Segunda Entrega Parcial

Sistema de Optimización de Rutas usando Árboles

1. Problema

El objetivo de esta segunda fase del proyecto es extender y mejorar la representación de rutas utilizando estructuras de datos basadas en **árboles**, específicamente árboles generales
gestionados mediante la librería **BigTree** en Python. El problema se centra en optimizar la búsqueda de la mejor ruta entre dos ubicaciones en una red de ciudades, considerando eficiencia, organización jerárquica y complejidad computacional respecto a la implementación previa con listas enlazadas.

2. Contexto

En el ámbito de la logística, redes de transporte y sistemas de navegación, encontrar rutas óptimas entre distintos puntos es fundamental para reducir costos y tiempos de desplazamiento.
En este proyecto, se modelan conexiones reales entre ciudades colombianas como Piedecuesta, Bucaramanga, Bogotá, Medellín, La Guajira, Leticia y Cúcuta, incorporando también Caracas como punto externo.
La implementación previa basada en listas enlazadas permitió gestionar las rutas de forma secuencial; sin embargo, el crecimiento de la red de ubicaciones y rutas requería una estructura más eficiente y organizada, motivando así la adopción de **árboles** para representar las rutas de manera más escalable y rápida.

---

3. Librerías Utilizadas

- **BigTree** (bigtree):
    - Permite crear árboles genéricos con múltiples hijos.
    - Facilita la navegación, búsqueda, y visualización de estructuras jerárquicas.
    - Funciones usadas: Node, find_name, descendants, show, vshow, entre otras.
- **OS** (os):
    - Utilizado en esta entrega para operaciones de sistema mínimas (por ejemplo, manejo de pantalla), aunque su uso fue marginal.

---

4. Documentación del Código

4.1 Estructura del Sistema

- **Árbol General**: Representa todas las ubicaciones principales (Piedecuesta, Floridablanca, Bucaramanga, Girón, Bogotá, Medellín y La Guajira).

- **Árboles de Rutas**: Cada ubicación tiene uno o varios árboles que representan posibles caminos desde esa ciudad hacia otras, con pesos asociados (distancia).

4.2Funciones Principales

- locations(root): Calcula el número de ubicaciones en el árbol principal.
- contarRutas(node): Cuenta cuántos árboles de rutas tiene una ubicación.
- imprimirNumeroDeRutas(raiz): Muestra el número de rutas por ubicación.
- nombresNodos(nodoRaiz): Extrae los nombres de todas las ubicaciones disponibles.
- distanciaMasCorta(root, inicio, final): Encuentra la ruta de menor distancia entre dos ubicaciones.
- main(): Función principal que permite la interacción del usuario con el sistema.

4.3Ejecución

El programa permite al usuario:

- Verificar el número de ubicaciones y rutas disponibles.
- Visualizar el árbol completo de ubicaciones.
- Ingresar origen y destino para encontrar la ruta más corta.
- Recibir mensajes de confirmación si la llegada es a **Piedecuesta** (mensaje especial de bienvenida).

---

5. ¿Por qué los Árboles son Mejores que las Listas para este Problema?

Aspecto	Listas Enlazadas	Árboles
Organización	Lineal y secuencial, difícil de escalar.	Jerárquica, facilita expansión y ramificación de rutas.
Búsqueda	O(n) en el peor caso (secuencial).	O(log n) o mejor si el árbol está equilibrado.
Flexibilidad	Difícil gestionar múltiples caminos y bifurcaciones.	Permite múltiples rutas y comparaciones directas.
Complejidad	Más sencilla de implementar inicialmente.	Más estructurada, pero más compleja de programar y mucho más eficiente a largo plazo.
Visualización	Pobre representación de jerarquías.	Representa claramente jerarquías y relaciones de caminos.

6. Análisis de eficiencia con Big O:

- **Método locations:**

```
def locations(self):
    return len(list(root.descendants)) + 1
```

El atributo descendants generalmente representa todos los nodos descendientes de un nodo dado. Si estamos recorriendo todos los nodos descendientes, la complejidad sería proporcional al número total de nodos descendientes de root, es decir, **O(n)**, donde **n** es el número total de nodos en el árbol.
Método contarRutas:

- **Método contarRutas:**

```
def contarRutas(node):
    return len([
        attr for attr, val in vars(node).items()
        if attr.startswith("rutas") and isinstance(val, Node)
    ])
```

Para cada nodo, iterar sobre todos sus atributos rutas y verificar si son instancias de Node también es una operación que, en el peor caso, requiere recorrer todos los atributos del nodo.Si el número de atributos de un nodo es **m**, entonces la complejidad de este método es **O(m)** por nodo.

- **Método imprimirNumeroDeRutas:**

```
def imprimirNumeroDeRutas(raiz):
    resultado = {}
    for node in [raiz] + list(raiz.descendants):
        resultado[node.name] = contarRutas(node)
    return resultado
```

El bucle itera sobre todos los nodos, incluyendo la raíz y todos los descendientes del árbol. Si **n** es el número de nodos, entonces el bucle tiene **n** iteraciones.Tiene una complejidad total de **O(n · m)**, donde **n** es el número de nodos y **m** es el número de atributos por nodo.

- **Método nombresNodos**

```
def nombresNodos(nodoRaiz):
    nodeNames = [nodoRaiz.name]  # Empieza con el nombre de la raiz
    nodeNames.extend([node.name for node in nodoRaiz.descendants])
    return nodeNames
```

Iterar sobre los descendientes del nodo raíz requiere **O(n)** donde **n** es el número de nodos descendientes.

- **Método distanciaMasCorta:**

```
def distanciaMasCorta(root, inicio, final):
    # 1) Encontrar el nodo de partida en el árbol principal
    nodo_inicio = find_name(root, inicio)
    if not nodo_inicio:
        return -1, None

    candidatas = []
    # 2) Iterar sobre todos los subárboles de rutas (ej: rutas0, rutas1)
    for attr in vars(nodo_inicio):
        if not attr.startswith("rutas"):
            continue
        subarbol = getattr(nodo_inicio, attr)  # Subárbol actual (ej: R0PIEDECUESTA)
```

```
    # 3) Buscar todos los nodos en el subárbol que coincidan con el destino
    for nodo in subarbol.descendants:
        if nodo.name.endswith(final):
            # Construir la ruta relativa DESDE subarbol hasta nodo
            path_parts = []
            current = nodo
            while current != subarbol and current is not None:
                path_parts.append(current.name)
                current = current.parent
            if current != subarbol:
                continue  # Si no está bajo el subarbol, ignorar
            path_parts.reverse()
```

```
                # 4) Reconstruir el camino desde el nodo de inicio del subárbol
                camino = [subarbol]  # Incluir la raíz del subárbol
                for part in path_parts:
                    current_node = find_name(subarbol, part)
                    if current_node:
                        camino.append(current_node)

                if camino:
                    # La distancia ya está precalculada en el nodo destino
                    dist = getattr(nodo, "distance", 0)
                    candidatas.append((dist, camino))
```

```python
# 5) Si no hay rutas válidas
if not candidatas:
    return -1, None

# 6) Seleccionar la ruta con la menor distancia
distancia, camino = min(candidatas, key=lambda x: x[0])
print("Camino más corto:", [n.name for n in camino])
return distancia, camino
```

El método busca un nodo de inicio, lo cual es una operación de búsqueda, **O(n)** donde **n** es el número de nodos.Luego itera sobre los subárboles de rutas, donde se evalúan todos los atributos que empiezan con rutas. Si hay **k** atributos de ruta, la complejidad de este paso es **O(k)**. Para cada subárbol, recorre todos los descendientes, lo que puede tener una complejidad de **O(n)**.En el peor de los casos, el número de nodos revisados será **O(n · k)**, ya que por cada atributo de ruta se revisan todos los nodos. La complejidad total sería **O(n · k · m)**, donde **n** es el número total de nodos, **k** es el número de atributos rutas, y **m** es el número de nodos en cada subárbol.

Conclusión:

El uso de árboles permite modelar sistemas de navegación de manera mucho más realista, soportando múltiples rutas alternativas entre ubicaciones. Esto facilita encontrar caminos más cortos, analizar el sistema como una red completa y gestionar dinámicamente nuevas rutas o cambios. Además, mejora significativamente la eficiencia en la búsqueda de rutas, optimizando el tiempo de ejecución conforme crece el número de ubicaciones.

---

7. Integrantes del Grupo

- Juan Daniel Torres Ramírez - 2240082
- Juan David Mejía Fragoso - 2240085
- Miguel Ángel Aguilar Rodríguez - 2240030
- Sebastián Nossa Agudelo - 2211555