarios y luego analizar cada archivo del proyecto para tu documentación técnica. Ahora voy a analizar cada archivo para tu documentación técnica:

## 📋 \*\*ANÁLISIS COMPLETO DE ARCHIVOS PARA DOCUMENTACIÓN TÉCNICA\*\*

### \*\*🎯 ARCHIVOS PRINCIPALES (Raíz del Proyecto)\*\*

#### \*\*1. app.py - ARCHIVO PRINCIPAL DE LA APLICACIÓN\*\*

- \*\*Qué hace\*\*: Punto de entrada principal del sistema Flask

- \*\*Para qué\*\*: Configura y ejecuta toda la aplicación web

- \*\*Por qué\*\*: Centraliza la configuración de Flask, base de datos, autenticación y rutas

- \*\*Funciones clave\*\*:

- `create\_app()`: Factory pattern para crear la aplicación

- Configuración de Flask-Login para autenticación

- Inicialización de SQLAlchemy y bcrypt

- Registro de blueprints (rutas modulares)

- Manejo de errores 404/500

- Auto-creación de tablas de base de datos

- Verificación del estado del sistema al inicio

#### \*\*2. requirements.txt - DEPENDENCIAS DEL PROYECTO\*\*

- \*\*Qué hace\*\*: Lista todas las librerías Python necesarias

- \*\*Para qué\*\*: Gestión de dependencias del proyecto

- \*\*Por qué\*\*: Permite instalación automática con `pip install -r requirements.txt`

- \*\*Dependencias clave\*\*: Flask, SQLAlchemy, NetworkX, matplotlib, bcrypt

#### \*\*3. .env - CONFIGURACIÓN DEL ENTORNO\*\*

- \*\*Qué hace\*\*: Variables de configuración sensibles

- \*\*Para qué\*\*: Configuración de base de datos y secretos

- \*\*Por qué\*\*: Separar configuración del código por seguridad

- \*\*Contiene\*\*: Host DB, puerto 3307, credenciales, secret key

#### \*\*4. README.md - DOCUMENTACIÓN BÁSICA\*\*

- \*\*Qué hace\*\*: Documentación básica del proyecto

- \*\*Para qué\*\*: Instrucciones de instalación y uso

- \*\*Por qué\*\*: Primera información que ve cualquier desarrollador

#### \*\*5. DOCUMENTACION\_COMPLETA.md - DOCUMENTACIÓN TÉCNICA\*\*

- \*\*Qué hace\*\*: Documentación técnica detallada

- \*\*Para qué\*\*: Referencia completa de la arquitectura

- \*\*Por qué\*\*: Facilita mantenimiento y comprensión del sistema

---

### \*\*📂 CARPETA models - CAPA DE DATOS\*\*

#### \*\*\_\_init\_\_.py\*\*

- \*\*Qué hace\*\*: Inicializa el paquete models

- \*\*Para qué\*\*: Centraliza importaciones de modelos

- \*\*Por qué\*\*: Facilita imports y organización modular

#### \*\*config.py\*\*

- \*\*Qué hace\*\*: Configuración de la aplicación Flask

- \*\*Para qué\*\*: Centralizar configuración de base de datos

- \*\*Por qué\*\*: Separar configuración de lógica de negocio

- \*\*Configuraciones\*\*: URI MySQL, charset UTF-8, secret key

#### \*\*database.py\*\*

- \*\*Qué hace\*\*: Instancias globales de SQLAlchemy y bcrypt

- \*\*Para qué\*\*: Evitar importaciones circulares

- \*\*Por qué\*\*: Patrón singleton para recursos compartidos

#### \*\*models.py\*\*

- \*\*Qué hace\*\*: Importaciones centralizadas de todos los modelos

- \*\*Para qué\*\*: Facilitar acceso a modelos desde app.py

- \*\*Por qué\*\*: Simplificar importaciones en el archivo principal

### \*\*📊 SUBCARPETA entities - MODELOS DE BASE DE DATOS\*\*

#### \*\*user.py\*\*

- \*\*Qué hace\*\*: Modelo de usuario con autenticación

- \*\*Para qué\*\*: Gestión de usuarios y roles del sistema

- \*\*Por qué\*\*: Autenticación y autorización seguras

- \*\*Funcionalidades\*\*:

- Encriptación de contraseñas con bcrypt

- Integración con Flask-Login

- Campo `es\_admin` para roles

- Métodos de verificación

#### \*\*`entities/provincia.py`\*\*

- \*\*Qué hace\*\*: Modelo de provincias de Ecuador

- \*\*Para qué\*\*: Organización territorial del país

- \*\*Por qué\*\*: Estructura jerárquica provincia → ciudad

- \*\*Relación\*\*: Una provincia tiene muchas ciudades

#### \*\*`entities/ciudad.py`\*\*

- \*\*Qué hace\*\*: Modelo de ciudades con atributos especiales

- \*\*Para qué\*\*: Nodos del grafo de rutas

- \*\*Por qué\*\*: Cada ciudad es un vértice en el algoritmo de grafos

- \*\*Atributos especiales\*\*:

- `es\_costera`: Marcador para ciudades costeras

- Relación con provincia

- Base para origen/destino de rutas

#### \*\*`entities/ruta.py`\*\*

- \*\*Qué hace\*\*: Modelo de conexiones entre ciudades

- \*\*Para qué\*\*: Aristas del grafo con pesos (costos)

- \*\*Por qué\*\*: Implementa las conexiones para algoritmo de Dijkstra

- \*\*Campos clave\*\*:

- `ciudad\_origen\_id`: Ciudad de salida

- `ciudad\_destino\_id`: Ciudad de llegada

- `peso`: Costo/distancia de la ruta

### \*\*🗄️ SUBCARPETA repositories - ACCESO A DATOS\*\*

#### \*\*`repositories/base\_repository.py`\*\*

- \*\*Qué hace\*\*: Clase base con operaciones CRUD genéricas

- \*\*Para qué\*\*: Evitar repetición de código en repositorios

- \*\*Por qué\*\*: Principio DRY (Don't Repeat Yourself)

- \*\*Métodos\*\*: create, get\_by\_id, get\_all, update, delete

#### \*\*`repositories/user\_repository.py`\*\*

- \*\*Qué hace\*\*: Operaciones específicas de usuarios

- \*\*Para qué\*\*: Consultas complejas de autenticación

- \*\*Por qué\*\*: Encapsular lógica de acceso a datos de usuarios

#### \*\*`repositories/ciudad\_repository.py`\*\*

- \*\*Qué hace\*\*: Operaciones específicas de ciudades

- \*\*Para qué\*\*: Consultas para construcción del grafo

- \*\*Por qué\*\*: Optimizar consultas de ciudades para algoritmos

#### \*\*`repositories/provincia\_repository.py`\*\*

- \*\*Qué hace\*\*: Operaciones específicas de provincias

- \*\*Para qué\*\*: Gestión administrativa de provincias

- \*\*Por qué\*\*: Separar responsabilidades por entidad

#### \*\*`repositories/ruta\_repository.py`\*\*

- \*\*Qué hace\*\*: Operaciones específicas de rutas

- \*\*Para qué\*\*: Consultas para construcción del grafo

- \*\*Por qué\*\*: Base fundamental para algoritmos de rutas

### \*\*🔧 SUBCARPETA services - LÓGICA DE NEGOCIO\*\*

#### \*\*`services/user\_service.py`\*\*

- \*\*Qué hace\*\*: Lógica de negocio para usuarios

- \*\*Para qué\*\*: Procesos complejos de autenticación

- \*\*Por qué\*\*: Separar lógica de negocio del acceso a datos

#### \*\*`services/grafo\_service.py`\*\*

- \*\*Qué hace\*\*: Lógica de negocio para grafos y algoritmos

- \*\*Para qué\*\*: Construcción de grafos y cálculos de rutas

- \*\*Por qué\*\*: Encapsular algoritmos complejos

---

### \*\*🌐 CARPETA routes - CONTROLADORES WEB\*\*

#### \*\*\_\_init\_\_.py\*\*

- \*\*Qué hace\*\*: Registra todos los blueprints

- \*\*Para qué\*\*: Modularización de rutas

- \*\*Por qué\*\*: Organización limpia del código

#### \*\*auth\_routes.py\*\*

- \*\*Qué hace\*\*: Maneja autenticación (login/register/logout)

- \*\*Para qué\*\*: Control de acceso al sistema

- \*\*Por qué\*\*: Seguridad y gestión de sesiones

- \*\*Rutas\*\*: `/login`, `/register`, `/logout`

#### \*\*home\_routes.py\*\*

- \*\*Qué hace\*\*: Página principal después del login

- \*\*Para qué\*\*: Dashboard del usuario

- \*\*Por qué\*\*: Punto central de navegación

#### \*\*admin\_routes.py\*\*

- \*\*Qué hace\*\*: Panel de administración completo

- \*\*Para qué\*\*: Gestión CRUD de todas las entidades

- \*\*Por qué\*\*: Interfaz administrativa para el sistema

- \*\*Funcionalidades\*\*: Gestión de usuarios, provincias, ciudades, rutas

#### \*\*rutaeconomica\_routes.py\*\*

- \*\*Qué hace\*\*: Búsqueda de rutas óptimas entre ciudades

- \*\*Para qué\*\*: Funcionalidad principal del sistema

- \*\*Por qué\*\*: Implementa el algoritmo de Dijkstra

- \*\*Características\*\*:

- Algoritmo de camino más corto

- Visualización de grafos

- Resultados con costos

#### \*\*ruta\_fija\_routes.py\*\*

- \*\*Qué hace\*\*: Ruta predefinida específica Ibarra→Loja

- \*\*Para qué\*\*: Demostración de ruta fija del sistema

- \*\*Por qué\*\*: Ejemplo concreto de funcionalidad

---

### \*\*🎮 CARPETA controllers - LÓGICA DE GRAFOS\*\*

#### \*\*grafo\_utils.py - NÚCLEO DEL SISTEMA\*\*

- \*\*Qué hace\*\*: Implementa toda la lógica de grafos y algoritmos

- \*\*Para qué\*\*: Core del sistema de rutas

- \*\*Por qué\*\*: Corazón matemático de la aplicación

- \*\*Funciones principales\*\*:

##### \*\*`construir\_grafo()`\*\*

- \*\*Qué hace\*\*: Construye grafo de NetworkX desde base de datos

- \*\*Para qué\*\*: Convertir datos relacionales en estructura de grafo

- \*\*Por qué\*\*: Base para todos los algoritmos

##### \*\*`grafo\_a\_imagen()`\*\*

- \*\*Qué hace\*\*: Genera visualización PNG del grafo

- \*\*Para qué\*\*: Representación visual de la red de rutas

- \*\*Por qué\*\*: Facilita comprensión del usuario

- \*\*Tecnología\*\*: matplotlib + NetworkX con layout spring

##### \*\*`camino\_optimo\_con\_costera()`\*\*

- \*\*Qué hace\*\*: Implementa algoritmo de Dijkstra modificado

- \*\*Para qué\*\*: Encontrar ruta más económica entre ciudades

- \*\*Por qué\*\*: Algoritmo óptimo para caminos más cortos

- \*\*Características\*\*:

- Algoritmo de Dijkstra clásico

- Detección de ciudades costeras en la ruta

- Retorna camino completo con costo total

##### \*\*`obtener\_ciudades()`\*\*

- \*\*Qué hace\*\*: Lista todas las ciudades disponibles

- \*\*Para qué\*\*: Poblar formularios de selección

- \*\*Por qué\*\*: Interfaz usuario-friendly

##### \*\*`estadisticas\_grafo()`\*\*

- \*\*Qué hace\*\*: Calcula estadísticas del grafo

- \*\*Para qué\*\*: Información analítica del sistema

- \*\*Por qué\*\*: Insights sobre la red de rutas

---

### \*\*🎨 CARPETA templates - INTERFAZ DE USUARIO\*\*

#### \*\*`base.html`\*\*

- \*\*Qué hace\*\*: Plantilla base con navegación y estilos

- \*\*Para qué\*\*: Layout común para todas las páginas

- \*\*Por qué\*\*: Consistencia visual y DRY en frontend

- \*\*Tecnología\*\*: AdminLTE + Bootstrap

#### \*\*`home.html`\*\*

- \*\*Qué hace\*\*: Dashboard principal del usuario

- \*\*Para qué\*\*: Punto de entrada después del login

- \*\*Por qué\*\*: Navegación centralizada a funcionalidades

#### \*\*`formulario\_simple.html`\*\*

- \*\*Qué hace\*\*: Interfaz para búsqueda de rutas económicas

- \*\*Para qué\*\*: Funcionalidad principal del sistema

- \*\*Por qué\*\*: Interfaz amigable para algoritmo complejo

- \*\*Características\*\*:

- Selección de origen y destino

- Visualización de resultados

- Mostrar grafo generado

#### \*\*`ruta\_fija.html`\*\*

- \*\*Qué hace\*\*: Muestra ruta predefinida Ibarra→Loja

- \*\*Para qué\*\*: Demostración específica

- \*\*Por qué\*\*: Ejemplo concreto de uso

#### \*\*`error.html`\*\*

- \*\*Qué hace\*\*: Páginas de error personalizadas

- \*\*Para qué\*\*: Manejo elegante de errores

- \*\*Por qué\*\*: Experiencia de usuario profesional

### \*\*📁 SUBCARPETA auth\*\*

#### \*\*`login.html` / `register.html`\*\*

- \*\*Qué hace\*\*: Formularios de autenticación

- \*\*Para qué\*\*: Control de acceso al sistema

- \*\*Por qué\*\*: Seguridad y gestión de usuarios

### \*\*📁 SUBCARPETA admin\*\*

#### \*\*`dashboard.html`\*\*

- \*\*Qué hace\*\*: Panel de control administrativo

- \*\*Para qué\*\*: Vista general del sistema para admin

- \*\*Por qué\*\*: Gestión centralizada

#### \*\*`ciudades.html`, `provincias.html`, `rutas.html`\*\*

- \*\*Qué hace\*\*: Interfaces CRUD para cada entidad

- \*\*Para qué\*\*: Gestión completa de datos

- \*\*Por qué\*\*: Administración sin necesidad de base de datos directa

---

### \*\*🎨 CARPETA static - RECURSOS ESTÁTICOS\*\*

#### \*\*img\*\*

- \*\*Qué hace\*\*: Imágenes del sistema (logos, iconos)

- \*\*Para qué\*\*: Identidad visual de la aplicación

- \*\*Por qué\*\*: Branding y experiencia visual

#### \*\*plugins\*\*

- \*\*Qué hace\*\*: Librerías frontend (Bootstrap, AdminLTE, jQuery)

- \*\*Para qué\*\*: Funcionalidad y estilos avanzados

- \*\*Por qué\*\*: Interfaz profesional sin desarrollo desde cero

---

### \*\*📄 CARPETA codigos\_sql\*\*

#### \*\*laragon\_database.sql\*\*

- \*\*Qué hace\*\*: Script SQL para inicialización de base de datos

- \*\*Para qué\*\*: Crear estructura inicial y datos de prueba

- \*\*Por qué\*\*: Deployment automático de la base de datos

---

## 🎯 \*\*ARQUITECTURA DEL SISTEMA DE GRAFOS\*\*

### \*\*📊 Cómo Funciona el Grafo:\*\*

1. \*\*Nodos (Vértices)\*\*: Cada ciudad es un nodo

2. \*\*Aristas (Edges)\*\*: Cada ruta es una arista con peso (costo)

3. \*\*Algoritmo\*\*: Dijkstra para camino más corto

4. \*\*Visualización\*\*: NetworkX + matplotlib con layout spring

### \*\*🔄 Lógica de las Rutas:\*\*

1. \*\*Construcción\*\*: `construir\_grafo()` lee BD y crea grafo NetworkX

2. \*\*Cálculo\*\*: `camino\_optimo\_con\_costera()` aplica Dijkstra

3. \*\*Resultado\*\*: Retorna camino completo + costo total + ciudades costeras

4. \*\*Visualización\*\*: `grafo\_a\_imagen()` genera PNG del grafo

Ahora te explico qué hace la carpeta .venv:

## 📂 \*\*CARPETA .venv - ENTORNO VIRTUAL DE PYTHON\*\*

### \*\*🔍 ¿Qué hace la carpeta .venv?\*\*

La carpeta .venv es un \*\*entorno virtual de Python\*\* (virtual environment). Es una carpeta especial que contiene una instalación aislada de Python con sus propias librerías.

### \*\*🎯 ¿Para qué sirve?\*\*

#### \*\*1. AISLAMIENTO DE DEPENDENCIAS\*\*

- \*\*Problema\*\*: Si instalas librerías globalmente en Python, pueden haber conflictos entre proyectos

- \*\*Solución\*\*: Cada proyecto tiene su propio entorno aislado

- \*\*Ejemplo\*\*:

- Proyecto A necesita Flask 2.0

- Proyecto B necesita Flask 1.1

- Sin .venv → Conflicto

- Con .venv → Cada uno en su entorno

#### \*\*2. GESTIÓN LIMPIA DE VERSIONES\*\*

- \*\*Tu proyecto\*\*: Usa versiones específicas definidas en requirements.txt

- \*\*Otros proyectos\*\*: No se afectan por las librerías de este proyecto

- \*\*Sistema\*\*: Python global queda limpio

#### \*\*3. REPRODUCIBILIDAD\*\*

- \*\*En tu máquina\*\*: Instalas librerías en .venv

- \*\*En otra máquina\*\*: Se puede recrear exactamente el mismo entorno

- \*\*Comando\*\*: `pip install -r requirements.txt`

### \*\*🗂️ ¿Qué contiene la carpeta .venv?\*\* ### \*\*📋 Estructura de .venv:\*\*

#### \*\*`Scripts/` (Windows) o `bin/` (Linux/Mac)\*\*

- \*\*Qué contiene\*\*: Ejecutables de Python y activación del entorno

- \*\*Archivos importantes\*\*:

- `python.exe` - Intérprete de Python del entorno

- `pip.exe` - Gestor de paquetes del entorno

- `activate.bat` - Script para activar el entorno

#### \*\*`Lib/` - LIBRERÍAS INSTALADAS\*\*

- \*\*Qué contiene\*\*: Todas las librerías Python del proyecto

- \*\*Subcarpetas\*\*:

- `site-packages/` - Donde se instalan las librerías con pip

- Aquí están Flask, SQLAlchemy, NetworkX, etc.

#### \*\*`Include/`\*\*

- \*\*Qué contiene\*\*: Archivos header de Python

- \*\*Para qué\*\*: Compilación de extensiones C/C++

#### \*\*`pyvenv.cfg`\*\*

- \*\*Qué contiene\*\*: Configuración del entorno virtual

- \*\*Para qué\*\*: Define versión de Python y configuraciones

### \*\*🔧 ¿Por qué es importante para tu proyecto?\*\*

#### \*\*1. PARA DESARROLLO\*\*

```bash

# Activar entorno (Windows)

.venv\Scripts\activate

# Instalar dependencias

pip install -r requirements.txt

# Ahora todas las librerías están aisladas

python app.py

```

#### \*\*2. PARA PRESENTACIÓN\*\*

- \*\*Ventaja\*\*: Demuestras buenas prácticas de desarrollo

- \*\*Profesional\*\*: Los entornos virtuales son estándar en la industria

- \*\*Reproducible\*\*: El profesor puede recrear tu entorno exacto

#### \*\*3. PARA DEPLOYMENT\*\*

- \*\*Documentación\*\*: Explicas cómo configurar el entorno

- \*\*Replicabilidad\*\*: Funciona igual en cualquier máquina

### \*\*⚠️ ¿Debo incluir .venv en la documentación?\*\*

#### \*\*SÍ, porque demuestra:\*\*

- ✅ \*\*Buenas prácticas\*\* de desarrollo Python

- ✅ \*\*Gestión profesional\*\* de dependencias

- ✅ \*\*Conocimiento\*\* de herramientas estándar

- ✅ \*\*Código reproducible\*\* y profesional

#### \*\*En tu documentación técnica, incluye:\*\*

\*\*Para la sección de "Arquitectura del Sistema":\*\*

### \*\*🔧 Gestión de Entorno Virtual (.venv/)\*\*

- \*\*Qué hace\*\*: Aislamiento de dependencias Python específicas del proyecto

- \*\*Para qué\*\*: Evitar conflictos entre librerías de diferentes proyectos

- \*\*Por qué\*\*: Buena práctica estándar en desarrollo Python profesional

- \*\*Beneficios\*\*:

- Reproducibilidad del entorno

- Gestión limpia de versiones

- Aislamiento de dependencias

- Deployment consistente

\*\*Comandos importantes:\*\*

```bash

# Crear entorno virtual

python -m venv .venv

# Activar entorno (Windows)

.venv\Scripts\activate

# Instalar dependencias

pip install -r requirements.txt

# Desactivar entorno

deactivate

```

### \*\*📊 Relación con requirements.txt:\*\*

- .venv → Contiene las librerías instaladas

- requirements.txt → Lista las librerías necesarias

- \*\*Flujo\*\*: requirements.txt define → `pip install` → se guarda en .venv

\*\*¡Es una carpeta muy importante que demuestra profesionalismo en tu proyecto!\*\* 🎯

**# 📋 DOCUMENTACIÓN COMPLETA DEL PROYECTO**

**## Sistema de Rutas de Ecuador - Flask Web Application**

---

**## 🔍 \*\*EXPLICACIÓN DE TABLAS DE BASE DE DATOS\*\***

**### \*\*¿Por qué hay "users" y "usuarios"?\*\***

- **\*\*El sistema USA SOLO la tabla "usuarios"\*\*** (definida en `models/entities/user.py`)

- **\*\*La tabla "users" puede ser residual\*\*** de configuraciones anteriores de MySQL

- **\*\*IMPORTANTE\*\***: El modelo `User` especifica `\_\_tablename\_\_ = 'usuarios'`

- **\*\*Todos los registros se guardan en "usuarios"\*\*** ✅

**\*\*Verificación\*\***: El sistema funciona correctamente con la tabla "usuarios"

---

**## 📁 \*\*ESTRUCTURA DEL PROYECTO\*\***

**### \*\*🎯 ARCHIVOS PRINCIPALES\*\***

**#### \*\*`app.py`\*\* - ARCHIVO PRINCIPAL**

- **\*\*Función\*\***: Punto de entrada de la aplicación Flask

- **\*\*Qué hace\*\***:

  - Configura Flask, base de datos, autenticación

  - Registra blueprints (rutas)

  - Maneja errores 404/500

  - Verifica estado de la base de datos

- **\*\*Puerto\*\***: 4000

- **\*\*Para ejecutar\*\***: `python app.py`

**#### \*\*`forms.py`\*\* - FORMULARIOS WEB**

- **\*\*Función\*\***: Define formularios usando Flask-WTF

- **\*\*Qué contiene\*\***:

  - LoginForm: Formulario de login

  - RegistroForm: Formulario de registro

  - Validaciones de campos

- **\*\*Por qué\*\***: Seguridad y validación de datos

**#### \*\*`requirements.txt`\*\* - DEPENDENCIAS**

- **\*\*Función\*\***: Lista todas las librerías Python necesarias

- **\*\*Para instalar\*\***: `pip install -r requirements.txt`

- **\*\*Contiene\*\***: Flask, SQLAlchemy, NetworkX, matplotlib, etc.

**#### \*\*`init\_db.py`\*\* - INICIALIZACIÓN DE BASE DE DATOS**

- **\*\*Función\*\***: Script para crear datos iniciales

- **\*\*Qué hace\*\***:

  - Crea 6 provincias de Ecuador

  - Crea 6 ciudades principales

  - Crea 16 rutas entre ciudades

  - Crea usuario admin (admin/admin)

- **\*\*Para ejecutar\*\***: `python init\_db.py`

---

**### \*\*📂 CARPETA `models/`\*\* - MODELOS DE BASE DE DATOS**

**#### \*\*`models/\_\_init\_\_.py`\*\***

- **\*\*Función\*\***: Hace que `models` sea un paquete Python

- **\*\*Importa\*\***: Todos los modelos para uso fácil

**#### \*\*`models/config.py`\*\* - CONFIGURACIÓN**

- **\*\*Función\*\***: Configuración de la base de datos

- **\*\*Contiene\*\***:

  - URI de MySQL (puerto 3307)

  - Configuración de charset UTF-8

  - Secret key para sesiones

**#### \*\*`models/database.py`\*\* - INSTANCIAS DE BD**

- **\*\*Función\*\***: Instancias globales de SQLAlchemy y bcrypt

- **\*\*Por qué\*\***: Evita importaciones circulares

**#### \*\*`models/models.py`\*\* - IMPORTACIONES CENTRALES**

- **\*\*Función\*\***: Importa todos los modelos en un solo lugar

- **\*\*Para\*\***: Facilitar imports en otras partes

**### \*\*📊 ENTIDADES (Modelos de BD)\*\***

**#### \*\*`models/entities/user.py`\*\* - MODELO USUARIO**

- **\*\*Tabla\*\***: `usuarios` (NO "users")

- **\*\*Campos\*\***: id, username, email, password\_hash, es\_admin, created\_at

- **\*\*Métodos\*\***:

  - `set\_password()`: Encripta contraseñas

  - `check\_password()`: Verifica contraseñas

  - `is\_admin()`: Verifica si es administrador

- **\*\*Autenticación\*\***: Usa Flask-Login

**#### \*\*`models/entities/provincia.py`\*\* - MODELO PROVINCIA**

- **\*\*Tabla\*\***: `provincias`

- **\*\*Campos\*\***: id, nombre, codigo, created\_at

- **\*\*Relación\*\***: Una provincia tiene muchas ciudades

**#### \*\*`models/entities/ciudad.py`\*\* - MODELO CIUDAD**

- **\*\*Tabla\*\***: `ciudades`

- **\*\*Campos\*\***: id, nombre, provincia\_id, es\_costera, created\_at

- **\*\*Relaciones\*\***:

  - Pertenece a una provincia

  - Puede ser origen/destino de rutas

**#### \*\*`models/entities/ruta.py`\*\* - MODELO RUTA**

- **\*\*Tabla\*\***: `rutas`

- **\*\*Campos\*\***: id, ciudad\_origen\_id, ciudad\_destino\_id, peso, created\_at

- **\*\*Función\*\***: Define conexiones entre ciudades con costos

**### \*\*🗄️ REPOSITORIOS (Acceso a Datos)\*\***

**#### \*\*`repositories/base\_repository.py`\*\***

- **\*\*Función\*\***: Clase base para operaciones CRUD

- **\*\*Métodos\*\***: create, get\_by\_id, get\_all, update, delete

- **\*\*Principio\*\***: DRY (Don't Repeat Yourself)

**#### \*\*`repositories/user\_repository.py`\*\***

- **\*\*Función\*\***: Operaciones específicas de usuarios

- **\*\*Métodos\*\***: get\_by\_username, get\_by\_email

**#### \*\*`repositories/ciudad\_repository.py`\*\***

- **\*\*Función\*\***: Operaciones específicas de ciudades

- **\*\*Métodos\*\***: get\_by\_provincia, get\_costeras

**#### \*\*`repositories/provincia\_repository.py`\*\***

- **\*\*Función\*\***: Operaciones específicas de provincias

**#### \*\*`repositories/ruta\_repository.py`\*\***

- **\*\*Función\*\***: Operaciones específicas de rutas

- **\*\*Métodos\*\***: get\_by\_origen, get\_by\_destino

**### \*\*🔧 SERVICIOS (Lógica de Negocio)\*\***

**#### \*\*`services/user\_service.py`\*\***

- **\*\*Función\*\***: Lógica de negocio para usuarios

- **\*\*Métodos\*\***: register\_user, authenticate\_user

**#### \*\*`services/grafo\_service.py`\*\***

- **\*\*Función\*\***: Lógica de grafos y algoritmos

- **\*\*Métodos\*\***: construir\_grafo, calcular\_ruta\_optima

---

**### \*\*🌐 CARPETA `routes/`\*\* - RUTAS WEB**

**#### \*\*`routes/\_\_init\_\_.py`\*\***

- **\*\*Función\*\***: Registra todos los blueprints

- **\*\*Qué hace\*\***: Centraliza el registro de rutas

**#### \*\*`routes/auth\_routes.py`\*\* - AUTENTICACIÓN**

- **\*\*Rutas\*\***:

  - `/login` - Página de login

  - `/register` - Página de registro

  - `/logout` - Cerrar sesión

- **\*\*Función\*\***: Maneja autenticación de usuarios

**#### \*\*`routes/home\_routes.py`\*\* - PÁGINA PRINCIPAL**

- **\*\*Rutas\*\***:

  - `/home` - Dashboard del usuario

- **\*\*Función\*\***: Página principal después del login

**#### \*\*`routes/admin\_routes.py`\*\* - ADMINISTRACIÓN**

- **\*\*Rutas\*\***:

  - `/admin/dashboard` - Panel de control

  - `/admin/usuarios` - Gestión de usuarios

  - `/admin/provincias` - Gestión de provincias

  - `/admin/ciudades` - Gestión de ciudades

  - `/admin/rutas` - Gestión de rutas

- **\*\*Función\*\***: Panel de administración completo

- **\*\*Protección\*\***: Solo usuarios admin

**#### \*\*`routes/rutaeconomica\_routes.py`\*\* - BÚSQUEDA DE RUTAS**

- **\*\*Rutas\*\***:

  - `/ruta\_economica` - Buscar ruta óptima

  - `/grafo` - Ver grafo de rutas

- **\*\*Función\*\***: Algoritmo de Dijkstra para ruta más económica

**#### \*\*`routes/ruta\_fija\_routes.py`\*\* - RUTA FIJA**

- **\*\*Rutas\*\***:

  - `/ruta\_fija` - Ruta predefinida Ibarra→Loja

  - `/ruta\_fija/grafo` - Grafo de la ruta fija

- **\*\*Función\*\***: Muestra ruta específica con costo

---

**### \*\*🎮 CARPETA `controllers/`\*\* - CONTROLADORES**

**#### \*\*`controllers/grafo\_utils.py`\*\* - UTILIDADES DE GRAFOS**

- **\*\*Funciones principales\*\***:

  - `construir\_grafo()`: Crea grafo de NetworkX desde BD

  - `grafo\_a\_imagen()`: Genera imagen PNG del grafo

  - `camino\_optimo\_con\_costera()`: Algoritmo de Dijkstra

  - `obtener\_ciudades()`: Lista todas las ciudades

  - `estadisticas\_grafo()`: Estadísticas del grafo

- **\*\*Tecnologías\*\***: NetworkX, matplotlib

- **\*\*Algoritmo\*\***: Dijkstra para camino más corto

---

**### \*\*🎨 CARPETA `templates/`\*\* - PLANTILLAS HTML**

**#### \*\*`base.html`\*\* - PLANTILLA BASE**

- **\*\*Función\*\***: Layout base para todas las páginas

- **\*\*Contiene\*\***: Navegación, estilos, scripts comunes

- **\*\*Framework\*\***: AdminLTE (Bootstrap)

**#### \*\*`home.html`\*\* - PÁGINA PRINCIPAL**

- **\*\*Función\*\***: Dashboard del usuario logueado

- **\*\*Características\*\***: Navegación a todas las funcionalidades

**#### \*\*`formulario\_simple.html`\*\* - BÚSQUEDA DE RUTAS**

- **\*\*Función\*\***: Formulario para buscar rutas entre ciudades

- **\*\*Características\*\***:

  - Dropdown de ciudades

  - Muestra resultado con costo

  - Visualización del grafo

**#### \*\*`ruta\_fija.html`\*\* - RUTA FIJA**

- **\*\*Función\*\***: Muestra ruta predefinida Ibarra→Loja

- **\*\*Características\*\***:

  - Ruta fija calculada

  - Costo total

  - Indicador de ciudades costeras

**#### \*\*`error.html`\*\* - PÁGINA DE ERRORES**

- **\*\*Función\*\***: Maneja errores 404, 500, etc.

- **\*\*Características\*\***: Mensajes de error amigables

**### \*\*📁 CARPETA `templates/auth/`\*\* - AUTENTICACIÓN**

**#### \*\*`login.html`\*\* - PÁGINA DE LOGIN**

- **\*\*Función\*\***: Formulario de inicio de sesión

- **\*\*Campos\*\***: Username/email, contraseña

**#### \*\*`register.html`\*\* - PÁGINA DE REGISTRO**

- **\*\*Función\*\***: Formulario de registro de usuarios

- **\*\*Campos\*\***: Username, email, contraseña, confirmación

**### \*\*📁 CARPETA `templates/admin/`\*\* - ADMINISTRACIÓN**

**#### \*\*`dashboard.html`\*\* - DASHBOARD ADMIN**

- **\*\*Función\*\***: Panel de control del administrador

- **\*\*Estadísticas\*\***: Usuarios, provincias, ciudades, rutas

**#### \*\*`usuarios.html`\*\* - GESTIÓN DE USUARIOS**

- **\*\*Función\*\***: Lista y administra usuarios del sistema

**#### \*\*`provincias.html`\*\* - GESTIÓN DE PROVINCIAS**

- **\*\*Función\*\***: CRUD completo de provincias

**#### \*\*`ciudades.html`\*\* - GESTIÓN DE CIUDADES**

- **\*\*Función\*\***: CRUD completo de ciudades

- **\*\*Características\*\***: Asignación a provincias, marca costera

**#### \*\*`rutas.html`\*\* - GESTIÓN DE RUTAS**

- **\*\*Función\*\***: CRUD completo de rutas entre ciudades

- **\*\*Características\*\***: Origen, destino, peso/costo

---

**### \*\*🎨 CARPETA `static/`\*\* - ARCHIVOS ESTÁTICOS**

**#### \*\*`static/img/`\*\* - IMÁGENES**

- **\*\*`Logo\_J.png`\*\***: Logo de la aplicación

- **\*\*`kuchau.png`\*\***: Imagen decorativa

**#### \*\*`static/plugins/`\*\* - LIBRERÍAS FRONTEND**

- **\*\*Bootstrap\*\***: Framework CSS

- **\*\*AdminLTE\*\***: Tema de administración

- **\*\*FontAwesome\*\***: Iconos

- **\*\*jQuery\*\***: JavaScript

**#### \*\*`static/dist/`\*\* - ARCHIVOS COMPILADOS**

- **\*\*CSS\*\***: Estilos de AdminLTE

- **\*\*JS\*\***: Scripts de AdminLTE

---

**### \*\*🔧 CARPETA `.venv/`\*\* - ENTORNO VIRTUAL PYTHON**

**#### \*\*¿Qué hace?\*\***

- **\*\*Función\*\***: Entorno virtual aislado con todas las dependencias Python

- **\*\*Para qué\*\***: Evita conflictos entre librerías de diferentes proyectos

- **\*\*Por qué\*\***: Buena práctica estándar en desarrollo Python profesional

**#### \*\*Contenido:\*\***

- **\*\*`Scripts/`\*\***: Ejecutables de Python y pip del entorno

- **\*\*`Lib/site-packages/`\*\***: Todas las librerías instaladas (Flask, SQLAlchemy, NetworkX, etc.)

- **\*\*`pyvenv.cfg`\*\***: Configuración del entorno virtual

**#### \*\*Comandos importantes:\*\***

```bash

# Activar entorno (NECESARIO antes de ejecutar)

.venv\Scripts\activate

# Instalar dependencias

pip install -r requirements.txt

# Ejecutar aplicación

python app.py

# Desactivar entorno

deactivate

```

**#### \*\*⚠️ IMPORTANTE:\*\***

- **\*\*NO borrar esta carpeta\*\*** - contiene todas las librerías necesarias

- **\*\*Sin .venv la aplicación NO funciona\*\***

- **\*\*Demuestra desarrollo profesional Python\*\***

---

**## 🚀 \*\*FLUJO DE LA APLICACIÓN\*\***

**### \*\*1. Inicio\*\***

1. Usuario accede a `http://localhost:4000`

2. `app.py` detecta si está logueado

3. Si no: redirige a `/login`

4. Si sí: redirige a `/home` o `/admin/dashboard`

**### \*\*2. Autenticación\*\***

1. Usuario ingresa credenciales en `login.html`

2. `auth\_routes.py` procesa el login

3. `user\_service.py` verifica credenciales

4. Flask-Login maneja la sesión

**### \*\*3. Funcionalidades\*\***

- **\*\*Usuario normal\*\***: Puede buscar rutas económicas

- **\*\*Administrador\*\***: Puede gestionar todo el sistema

**### \*\*4. Búsqueda de Rutas\*\***

1. Usuario selecciona origen y destino

2. `rutaeconomica\_routes.py` recibe la petición

3. `grafo\_utils.py` ejecuta algoritmo de Dijkstra

4. Retorna ruta óptima con costo

---

**## 🔧 \*\*TECNOLOGÍAS UTILIZADAS\*\***

**### \*\*Backend\*\***

- **\*\*Flask\*\***: Framework web de Python

- **\*\*SQLAlchemy\*\***: ORM para base de datos

- **\*\*Flask-Login\*\***: Manejo de sesiones

- **\*\*bcrypt\*\***: Encriptación de contraseñas

- **\*\*NetworkX\*\***: Algoritmos de grafos

- **\*\*matplotlib\*\***: Generación de gráficos

**### \*\*Frontend\*\***

- **\*\*AdminLTE\*\***: Framework de administración

- **\*\*Bootstrap\*\***: Framework CSS

- **\*\*jQuery\*\***: JavaScript

- **\*\*FontAwesome\*\***: Iconos

**### \*\*Base de Datos\*\***

- **\*\*MySQL\*\***: Sistema de gestión de base de datos

- **\*\*Charset UTF-8\*\***: Soporte para acentos españoles

---

**## 📊 \*\*DATOS DEL SISTEMA\*\***

**### \*\*6 Provincias de Ecuador\*\***

1. Pichincha (PIC)

2. Guayas (GUA)

3. Azuay (AZU)

4. Manabí (MAN)

5. Imbabura (IMB)

6. Loja (LOJ)

**### \*\*6 Ciudades Principales\*\***

1. **\*\*Quito\*\*** (Pichincha) - Capital

2. **\*\*Guayaquil\*\*** (Guayas) - Costera ⚓

3. **\*\*Cuenca\*\*** (Azuay)

4. **\*\*Portoviejo\*\*** (Manabí) - Costera ⚓

5. **\*\*Ibarra\*\*** (Imbabura)

6. **\*\*Loja\*\*** (Loja)

**### \*\*16 Rutas Principales\*\***

- Conexiones bidireccionales entre ciudades

- Distancias reales en kilómetros

- Ejemplo: Quito ↔ Guayaquil (420 km)

---

**## 🎯 \*\*PUNTOS CLAVE PARA LA EXPOSICIÓN\*\***

**### \*\*1. Arquitectura MVC\*\***

- **\*\*Modelos\*\***: Entidades de base de datos

- **\*\*Vistas\*\***: Templates HTML

- **\*\*Controladores\*\***: Routes y services

**### \*\*2. Principios SOLID\*\***

- **\*\*Single Responsibility\*\***: Cada clase tiene una función específica

- **\*\*Open/Closed\*\***: Extensible sin modificar código existente

- **\*\*Dependency Injection\*\***: Repositorios inyectados en servicios

**### \*\*3. Seguridad\*\***

- **\*\*Autenticación\*\***: Flask-Login

- **\*\*Encriptación\*\***: bcrypt para contraseñas

- **\*\*Autorización\*\***: Decoradores admin\_required

**### \*\*4. Algoritmos\*\***

- **\*\*Dijkstra\*\***: Para encontrar la ruta más económica

- **\*\*NetworkX\*\***: Librería especializada en grafos

**### \*\*5. Base de Datos\*\***

- **\*\*Relaciones\*\***: Foreign keys entre tablas

- **\*\*Integridad\*\***: Cascadas para eliminación

- **\*\*UTF-8\*\***: Soporte completo para español

---

**## ⚡ \*\*COMANDOS IMPORTANTES\*\***

```bash

# Inicializar base de datos

python init\_db.py

# Ejecutar aplicación

python app.py

# Acceder al sistema

http://localhost:4000

# Credenciales admin

Usuario: admin

Contraseña: admin

```

---

**## ✅ \*\*SISTEMA COMPLETAMENTE FUNCIONAL\*\***

- ✅ Autenticación de usuarios

- ✅ Panel de administración

- ✅ Gestión completa de CRUD

- ✅ Algoritmo de rutas óptimas

- ✅ Visualización de grafos

- ✅ Interfaz responsive

- ✅ Base de datos UTF-8

- ✅ Manejo de errores

- ✅ Seguridad implementada

**\*\*El proyecto está listo para producción y exposición\*\*** 🎉

Esta estructura te permitirá crear una documentación técnica completa explicando cada componente del sistema. 🎉

Perfecto, te voy a explicar \*\*TODO EL CÓDIGO\*\* para que puedas responder cualquier pregunta del profesor. Te preparo para las preguntas más probables:

## 🎯 \*\*PREGUNTAS ESPERADAS DEL PROFESOR Y RESPUESTAS\*\*

### \*\*1. ¿Cómo funciona la creación de usuarios?\*\*

\*\*Código en auth\_routes.py:\*\*

```python

@auth.route('/register', methods=['GET', 'POST'])

def register():

if request.method == 'POST':

# Obtener datos del formulario

username = request.form.get('username')

email = request.form.get('email')

password = request.form.get('password')

# Verificar si ya existe

existing\_user = User.query.filter\_by(username=username).first()

if existing\_user:

flash('Usuario ya existe')

return redirect(url\_for('auth.register'))

# Crear nuevo usuario

new\_user = User(username=username, email=email, es\_admin=False)

new\_user.set\_password(password) # Encripta con bcrypt

# Guardar en base de datos

db.session.add(new\_user)

db.session.commit()

```

\*\*Tu respuesta:\*\*

> "Los usuarios se crean en la ruta `/register`. Primero verifico si el username ya existe en la base de datos. Si no existe, creo un nuevo objeto `User`, uso el método `set\_password()` que encripta la contraseña con bcrypt por seguridad, y lo guardo en la tabla `usuarios` usando SQLAlchemy."

### \*\*2. ¿Cómo funcionan los grafos en doble dirección?\*\*

\*\*Código en grafo\_utils.py:\*\*

```python

def construir\_grafo():

"""Construye el grafo desde la base de datos"""

G = nx.Graph() # Grafo NO dirigido (automáticamente bidireccional)

# Agregar ciudades como nodos

ciudades = Ciudad.query.all()

for ciudad in ciudades:

G.add\_node(ciudad.id,

nombre=ciudad.nombre,

es\_costera=ciudad.es\_costera)

# Agregar rutas como aristas

rutas = Ruta.query.all()

for ruta in rutas:

# Una sola ruta en BD = dos direcciones en el grafo

G.add\_edge(ruta.ciudad\_origen\_id,

ruta.ciudad\_destino\_id,

weight=ruta.peso)

```

\*\*Tu respuesta:\*\*

> "Uso NetworkX con `nx.Graph()` que es un grafo NO dirigido, esto significa que automáticamente cada arista funciona en ambas direcciones. Si agrego una ruta de Quito a Guayaquil con peso 420, automáticamente puedo ir de Guayaquil a Quito también con el mismo peso. No necesito crear dos rutas en la base de datos."

### \*\*3. ¿Cómo funciona el algoritmo de Dijkstra?\*\*

\*\*Código en grafo\_utils.py:\*\*

```python

def camino\_optimo\_con\_costera(origen\_id, destino\_id):

G = construir\_grafo()

try:

# Dijkstra automático de NetworkX

camino = nx.shortest\_path(G, origen\_id, destino\_id, weight='weight')

costo\_total = nx.shortest\_path\_length(G, origen\_id, destino\_id, weight='weight')

# Verificar ciudades costeras en el camino

ciudades\_costeras = []

for ciudad\_id in camino:

ciudad = Ciudad.query.get(ciudad\_id)

if ciudad.es\_costera:

ciudades\_costeras.append(ciudad.nombre)

return {

'camino': camino,

'costo': costo\_total,

'ciudades\_costeras': ciudades\_costeras,

'valido': True

}

except nx.NetworkXNoPath:

return {'valido': False, 'error': 'No hay ruta disponible'}

```

\*\*Tu respuesta:\*\*

> "Uso la implementación de Dijkstra de NetworkX con `shortest\_path()`. Le paso el grafo, origen, destino y `weight='weight'` para que use los pesos de las rutas. El algoritmo encuentra automáticamente el camino más económico. Después recorro ese camino para identificar si hay ciudades costeras en la ruta."

### \*\*4. ¿Cómo se visualiza el grafo?\*\*

\*\*Código en grafo\_utils.py:\*\*

```python

def grafo\_a\_imagen(grafo, camino\_optimo=None):

plt.figure(figsize=(12, 8))

# Layout spring para distribución automática

pos = nx.spring\_layout(grafo, k=3, iterations=50)

# Colores: ruta óptima en rojo, otros nodos en azul

node\_colors = []

for node in grafo.nodes():

if camino\_optimo and node in camino\_optimo:

node\_colors.append('red') # Nodos de la ruta

else:

node\_colors.append('lightblue') # Otros nodos

# Dibujar el grafo

nx.draw(grafo, pos,

node\_color=node\_colors,

with\_labels=True,

node\_size=3000,

font\_size=10)

# Guardar como PNG

plt.savefig('static/img/grafo.png', dpi=300, bbox\_inches='tight')

```

\*\*Tu respuesta:\*\*

> "Uso matplotlib con NetworkX. El `spring\_layout` distribuye los nodos automáticamente para evitar solapamientos. Si hay una ruta calculada, la pinto en rojo para resaltar el camino óptimo. Guardo la imagen como PNG en la carpeta static para mostrarla en la web."

### \*\*5. ¿Cómo funciona la autenticación?\*\*

\*\*Código en user.py:\*\*

```python

def set\_password(self, password):

"""Encripta la contraseña con bcrypt"""

self.password\_hash = bcrypt.generate\_password\_hash(password).decode('utf-8')

def check\_password(self, password):

"""Verifica la contraseña"""

return bcrypt.check\_password\_hash(self.password\_hash, password)

def is\_admin(self):

"""Verifica si es administrador"""

return self.es\_admin

```

\*\*Tu respuesta:\*\*

> "Uso Flask-Login para manejar sesiones y bcrypt para encriptar contraseñas. Cuando un usuario se registra, `set\_password()` encripta su contraseña. Al hacer login, `check\_password()` compara la contraseña ingresada con el hash guardado. El campo `es\_admin` determina si puede acceder al panel administrativo."

### \*\*6. ¿Cómo funciona el patrón Repository?\*\*

\*\*Código en base\_repository.py:\*\*

```python

class BaseRepository:

def \_\_init\_\_(self, model):

self.model = model

def create(self, \*\*kwargs):

instance = self.model(\*\*kwargs)

db.session.add(instance)

db.session.commit()

return instance

def get\_all(self):

return self.model.query.all()

def get\_by\_id(self, id):

return self.model.query.get(id)

```

\*\*Tu respuesta:\*\*

> "Implemento el patrón Repository para separar la lógica de acceso a datos. `BaseRepository` tiene operaciones CRUD genéricas que todos los repositorios heredan. Esto evita repetir código y facilita el mantenimiento. Por ejemplo, `UserRepository` hereda de `BaseRepository` pero agrega métodos específicos como `get\_by\_username()`."

### \*\*7. ¿Cómo manejas los errores?\*\*

\*\*Código en app.py:\*\*

```python

@app.errorhandler(500)

def handle\_internal\_error(error):

if "Can't connect to MySQL server" in str(error):

return render\_template('error.html',

error="Error de conexión a la base de datos",

message="Asegúrate de que Laragon esté ejecutándose"), 500

return render\_template('error.html',

error="Error interno del servidor"), 500

@app.errorhandler(404)

def handle\_not\_found(error):

return render\_template('error.html',

error="Página no encontrada"), 404

```

\*\*Tu respuesta:\*\*

> "Implemento manejadores de errores personalizados. El decorador `@app.errorhandler()` captura errores específicos y muestra páginas amigables en lugar de errores técnicos. Distingo entre errores de base de datos y otros errores internos para dar mensajes más útiles al usuario."

### \*\*8. ¿Cómo funciona la arquitectura MVC?\*\*

\*\*Tu respuesta:\*\*

> - \*\*Modelos\*\* (models): Representan los datos (User, Ciudad, Ruta, Provincia) y la lógica de base de datos

> - \*\*Vistas\*\* (templates): Interfaces HTML que el usuario ve, usan AdminLTE para ser responsive

> - \*\*Controladores\*\* (routes): Manejan las peticiones HTTP, procesan datos y deciden qué vista mostrar

> - \*\*Separación clara\*\*: Los controladores llaman a servicios, los servicios usan repositorios, los repositorios acceden a los modelos

### \*\*9. ¿Por qué usaste NetworkX?\*\*

\*\*Tu respuesta:\*\*

> "NetworkX es la librería estándar para grafos en Python. Tiene implementaciones optimizadas de algoritmos como Dijkstra, soporte nativo para grafos dirigidos y no dirigidos, y excelente integración con matplotlib para visualización. Esto me ahorra implementar Dijkstra desde cero y me da herramientas profesionales de análisis de grafos."

### \*\*10. ¿Cómo funciona el entorno virtual?\*\*

\*\*Tu respuesta:\*\*

> ".venv es un entorno virtual que aísla las dependencias del proyecto. Contiene su propia instalación de Python y librerías específicas listadas en requirements.txt. Esto evita conflictos con otros proyectos y garantiza que el código funcione igual en cualquier máquina. Es una buena práctica profesional en desarrollo Python."

## 🔥 \*\*PREGUNTAS TÉCNICAS AVANZADAS POSIBLES:\*\*

### \*\*11. ¿Qué complejidad tiene tu algoritmo?\*\*

> "Dijkstra tiene complejidad O(V²) en el peor caso donde V es el número de vértices (ciudades). Con NetworkX está optimizado y en mi caso con solo 6 ciudades es instantáneo. Para grafos más grandes se puede optimizar a O((V+E)logV) con heaps."

### \*\*12. ¿Cómo escalas el sistema?\*\*

> "El diseño modular permite escalabilidad fácil: agregar más ciudades solo requiere insertarlas en la base de datos, el grafo se construye automáticamente. El patrón Repository facilita cambiar el almacenamiento. Los services permiten agregar algoritmos alternativos sin tocar la interfaz."

### \*\*13. ¿Qué medidas de seguridad implementaste?\*\*

> "Autenticación con Flask-Login, encriptación de contraseñas con bcrypt, autorización por roles (admin/usuario), validación de formularios con Flask-WTF, y manejo seguro de sesiones. Las rutas administrativas requieren decorador `@admin\_required`."

## 🎯 \*\*TIPS PARA LA PRESENTACIÓN:\*\*

1. \*\*Muestra el flujo completo\*\*: Login → Búsqueda → Resultado → Grafo

2. \*\*Demuestra el panel admin\*\*: CRUD de ciudades, rutas, usuarios

3. \*\*Explica la ruta fija\*\*: Ibarra→Loja como ejemplo predefinido

4. \*\*Muestra el código clave\*\*: `construir\_grafo()` y `camino\_optimo\_con\_costera()`

5. \*\*Destaca las buenas prácticas\*\*: MVC, Repository, entorno virtual

¡Con esta preparación puedes responder cualquier pregunta técnica del profesor! 🚀

Te explico \*\*TODO\*\* sobre los Repositories. Tu profesor no te enseñó porque es un patrón \*\*avanzado\*\* que se usa en desarrollo profesional:

## 🏗️ \*\*¿QUÉ SON LOS REPOSITORIES?\*\*

### \*\*📋 Definición Simple:\*\*

Un Repository es una \*\*"caja de herramientas"\*\* que contiene todas las formas de acceder y manipular datos de una tabla específica.

### \*\*🎯 ¿PARA QUÉ SIRVEN?\*\*

#### \*\*Problema SIN Repository:\*\*

```python

# En diferentes partes del código tienes que repetir:

users = User.query.all() # En admin\_routes.py

user = User.query.filter\_by(username='admin').first() # En auth\_routes.py

User.query.filter\_by(email='test@test.com').first() # En register.py

```

#### \*\*Solución CON Repository:\*\*

```python

# Una sola clase centraliza TODAS las consultas:

user\_repo = UserRepository()

users = user\_repo.get\_all()

user = user\_repo.get\_by\_username('admin')

user = user\_repo.get\_by\_email('test@test.com')

```

### \*\*🔥 ¿POR QUÉ SON IMPORTANTES?\*\*

1. \*\*DRY (Don't Repeat Yourself)\*\*: No repites código de consultas

2. \*\*Centralización\*\*: Todas las consultas en un lugar

3. \*\*Mantenimiento\*\*: Si cambias la base de datos, solo cambias el Repository

4. \*\*Testeo\*\*: Más fácil hacer pruebas unitarias

5. \*\*Profesionalismo\*\*: Es un patrón usado en empresas reales

---

## 🧩 \*\*ARQUITECTURA DE REPOSITORIES\*\*

### \*\*1. BaseRepository - La Clase Padre\*\*

```python

class BaseRepository:

def \_\_init\_\_(self, model):

self.model = model # User, Ciudad, Ruta, etc.

def create(self, \*\*kwargs):

"""Crear nuevo registro"""

instance = self.model(\*\*kwargs)

db.session.add(instance)

db.session.commit()

return instance

def get\_all(self):

"""Obtener todos los registros"""

return self.model.query.all()

def get\_by\_id(self, id):

"""Obtener por ID"""

return self.model.query.get(id)

def update(self, instance, \*\*kwargs):

"""Actualizar registro"""

for key, value in kwargs.items():

setattr(instance, key, value)

db.session.commit()

return instance

def delete(self, instance):

"""Eliminar registro"""

db.session.delete(instance)

db.session.commit()

def count(self):

"""Contar registros"""

return self.model.query.count()

```

\*\*¿Por qué BaseRepository?\*\*

- Todas las tablas necesitan operaciones básicas (CRUD)

- En lugar de repetir código, lo escribes una vez

- Cada Repository específico hereda estas funciones

---

## 🛣️ \*\*RUTA\_REPOSITORY EXPLICADO LÍNEA POR LÍNEA\*\*

### \*\*🏗️ Estructura Básica:\*\*

```python

class RutaRepository(BaseRepository):

def \_\_init\_\_(self):

super().\_\_init\_\_(Ruta) # Hereda de BaseRepository para tabla Ruta

```

\*\*¿Qué significa esto?\*\*

- `RutaRepository` hereda todas las funciones básicas de `BaseRepository`

- Automáticamente tiene: `create()`, `get\_all()`, `get\_by\_id()`, etc.

- Pero además agrega funciones \*\*específicas\*\* para rutas

---

### \*\*🔍 FUNCIONES ESPECÍFICAS DE RUTAS:\*\*

#### \*\*1. Buscar Ruta Entre Dos Ciudades:\*\*

```python

def get\_by\_ciudades(self, origen\_id: int, destino\_id: int) -> Optional[Ruta]:

return Ruta.query.filter\_by(

ciudad\_origen\_id=origen\_id,

ciudad\_destino\_id=destino\_id

).first()

```

\*\*¿Para qué?\*\*

- Verificar si existe ruta directa Quito→Guayaquil

- \*\*Uso real:\*\* "¿Hay conexión directa entre estas ciudades?"

#### \*\*2. Rutas Que Salen de Una Ciudad:\*\*

```python

def get\_desde\_ciudad(self, ciudad\_id: int) -> List[Ruta]:

return Ruta.query.filter\_by(ciudad\_origen\_id=ciudad\_id).all()

```

\*\*¿Para qué?\*\*

- Ver todas las ciudades a las que puedes ir desde Quito

- \*\*Uso real:\*\* "¿A dónde puedo viajar desde aquí?"

#### \*\*3. Rutas Que Llegan a Una Ciudad:\*\*

```python

def get\_hacia\_ciudad(self, ciudad\_id: int) -> List[Ruta]:

return Ruta.query.filter\_by(ciudad\_destino\_id=ciudad\_id).all()

```

\*\*¿Para qué?\*\*

- Ver desde dónde puedes llegar a Loja

- \*\*Uso real:\*\* "¿Desde dónde puedo llegar a esta ciudad?"

#### \*\*4. Todas las Rutas de Una Ciudad:\*\*

```python

def get\_rutas\_ciudad(self, ciudad\_id: int) -> List[Ruta]:

return Ruta.query.filter(

(Ruta.ciudad\_origen\_id == ciudad\_id) |

(Ruta.ciudad\_destino\_id == ciudad\_id)

).all()

```

\*\*¿Para qué?\*\*

- Ver TODAS las conexiones de una ciudad (saliendo Y llegando)

- \*\*Uso real:\*\* "¿Qué tan conectada está esta ciudad?"

---

### \*\*🔄 FUNCIONES DE BIDIRECCIONALIDAD:\*\*

#### \*\*5. Verificar Si Existe Ruta:\*\*

```python

def ruta\_existe(self, origen\_id: int, destino\_id: int) -> bool:

return self.get\_by\_ciudades(origen\_id, destino\_id) is not None

```

\*\*¿Para qué?\*\*

- Verificación rápida antes de agregar rutas

- \*\*Uso real:\*\* "¿Ya existe esta conexión?"

#### \*\*6. Rutas Bidireccionales:\*\*

```python

def get\_rutas\_bidireccionales(self) -> List[Tuple[Ruta, Ruta]]:

rutas = self.get\_all()

bidireccionales = []

for ruta in rutas:

# Buscar ruta inversa

ruta\_inversa = self.get\_by\_ciudades(ruta.ciudad\_destino\_id, ruta.ciudad\_origen\_id)

if ruta\_inversa and ruta.id < ruta\_inversa.id: # Evitar duplicados

bidireccionales.append((ruta, ruta\_inversa))

return bidireccionales

```

\*\*¿Para qué?\*\*

- Encontrar pares de rutas que van en ambas direcciones

- \*\*Ejemplo:\*\* Quito→Guayaquil Y Guayaquil→Quito

- \*\*Uso real:\*\* "¿Qué rutas permiten viaje de ida y vuelta?"

#### \*\*7. Rutas Unidireccionales:\*\*

```python

def get\_rutas\_unidireccionales(self) -> List[Ruta]:

rutas = self.get\_all()

unidireccionales = []

for ruta in rutas:

if not self.ruta\_existe(ruta.ciudad\_destino\_id, ruta.ciudad\_origen\_id):

unidireccionales.append(ruta)

return unidireccionales

```

\*\*¿Para qué?\*\*

- Encontrar rutas que solo van en una dirección

- \*\*Uso real:\*\* "¿Qué rutas necesitan conexión de regreso?"

---

### \*\*📊 FUNCIONES DE ANÁLISIS:\*\*

#### \*\*8. Ruta Más Corta y Más Larga:\*\*

```python

def get\_ruta\_mas\_corta(self) -> Optional[Ruta]:

return Ruta.query.order\_by(Ruta.peso.asc()).first()

def get\_ruta\_mas\_larga(self) -> Optional[Ruta]:

return Ruta.query.order\_by(Ruta.peso.desc()).first()

```

\*\*¿Para qué?\*\*

- Estadísticas del sistema

- \*\*Uso real:\*\* "¿Cuál es la conexión más corta del país?"

#### \*\*9. Rutas Ordenadas por Distancia:\*\*

```python

def get\_rutas\_ordenadas\_por\_distancia(self, ascendente: bool = True) -> List[Ruta]:

if ascendente:

return Ruta.query.order\_by(Ruta.peso.asc()).all()

else:

return Ruta.query.order\_by(Ruta.peso.desc()).all()

```

\*\*¿Para qué?\*\*

- Mostrar rutas de menor a mayor distancia o viceversa

- \*\*Uso real:\*\* Rankings de rutas por distancia

#### \*\*10. Matriz de Adyacencia:\*\*

```python

def get\_matriz\_adyacencia(self) -> dict:

rutas = self.get\_all()

matriz = {}

for ruta in rutas:

if ruta.ciudad\_origen\_id not in matriz:

matriz[ruta.ciudad\_origen\_id] = {}

matriz[ruta.ciudad\_origen\_id][ruta.ciudad\_destino\_id] = float(ruta.peso)

return matriz

```

\*\*¿Para qué?\*\*

- Crear representación matemática del grafo

- \*\*Resultado:\*\* `{1: {2: 420.0, 3: 180.0}, 2: {1: 420.0}}`

- \*\*Uso real:\*\* Algoritmos avanzados de grafos

#### \*\*11. Estadísticas Completas:\*\*

```python

def get\_estadisticas(self) -> dict:

total\_rutas = self.count()

bidireccionales = len(self.get\_rutas\_bidireccionales())

unidireccionales = len(self.get\_rutas\_unidireccionales())

ruta\_corta = self.get\_ruta\_mas\_corta()

ruta\_larga = self.get\_ruta\_mas\_larga()

return {

'total\_rutas': total\_rutas,

'rutas\_bidireccionales': bidireccionales \* 2,

'rutas\_unidireccionales': unidireccionales,

'distancia\_minima': float(ruta\_corta.peso) if ruta\_corta else 0,

'distancia\_maxima': float(ruta\_larga.peso) if ruta\_larga else 0

}

```

\*\*¿Para qué?\*\*

- Dashboard administrativo con métricas del sistema

- \*\*Uso real:\*\* Panel de control con estadísticas

---

## 🎯 \*\*¿CÓMO SE USA EN LA PRÁCTICA?\*\*

### \*\*En los Controllers/Routes:\*\*

```python

# En lugar de escribir consultas SQL complejas:

ruta\_repo = RutaRepository()

# Operaciones simples

todas\_rutas = ruta\_repo.get\_all()

ruta\_especifica = ruta\_repo.get\_by\_ciudades(1, 2)

# Operaciones complejas

rutas\_desde\_quito = ruta\_repo.get\_desde\_ciudad(1)

estadisticas = ruta\_repo.get\_estadisticas()

matriz = ruta\_repo.get\_matriz\_adyacencia()

```

### \*\*Ventajas:\*\*

1. \*\*Código limpio\*\* en los controladores

2. \*\*Reutilización\*\* de consultas complejas

3. \*\*Testeo fácil\*\* - mockeas el repository

4. \*\*Mantenimiento\*\* - cambios centralizados

---

## 🚀 \*\*PREGUNTAS QUE PUEDE HACER EL PROFESOR:\*\*

### \*\*1. "¿Por qué usas Repository?"\*\*

\*\*Tu respuesta:\*\*

> "Para separar la lógica de acceso a datos del resto de la aplicación. Centralizo todas las consultas de rutas en una clase, evito repetir código y facilito el mantenimiento. Es un patrón profesional que mejora la arquitectura."

### \*\*2. "¿Qué es la matriz de adyacencia?"\*\*

\*\*Tu respuesta:\*\*

> "Es una representación matemática del grafo donde cada ciudad es un índice y los valores son las distancias. Por ejemplo, `matriz[1][2] = 420` significa que de la ciudad 1 a la ciudad 2 hay 420 km. Es útil para algoritmos de grafos."

### \*\*3. "¿Cómo detectas rutas bidireccionales?"\*\*

\*\*Tu respuesta:\*\*

> "Recorro todas las rutas y para cada una busco si existe la ruta inversa. Si encuentro Quito→Guayaquil y también Guayaquil→Quito, las marco como bidireccionales. Uso `ruta.id < ruta\_inversa.id` para evitar duplicados."

### \*\*4. "¿Para qué sirven las estadísticas?"\*\*

\*\*Tu respuesta:\*\*

> "Para el panel administrativo. Permiten ver métricas como total de rutas, cuántas son bidireccionales, cuál es la distancia mínima y máxima. Ayuda a analizar la conectividad del sistema de rutas."

\*\*¡Con esta explicación entiendes completamente el patrón Repository y puedes responder cualquier pregunta!\*\* 🎉

Summarized conversation historyTe explico paso a paso el archivo `grafo\_service.py` para que entiendas completamente qué hace cada método y para qué sirve:

## ¿Qué es el GrafoService?

Es una clase que se encarga de \*\*analizar las rutas entre ciudades\*\* usando algoritmos de grafos. Piensa en las ciudades como puntos en un mapa y las rutas como líneas que los conectan.

## Métodos principales y para qué sirven:

### 1. `crear\_grafo()` - Base de todo

```python

def crear\_grafo(self) -> nx.Graph:

```

\*\*¿Qué hace?\*\* Crea un "mapa matemático" con:

- \*\*Nodos\*\* = Ciudades (con información como nombre, provincia, si es costera)

- \*\*Aristas\*\* = Rutas entre ciudades (con peso = distancia/costo)

\*\*¿Para qué sirve?\*\* Es la base para todos los cálculos. Sin esto no funcionaría nada.

### 2. `calcular\_ruta\_mas\_corta()` - GPS del sistema

```python

def calcular\_ruta\_mas\_corta(self, origen\_id: int, destino\_id: int)

```

\*\*¿Qué hace?\*\* Como un GPS, encuentra el camino más corto entre dos ciudades.

\*\*Ejemplo práctico:\*\*

- Usuario quiere ir de Quito a Guayaquil

- El método devuelve: "Quito → Latacunga → Ambato → Riobamba → Guayaquil"

- Con distancia total: 420 km

\*\*¿Para qué sirve?\*\* Para que los usuarios encuentren la ruta más eficiente.

### 3. `obtener\_rutas\_alternativas()` - Opciones de viaje

```python

def obtener\_rutas\_alternativas(self, origen\_id: int, destino\_id: int, k: int = 3)

```

\*\*¿Qué hace?\*\* Encuentra diferentes caminos para llegar al mismo destino.

\*\*¿Por qué es útil?\*\*

- Si hay tráfico en la ruta principal

- Si una carretera está cerrada

- Si quieres un camino más escénico (aunque sea más largo)

\*\*Ejemplo:\*\*

- Ruta 1: Quito → Guayaquil (directo) - 420 km

- Ruta 2: Quito → Cuenca → Guayaquil - 480 km

- Ruta 3: Quito → Manta → Guayaquil - 520 km

### 4. `validar\_ruta\_directa()` - ¿Hay vuelo directo?

```python

def validar\_ruta\_directa(self, origen\_id: int, destino\_id: int)

```

\*\*¿Qué hace?\*\* Verifica si existe una conexión directa entre dos ciudades.

\*\*¿Para qué sirve?\*\*

- Como preguntarte: "¿Hay vuelo directo o necesito hacer escalas?"

- En el sistema: "¿Puedo ir directo de Quito a Cuenca?"

- Si no hay ruta directa, necesitas pasar por otras ciudades

### 5. `obtener\_estadisticas\_grafo()` - Dashboard administrativo

```python

def obtener\_estadisticas\_grafo(self)

```

\*\*¿Qué hace?\*\* Analiza todo el sistema de rutas y da estadísticas generales.

\*\*Información que proporciona:\*\*

- \*\*Número total de ciudades y rutas\*\*

- \*\*¿Están todas las ciudades conectadas?\*\* (puedes llegar a cualquier lugar)

- \*\*Ciudad más conectada\*\* (como un "hub" de aeropuerto)

- \*\*Diámetro del grafo\*\* (la distancia más larga entre cualquier par de ciudades)

\*\*¿Para qué sirve?\*\* Para administradores que quieren entender:

- ¿Qué tan bien conectado está el país?

- ¿Qué ciudades son más importantes?

- ¿Hay ciudades aisladas?

### 6. `obtener\_ciudades\_por\_conectividad()` - Ranking de importancia

```python

def obtener\_ciudades\_por\_conectividad(self)

```

\*\*¿Qué hace?\*\* Ordena las ciudades por cuántas conexiones tienen.

\*\*Ejemplo de resultado:\*\*

1. Quito - 8 conexiones

2. Guayaquil - 6 conexiones

3. Cuenca - 4 conexiones

4. Loja - 2 conexiones

\*\*¿Para qué sirve?\*\*

- \*\*Para planificación urbana:\*\* Las ciudades más conectadas son más importantes económicamente

- \*\*Para turismo:\*\* Ciudades con más conexiones son más fáciles de visitar

- \*\*Para logística:\*\* Mejor ubicar centros de distribución en ciudades bien conectadas

## ¿Cómo se usa todo esto en la aplicación?

### En la interfaz de usuario:

1. \*\*Formulario de rutas:\*\* Usuario selecciona origen y destino → usa `calcular\_ruta\_mas\_corta()`

2. \*\*"Ver alternativas"\*\* → usa `obtener\_rutas\_alternativas()`

3. \*\*Dashboard admin\*\* → usa `obtener\_estadisticas\_grafo()`

### En el código:

```python

# En las rutas de Flask

from models.services.grafo\_service import GrafoService

@app.route('/calcular-ruta')

def calcular\_ruta():

grafo\_service = GrafoService()

resultado = grafo\_service.calcular\_ruta\_mas\_corta(origen\_id, destino\_id)

return render\_template('resultado.html', ruta=resultado)

```

## Algoritmos que usa internamente:

- \*\*NetworkX biblioteca:\*\* Para manejar grafos

- \*\*Dijkstra:\*\* Para encontrar rutas más cortas (lo hace NetworkX automáticamente)

- \*\*Análisis de conectividad:\*\* Para estadísticas del grafo

¿Te queda más claro ahora? ¿Hay algún método específico que quieras que explique más a fondo?

Te explico paso a paso cómo funciona el sistema de autenticación en los templates y qué hace cada uno:

## 🔐 \*\*Sistema de Autenticación - Visión General\*\*

El sistema de autenticación tiene \*\*2 templates principales\*\* que manejan el \*\*login\*\* y \*\*registro\*\* de usuarios.

## 📁 \*\*Templates de Autenticación\*\*

### 1. \*\*login.html\*\* - Pantalla de Inicio de Sesión ### 2. \*\*register.html\*\* - Pantalla de Registro Ahora voy a revisar cómo funciona el template base para entender el contexto: Ahora voy a revisar las rutas de autenticación para entender el flujo: ## 🎯 \*\*¿Qué hace cada template de autenticación?\*\*

### 🔑 \*\*1. login.html - Pantalla de Login\*\*

\*\*¿Qué muestra?\*\*

- \*\*Formulario de inicio de sesión\*\* con campos:

- Nombre de usuario

- Contraseña

- \*\*Enlace para registrarse\*\* si no tienes cuenta

- \*\*Credenciales de prueba\*\* (admin/admin123)

\*\*¿Cuándo aparece?\*\*

- Cuando vas a `/login`

- Cuando intentas acceder a una página protegida sin estar logueado

- Cuando haces clic en "Iniciar Sesión" en el navbar

\*\*¿Qué hace cuando envías el formulario?\*\*

1. Valida usuario y contraseña

2. Si es \*\*admin\*\* → redirige al \*\*Panel Admin\*\*

3. Si es \*\*usuario normal\*\* → redirige al \*\*Home\*\*

4. Si hay error → muestra mensaje de error

### 📝 \*\*2. register.html - Pantalla de Registro\*\*

\*\*¿Qué muestra?\*\*

- \*\*Formulario de registro\*\* con campos:

- Nombre de usuario (mínimo 3 caracteres)

- Email

- Contraseña (mínimo 6 caracteres)

- Confirmar contraseña

- \*\*Enlace para login\*\* si ya tienes cuenta

\*\*¿Cuándo aparece?\*\*

- Cuando vas a `/register`

- Cuando haces clic en "Registrarse" en el navbar

- Cuando haces clic en "Regístrate aquí" desde el login

\*\*¿Qué hace cuando envías el formulario?\*\*

1. Valida todos los campos

2. Verifica que las contraseñas coincidan

3. Crea el usuario en la base de datos

4. Redirige al login para que inicies sesión

## 🔄 \*\*Flujo completo de autenticación:\*\*

### \*\*Usuario NO autenticado:\*\*

```

1. Ve navbar → botones "Iniciar Sesión" + "Registrarse"

2. En sidebar → aparece "Invitado"

3. Solo ve: Home, Contacto

```

### \*\*Usuario autenticado normal:\*\*

```

1. Ve navbar → dropdown con su nombre + "Cerrar Sesión"

2. En sidebar → su nombre y rol "User"

3. Ve: Home, Buscar Rutas, Contacto

```

### \*\*Usuario admin:\*\*

```

1. Ve navbar → dropdown + enlace "Panel Admin"

2. En sidebar → su nombre y rol "Admin"

3. Ve: Home, Buscar Rutas, Panel Admin, Contacto

```

## 🛡️ \*\*Sistema de protección en templates:\*\*

\*\*En base.html hay lógica condicional:\*\*

```html

{% if current\_user.is\_authenticated %}

<!-- Usuario logueado: muestra menú completo -->

{% if current\_user.is\_admin() %}

<!-- Admin: muestra Panel Admin -->

{% endif %}

{% else %}

<!-- Usuario no logueado: muestra Login/Register -->

{% endif %}

```

## 🎨 \*\*Estilos y diseño:\*\*

- \*\*Login:\*\* Usa \*\*tema azul\*\* (card-primary)

- \*\*Register:\*\* Usa \*\*tema verde\*\* (card-success)

- \*\*Credenciales de prueba:\*\* Solo en login para facilitar testing

- \*\*Enlaces cruzados:\*\* Desde login puedes ir a register y viceversa

¿Te queda claro cómo funciona todo el sistema de autenticación?