Pontificia Universidad Católica del Ecuador – Sede Ambato

Informe Técnico - Sistema de Rutas con Grafos

Realizado por: Joaquín Esteban Bermeo Quispe

Materia: Programación IV

Docente: Edison Meneses

Fecha de Entrega: 18-06-2025

Contenido

1. Introducción

El presente informe detalla el desarrollo de un sistema de gestión y cálculo de rutas, implementado en lenguaje Python utilizando el framework Flask para la creación de la interfaz web. El sistema ha sido diseñado con el objetivo de facilitar la administración de una red de ubicaciones geográficas, permitiendo a los usuarios registrar provincias, ciudades y las rutas que las conectan. A través de una interfaz intuitiva desarrollada con HTML y Flask, los usuarios pueden crear, modificar, visualizar o eliminar tanto nodos como conexiones del sistema.

Uno de los componentes centrales del proyecto es la aplicación del algoritmo de Dijkstra, el cual permite calcular la ruta óptima entre dos puntos dentro de la red de ciudades registrada. Además, el sistema incluye la generación automática de gráficos que representan visualmente las rutas establecidas, ofreciendo así una perspectiva clara y organizada del mapa de conexiones.

Como complemento final, se ha integrado una funcionalidad que permite generar un documento en formato PDF, el cual resume detalladamente el cálculo de la ruta óptima, los puntos involucrados y sus respectivas distancias. Este enfoque no solo brinda una herramienta interactiva y funcional, sino que también fortalece el análisis y documentación de las rutas gestionadas dentro del sistema.

2. Arquitectura del Sistema

2.1 Estructura del Proyecto

```
Proyecto_Final_JB/
├── controllers/        # Controladores MVC
│   ├── admin_controller.py
│   ├── auth_controller.py
│   ├── grafo_controller.py
│   └── home_controller.py
├── models/             # Modelos de datos
│   ├── ciudad.py
│   ├── provincia.py
│   ├── ruta.py
│   └── user.py
├── routes/             # Definición de rutas web
│   ├── admin_routes.py
│   ├── auth_routes.py
│   ├── grafos_routes.py
│   └── home_routes.py
├── static/             # Archivos estáticos (CSS, JS, imágenes)
├── templates/          # Plantillas HTML
├── utils/              # Utilidades
│   └── grafo_db_utils.py
└── app.py              # Aplicación principal
```

2.2 Patrón de Arquitectura

El sistema implementa el patrón **MVC (Modelo-Vista-Controlador)**:

- Modelo: Clases en */models/* que representan las entidades de datos
- Vista: Plantillas HTML en */templates/*
- Controlador: Lógica de negocio en */controllers/*

3. Componentes Principales

3.1 Modelos de Datos

- Ciudad (ciudad.py)
    - Representa una ciudad en el grafo con sus atributos:
        - ID único
        - Nombre
        - Provincia asociada
- Ruta (ruta.py)
    - Representa las conexiones entre ciudades:
        - Ciudad origen
        - Ciudad destino
        - Costo
        - Tipo de ruta
- Provincia (provincia.py)
    - Agrupa ciudades por división territorial:
        - ID y nombre de provincia
        - Lista de ciudades asociadas

3.2 Controladores

- Admin Controller (admin_controller.py)
    - Maneja las funcionalidades administrativas del sistema:
        - **Gestión de usuarios:** Crear, editar, eliminar y listar usuarios
        - **Administración de ciudades:** CRUD completo de ciudades con validaciones
        - **Gestión de rutas:** Crear y modificar conexiones entre ciudades
        - **Panel de control:** Dashboard con estadísticas del sistema
        - **Validación de permisos:** Verificación de roles administrativos

- Auth Controller (auth_controller.py)
    - Controla la autenticación y autorización de usuarios:
        - **Login de usuarios:** Validación de credenciales y creación de sesiones
        - **Registro de nuevos usuarios:** Creación de cuentas con validaciones
        - **Logout:** Cierre seguro de sesiones
        - **Recuperación de contraseñas:** Proceso de reset de credenciales
        - **Middleware de autenticación:** Protección de rutas que requieren login
- Grafo Controller (grafo_controller.py)
    - Contiene la lógica principal para el manejo de grafos:
        - **Construcción del grafo:** Generación desde la base de datos
        - **Implementación del algoritmo de Dijkstra:** Cálculo de rutas óptimas
        - **Validación de entrada:** Verificación de ciudades origen y destino
        - **Procesamiento de resultados:** Formateo de caminos y costos
        - **Visualización de grafos:** Generación de imágenes del grafo
        - **Estadísticas del grafo:** Cálculo de métricas y análisis de conectividad
        - **Manejo de ciudades costeras:** Lógica específica para rutas que incluyan costa
- Home Controller (home_controller.py)
    - Gestiona las páginas principales y navegación del sitio:
        - **Página principal:** Renderizado del homepage con información general
        - **Navegación:** Control de menús y enlaces principales
        - **Presentación del sistema:** Información sobre funcionalidades
        - **Dashboard público:** Estadísticas generales sin requerir login
        - **Páginas informativas:** Secciones de ayuda y documentación
        - **Integración de componentes:** Coordinación entre diferentes módulos del sistema

3.3 Utilidades de Grafo (grafo_db_utils.py)

- Funciones auxiliares para:
    - Conexión con base de datos
    - Conversión de datos a estructura de grafo
    - Validación de datos de entrada

4. Algoritmo de Dijkstra

4.1 Descripción del Algoritmo

El algoritmo de Dijkstra encuentra el camino más corto entre dos nodos en un grafo ponderado con pesos no negativos. En este proyecto se utiliza para calcular rutas óptimas entre ciudades donde cada arista representa la distancia o costo de viaje.

4.2 Implementación en el Proyecto

El sistema utiliza la librería NetworkX de Python para implementar el algoritmo de Dijkstra de manera eficiente. La implementación se encuentra principalmente en el archivo *grafo_db_utils.py*.

4.2.1 Construcción del Grafo

```python
def construir_grafo():
    """Construye el grafo no dirigido y ponderado desde la base de datos"""
    G = nx.Graph()  # Grafo no dirigido

    # Obtener todas las rutas de la base de datos
    rutas = Ruta.query.all()

    for ruta in rutas:
        origen = ruta.ciudad_origen.nombre
        destino = ruta.ciudad_destino.nombre
        costo = float(ruta.costo)
        # En un grafo no dirigido, una arista conecta en ambas direcciones automáticamente
        G.add_edge(origen, destino, weight=costo)

    return G
```

4.2.2 Cálculo de Camino Óptimo

```python
def camino_optimo_con_costera(origen='Ibarra', destino='Loja'):
    """
    Calcula el camino óptimo entre dos ciudades que pase por al menos una ciudad costera
    Utiliza el algoritmo de Dijkstra implementado en NetworkX
    """
    G = construir_grafo()
    costeras = obtener_ciudades_costeras()

    try:
        # Aplicación directa del algoritmo de Dijkstra
        camino = nx.dijkstra_path(G, origen, destino, weight='weight')
        costo = nx.dijkstra_path_length(G, origen, destino, weight='weight')

        # Verificar si el camino pasa por al menos una ciudad costera
        contiene_costera = any(ciudad in costeras for ciudad in camino)

        return {
            "camino": camino,
            "costo": costo,
            "valido": contiene_costera,
            "ciudades_costeras_en_ruta": [c for c in camino if c in costeras]
        }
    except nx.NetworkXNoPath:
        return {
            "camino": [],
            "costo": None,
            "valido": False,
            "ciudades_costeras_en_ruta": []
        }
```

4.2.3 Validación de Entrada

```python
def validar_ciudades_existen(origen, destino):
    """Valida que las ciudades de origen y destino existan en la base de datos"""
    ciudad_origen = Ciudad.buscar_por_nombre(origen)
    ciudad_destino = Ciudad.buscar_por_nombre(destino)

    if not ciudad_origen:
        return False, f"La ciudad de origen '{origen}' no existe"

    if not ciudad_destino:
        return False, f"La ciudad de destino '{destino}' no existe"

    return True, "Ciudades válidas"
```

4.3 Características del Grafo Implementado

4.3.1 Tipo de Grafo

- **No Dirigido (Undirected):** Las rutas son bidireccionales
- **Ponderado:** Cada arista tiene un peso (costo/distancia)
- **Conexo:** Se asume que existe al menos un camino entre cualquier par de ciudades

4.3.2 Estructura de Datos

```
# Ejemplo de estructura del grafo en NetworkX:
G = {
    'Quito': {'Ambato': {'weight': 120}, 'Latacunga': {'weight': 90}},
    'Ambato': {'Quito': {'weight': 120}, 'Riobamba': {'weight': 50}},
    'Latacunga': {'Quito': {'weight': 90}, 'Riobamba': {'weight': 65}},
    # ... más conexiones
}
```

4.4 Ventajas de Usar NetworkX

4.4.1 Eficiencia

- **Implementación optimizada:** NetworkX usa heap binario internamente
- **Memoria optimizada:** Manejo eficiente de grafos grandes

4.4.2 Funcionalidades Adicionales

```
# Funciones adicionales disponibles:
camino = nx.dijkstra_path(G, origen, destino, weight='weight')          # Camino óptimo
costo = nx.dijkstra_path_length(G, origen, destino, weight='weight')    # Costo mínimo
existe = nx.has_path(G, origen, destino)                                  # Verificar conectividad
```

4.5 Integración con Base de Datos

4.5.1 Flujo de Datos

```
Base de Datos → Modelos ORM → Construcción Grafo → Dijkstra → Resultados
```

4.5.2 Obtención de Conexiones

```python
def obtener_rutas_desde_ciudad(ciudad_nombre):
    """Obtiene todas las rutas conectadas a una ciudad específica"""
    ciudad = Ciudad.buscar_por_nombre(ciudad_nombre)
    if not ciudad:
        return []

    rutas = Ruta.obtener_por_ciudad(ciudad.id)
    conexiones = []

    for ruta in rutas:
        ciudad_conectada = ruta.get_ciudad_conectada(ciudad.id)
        if ciudad_conectada:
            conexiones.append((ciudad_conectada.nombre, float(ruta.costo)))

    return conexiones
```

4.6 Manejo de Errores

4.6.1 Casos de Error Contemplados

- **NetworkXNoPath:** No existe camino entre origen y destino
- **Ciudades inexistentes:** Validación previa de entrada
- **Grafo vacío:** Verificación de datos en base de datos

4.6.2 Respuestas del Sistema

```python
# Respuesta exitosa:
{
    "camino": ["Quito", "Ambato", "Riobamba", "Guayaquil"],
    "costo": 350.5,
    "valido": True,
    "ciudades_costeras_en_ruta": ["Guayaquil"]
}

# Respuesta de error:
{
    "camino": [],
    "costo": None,
    "valido": False,
    "ciudades_costeras_en_ruta": []
}
```

4.7 Visualización del Resultado

4.7.1 Generación de Imagen con Camino Resaltado

```python
def grafo_a_imagen_camino(camino):
    """Genera la imagen del grafo con el camino resaltado"""
    G = construir_grafo()
    pos = nx.spring_layout(G, seed=8)

    # Resaltar el camino encontrado por Dijkstra
    if camino and len(camino) > 1:
        edges_camino = list(zip(camino[:-1], camino[1:]))
        nx.draw_networkx_edges(G, pos, edgelist=edges_camino,
                               edge_color='red', width=4)
        nx.draw_networkx_nodes(G, pos, nodelist=camino,
                               node_color='orange', node_size=2200)
```

4.8 Ejemplo de Uso Completo

```python
# 1. Usuario selecciona ciudades
origen = "Quito"
destino = "Guayaquil"

# 2. Validar entrada
valido, mensaje = validar_ciudades_existen(origen, destino)
if not valido:
    return {"error": mensaje}

# 3. Calcular ruta óptima
resultado = camino_optimo_con_costera(origen, destino)

# 4. Resultado:
# {
#   "camino": ["Quito", "Latacunga", "Ambato", "Riobamba", "Guayaquil"],
#   "costo": 270.0,
#   "valido": True,
#   "ciudades_costeras_en_ruta": ["Guayaquil"]
# }
```

Esta implementación aprovecha la robustez y eficiencia de NetworkX mientras mantiene la flexibilidad para trabajar con datos dinámicos desde la base de datos.

5. Funcionalidades del Sistema

5.1 Gestión de Datos

- **CRUD de Ciudades:** Crear, leer, actualizar y eliminar ciudades
- **CRUD de Rutas:** Gestión de conexiones entre ciudades con sus distancias
- **Validación de Datos:** Verificación de integridad y consistencia

5.2 Cálculo de Rutas

- **Ruta Más Corta:** Utilizando algoritmo de Dijkstra
- **Distancia Total:** Cálculo preciso de kilómetros
- **Ciudades Intermedias:** Lista ordenada de paradas en la ruta óptima

5.3 Visualización

- **Interfaz Web Intuitiva:** Formularios para selección de origen y destino
- **Resultados Detallados:** Distancia total, ciudades intermedias, tiempo estimado
- **Representación del Camino:** Visualización step-by-step de la ruta

6. Flujo de Funcionamiento

6.1 Proceso de Cálculo de Rutas

1. **Entrada del Usuario:** Selección de ciudad origen y destino
2. **Construcción del Grafo:** Carga de ciudades y rutas desde base de datos
3. **Aplicación de Dijkstra:** Ejecución del algoritmo para encontrar camino óptimo
4. **Procesamiento de Resultados:** Formateo de la ruta y cálculo de totales
5. **Presentación:** Muestra de resultados en interfaz web

6.2 Ejemplo de Uso

```
Usuario selecciona: Quito → Guayaquil

Sistema construye grafo:
{
  'Quito': {'Ambato': 120, 'Latacunga': 90},
  'Ambato': {'Quito': 120, 'Riobamba': 50, 'Latacunga': 45},
  'Latacunga': {'Quito': 90, 'Ambato': 45, 'Riobamba': 65},
  'Riobamba': {'Ambato': 50, 'Latacunga': 65, 'Guayaquil': 180},
  'Guayaquil': {'Riobamba': 180}
}

Dijkstra encuentra: Quito → Latacunga → Ambato → Riobamba → Guayaquil
Distancia total: 270 km
```

7. Base de Datos

7.1 Esquema de Tablas

```sql
-- Tabla de provincias
CREATE TABLE provincias (
    id INTEGER PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL
);

-- Tabla de ciudades
CREATE TABLE ciudades (
    id INTEGER PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    provincia_id INTEGER,
    latitud DECIMAL(10,8),
    longitud DECIMAL(11,8),
    FOREIGN KEY (provincia_id) REFERENCES provincias(id)
);

-- Tabla de rutas
CREATE TABLE rutas (
    id INTEGER PRIMARY KEY,
    ciudad_origen_id INTEGER,
    ciudad_destino_id INTEGER,
    distancia DECIMAL(10,2) NOT NULL,
    tiempo_estimado INTEGER,
    FOREIGN KEY (ciudad_origen_id) REFERENCES ciudades(id),
    FOREIGN KEY (ciudad_destino_id) REFERENCES ciudades(id)
);
```

8. Tecnologías Utilizadas

- **Backend:** Python 3.13.5, Flask
- **Base de Datos:** MySQL
- **Frontend:** HTML5, CSS3, JavaScript
- **Librerías:**
    - Flask-SQLAlchemy (ORM)
    - Flask-Login (Autenticación)
    - Jinja2 (Plantillas)

9. Consideraciones de Rendimiento

9.1 Optimizaciones para Dijkstra

- **Terminación Temprana**: El algoritmo se detiene al encontrar el destino
- **Caché de Grafos:** Evita reconstrucción innecesaria del grafo
- **Validación Previa:** Verifica que origen y destino existan antes del cálculo

9.2 Escalabilidad

- Para grafos grandes, se puede implementar Dijkstra con heap (priority queue)
- Posibilidad de implementar A* para casos específicos con heurística
- Base de datos indexada para consultas rápidas

10. Conclusiones

En conclusión, el desarrollo de este sistema ha permitido integrar de forma efectiva múltiples componentes tecnológicos para resolver un problema real de planificación de rutas. La combinación de una arquitectura modular basada en el patrón MVC, una interfaz web amigable con Flask y HTML, junto con la robustez del algoritmo de Dijkstra implementado mediante la biblioteca NetworkX, da como resultado una herramienta poderosa y escalable. Además, la generación de representaciones gráficas y reportes PDF aporta valor agregado al usuario final, facilitando la interpretación y documentación de los resultados. Este proyecto no solo representa un ejercicio académico completo, sino también una base sólida para futuras mejoras o aplicaciones reales en ámbitos de transporte, logística o planificación territorial.

10.1 Fortalezas

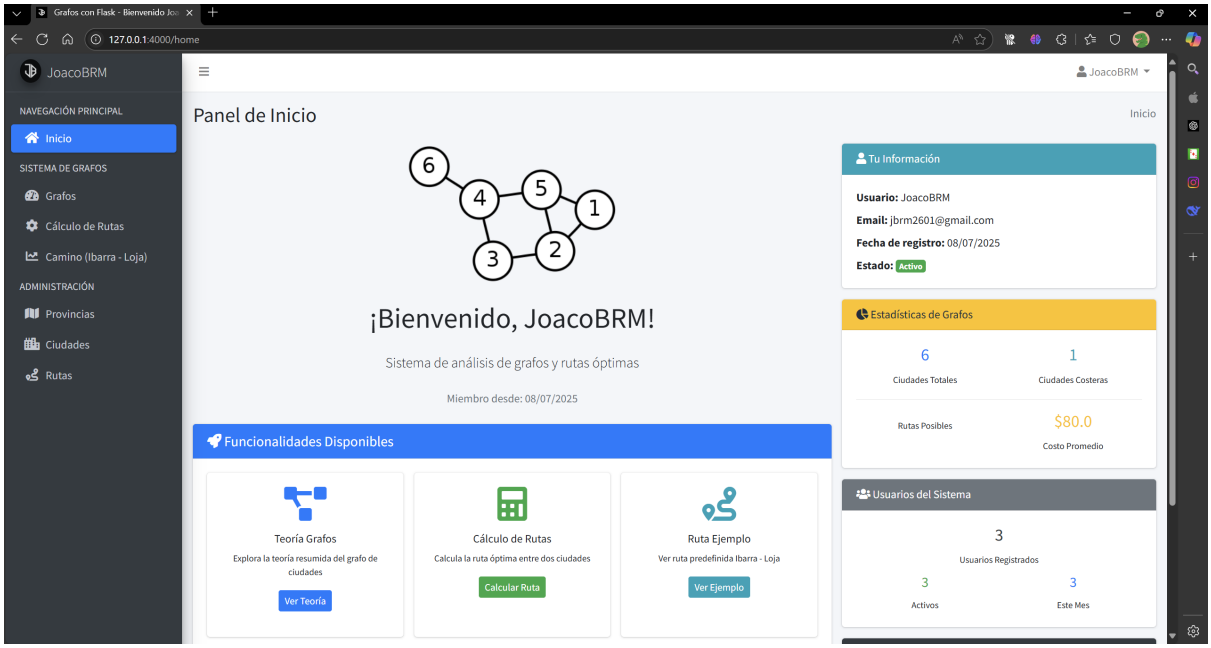
- Implementación correcta y eficiente de Dijkstra
- Garantía de encontrar la ruta más corta
- Interfaz de usuario clara e intuitiva
- Código modular y mantenible

10.2 Aplicaciones Prácticas

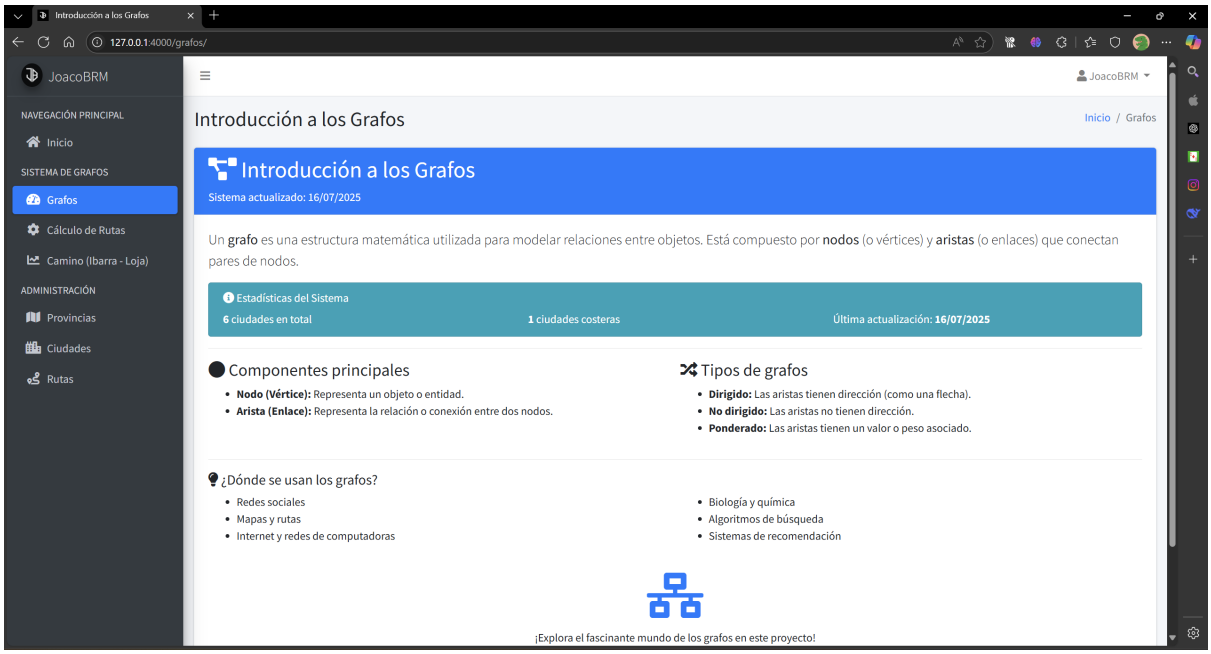
- Sistemas de navegación GPS
- Planificación de rutas logísticas
- Optimización de costos de transporte
- Análisis de redes de comunicación

11. Anexos

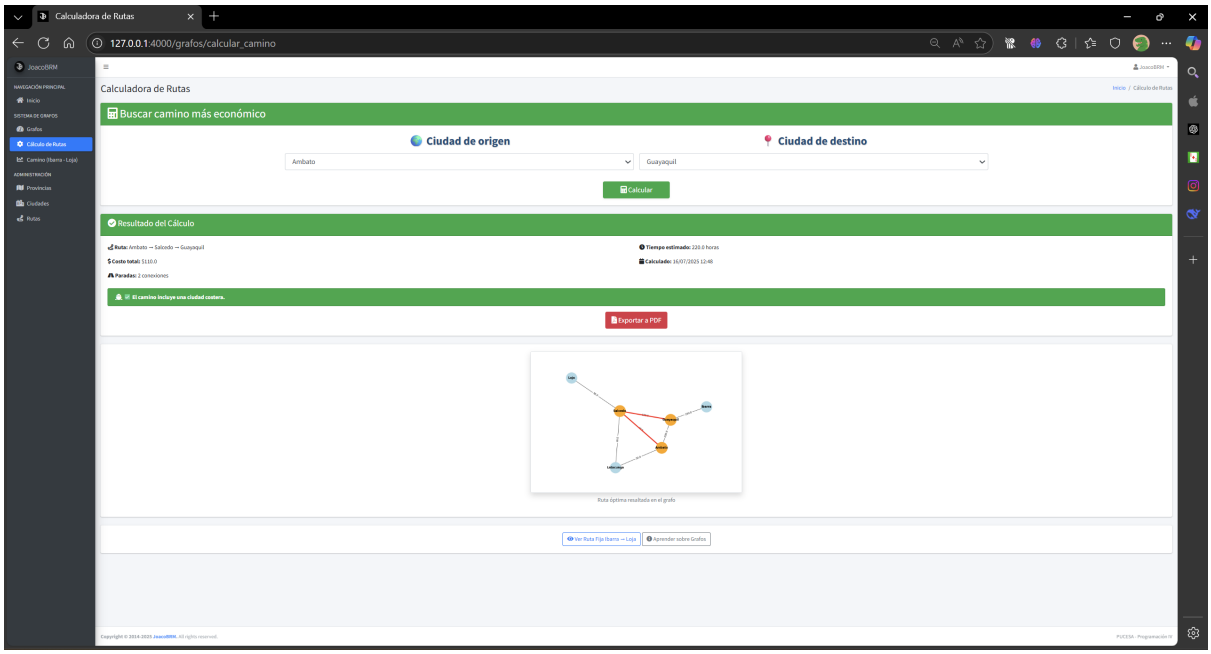
11.1 Panel de Inicio



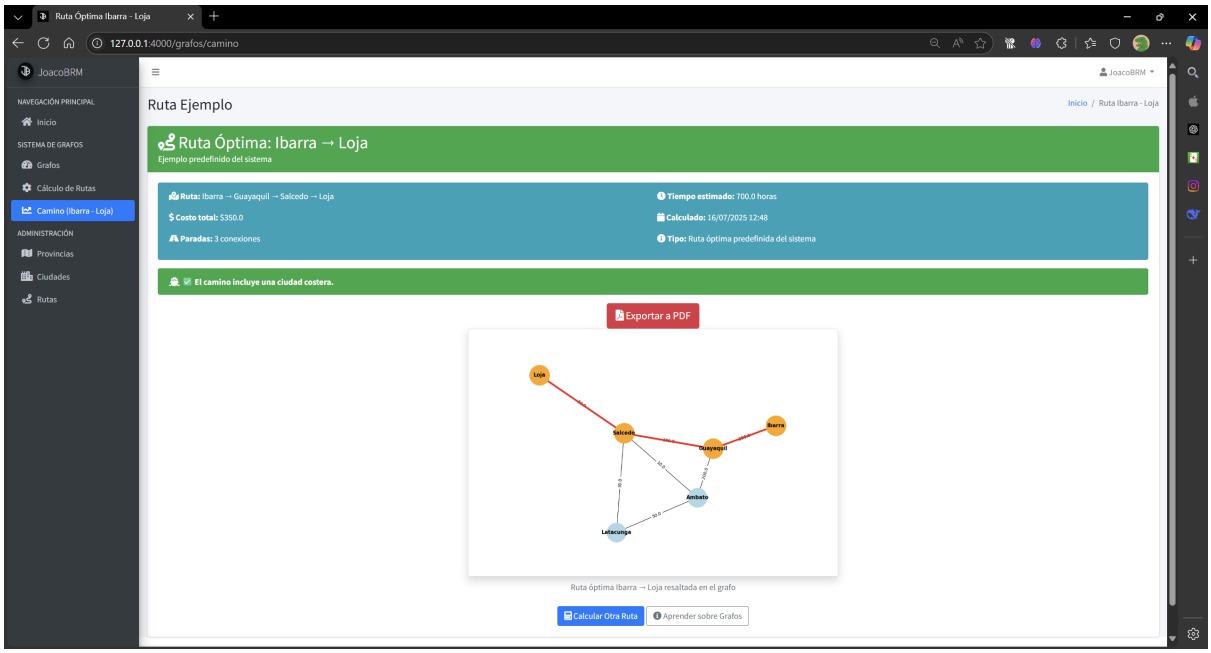
11.2 Introducción a los Grafos



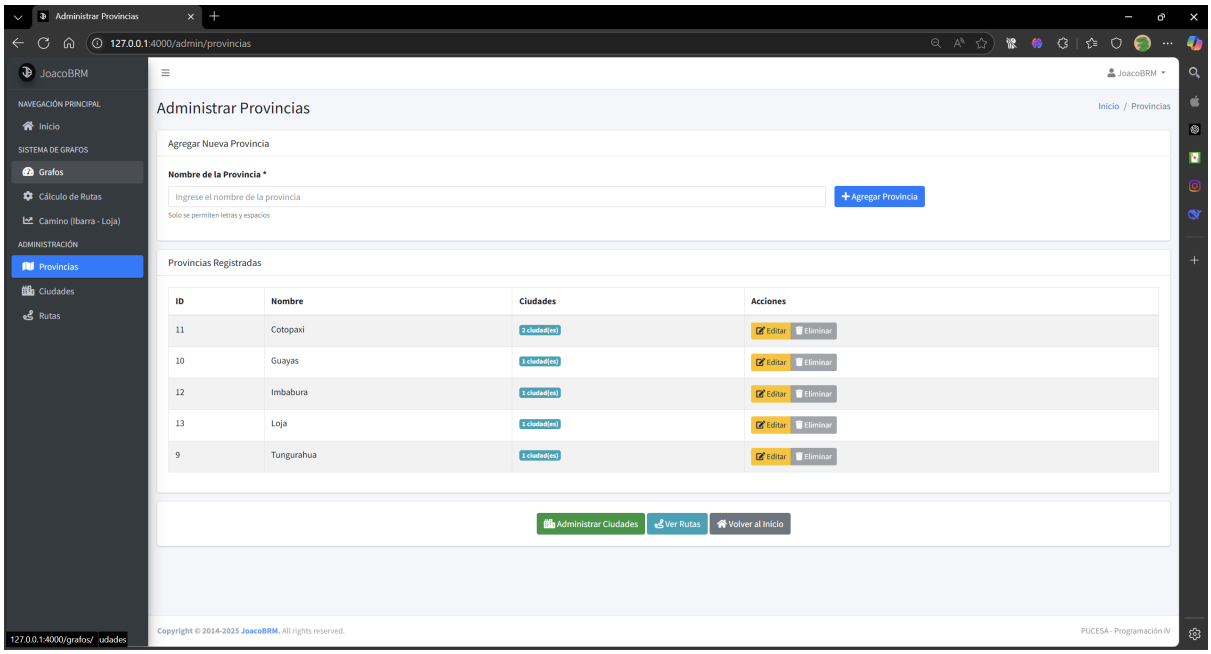
11.3 Calculadora de Rutas



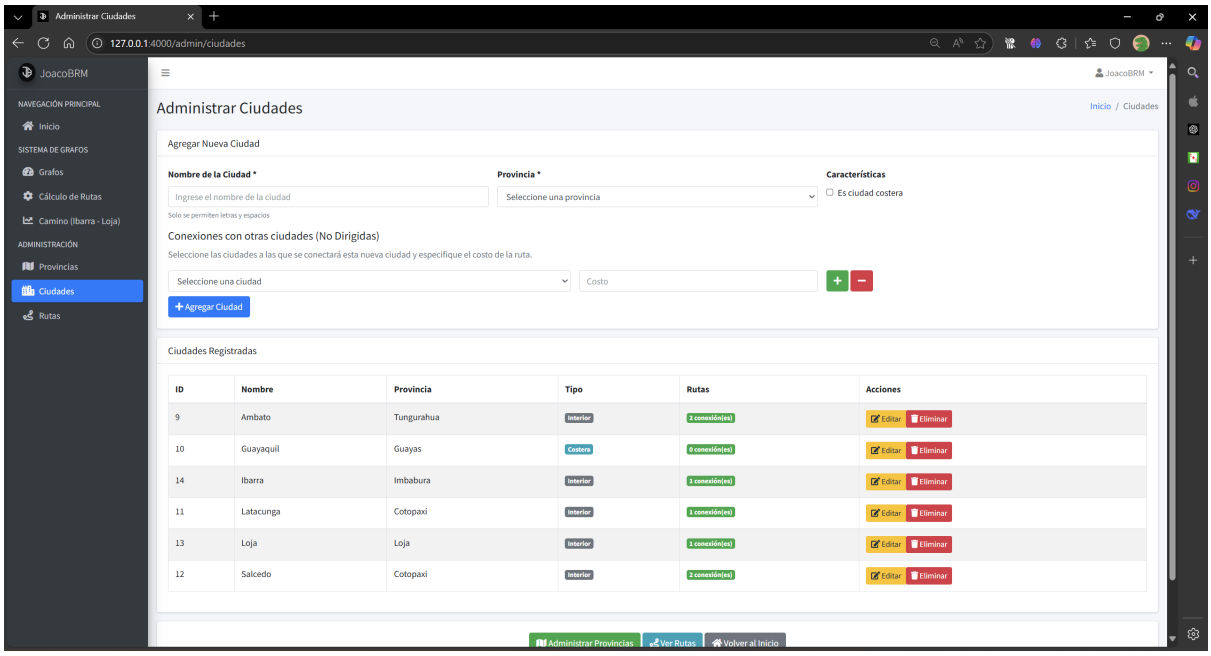
11.4 Ruta Fija: Ibarra → Loja



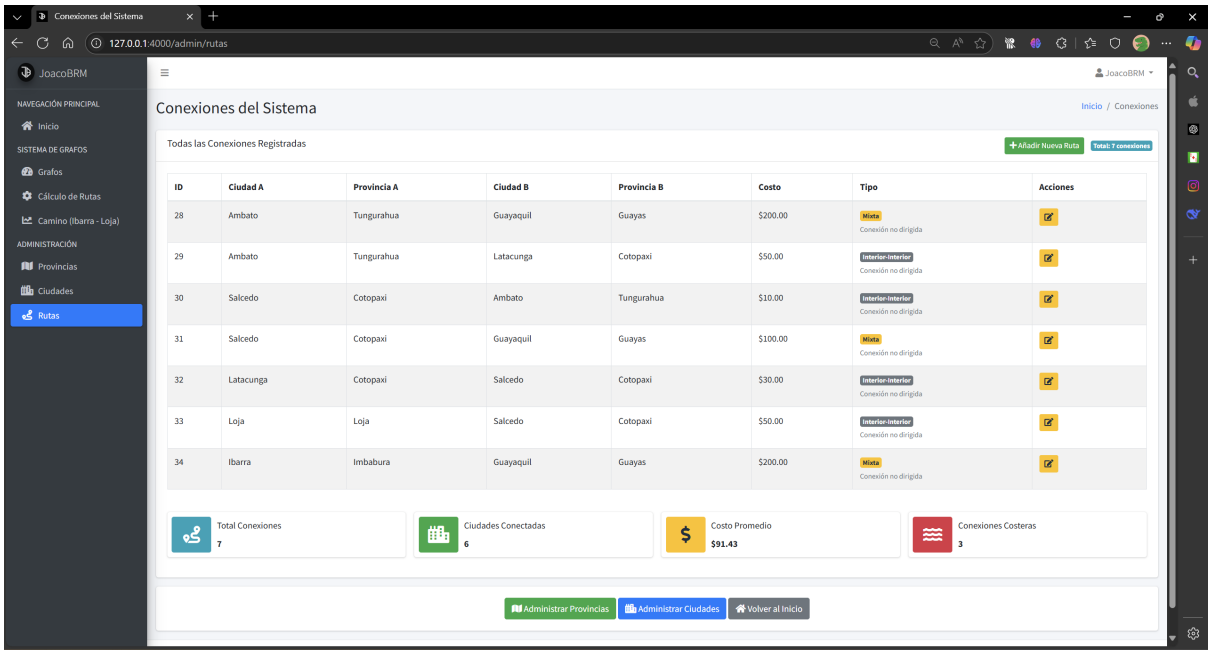
11.5 Administrar Provincias



11.6 Administrar Ciudades



11.7 Administración de Rutas



11.8 Enlace al repositorio del proyecto

https://github.com/JoacoBRM/Proyecto_Final_JB