Pontificia Universidad Católica del Ecuador | Sede Ambato

Sistemas de la Información

Programación IV

Docente: Edison Fernando Meneses Torres

Estudiante: Emilio Jiménez

INFORME TÉCNICO - SISTEMA RUTIX

Jueves, 17 de julio 2025

Contenidos

INTRODUCCIÓN

RUTIX es un sistema web desarrollado en Flask que permite la gestión y visualización de rutas entre ciudades utilizando algoritmos de grafos. El sistema implementa el algoritmo de Dijkstra para encontrar el camino más corto entre dos ciudades, con la particularidad de que las rutas deben pasar por ciudades costeras.

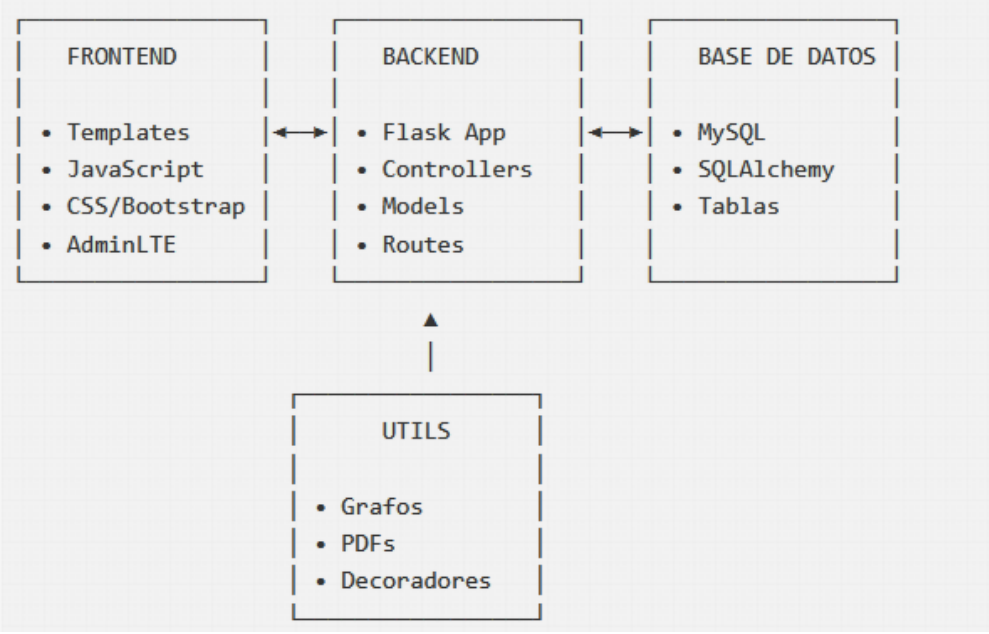
Objetivos del Sistema:

- Gestionar ciudades, provincias y conexiones entre ciudades
- Calcular rutas óptimas utilizando algoritmos de grafos
- Visualizar grafos de manera interactiva
- Administrar usuarios con diferentes niveles de permisos
- Generar reportes en PDF

ARQUITECTURA DEL SISTEMA

El sistema RUTIX sigue el patrón de arquitectura MVC (Modelo-Vista-Controlador) con una estructura modular que separa las responsabilidades:

Arquitectura General



Componentes Principales

1. **Capa de Presentación (Templates)**
    a. Interfaz de usuario basada en HTML/CSS/JavaScript
    b. Framework AdminLTE para diseño responsivo
    c. Visualización de grafos con matplotlib
2. **Capa de Lógica de Negocio (Controllers)**
    a. Procesamiento de formularios
    b. Lógica de autenticación
3. **Capa de Datos (Models)**
    a. Modelos SQLAlchemy
    b. Relaciones entre entidades
    c. Operaciones CRUD
4. **Capa de Rutas (Routes)**
    a. Blueprints para organización modular
    b. Endpoints REST
    c. Manejo de peticiones HTTP
5. **Capa de Utilidades (Utils)**
    a. Algoritmos de grafos (NetworkX + Matplotlib)
    b. Generación de PDFs (xhtml2pdf)
    c. Decoradores de seguridad y permisos

TECNOLOGÍAS UTILIZADAS

Backend

- **Flask:** Framework web principal
- **SQLAlchemy:** ORM para base de datos
- **Flask-Login:** Gestión de autenticación
- **NetworkX:** Biblioteca para algoritmos de grafos
- **Matplotlib:** Generación de gráficos
- **PyMySQL:** Conector para MySQL
- **xhtml2pdf:** Generación de documentos PDF desde HTML

Frontend

- **HTML5/CSS3:** Estructura y estilos
- **JavaScript:** Interactividad del lado cliente
- **Bootstrap:** Framework CSS responsivo
- **AdminLTE:** Template administrativo
- **jQuery:** Manipulación del DOM

Base de Datos

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

Herramientas de Desarrollo

- **Python 3.13:** Lenguaje de programación
- **pip:** Gestor de paquetes
- **python-dotenv:** Gestión de variables de entorno

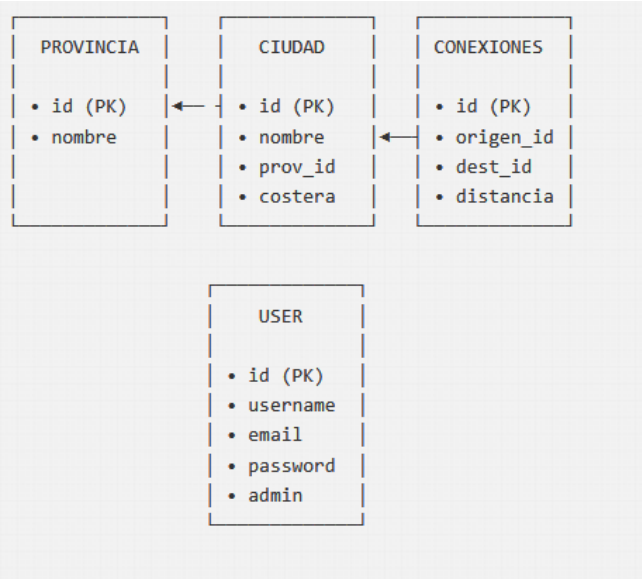
ESTRUCTURA DEL PROYECTO

```
Rutix/
├── app.py                      # Punto de entrada de la aplicación
├── config.py                   # Configuración de la aplicación
├── extensions.py               # Extensiones de Flask (DB, Login)
├── requirements.txt            # Dependencias del proyecto
├── rutix.sql                   # Schema de base de datos
│
├── controllers/                # Lógica de negocio
│   ├── home.py                # Controlador principal
│   ├── agregar/               # Controladores CRUD
│   ├── editar/
│   ├── eliminar/
│   ├── grafos/                # Lógica de grafos
│   ├── users/                 # Gestión de usuarios
│   └── visualizar/            # Visualización de datos
│
├── models/                     # Modelos de datos
│   ├── agregar/
│   │   ├── ciudad.py          # Modelo de ciudades
│   │   ├── conexiones.py      # Modelo de conexiones
│   │   └── provincia.py       # Modelo de provincias
│   └── users/
│       └── User.py            # Modelo de usuarios
│
├── routes/                     # Definición de rutas
│   ├── __init__.py            # Registro de blueprints
│   ├── home_routes.py         # Rutas principales
│   ├── advertencia/           # Rutas de errores
│   ├── grafos/                # Rutas de grafos
│   ├── users/                 # Rutas de usuarios
│   └── visualizar/            # Rutas de visualización
```

```
├── static/                     # Archivos estáticos
│   ├── adminlte/              # Framework AdminLTE
│   ├── img/                   # Imágenes generadas
│   ├── scripts/               # JavaScript personalizado
│   └── styles/                # CSS personalizado
│
├── templates/                  # Plantillas HTML
│   ├── base.html              # Plantilla base
│   ├── dashboard.html         # Panel principal
│   ├── advertencia/           # Templates de error
│   ├── grafos/                # Templates de grafos
│   ├── pdf/                   # Templates para PDF
│   ├── users/                 # Templates de usuarios
│   └── visualizar/            # Templates de visualización
│
└── utils/                      # Utilidades
    ├── decorators/            # Decoradores personalizados
    │   └── admin_required.py  # Decorador de permisos admin
    ├── generar_pdf/           # Generación de PDFs
    │   └── tablas.py          # Funciones PDF con xhtml2pdf
    └── grafos/                # Utilidades de grafos
        └── grafo_utils.py     # Algoritmos principales
```

MODELOS DE DATOS

Diagrama Entidad-Relación



Descripción de Entidades

Provincia

```python
class Provincia(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    nombre = db.Column(db.String(100), nullable=False)
    # Relación: Una provincia tiene muchas ciudades
```

Ciudad

```python
class Ciudad(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    nombre = db.Column(db.String(100), nullable=False)
    provincia_id = db.Column(db.Integer, ForeignKey('provincias.id'))
    costera = db.Column(db.Boolean, default=False)
    # Relación: Una ciudad pertenece a una provincia
```

Conexiones

```python
class Conexiones(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    origen_id = db.Column(db.Integer, ForeignKey('ciudades.id'))
    destino_id = db.Column(db.Integer, ForeignKey('ciudades.id'))
    distancia = db.Column(db.Float, nullable=False)
    # Relación: Una conexión une dos ciudades
```

User

```python
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True)
    email = db.Column(db.String(120), unique=True)
    password = db.Column(db.String(128))  # Hash
    admin = db.Column(db.Boolean, default=False)
```

LÓGICA DE GRAFOS Y RUTAS

Algoritmo Principal: Dijkstra Modificado

El sistema implementa una versión modificada del algoritmo de Dijkstra que tiene la restricción de que todas las rutas válidas deben pasar por al menos una ciudad costera.

Implementación del Grafo

```python
def construir_grafo(aristas):
    """
    Construye un grafo no dirigido y ponderado usando NetworkX

    Args:
        aristas: Lista de tuplas (origen, destino, costo)

    Returns:
        nx.Graph: Grafo construido
    """
    G = nx.Graph()
    for origen, destino, costo in aristas:
        G.add_edge(origen, destino, weight=costo)
    return G
```

Algoritmo de Ruta Óptima

```python
def camino_optimo_con_costera(origen, destino, G, COSTERAS):
    """
    Calcula el camino más corto con restricción de ciudades costeras

    Args:
        origen: Ciudad de origen
        destino: Ciudad de destino
        G: Grafo de ciudades
        COSTERAS: Lista de ciudades costeras

    Returns:
        dict: {camino, costo, valido}
    """
    try:
        # Algoritmo de Dijkstra para encontrar el camino más corto
        camino = nx.dijkstra_path(G, origen, destino, weight='weight')
        costo = nx.dijkstra_path_length(G, origen, destino, weight='weight')

        # Verificar si el camino contiene al menos una ciudad costera
        contiene_costera = any(ciudad in COSTERAS for ciudad in camino)

        return {
            "camino": camino,
            "costo": costo,
            "valido": contiene_costera
        }
    except nx.NetworkXNoPath:
        return {
            "camino": [],
            "costo": None,
            "valido": False
        }
```

Visualización de Grafos

El sistema genera dos tipos de visualizaciones

Grafo Completo

```python
def grafo_a_imagen(nombre_archivo='grafo.png', G=None):
    """
    Genera una imagen del grafo completo
    - Nodos en azul (#0aa4e6)
    - Aristas en gris
    - Etiquetas con pesos
    """
    pos = nx.spring_layout(G, seed=8)
    pesos = nx.get_edge_attributes(G, 'weight')

    fig, ax = plt.subplots(figsize=(10, 7))
    nx.draw(G, pos, with_labels=True, node_color="#0aa4e6",
            node_size=2000, font_weight='bold', arrows=True)
    nx.draw_networkx_edge_labels(G, pos, edge_labels=pesos)

    plt.savefig(ruta_archivo, format='png')
    plt.close()
```

Grafo con Ruta Resaltada

```python
def grafo_a_imagen_marcado(nombre_archivo='grafo.png', camino=None, G=None):
    """
    Genera una imagen del grafo con el camino óptimo resaltado
    - Camino óptimo en verde (#80fa6d)
    - Resto del grafo en colores normales
    """
    # Dibuja el grafo base
    nx.draw(G, pos, with_labels=True, node_color="#0aa4e6",
            node_size=2000, font_weight='bold', arrows=True,
            edge_color='gray')

    # Resalta el camino encontrado
    if camino and len(camino) >= 2:
        aristas_camino = list(zip(camino, camino[1:]))
        nx.draw_networkx_edges(G, pos, edgelist=aristas_camino,
                               edge_color="#80fa6d", width=4, arrows=True)
```

Características del Algoritmo

- **Complejidad Temporal:** O((V + E) log V) donde V = vértices, E = aristas
- **Garantía de Optimalidad:** Encuentra siempre el camino más corto
- **Restricción Costera:** Valida que la ruta pase por ciudades costeras
- **Manejo de Errores:** Captura casos donde no existe camino válido

SISTEMA DE AUTENTICACIÓN

Arquitectura de Autenticación

El sistema utiliza Flask-Login para gestionar la autenticación de usuarios con dos niveles de permisos.

Configuración Base

```
# extensions.py
lm = LoginManager()

@lm.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))

# app.py
lm.login_view = 'login.login'   # Redirección para usuarios no autenticados
```

Modelo de Usuario

```python
class User(db.Model, UserMixin):
    # Campos de usuario
    username = db.Column(db.String(80), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    password = db.Column(db.String(128), nullable=False)  # Hash SHA-256
    admin = db.Column(db.Boolean, default=False)

    # Métodos de seguridad
    def check_password(self, password):
        return check_password_hash(self.password, password)

    @classmethod
    def create_user(cls, username, email, password):
        hashed_password = generate_password_hash(password)
        # ... resto del método
```

Niveles de Permisos

1. Usuario Regular
    a. Visualizar datos (ciudades, provincias, conexiones)
    b. Calcular rutas
    c. Ver grafos
    d. Generar PDFs de consulta
2. Usuario Administrador
    a. Todas las funciones de usuario regular
    b. Crear, editar y eliminar ciudades
    c. Crear, editar y eliminar provincias
    d. Crear, editar y eliminar conexiones
    e. Gestionar otros usuarios

Decorador de Seguridad

```python
# utils/decorators/admin_required.py
def admin_required(f):
    @wraps(f)
        def decorated_function(*args, **kwargs):
            if not current_user.is_authenticated or not current_user.admin:
                return redirect(url_for('error_admins.error_admin'))
            return f(*args, **kwargs)
        return decorated_function
```

INTERFACES DE USUARIO

Framework y Diseño

- **AdminLTE 3:** Template administrativo moderno y responsivo
- **Bootstrap 4:** Framework CSS para componentes responsive
- **Font Awesome:** Iconografía
- **jQuery:** Interactividad del lado cliente

Principales Interfaces

1. **Dashboard Principal**
    a. **Ruta:** */dashboard*
    b. **Descripción:** Panel principal con navegación a todas las funcionalidades
    c. Componentes:
        i. Cards informativos
        ii. Menú lateral
        iii. Breadcrumbs
        iv. Footer con información del sistema
2. **Gestión de Ciudades**
    a. **Ruta:** */visualizar-ciudades*
    b. Funcionalidades:
        i. Tabla paginada con búsqueda
        ii. Filtros por provincia
        iii. Indicador de ciudades costeras

iv. Acciones CRUD (solo administradores)

3. **Visualización de Grafos**
    a. **Ruta:** */info-grafos* y */grafo*
    b. Características:
        i. Generación dinámica de grafos
        ii. Resaltado de rutas óptimas
        iii. Formulario de búsqueda de rutas
        iv. Información de costos y validez
4. **Gestión de Usuarios**
    a. **Rutas:** */login, /register*
    b. Funcionalidades:
        i. Formularios de autenticación
        ii. Validación de datos
        iii. Mensajes de error/éxito
        iv. Redirección automática

Elementos de UX/UI

1. **Navegación Intuitiva**
    a. Menú lateral colapsible
    b. Breadcrumbs contextuales
    c. Botones de acción claramente identificados
2. **Feedback Visual**
    a. Alertas de Bootstrap para mensajes
    b. Indicadores de carga
    c. Estados de botones
3. **Responsividad**
    a. Diseño adaptable a móviles
    b. Tablas con scroll horizontal
    c. Menús colapsibles

FUNCIONALIDADES PRINCIPALES

Gestión de Datos Geográficos

Provincias

- CRUD completo para administradores
- Validación de nombres únicos
- Relación cascad con ciudades

Ciudades

- CRUD completo con validaciones
- Clasificación como costera/no costera
- Relación con provincias
- Verificación de dependencias antes de eliminar

Conexiones

- Gestión de rutas entre ciudades
- Validación de distancias (números positivos)
- Prevención de conexiones duplicadas
- Eliminación automática al eliminar ciudades

Algoritmos de Grafos

Construcción del Grafo

```
# Proceso de construcción
conexiones = Conexiones.obtener_conexiones()  # BD → Lista de tuplas
grafo = construir_grafo(conexiones)           # Lista → NetworkX Graph
```

Cálculo de Rutas

```
# Flujo de cálculo
1. Recibir origen y destino del formulario
2. Construir grafo desde base de datos
3. Obtener lista de ciudades costeras
4. Aplicar Dijkstra con validación costera
5. Generar imagen del grafo con ruta resaltada
6. Retornar resultado al usuario
```

Visualización Dinámica

Generación de Imágenes

- **Matplotlib + NetworkX** para renderizado
- **Layout spring** para distribución automática
- **Colores diferenciados** para estados del grafo
- **Almacenamiento en** */static/img/*

Estados Visuales

- **Grafo Normal:** Nodos azules, aristas grises
- **Ruta Resaltada:** Camino en verde, resto normal
- **Sin Conexión:** Mensaje de error, grafo normal

Generación de Reportes

PDFs Dinámicos con xhtml2pdf

La librería xhtml2pdf permite convertir templates HTML/CSS directamente a formato PDF, manteniendo el diseño y estilos de la interfaz web.

Proceso de Generación:

```python
from xhtml2pdf import pisa

def generar_pdf_desde_template(template_html, context):
    """
    Convierte un template HTML a PDF usando xhtml2pdf

    Args:
        template_html: Template renderizado con datos
        context: Datos para el template

    Returns:
        PDF generado como respuesta HTTP
    """
    # Renderizar template con contexto
    html_renderizado = render_template(template_html, **context)

    # Convertir HTML a PDF
    response = make_response()
    pdf = pisa.pisaDocument(BytesIO(html_renderizado.encode("UTF-8")), response)

    if not pdf.err:
        response.headers['Content-Type'] = 'application/pdf'
        response.headers['Content-Disposition'] = 'attachment; filename=reporte.pdf'

    return response
```

Tipos de Reportes Disponibles

- **Tablas de ciudades** con filtros por provincia y estado costera
- **Listado de provincias** con conteo de ciudades asociadas
- **Conexiones detalladas** con información de distancias
- **Rutas calculadas** con camino completo, costo total y validación costera

Características de los PDFs

- **Templates HTML especializados** en `/templates/pdf/`

- **Estilos CSS específicos** para formato de impresión
- **Header personalizado** con logo y información del sistema
- **Formato profesional** con tablas estructuradas
- **Datos actualizados** en tiempo real desde la base de datos
- **Descarga directa** desde el navegador sin almacenamiento temporal

Implementación Específica en RUTIX

```python
# utils/generar_pdf/tablas.py
from xhtml2pdf import pisa
from flask import make_response, render_template
from datetime import datetime

def generar_pdf_ciudades(ciudades, provincias):
    """
    Genera PDF de ciudades con información completa

    Features:
    - Timestamp automático en nombre de archivo
    - Fecha de generación en el documento
    - Manejo de errores con try/catch
    - Headers HTTP apropiados para descarga
    """
    try:
        html_content = render_template('pdf/ciudades_pdf.html',
                                       ciudades=ciudades,
                                       provincias=provincias,
                                       fecha_generacion=datetime.now().strftime("%d/%m/%Y %H:%M"))

        response = make_response()
        response.headers['Content-Type'] = 'application/pdf'
        response.headers['Content-Disposition'] = f'attachment; filename=ciudades_{datetime.now().strftime("%Y%m%d_%H%M%S")}.pdf'

        pisa_status = pisa.CreatePDF(html_content, dest=response.stream)

        if pisa_status.err:
            return None

        return response
    except Exception:
        flash('Error al generar el PDF', 'error')
        return None
```

Templates PDF Especializados

- **ciudades_pdf.html:** Lista completa de ciudades con provincia y estado costera
- **provincias_pdf.html:** Información de provincias con contadores
- **conexiones_pdf.html:** Tabla de conexiones con distancias
- **rutas_pdf.html:** Rutas calculadas con detalles del camino

Ventajas de xhtml2pdf en RUTIX

- **Familiaridad:** Utiliza HTML/CSS estándar que ya conoce el equipo

- **Integración Perfecta:** Se integra sin problemas con templates de Flask/Jinja2
- **Flexibilidad de Diseño:** Permite estilos complejos y diseños personalizados
- **Rendimiento Óptimo:** Generación eficiente sin dependencias externas pesadas
- **Mantenibilidad:** Los templates PDF comparten estilos con la interfaz web
- **Control Total:** Manejo completo del formato y presentación de datos

Sistema de Permisos

Middleware de Autenticación

```
@login_required  # Flask-Login
@admin_required  # Decorador personalizado
def funcion_admin():
    # Solo accesible para administradores
```

Control de Acceso

- **Vistas diferenciadas** según tipo de usuario
- **Botones condicionales** en templates
- **Redirecciones automáticas** para accesos no autorizados
- **Mensajes informativos** sobre permisos

CONFIGURACIÓN Y DESPLIEGUE

Variables de Entorno

El sistema utiliza un archivo *.env* para configuración:

```
# Base de datos
DB_HOST=localhost
DB_PORT=3306
DB_USER=root
DB_PASSWORD=tu_password
DB_NAME=rutix

# Seguridad
SECRET_KEY=tu_clave_secreta_muy_segura
```

Configuración de la Aplicación

```python
# config.py
class Config:
    # URI de conexión a MySQL
    SQLALCHEMY_DATABASE_URI = f'mysql+pymysql://{os.getenv('DB_USER')}:{os.getenv('DB_PASSWORD')}@{os.getenv('DB_HOST')}:{os.getenv('DB_PORT')}/{os.getenv('DB_NAME')}

    # Configuraciones de seguridad
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SECRET_KEY = os.getenv('SECRET_KEY')
```

Dependencias

```
Flask==2.3.3
matplotlib==3.7.2
networkx==3.1
Flask-SQLAlchemy==3.0.5
python-dotenv==1.0.0
PyMySQL==1.1.0
Flask-Login==0.6.2
Werkzeug==2.3.7
xhtml2pdf==0.2.11
```

Proceso de Instalación

1. Clonar el repositorio

```
git clone https://github.com/BRUMILO/Rutix
cd Rutix
```

2. Crear entorno virtual

```
python -m venv venv
venv\Scripts\activate  # Windows
```

3. Instalar dependencias

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

4. Configurar base de datos

```
# Crear base de datos MySQL
mysql -u root -p
CREATE DATABASE Rutix_DB;
```

5. Configurar variables de entorno

```
# Crear archivo .env con las variables necesarias
```

6. Ejecutar la aplicación

```
python app.py
```

Inicialización Automática

El sistema incluye funcionalidad de inicialización automática.

```
def create_user_admin(app):
    """Crea un usuario administrador por defecto"""
    with app.app_context():
        if not User.query.filter_by(username='admin').first():
            User.create_user('admin', 'admin@email.com', 'admin123')
            admin = User.query.filter_by(username='admin').first()
            admin.admin = True
            db.session.commit()
```

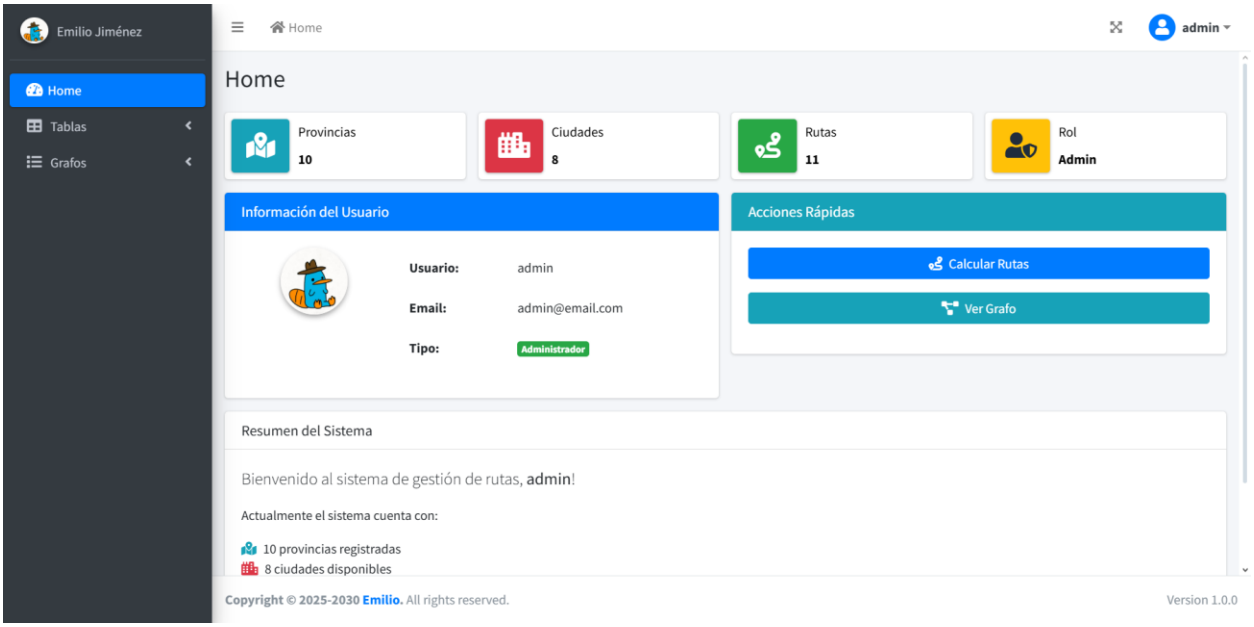
Credenciales por defecto

- **Usuario:** admin
- **Contraseña:** admin123
- **Email:** admin@email.com

CAPTURAS DEL SISTEMA

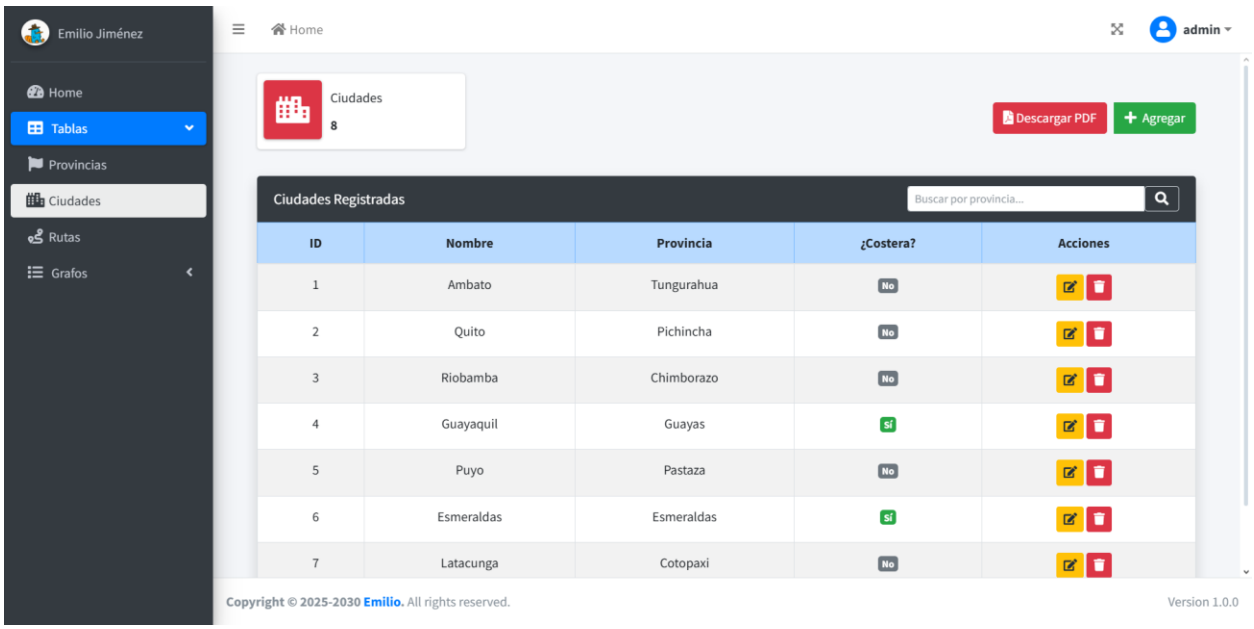
Dashboard Principal

- Panel de control centralizado
- Navegación intuitiva por módulos
- Información de estado del sistema



Gestión de Ciudades

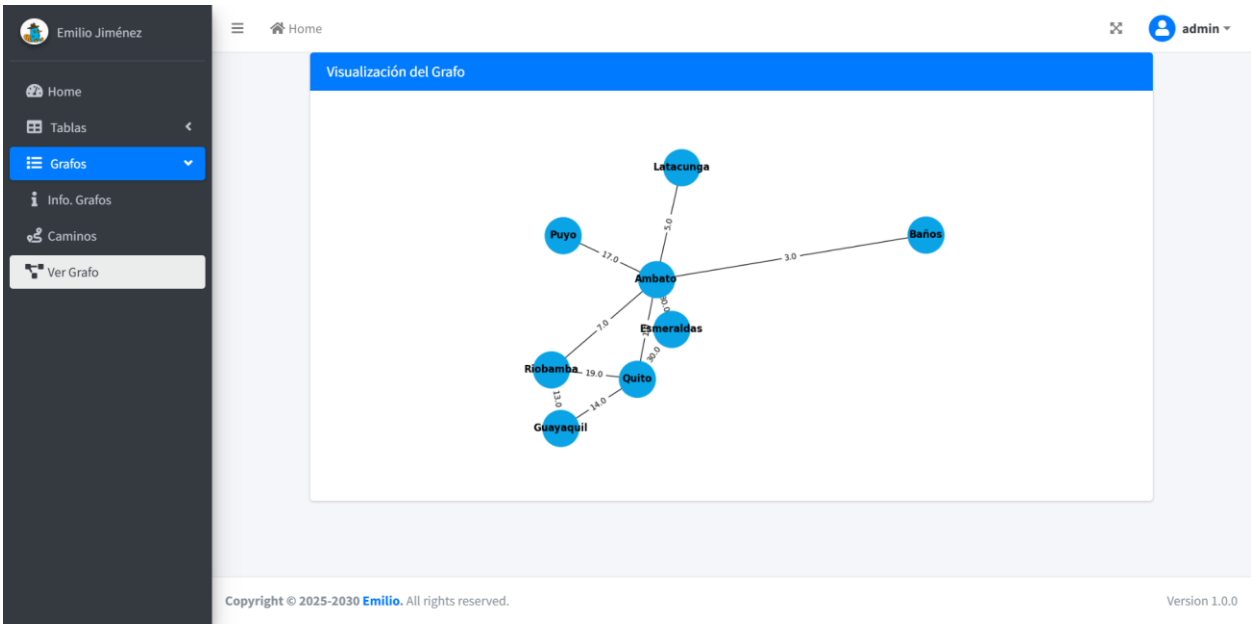
- Tabla con información completa
- Indicadores visuales para ciudades costeras
- Funciones CRUD accesibles



Visualización de Grafos

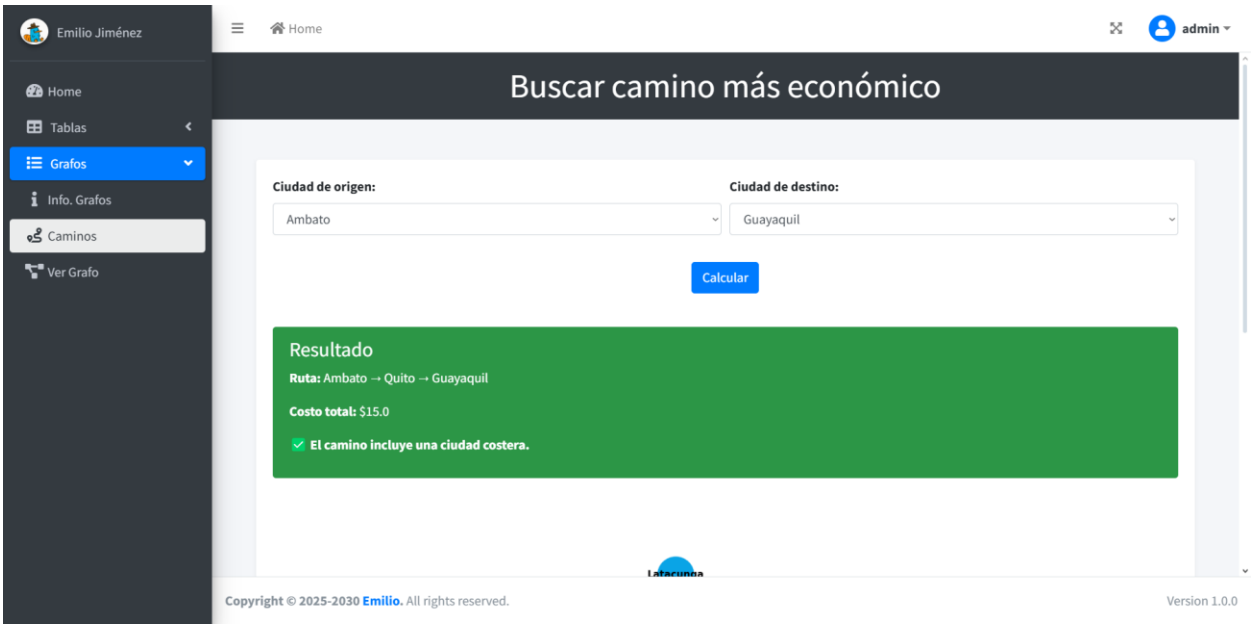
- Representación gráfica del mapa de ciudades

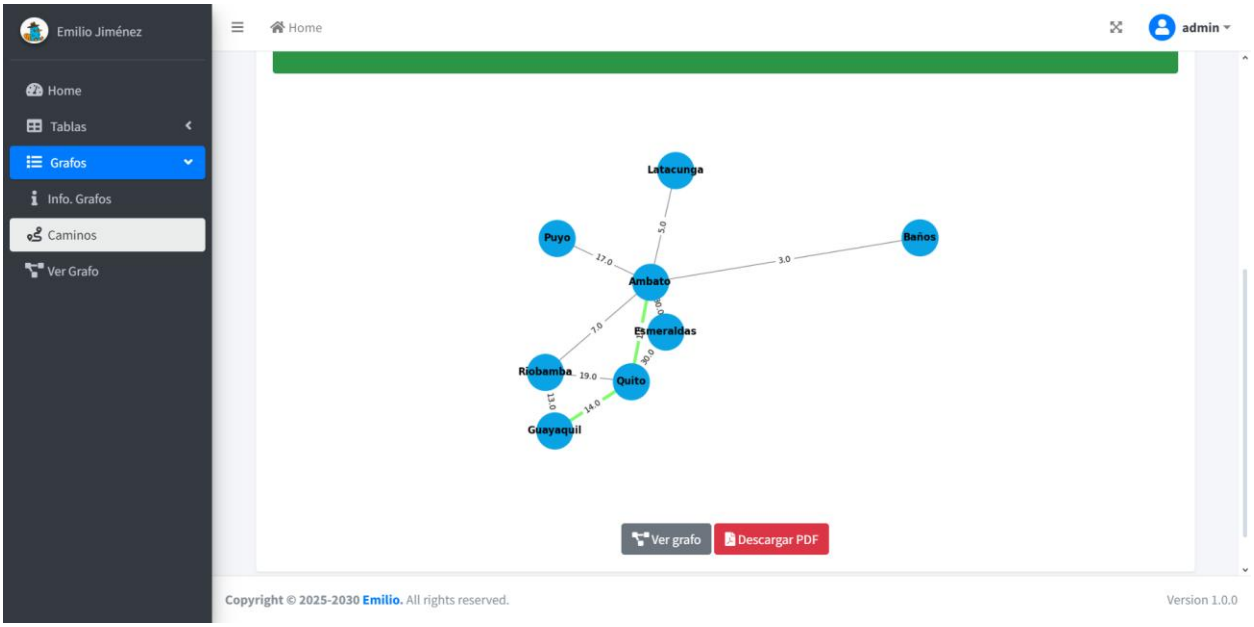
- Resaltado de rutas óptimas
- Información detallada de costos



Cálculo de Rutas

- Formulario de búsqueda intuitivo
- Resultados con validación costera
- Visualización del camino encontrado





CONCLUSIONES

Logros Técnicos

- **Arquitectura Sólida:** Implementación exitosa del patrón MVC con separación clara de responsabilidades.
- **Algoritmos Eficientes:** Uso apropiado del algoritmo de Dijkstra con modificaciones para restricciones específicas del dominio.
- **Interfaz Intuitiva:** Desarrollo de una interfaz moderna y responsiva utilizando frameworks estándar de la industria.
- **Seguridad Robusta:** Implementación de autenticación y autorización con diferentes niveles de permisos.
- **Escalabilidad**: Estructura modular que permite fácil expansión y mantenimiento.

Características Destacadas

- **Visualización Dinámica:** Generación automática de grafos con resaltado de rutas
- **Validación Inteligente:** Algoritmo que respeta restricciones de negocio (ciudades costeras)
- **Gestión Completa:** CRUD completo para todas las entidades del sistema
- **Reportería:** Generación de PDFs dinámicos y profesionales

- **Usabilidad:** Interfaz intuitiva adaptable a diferentes dispositivos

Tecnologías Apropiadas

La selección de tecnologías demuestra un balance adecuado entre:

- **Simplicidad vs Funcionalidad:** Flask ofrece flexibilidad sin complejidad excesiva
- **Rendimiento vs Facilidad de Uso:** NetworkX proporciona algoritmos optimizados con API simple
- **Estética vs Practicidad:** AdminLTE ofrece diseño profesional con funcionalidad completa.

Potencial de Mejora

El sistema establece una base sólida que permite futuras expansiones como:

- API REST para integración con otros sistemas
- Algoritmos adicionales (A*, Floyd-Warshall)
- Mapas interactivos con bibliotecas como Leaflet
- Optimización de rutas considerando múltiples criterios
- Sistema de notificaciones en tiempo real

Valor del Proyecto

RUTIX demuestra la aplicación práctica de conceptos fundamentales de programación:

- Estructuras de datos (grafos)
- Algoritmos de optimización
- Diseño de software
- Desarrollo web full-stack
- Gestión de bases de datos

El sistema resuelve un problema real de optimización de rutas con restricciones específicas, proporcionando una herramienta útil y bien diseñada para la gestión de información geográfica y cálculo de rutas.

Anexos

Repositorio: https://github.com/BRUMILO/Rutix