Universidad Mariano Gálvez de Guatemala

Facultad de Ingeniería en Sistemas de información

Proyecto Final – Programación III

Ing. José Miguel Villatoro Hidalgo

# Manual Técnico


| | |
|---|---|
| Edilson Enrique Villeda García | 9490-23-2637 |
| Dayna Marianne Meza Maltez | 9490-23-3808 |
| Luis Samuel Menchú Tun | 9490-23-6285 |

## Tabla de contenido

## 1. Introducción

Este manual técnico documenta el desarrollo e implementación del proyecto final para el curso de Programación III. El proyecto consiste en una aplicación web que permite al usuario obtener recomendaciones de rutas turísticas en Guatemala, basadas en criterios como presupuesto, tiempo disponible y ubicación de partida. El sistema permite la carga de datos desde archivos CSV y ofrece funcionalidades como calificación de lugares y comentarios.

El sistema ha sido diseñado para facilitar la planificación de viajes personalizados y eficientes, especialmente dentro del territorio nacional.

La documentación incluye detalles sobre la instalación, funcionamiento, estructuras de datos utilizadas, módulos implementados y lógica detrás del algoritmo de recomendación.

## 2. Descripción general del sistema

La aplicación fue desarrollada utilizando Python y Flask como marco principal. Su objetivo es recomendar rutas turísticas completas que el usuario puede realizar, considerando criterios específicos como:

- Presupuesto disponible.
- Tiempo total en horas.
- Punto de partida (ubicación inicial).

El sistema carga la información de los lugares turísticos desde archivos CSV, la cual se organiza internamente utilizando un Árbol B para búsquedas eficientes y un grafo que representa las conexiones entre lugares mediante distancias geográficas. El usuario puede interactuar con la plataforma a través de una interfaz web sencilla que le permite:

- Ingresar sus criterios de búsqueda.
- Visualizar las rutas recomendadas.
- Calificar los lugares que ha visitado.
- Escribir comentarios sobre su experiencia.
- Añadir nuevos lugares.

El algoritmo genera múltiples rutas posibles y prioriza distintas estrategias, como, por ejemplo, mayor calificación, menor distancia, equilibrio entre costo y cantidad de lugares, entre otras. La aplicación también permite la carga manual de datos y ofrece funciones de exportación e importación.

## 3. Instalación

Se detallan los pasos y recomendaciones para instalar la aplicación web en el sistema y que se ejecute correctamente.

### 3.1 Requisitos del Sistema

El sistema fue desarrollado en Python y Flask y se puede ejecutar localmente en cualquier computador con Windows y Python. Para que la aplicación se pueda ejecutar correctamente es necesario que el sistema cumpla con ciertos requisitos.

- Sistema Operativo: Windows 10 en adelante
- Python: versión 3.00 o superior.
- PIP: Gestor de paquetes de Python
- Navegador Web: Google, Edge, FireFox, etc
- Conexión a internet.

### 3.2 Instalación

a) Si no se tuviera Python instalado se debe descargar e instalar desde el sitio oficial: ttps://www.python.org.
b) En la terminal de Pyhon se verifica que Python y pip estén correctamente instalados ejecutando el comando:

   *python --version*

   *pip --version*

c) Ubicarse dentro de la carpeta con el proyecto
d) Instalar todas las dependencias necesarias ejecutando el comando:

   *pip install -r requirements.txt*

### 3.3. Ejecución

Cuando ya tengamos todas las dependencias instaladas, procedemos a ejecutar el programa.

a) Abrimos la terminal
b) Ejecutamos el comando: *python app.py*

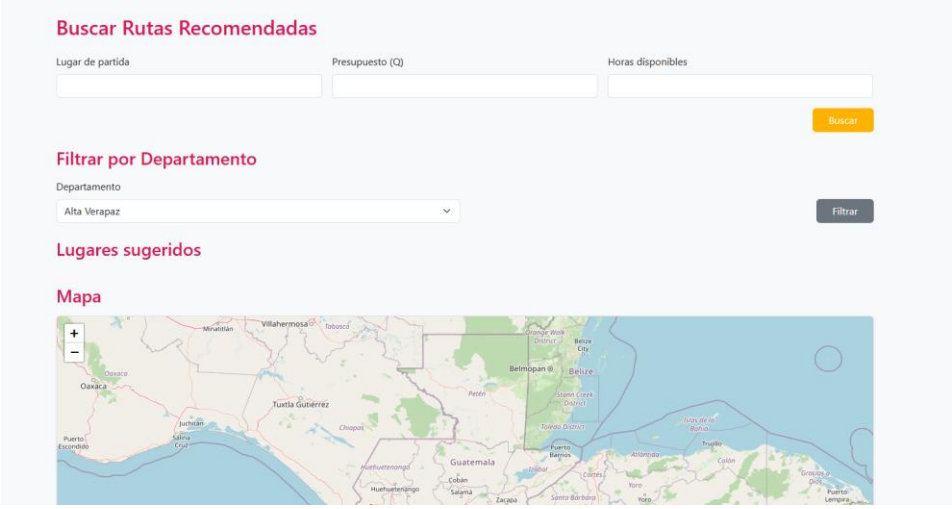
```
oyecto_rutas_con_coordenadas (1)/proyecto_rutas_con_coordenadas/proyecto_rutas/app.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 299-187-988
```

c) Acceso a la aplicación

Una vez este iniciado el servidor podemos acceder a la aplicación dejando el clic sobre la URL o ingresando en la dirección:

http://127.0.0.1:5000/

A partir de aquí el usuario podrá utilizar todas las funcionalidades del sistema.



## 4. Módulos del sistema / Estructura del código fuente.

El proyecto está organizado en múltiples módulos que trabajan en conjunto para ofrecer la funcionalidad completa del sistema de rutas turísticas. A continuación, se describe cada uno de los archivos más relevantes y su propósito dentro de la aplicación:

### 4.1. App.py

Este archivo es el núcleo de la aplicación web. Contiene toda la configuración y las rutas necesarias para mostrar la interfaz en el navegador, procesar formularios y coordinar las acciones del usuario con las estructuras internas como el Árbol B, el grafo y el recomendador de rutas. También gestiona las funciones de carga de archivos, generación de rutas, filtrado por departamento y exportación/importación de datos.

```python

from flask import Flask, render_template, request, redirect, send_file
import os
from arbol_b import ArbolB, Lugar
from utils.generador_mapa import generar_mapa
from utils.parser_csv import cargar_csv, exportar_csv, cargar_conexiones
from utils.serializador import guardar_arbol, cargar_arbol, guardar_grafo, cargar_grafo
from recomendador import recomendar_rutas



from grafo import Grafo

#grafo = Grafo() DONDE GUARDAR Y CARGAR AUTOMATICAMENTE LOS ARBOLES Y GRAFOS
ARCHIVO_ARBOL = os.path.join(os.path.dirname(__file__), "datos", "arbol_guardado.pkl")
ARCHIVO_GRAFO = os.path.join(os.path.dirname(__file__), "datos", "grafo_guardado.pkl")
arbol = cargar_arbol(ARCHIVO_ARBOL) or ArbolB(3)
grafo = cargar_grafo(ARCHIVO_GRAFO) or Grafo()
from utils.generador_mapa import generar_mapa
rutas = []

#INICIA EL PROGRAMA
app = Flask(__name__)
app.config['UPLOAD_FOLDER'] = os.path.join(os.path.dirname(__file__), 'datos')

@app.route("/")
def inicio():
    lugares = arbol.obtener_lugares()
    departamentos = sorted(set(l.departamento for l in lugares))
    depto_filtrado = request.args.get("departamento")
    calificar_lugar = request.args.get("calificar")

    lugares_filtrados = [l for l in lugares if l.departamento == depto_filtrado] if depto_filtrado else []
    generar_mapa(lugares_filtrados if depto_filtrado else [])

    return render_template(
        "inicio.html",
```

```python
@app.route("/recomendar", methods=["POST"])
def recomendar():
    global rutas
    origen = request.form["origen"].strip()
    presupuesto = float(request.form["presupuesto"])
    horas = float(request.form["horas"])
    rutas = recomendar_rutas(grafo, origen, presupuesto, horas)
    generar_mapa([], grafo=grafo, origen_nombre=origen)  # Mapa limpio, sin ruta aún
    return render_template("inicio.html", lugares=[], departamentos=[], rutas=rutas)

@app.route("/insertar", methods=["GET", "POST"])
def insertar():
    if request.method == "POST":
            nombre = request.form["nombre"]
            depto = request.form["departamento"]
            calif = float(request.form["calificacion"])
            costo = float(request.form["costo"])
            tiempo = float(request.form["tiempo"])
            lat = float(request.form["latitud"])
            lon = float(request.form["longitud"])
            lugar = Lugar(nombre, depto, calif, costo, tiempo, lat, lon)
            arbol.insertar(lugar)
            grafo.agregar_lugar(lugar.nombre, lugar)
            guardar_grafo(grafo, ARCHIVO_GRAFO)
            guardar_arbol(arbol, ARCHIVO_ARBOL)
            return redirect("/insertar")
    return render_template("insertar.html")

@app.route("/cargar_csv", methods=["POST"])
def cargar():
    archivo = request.files["archivo"]
    if archivo:
        path = os.path.join(app.config["UPLOAD_FOLDER"], archivo.filename)
        archivo.save(path)
        lugares = cargar_csv(path)
        for lugar in lugares:
            arbol.insertar(lugar)
```

### 4.2. Árbol_b.py

Contiene la implementación completa del Árbol B, una estructura de datos balanceada que permite almacenar, insertar y buscar lugares turísticos de forma eficiente. Es clave para mantener un acceso rápido a la información durante la ejecución de la aplicación. Cada lugar turístico se guarda como un nodo en esta estructura de datos implementada.

```
class NodoB:
    def insertar_no_lleno(self, clave):
            self.claves.append(clave)
            self.claves.sort(key=lambda x: x.nombre)
        else:
            i = len(self.claves) - 1
            while i >= 0 and clave.nombre < self.claves[i].nombre:
                i -= 1
            i += 1
            if len(self.hijos[i].claves) == (2 * self.t) - 1:
                self.hijos[i].dividir(self, i)
                if clave.nombre > self.claves[i].nombre:
                    i += 1
            self.hijos[i].insertar_no_lleno(clave)

    def recorrer_inorden(self):
        resultado = []
        for i in range(len(self.claves)):
            if not self.hoja:
                resultado += self.hijos[i].recorrer_inorden()
            resultado.append(self.claves[i])
        if not self.hoja:
            resultado += self.hijos[-1].recorrer_inorden()
        return resultado

class ArbolB:
    def __init__(self, t):
        self.raiz = NodoB(t)
        self.t = t

    def insertar(self, lugar):
        raiz = self.raiz
        if len(raiz.claves) == (2 * self.t) - 1:
            nueva_raiz = NodoB(self.t)
            nueva_raiz.hoja = False
            nueva_raiz.hijos.append(raiz)
            nueva_raiz.dividir(nueva_raiz, 0)
```

### 4.3. Generador_mapa.py

Es el encargado de generar el mapa interactivo utilizando la librería Folium. Esta funcionalidad permite visualizar los lugares turísticos y las rutas recomendadas directamente sobre un mapa en el navegador.

- *Crear un mapa base centrado en Guatemala*
  Utilizando coordenadas iniciales (por ejemplo, de la Ciudad de Guatemala o del primer lugar de la ruta).

```
def generar_mapa(lugares, grafo=None, origen_nombre=None):
    if not lugares:
        # Si no hay lugares, usar mapa por defecto
        mapa = folium.Map(location=[15.5, -90.25], zoom_start=7)
    else:
        promedio_lat = sum([l.latitud for l in lugares]) / len(lugares)
        promedio_lon = sum([l.longitud for l in lugares]) / len(lugares)
        mapa = folium.Map(location=[promedio_lat, promedio_lon], zoom_start=7)
```

- *Agregar marcadores para cada lugar turístico.*
  Por cada lugar en la ruta, se añade un marcador con su nombre, calificación y un texto que aparece al hacer clic.

```
    folium.Marker(
        location=[lugar.latitud, lugar.longitud],
        popup=folium.Popup(popup_html, max_width=250),
        tooltip=lugar.nombre
    ).add_to(mapa)

    coords[lugar.nombre] = (lugar.latitud, lugar.longitud)
```

- *Dibujar la ruta entre los lugares*.
  Si se genera una ruta con varios lugares, se conectan entre sí con una línea, formando visualmente el recorrido sugerido.

```
    # Dibujar líneas si hay grafo y origen
    if grafo and origen_nombre in grafo.nodos:
        if origen_nombre and len(lugares) > 1:
            secuencia_coords = [(l.latitud, l.longitud) for l in lugares]
            folium.PolyLine(
                secuencia_coords,
                color="blue",
                weight=4,
                opacity=0.6
            ).add_to(mapa)
```

- *Guardar el mapa como archivo HTML*
  Una vez agregados los elementos al mapa, este se guarda en un archivo llamado mapa.html, que luego es abierto desde el navegador por Flask.

```
# Guardar el mapa
base_dir = os.path.abspath(os.path.dirname(__file__))
ruta_mapa = os.path.join(base_dir, "..", "static", "mapa.html")
ruta_mapa = os.path.normpath(ruta_mapa)
os.makedirs(os.path.dirname(ruta_mapa), exist_ok=True)
mapa.save(ruta_mapa)
```

### 4.4. Grafo.py

Este módulo define y administra la estructura de grafo, la cual representa las conexiones entre diferentes lugares turísticos. Utiliza un diccionario de adyacencias donde cada lugar está conectado con otros, junto con la distancia entre ellos. Esto permite calcular rutas lógicas y eficientes, considerando la ubicación geográfica de cada sitio.


```python

class NodoGrafo:
    def __init__(self, lugar):
        self.lugar = lugar

class Grafo:
    def __init__(self):
        self.nodos = {}
        self.conexiones = {}

    def agregar_lugar(self, nombre, lugar):
        if nombre not in self.nodos:
            self.nodos[nombre] = NodoGrafo(lugar)
            self.conexiones[nombre] = []

    def agregar_conexion(self, origen, destino, distancia, tiempo):
        if origen in self.nodos and destino in self.nodos:
            self.conexiones[origen].append((destino, distancia, tiempo))
            self.conexiones[destino].append((origen, distancia, tiempo))

    def obtener_vecinos(self, nombre_lugar):
        return self.conexiones.get(nombre_lugar, [])

```

### 4.5. Generadosr_arbol.py

Para complementar la visualización del sistema, se incorporó un módulo adicional llamado generador_arbol.py, el cual permite generar automáticamente una imagen del Árbol B y descargarla desde la interfaz web. Esto es útil para depuración, revisión de estructura y demostración de cómo los datos están organizados internamente.

- *Visualización con Graphviz*
  El módulo utiliza la librería graphviz y su clase Digraph() para recorrer recursivamente el árbol desde la raíz, y representar cada nodo con sus claves.

- *Generación del archivo*
  La imagen se guarda como .png en la carpeta static.

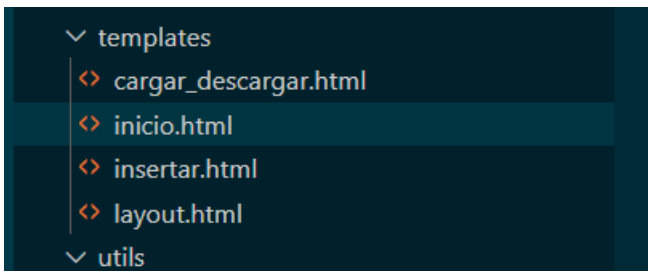
```python
from graphviz import Digraph

def graficar_arbol(arbol, path_salida="static/arbol_b.png"):
    dot = Digraph()
    dot.attr('node', shape='record')

    def agregar_nodo(nodo, id_nodo=0):
        if nodo is None:
            return id_nodo

        nodo_id = f"n{id_nodo}"
        etiquetas = "|".join([f"<f{i}> {lugar.nombre}" for i, lugar in enumerate(nodo.claves)])
        dot.node(nodo_id, f"{{{etiquetas}}}")

        id_actual = id_nodo + 1
        for i, hijo in enumerate(nodo.hijos):
            hijo_id = f"n{id_actual}"
            id_actual = agregar_nodo(hijo, id_actual)
            dot.edge(nodo_id, hijo_id)

        return id_actual

    agregar_nodo(arbol.raiz)
    dot.render(filename=path_salida.replace(".png", ""), format='png', cleanup=True)
```

- *Ruta Flask para descargarlo*
  Se añadió una ruta /descargar_arbol en app.py que invoca la función graficar_arbol(...) y luego permite descargar la imagen desde el navegador.

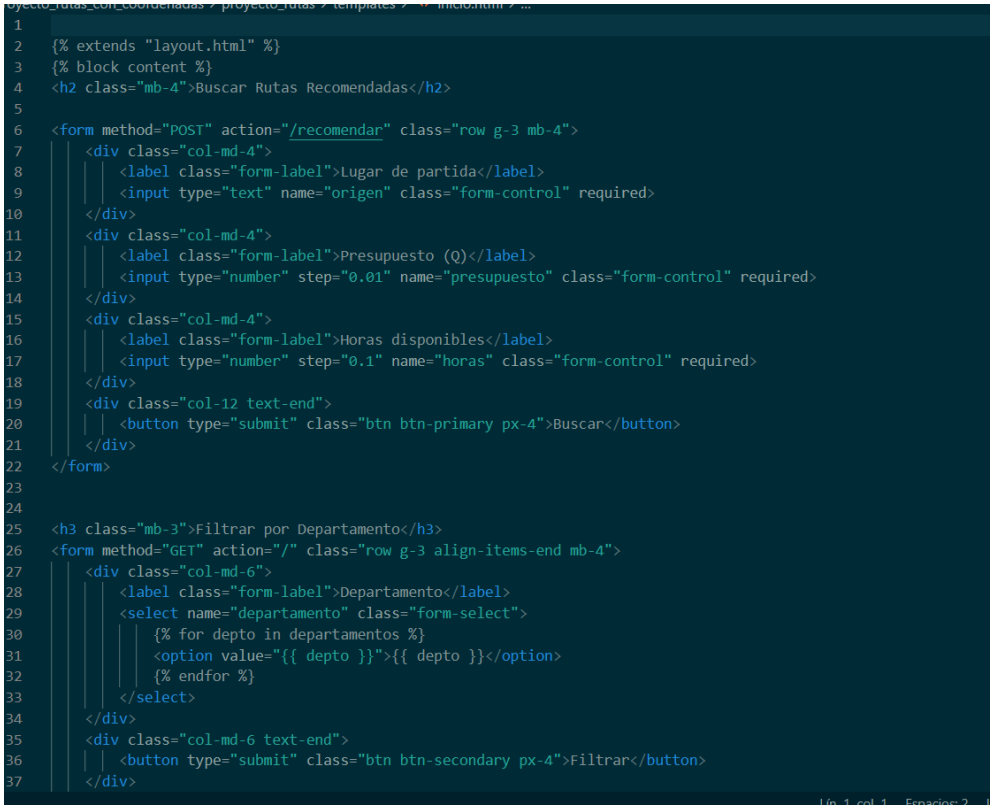
```python
@app.route("/descargar_arbol")
def descargar_arbol():
    output_path = os.path.join(os.path.dirname(__file__), "static", "arbol_b.png")
    graficar_arbol(arbol, output_path)
    return send_file(output_path, as_attachment=True)
```

### 4.6. HTML y CSS

El sistema utiliza plantillas HTML integradas con Flask para renderizar contenido dinámico en el navegador. Estas se encuentran dentro de la carpeta templates/ y cada archivo cumple un rol específico:

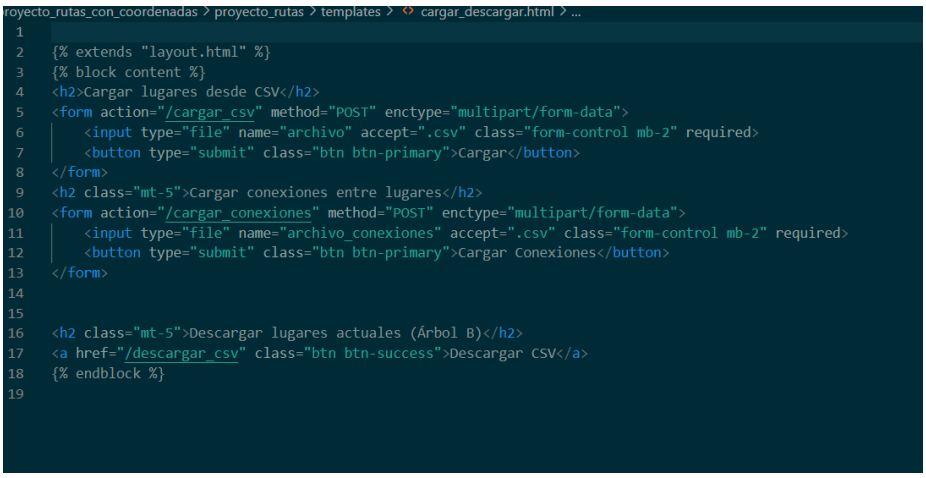


- **inicio.html**
  Página principal del sistema. Muestra el menú de navegación con acceso a las diferentes secciones del proyecto.



```
{% extends "layout.html" %}
{% block content %}
<h2 class="mb-4">Buscar Rutas Recomendadas</h2>

<form method="POST" action="/recomendar" class="row g-3 mb-4">
    <div class="col-md-4">
        <label class="form-label">Lugar de partida</label>
        <input type="text" name="origen" class="form-control" required>
    </div>
    <div class="col-md-4">
        <label class="form-label">Presupuesto (Q)</label>
        <input type="number" step="0.01" name="presupuesto" class="form-control" required>
    </div>
    <div class="col-md-4">
        <label class="form-label">Horas disponibles</label>
        <input type="number" step="0.1" name="horas" class="form-control" required>
    </div>
    <div class="col-12 text-end">
        <button type="submit" class="btn btn-primary px-4">Buscar</button>
    </div>
</form>


<h3 class="mb-3">Filtrar por Departamento</h3>
<form method="GET" action="/" class="row g-3 align-items-end mb-4">
    <div class="col-md-6">
        <label class="form-label">Departamento</label>
        <select name="departamento" class="form-select">
            {% for depto in departamentos %}
            <option value="{{ depto }}">{{ depto }}</option>
            {% endfor %}
        </select>
    </div>
    <div class="col-md-6 text-end">
        <button type="submit" class="btn btn-secondary px-4">Filtrar</button>
    </div>
```

- **cargar_descargar.html**
  Permite cargar archivos CSV con información turística o descargar los datos actuales desde el Árbol B.



```
{% extends "layout.html" %}
{% block content %}
<h2>Cargar lugares desde CSV</h2>
<form action="/cargar_csv" method="POST" enctype="multipart/form-data">
    <input type="file" name="archivo" accept=".csv" class="form-control mb-2" required>
    <button type="submit" class="btn btn-primary">Cargar</button>
</form>
<h2 class="mt-5">Cargar conexiones entre lugares</h2>
<form action="/cargar_conexiones" method="POST" enctype="multipart/form-data">
    <input type="file" name="archivo_conexiones" accept=".csv" class="form-control mb-2" required>
    <button type="submit" class="btn btn-primary">Cargar Conexiones</button>
</form>


<h2 class="mt-5">Descargar lugares actuales (Árbol B)</h2>
<a href="/descargar_csv" class="btn btn-success">Descargar CSV</a>
{% endblock %}
```

- **insertar.html**
  Contiene un formulario para ingresar manualmente nuevos lugares turísticos con sus atributos: nombre, coordenadas, calificación, etc.

```html
<h2>Insertar Lugar Turístico</h2>
<form method="POST" class="row g-3">
    <div class="col-md-6">
        <label>Nombre</label>
        <input type="text" name="nombre" class="form-control" required>
    </div>
    <div class="col-md-6">
        <label>Departamento</label>
        <input type="text" name="departamento" class="form-control" required>
    </div>
    <div class="col-md-4">
        <label>Calificación (1-5)</label>
        <input type="number" name="calificacion" step="0.1" class="form-control" required>
    </div>
    <div class="col-md-4">
        <label>Costo (Q)</label>
        <input type="number" name="costo" step="0.1" class="form-control" required>
    </div>
    <div class="col-md-4">
        <label>Tiempo (hrs)</label>
        <input type="number" name="tiempo" step="0.1" class="form-control" required>
    </div>

    <div class="col-md-6">
        <label>Latitud</label>
        <input type="number" name="latitud" step="0.000001" class="form-control" required>
    </div>
    <div class="col-md-6">
        <label>Longitud</label>
        <input type="number" name="longitud" step="0.000001" class="form-control" required>
    </div>

    <div class="col-12">
        <button type="submit" class="btn btn-success">Insertar</button>
    </div>
</form>
{% endblock %}
```

- **layout.html**
  Plantilla base compartida por las demás vistas. Contiene la estructura común del sitio, como la barra de navegación, encabezado, etc. y aplica el diseño general mediante bloques de contenido heredables.

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Rutas Turísticas</title>
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css">
    <link rel="stylesheet" href="{{ url_for('static', filename='css/estilos.css') }}">

</head>
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js"></script>
<body class="bg-light">
<nav class="navbar navbar-expand-lg navbar-dark bg-dark mb-4">
    <div class="container-fluid">
        <a class="navbar-brand" href="/">Rutas Turísticas</a>
        <div class="navbar-nav">
            <a class="nav-link" href="/">Inicio</a>
            <a class="nav-link" href="/insertar">Insertar Lugar</a>
            <a class="nav-link" href="/cargar_descargar">Cargar/Descargar CSV</a>
        </div>
    </div>
</nav>
<div class="container py-4">
    {% block content %}{% endblock %}
</div>
</body>
</html>
```

### 4.7. Recomendador.py

## *Algoritmo de recomendación*

El algoritmo de recomendación es el núcleo del sistema, ya que se encarga de generar rutas turísticas personalizadas basadas en los criterios ingresados por el usuario: presupuesto disponible, cantidad de días y lugar de origen.

La lógica del algoritmo está implementada en el archivo recomendador.py, el cual trabaja directamente con los datos almacenados en el Árbol B y el grafo de conexiones. El proceso de recomendación se divide en varias etapas:

### *4.7.1. Filtrado inicial por distancia y presupuesto*

Se toma como referencia el lugar de origen ingresado por el usuario. A partir de allí, se exploran los lugares conectados en el grafo, validando que estén dentro del presupuesto y que la cantidad de días disponibles alcance para recorrerlos (considerando distancias).

### *4.7.2. Generación de múltiples rutas*

Una vez filtrados los posibles destinos, el sistema genera hasta cinco rutas distintas, cada una siguiendo un criterio de priorización diferente:

- **Ruta con mayor calificación total:** suma las calificaciones de todos los lugares en la ruta.
- **Ruta con mayor cantidad de lugares:** busca visitar el mayor número posible de sitios sin exceder el presupuesto.
- **Ruta más cercana:** prioriza los lugares más cercanos al origen.
- **Ruta equilibrada:** intenta balancear el costo, número de lugares y calificación.
- **Ruta personalizada:** puede adaptarse a condiciones futuras, como tipo de actividad, horario, etc.

| Criterio | Código | Explicación |
|---|---|---|
| Calificación | sum(...calificacion...) | Rutas con mayor calificación total |
| Cantidad | len(r[0]) | Rutas con más lugares visitados |
| Equilibrado | sum(...) / len(...) | Buen promedio de calificación por lugar |
| Cercanía | len(r[0]) (usado como aproximación) | Rutas más cortas |
| Costo | r[1] (menor costo total) | Ruta más económica |

### 4.7.3. Evaluación y selección

Cada ruta generada se compara en términos de costo total, distancia recorrida y calificación promedio. Si una ruta excede el presupuesto o los días, es descartada automáticamente.

### 4.7.4. Presentación de resultados

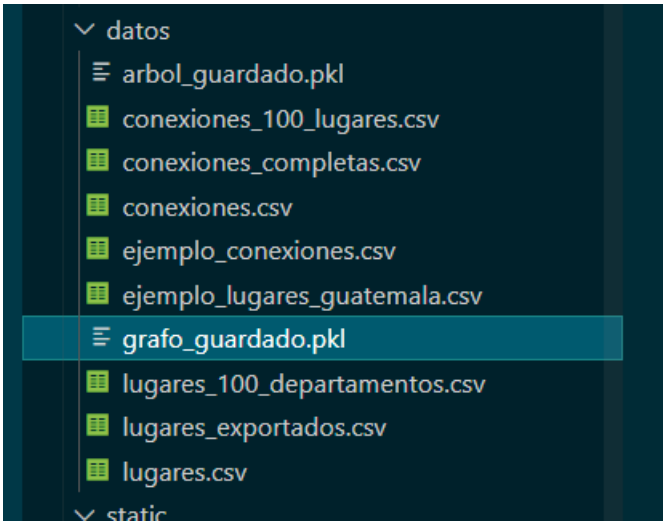
Las rutas válidas se presentan al usuario ordenadas por prioridad, y se visualizan tanto en forma de lista como en el mapa. Cada ruta incluye el orden de lugares, distancia total, tiempo estimado, y calificación promedio.

```
    return [r[0] for r in rutas_encontradas[:1]]
```

```
proyecto_rutas_con_coordenadas > proyecto_rutas > recomendador.py > recomendar_rutas > buscar_rutas
from collections import deque

def recomendar_rutas(grafo, origen, presupuesto_max, tiempo_max):
    rutas = []

    if origen not in grafo.nodos:
        return []
    def buscar_rutas(criterio, limite=5):
        rutas_encontradas = []
        visitadas = set()

        queue = deque()
        queue.append(([origen], 0.0, 0.0))  # (ruta, costo, tiempo)

        while queue and len(rutas_encontradas) < limite:
            ruta, costo_total, tiempo_total = queue.popleft()
            actual = ruta[-1]

            if len(ruta) > 1:
                rutas_encontradas.append((ruta, costo_total, tiempo_total))

            for vecino, dist, tiempo in grafo.obtener_vecinos(actual):
                if vecino in ruta:
                    continue
                lugar_vecino = grafo.nodos[vecino].lugar
                nuevo_costo = costo_total + lugar_vecino.costo
                nuevo_tiempo = tiempo_total + lugar_vecino.tiempo
                if nuevo_costo <= presupuesto_max and nuevo_tiempo <= tiempo_max:
                    queue.append((ruta + [vecino], nuevo_costo, nuevo_tiempo))

        # Aplicar criterio de ordenamiento
        if criterio == "calificacion":
            rutas_encontradas.sort(key=lambda r: sum(grafo.nodos[n].lugar.calificacion for n in r[0]), reverse=True)
        elif criterio == "cantidad":
            rutas_encontradas.sort(key=lambda r: len(r[0]), reverse=True)
        elif criterio == "equilibrado":
            rutas_encontradas.sort(key=lambda r: (sum(grafo.nodos[n].lugar.calificacion for n in r[0]) / len(r[0])) if len(r[0]) >
        elif criterio == "cercania":
            rutas_encontradas.sort(key=lambda r: len(r[0]))  # simulación simp
        elif criterio == "costo":
            rutas_encontradas.sort(key=lambda r: r[1])  # menor costo total

        return [r[0] for r in rutas_encontradas[:1]]  # solo la mejor por crit

    rutas += buscar_rutas("calificacion")
    rutas += buscar_rutas("cantidad")
    rutas += buscar_rutas("equilibrado")
    rutas += buscar_rutas("cercania")
    rutas += buscar_rutas("costo")

    return rutas
```

### 4.7.5. /datos/*.csv

La carpeta datos contiene los archivos .csv que se utilizan como fuente de entrada para los lugares turísticos y sus conexiones. Estos archivos pueden ser actualizados o reemplazados para modificar la base de datos del sistema sin necesidad de alterar el código fuente.



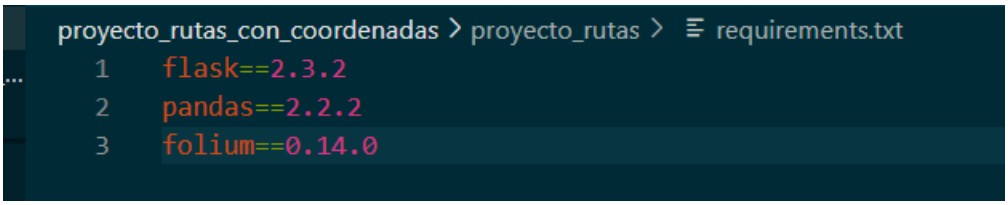

### 4.7.6. AbrirServidor.bat

Es un archivo ejecutable de Windows que sirve como acceso rápido para iniciar la aplicación. Al hacer doble clic, se abre la terminal y se ejecuta automáticamente el archivo app.py sin necesidad de comandos manuales.

### 4.7.7. Requirements.txt

Este archivo lista todas las librerías necesarias para que el sistema funcione. Es utilizado para instalar las dependencias automáticamente mediante el comando pip install -r requirements.txt.

- Flask: Es el motor principal que permite que la app tenga una interfaz web. Todo lo que ves en el navegador funciona gracias a Flask, que es un microframework de Python para construir aplicaciones web
- Pandas: Sirve para leer, procesar y analizar los archivos CSV que contienen los lugares turísticos y conexiones. Es una librería para manejo y análisis de datos en Python.
- Folium: Lo utilizamos para mostrar las rutas recomendadas sobre un mapa en la interfaz web. Los lugares y caminos se visualizan usando puntos, líneas, y zoom interactivo. Es una librería para crear **mapas interactivos** en Python utilizando Leaflet.js.




## 5. Estructuras de Datos Implementadas

Para el correcto funcionamiento del sistema de recomendaciones turísticas, se utilizaron dos estructuras de datos principales: un Árbol B y un grafo. Ambas fueron implementadas desde cero en Python y se integran directamente con la lógica del sistema.

### 5.1. Árbol B

El Árbol B es una estructura de datos especializada para almacenar y buscar información de forma eficiente, incluso cuando se trata de grandes volúmenes de datos. En este proyecto, se utilizó para guardar todos los lugares turísticos disponibles, permitiendo búsquedas rápidas y ordenadas.

El árbol B se implementó en el archivo árbol_b.py

### 5.2. Grafo

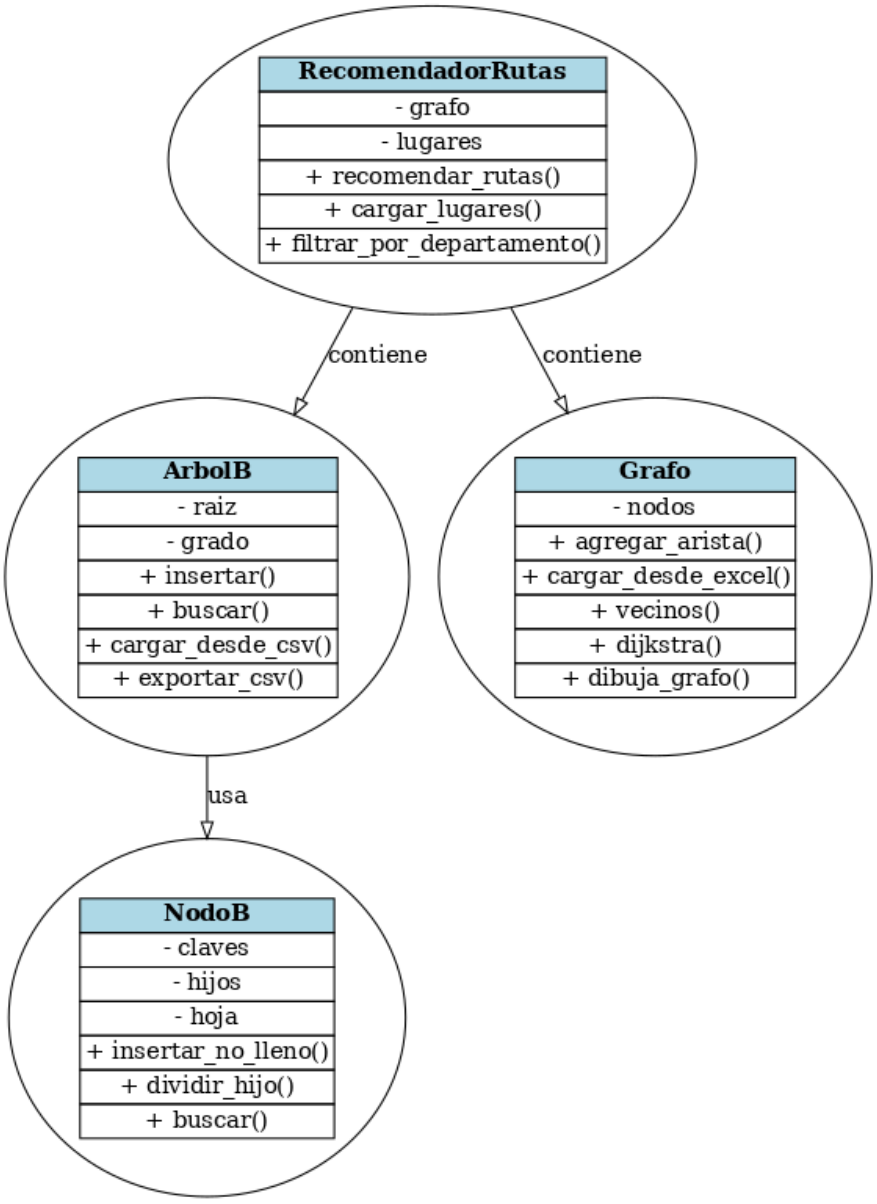
El grafo representa las conexiones entre los lugares turísticos. Cada vértice es un lugar, y cada arista representa una posible ruta entre dos puntos, con una distancia y tiempo de recorrido asociados.

El grafo es fundamental para calcular las posibles rutas entre los lugares.

El grafo se implemento en el archivo grafo.py

## 6. Diseño del Sistema

Para explicar el diseño del sistema de una forma más visual se realizó un diagrama de clases el cual muestra las entidades con las cuales podemos lograr las funcionalidades de nuestra aplicación.



### 6.1. RecomendadorRutas

Es la clase principal del sistema. Se encarga de coordinar el funcionamiento general y de ejecutar el algoritmo de recomendación. Contiene referencias a las estructuras de datos ArbolB y Grafo.

- ***Atributos:***
    - grafo: objeto que representa las conexiones entre lugares
    - lugares: lista de lugares turísticos cargados desde archivo

- ***Métodos:***
    - recomendar_rutas(): ejecuta el algoritmo de recomendación
    - cargar_lugares(): carga los lugares desde CSV
    - filtrar_por_departamento(): filtra lugares según su ubicación geográfica

### 6.2. ArbolB

Implementa la estructura de Árbol B para almacenar lugares turísticos de forma ordenada y eficiente.

- ***Atributos:***
    - raíz: nodo raíz del árbol
    - grado: define el grado mínimo del árbol
- ***Métodos:***
    - insertar(): agrega un nuevo lugar
    - buscar(): busca un lugar por su clave
    - cargar_desde_csv(): carga lugares desde un archivo CSV
    - exportar_csv(): guarda los lugares actuales a un archivo CSV

### 6.3. NodoB

Representa cada nodo del Árbol B. Puede contener varias claves (lugares) y referencias a nodos hijos.

- ***Atributos:***
    - claves: lista de claves/lugares en el nodo
    - hijos: referencias a nodos hijos
    - hoja: indica si el nodo es hoja o no
- ***Métodos:***
    - insertar_no_lleno(): inserta una clave en un nodo no lleno
    - dividir_hijo(): divide un nodo si está lleno
    - buscar(): busca una clave en el nodo o sus hijos

### 6.4. Grafo

Representa el conjunto de conexiones entre lugares. Se basa en un diccionario de adyacencia que almacena nodos y sus vecinos.

- ***Atributos:***
    - nodos: diccionario de nodos y sus conexiones
- ***Métodos:***
    - agregar_arista(): conecta dos lugares con una distancia
    - cargar_desde_excel(): carga conexiones desde un archivo Excel
    - vecinos(): obtiene los vecinos de un nodo dado
    - dijkstra(): calcula la ruta más corta entre dos lugares
    - dibuja_grafo(): genera una visualización del grafo

## 7. Funcionamiento del sistema

Para una descripción detallada de la interfaz y del uso paso a paso del sistema, se recomienda consultar el **Manual de Usuario**.
Este manual técnico se enfoca únicamente en los aspectos internos del sistema. A nivel estructural, la interfaz fue desarrollada utilizando el framework **Flask**, el cual renderiza formularios HTML que interactúan directamente con las estructuras del Árbol B, el grafo y el módulo de recomendaciones.

## 8. Conclusiones

El desarrollo de este sistema permitió aplicar los conocimientos adquiridos a lo largo del curso de Programación III, integrando estructuras de datos avanzadas como árboles B y grafos, junto con el uso de Flask para la creación de una interfaz web funcional e interactiva.

Se logró construir un sistema capaz de generar rutas turísticas personalizadas, considerando factores reales como presupuesto, distancia, tiempo y calificación, todo desde una lógica programada y optimizada para el usuario.

El diseño del proyecto facilita su mantenimiento y futura ampliación, permitiendo agregar nuevas funcionalidades como filtros por tipo de actividad, validación de clima, integración con bases de datos externas, o incluso exportación de rutas en formato PDF.

Una de las limitaciones que encontramos en el sistema es que se encuentra la dependencia de una base de datos estática cargada desde CSV, lo que podría ser optimizado usando una base de datos real en el futuro.