</ Implementación del proyecto con Listas

/>

} /> [

Miguel Angel Aguilar Rodriguez - 2240030.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Estructura utilizada

- Clase NodoRuta
- Clase NodoUbicacion
- Clase RedDeUbicaciones

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Librerías utilizadas

- Time

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 1 1 0 1

</ Métodos

```python
class NodoRuta:
    def __init__(self, destino, peso):
        self.destino = destino  # Nombre del destino
        self.peso = peso  # Peso de la ruta (tiempo)
        self.siguiente = None  # Apunta al siguiente vecino en la lista de rutas
        self.anterior = None  # Apunta al vecino anterior en la lista de rutas
```

```python
class NodoUbicacion:
    def __init__(self, ubicacion):
        self.ubicacion = ubicacion  # Nombre de la ubicación
        self.rutas = None  # Lista enlazada de rutas
        self.siguiente = None  # Apunta al siguiente nodo en la lista de ubicaciones
        self.anterior = None  # Apunta al nodo anterior en la lista de ubicaciones
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Métodos

```python
class RedDeUbicaciones:
    def __init__(self):
        self.head = None  # Primer nodo de la lista de ubicaciones
        self.tail = None  # Cola

    def esta_vacia(self):
        # Verifica si la lista de ubicaciones está vacía
        return self.head is None

    def agregar_ubicacion(self, ubicacion):
        # Agrega una ubicación al comienzo de la lista
        nuevo_nodo = NodoUbicacion(ubicacion)
        if self.head is None:
            self.head = nuevo_nodo
            self.tail = nuevo_nodo
        else:
            nuevo_nodo.siguiente = self.head
            self.head.anterior = nuevo_nodo
            self.head = nuevo_nodo
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Métodos

```python
def agregar_ruta(self, origen, destino, peso):
    nodo_origen = self.buscar_ubicacion(origen)
    if nodo_origen:
        nueva_ruta = NodoRuta(destino, peso)
        if nodo_origen.rutas is None:
            nodo_origen.rutas = nueva_ruta
        else:
            ruta_actual = nodo_origen.rutas
            while ruta_actual.siguiente:
                ruta_actual = ruta_actual.siguiente
            ruta_actual.siguiente = nueva_ruta
            nueva_ruta.anterior = ruta_actual

def ordenar_rutas(self, ubicacion):
    # Ordena las rutas de una ubicación específica con Merge Sort
    nodo_origen = self.buscar_ubicacion(ubicacion)
    if nodo_origen and nodo_origen.rutas:
        nodo_origen.rutas = self.merge_sort_rutas(nodo_origen.rutas)
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Métodos

```python
def actualizar_tail(self):
    # Actualiza la cola para que apunte al último nodo de la lista
    actual = self.head
    while actual and actual.siguiente:
        actual = actual.siguiente
    self.tail = actual

def ordenar_ubicaciones(self):
    # Ordena la lista de ubicaciones con Merge Sort y actualiza la cola
    self.head = self.merge_sort(self.head)
    self.actualizar_tail()
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Métodos

```python
def buscar_ubicacion(self, ubicacion):
    # Busca una ubicación en la lista de ubicaciones
    actual = self.head
    while actual:
        if actual.ubicacion == ubicacion:
            return actual
        if actual.ubicacion > ubicacion:
            return None
        actual = actual.siguiente
    return None

def contar_ubicaciones(self):
    # Cuenta cuántas ubicaciones hay en la lista
    contador = 0
    actual = self.head
    while actual:
        contador += 1
        actual = actual.siguiente
    return contador
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Métodos

```python
def encontrar_ruta(self, inicio, destino, visitados=None, camino=None, costo=0):
    # Buscar ruta entre dos ubicaciones, considerando rutas indirectas
    if visitados is None:
        visitados = set()
    if camino is None:
        camino = []

    nodo_inicio = self.buscar_ubicacion(inicio)
    if nodo_inicio is None:
        print("El nodo de inicio no existe.")
        return None, float('inf')

    # Si llegamos al destino, devolvemos el camino y el costo
    if inicio == destino:
        return camino + [inicio], costo

    # Marcar el nodo de inicio como visitado
    visitados.add(inicio)

    mejor_camino = None
    mejor_costo = float('inf')
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Métodos

```
# Explorar los vecinos
ruta_actual = nodo_inicio.rutas
while ruta_actual:
    if ruta_actual.destino not in visitados:
        nuevo_camino, nuevo_costo = self.encontrar_ruta(ruta_actual.destino, destino, visitados, camino + [inicio], costo + ruta_actual.peso)

        # Si encontramos un mejor camino, lo almacenamos
        if nuevo_camino and nuevo_costo < mejor_costo:
            mejor_camino = nuevo_camino
            mejor_costo = nuevo_costo

    ruta_actual = ruta_actual.siguiente

visitados.remove(inicio)
return mejor_camino, mejor_costo
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Pruebas

```
# Agregar ubicaciones
red.agregar_ubicacion("Piedecuesta")
red.agregar_ubicacion("Bucaramanga")
red.agregar_ubicacion("Bogota")
red.agregar_ubicacion("Medellin")
red.agregar_ubicacion("La Guajira")
red.agregar_ubicacion("Leticia")
red.agregar_ubicacion("Cucuta")
red.agregar_ubicacion("Caracas")
```

```
# Ordenar las ubicaciones después de agregarlas
red.ordenar_ubicaciones()

# Agregar rutas entre las ubicaciones
red.agregar_ruta("Piedecuesta", "Bucaramanga", 1.0)
red.agregar_ruta("Piedecuesta", "Bogota", 8.0)
red.agregar_ruta("Bucaramanga", "Medellin", 7.0)
red.agregar_ruta("Bucaramanga", "Leticia", 15.0)
red.agregar_ruta("Bogota", "Medellin", 6.5)
red.agregar_ruta("Bogota", "Caracas", 24.0)
red.agregar_ruta("Bogota", "La Guajira", 12.0)
red.agregar_ruta("Medellin", "Piedecuesta", 9.0)
red.agregar_ruta("Medellin", "Bogota", 9.5)
red.agregar_ruta("La Guajira", "Medellin", 14.0)
red.agregar_ruta("La Guajira", "Piedecuesta", 21.0)
red.agregar_ruta("Leticia", "Bogota", 13.0)
red.agregar_ruta("Leticia", "La Guajira", 19.0)
red.agregar_ruta("Cucuta", "Bucaramanga", 9.0)
red.agregar_ruta("Cucuta", "Leticia", 18.0)
red.agregar_ruta("Caracas", "Bogota", 24.0)
red.agregar_ruta("Caracas", "Piedecuesta", 22.0)
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Pruebas

```
# Ordenar las rutas de cada ubicación
red.ordenar_rutas("Piedecuesta")
red.ordenar_rutas("Bucaramanga")
red.ordenar_rutas("Bogota")
red.ordenar_rutas("Medellin")
red.ordenar_rutas("La Guajira")

# Imprimir la red de ubicaciones y rutas
print("Red de ubicaciones y rutas ordenadas:")
red.imprimir_red()

# Medir el tiempo para encontrar la ruta más corta de Bucaramanga a Caracas
inicio = "Bucaramanga"
fin = "Caracas"
print(f"\nBuscando la ruta más corta de {inicio} a {fin}...")
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Pruebas

```python
start_time = time.time()  # Tomar tiempo de inicio
camino, costo = red.encontrar_ruta(inicio, fin)
end_time = time.time()  # Tomar tiempo de fin

if camino:
    print(f"\nCamino más corto: {' -> '.join(camino)}")
    print(f"Costo total: {costo}")
else:
    print(f"No se encontró un camino desde {inicio} a {fin}.")

print(f"\nTiempo de búsqueda: {end_time - start_time} segundos.")
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Resultados

```
Red de ubicaciones y rutas ordenadas:
Ubicación: Bogota
   -> Ruta a Caracas con costo 24.0
   -> Ruta a La Guajira con costo 12.0
   -> Ruta a Medellin con costo 6.5
Ubicación: Bucaramanga
   -> Ruta a Leticia con costo 15.0
   -> Ruta a Medellin con costo 7.0
Ubicación: Caracas
   -> Ruta a Bogota con costo 24.0
   -> Ruta a Piedecuesta con costo 22.0
Ubicación: Cucuta
   -> Ruta a Bucaramanga con costo 9.0
   -> Ruta a Leticia con costo 18.0
```

```
Ubicación: La Guajira
   -> Ruta a Medellin con costo 14.0
   -> Ruta a Piedecuesta con costo 21.0
Ubicación: Leticia
   -> Ruta a Bogota con costo 13.0
   -> Ruta a La Guajira con costo 19.0
Ubicación: Medellin
   -> Ruta a Bogota con costo 9.5
   -> Ruta a Piedecuesta con costo 9.0
Ubicación: Piedecuesta
   -> Ruta a Bogota con costo 8.0
   -> Ruta a Bucaramanga con costo 1.0
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Resultados

```
Buscando la ruta más corta de Bucaramanga a Caracas...

Camino más corto: Bucaramanga -> Medellin -> Bogota -> Caracas
Costo total: 40.5

Tiempo de búsqueda: 8.153915405273438e-05 segundos.
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Implementación del proyecto final con Árboles />

} /> [

Juan Daniel Torres Ramirez - 2240082

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 1 0 1

</>

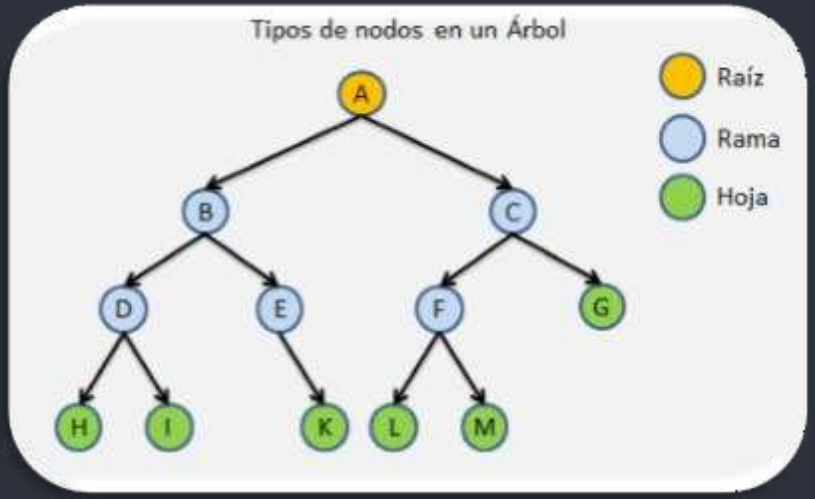
PARTE #1

01

} /> [

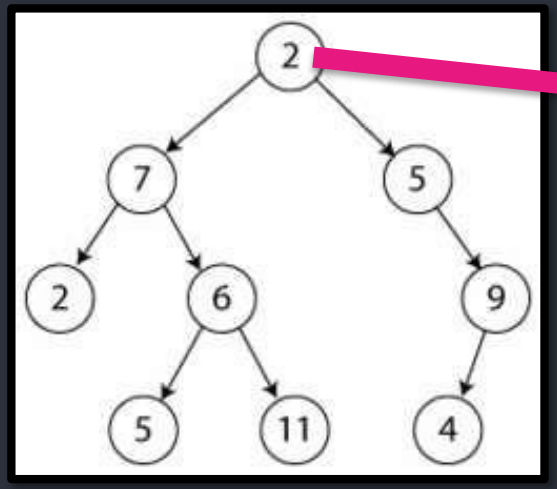
1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 1 0 1

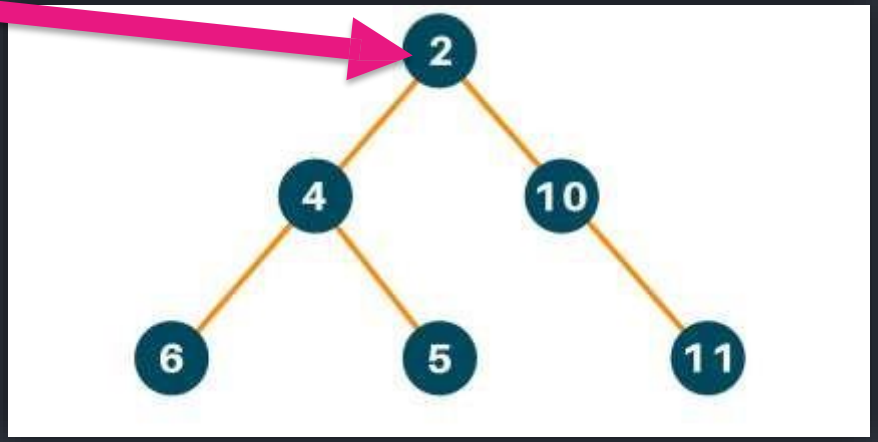
</ ¿Cómo se realizó?



1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 1 1 0 1

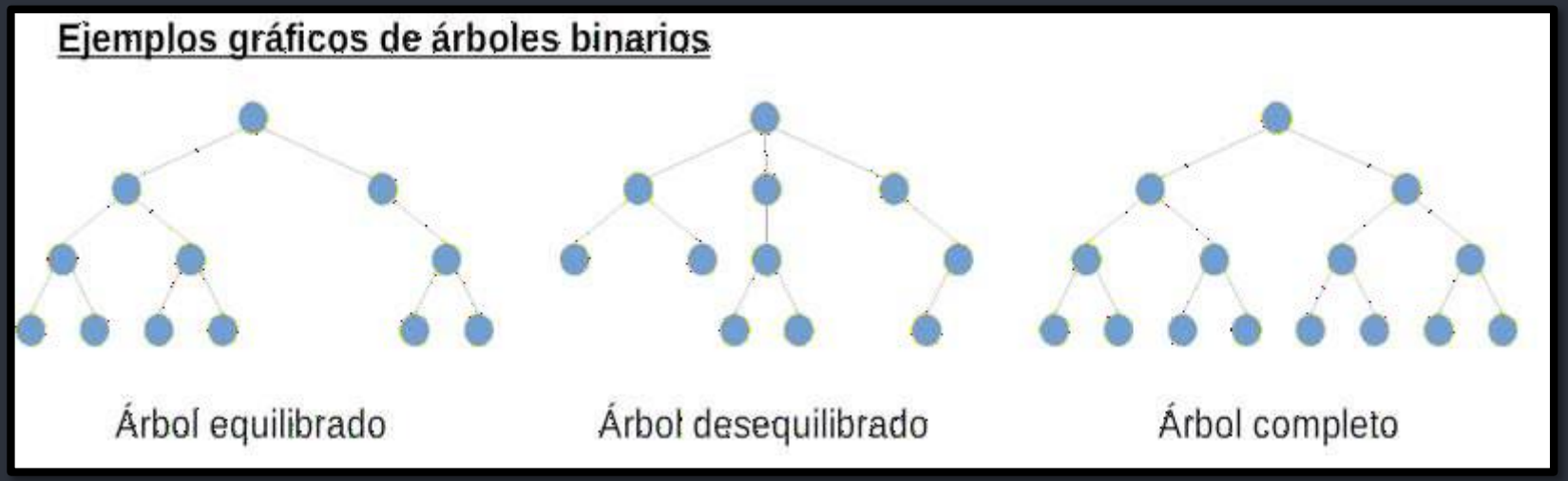
</ ¿Cómo se pensó?





1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 1 0 1

</ ¿qué tipo de arboles acepta?



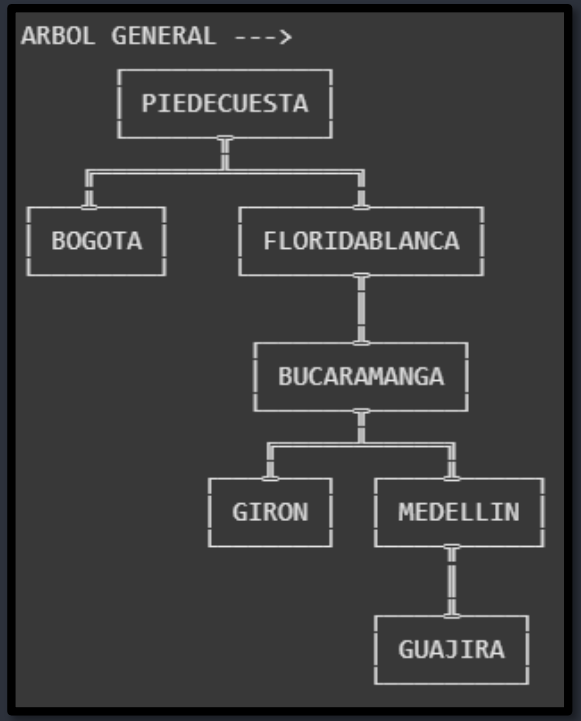
1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 1 0 1

</ ¿Cómo se pensó?

```
A continuacion, le mostraremos el numero de ubicaciones y el numero de rutas que posee cada ubicacion
El número de ubicaciones es: 7
El número de rutas por cada ubicación:
  - PIEDECUESTA: 2 rutas
  - BOGOTA: 2 rutas
  - FLORIDABLANCA: 2 rutas
  - BUCARAMANGA: 2 rutas
  - GIRON: 2 rutas
  - MEDELLIN: 2 rutas
  - GUAJIRA: 2 rutas
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Árbol General



1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Ubicaciones y sus relaciones

```
#Relaciones padre e hijos
R0PIEDECUESTA.children = [R0BOGOTA, R0FLORIDABLANCA]
R0BUCARAMANGA.children = [R0GIRON, R0MEDELLIN]
R0FLORIDABLANCA.children = [R0BUCARAMANGA]
R0MEDELLIN.children = [R0GUAJIRA]
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Ubicaciones y sus relaciones

```
#Para PIEDECUESTA (R0)
#Se definen los nodos
R0PIEDECUESTA = Node("R0PIEDECUESTA", distance = 0)
R0FLORIDABLANCA = Node("R0FLORIDABLANCA", parent=R0PIEDECUESTA, distance = 5)
R0BUCARAMANGA = Node("R0BUCARAMANGA", parent=R0FLORIDABLANCA, distance = (5+7))
R0GIRON = Node("R0GIRON", parent=R0BUCARAMANGA, distance = (5+7+4))
R0BOGOTA = Node("R0BOGOTA", parent=R0PIEDECUESTA, distance = (440))
R0MEDELLIN = Node("R0MEDELLIN", parent=R0BUCARAMANGA, distance = (5+7+4+400))
R0GUAJIRA = Node("R0GUAJIRA", parent=R0MEDELLIN, distance = (5+7+4+400+550)) #Vemos
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Ubicaciones y sus relaciones

```
# Para PIEDECUESTA (R01) - Segunda opción
R01PIEDECUESTA = Node("R01PIEDECUESTA", distance=0)
R01BOGOTA = Node("R01BOGOTA", parent=R01PIEDECUESTA, distance=450)  #Ruta direc
R01FLORIDABLANCA = Node("R01FLORIDABLANCA", parent=R01PIEDECUESTA, distance=6)
R01BUCARAMANGA = Node("R01BUCARAMANGA", parent=R01FLORIDABLANCA, distance=13)
R01GIRON = Node("R01GIRON", parent=R01BUCARAMANGA, distance=18)  #Ruta a Girón
R01MEDELLIN = Node("R01MEDELLIN", parent=R01BUCARAMANGA, distance=420)  #Ruta a
R01GUAJIRA = Node("R01GUAJIRA", parent=R01MEDELLIN, distance=970) #Ruta a la gu

R01PIEDECUESTA.children = [R01BOGOTA, R01FLORIDABLANCA]
R01FLORIDABLANCA.children = [R01BUCARAMANGA]
R01BUCARAMANGA.children = [R01GIRON, R01MEDELLIN]
R01MEDELLIN.children = [R01GUAJIRA]
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Rutas de Piedecuesta

```
****Rutas de PIEDECUESTA --->

R0PIEDECUESTA [distance=0]
|-- R0BOGOTA [distance=440]
`-- R0FLORIDABLANCA [distance=5]
    `-- R0BUCARAMANGA [distance=12]
        |-- R0GIRON [distance=16]
        `-- R0MEDELLIN [distance=416]
            `-- R0GUAJIRA [distance=966]

R01PIEDECUESTA [distance=0]
|-- R01BOGOTA [distance=450]
`-- R01FLORIDABLANCA [distance=6]
    `-- R01BUCARAMANGA [distance=13]
        |-- R01GIRON [distance=18]
        `-- R01MEDELLIN [distance=420]
            `-- R01GUAJIRA [distance=970]
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Resultados para una ruta específica

```
Ahora porfavor seleccione primero una ubicacion de partida y luego una de destino
Porfavor ingrese una ubicacion valida (Ingrese exactamente lo que aparece de nombre (Mayusculas incluidas)): PIEDECUESTA
Ahora porfavor ingrese la ubicacion de destino:
Porfavor ingrese una ubicacion valida (Ingrese exactamente lo que aparece de nombre (Mayusculas incluidas)): BUCARAMANGA

La respuesta es:

Camino más corto: ['R0PIEDECUESTA', 'R0FLORIDABLANCA', 'R0BUCARAMANGA']
La distancia más corta entre PIEDECUESTA y BUCARAMANGA es: 12
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</>

PARTE #2

02

} /> [

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1

</ Métodos

```python
#Se debe introducir el arbol raiz del arbol de ubicaciones (El general)
def locations(self):
  return len(list(root.descendants)) + 1

def contarRutas(node):
  #Cuenta el número de arboles de ruta (atributos de tipo arbol (nodos)) para un nodo dado.
  return len([
        attr for attr, val in vars(node).items()
        if attr.startswith("rutas") and isinstance(val, Node)
    ])
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Métodos

```python
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()
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Métodos

```
# 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))
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

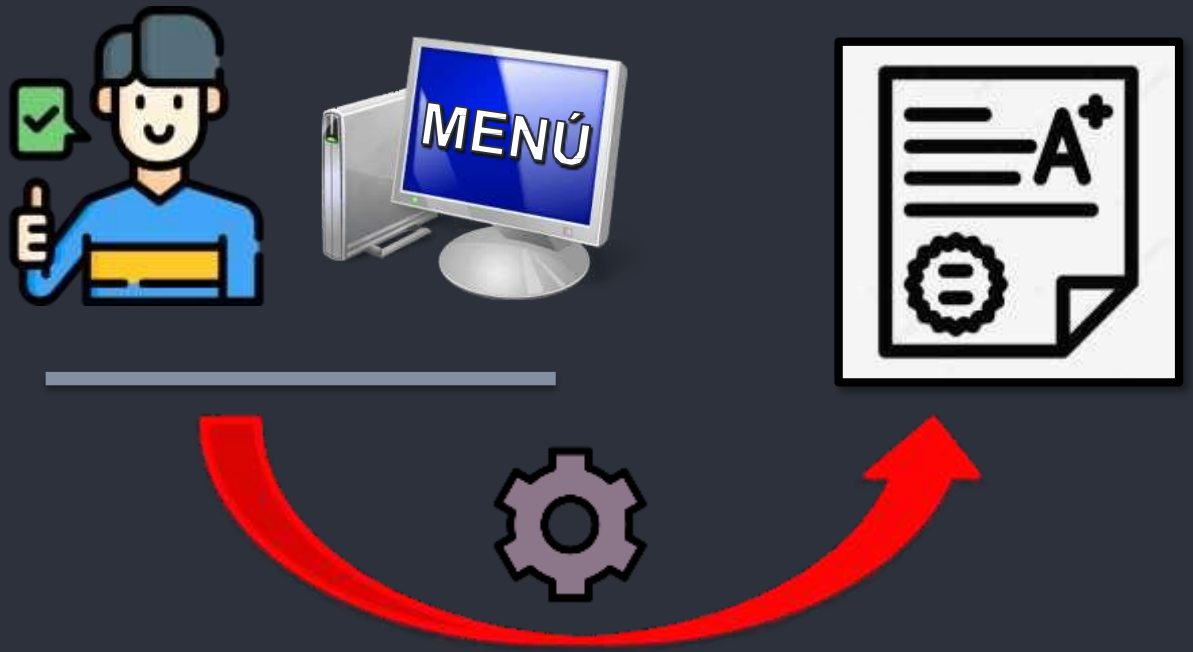
</ Métodos

```
# 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
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Flujo



1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Implementacion del proyecto final con grafos />

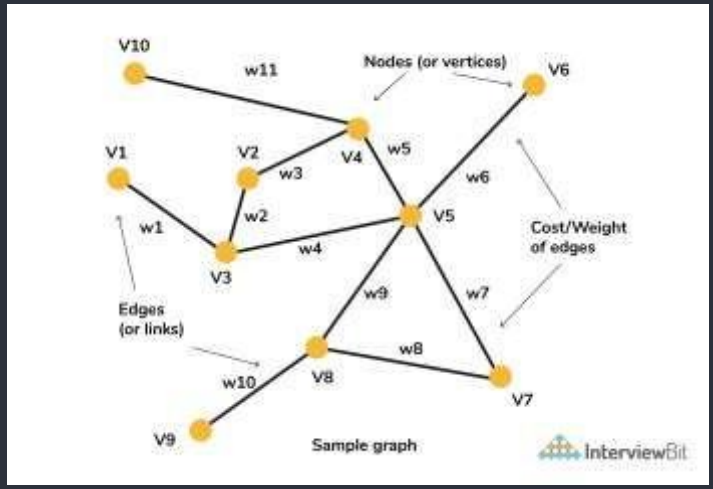
} /> [

Juan David Mejia Fragoso - 2240085.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Estructura utilizada

- Clase Grafo
- Algoritmo de Dijkstra
- Visualización
- Interfaz de usuario



</ Librerías utilizadas

- Heapq
- Networkx
- Matplotlib

NetworkX
Network Analysis in Python

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Métodos

```python
def añadir_arista(self, origen, destino, peso):
    if origen not in self.nodos:
        self.nodos[origen] = {}
    self.nodos[origen][destino] = peso
```

```python
def obtener_nodos(self):
    nodos = set(self.nodos.keys())
    for vecinos in self.nodos.values():
        nodos.update(vecinos.keys())
    return list(nodos)
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Métodos

```python
def dijkstra(grafo, inicio):
    distancias = {nodo: float('inf') for nodo in grafo.obtener_nodos()}
    distancias[inicio] = 0
    cola = [(0, inicio)]
    while cola:
        distancia_actual, nodo_actual = heapq.heappop(cola)
        if distancia_actual > distancias[nodo_actual]:
            continue
        for vecino, peso in grafo.nodos.get(nodo_actual, {}).items():
            distancia = distancia_actual + peso
            if distancia < distancias[vecino]:
                distancias[vecino] = distancia
                heapq.heappush(cola, (distancia, vecino))
    return distancias
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Métodos

```python
def visualizar_grafo(grafo):
    G = nx.DiGraph()
    for origen, destinos in grafo.nodos.items():
        for destino, peso in destinos.items():
            G.add_edge(origen, destino, weight=peso)
    if len(G.nodes) == 0:
        print("El grafo está vacío, no se puede visualizar.")
        return
    pos = nx.circular_layout(G)  # Layout compatible
    plt.figure(figsize=(8, 6))
    nx.draw(
        G, pos, with_labels=True, node_color='lightblue',
        node_size=2000, font_size=14, font_weight='bold', arrowsize=20
    )
    etiquetas_peso = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=etiquetas_peso, font_size=12)
    plt.title('Visualización del Grafo')
    plt.axis('off')
    plt.tight_layout()
    plt.show()
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Métodos

```python
def interfaz():
    g = Grafo()
    print("Bienvenido al sistema de creación de grafos.")
    while True:
        print("\nOpciones:")
        print("1. Añadir arista")
        print("2. Visualizar grafo")
        print("3. Calcular rutas mínimas (Dijkstra)")
        print("4. Salir")
        opcion = input("Elige una opción: ")
        if opcion == "1":
            origen = input("Nodo origen: ")
            destino = input("Nodo destino: ")
            try:
                peso = float(input("Peso de la arista: "))
            except ValueError:
                print("Peso inválido. Intenta de nuevo.")
                continue
            g.añadir_arista(origen, destino, peso)
            print(f"Arista {origen} -> {destino} con peso {peso} añadida.")
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Métodos

```
        elif opcion == "2":
            if not g.nodos:
                print("El grafo está vacío.")
            else:
                visualizar_grafo(g)
        elif opcion == "3":
            if not g.nodos:
                print("El grafo está vacío.")
            else:
                inicio = input("Nodo de inicio para Dijkstra: ")
                if inicio not in g.obtener_nodos():
                    print("Ese nodo no existe en el grafo.")
                else:
                    distancias = dijkstra(g, inicio)
                    print("Distancias mínimas desde", inicio)
                    for nodo, distancia in distancias.items():
                        print(f"{inicio} -> {nodo}: {distancia}")
        elif opcion == "4":
            print("¡Hasta luego!")
            break
        else:
            print("Opción no válida. Intenta de nuevo.")

if __name__ == "__main__":
    interfaz()
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

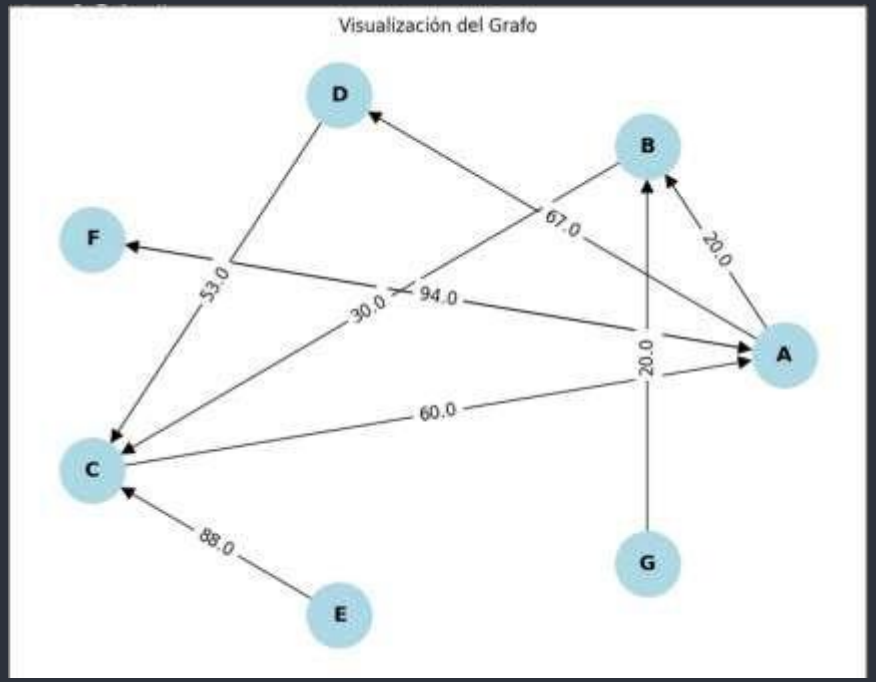
</ Resultados

```
Bienvenido al sistema de creación de grafos.

Opciones:
1. Añadir arista
2. Visualizar grafo
3. Calcular rutas mínimas (Dijkstra)
4. Salir
Elige una opción:
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Resultados



1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 1 0 1

</ Resultados

```
Opciones:
1. Añadir arista
2. Visualizar grafo
3. Calcular rutas mínimas (Dijkstra)
4. Salir
Elige una opción: 3
Nodo de inicio para Dijkstra: A
Distancias mínimas desde A
A -> G: inf
A -> A: 0
A -> D: 67.0
A -> F: 94.0
A -> B: 20.0
A -> C: 50.0
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Conclusiones Finales del Proyecto

/>

} /> [

Sebastián Nossa Agudelo-
2211555.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Conclusión General

A lo largo del proyecto se implementaron tres soluciones para encontrar rutas óptimas: listas enlazadas, árboles y grafos. Cada una permitió profundizar en estructuras de datos diferentes y evaluar su impacto en la eficiencia de búsqueda.

</ La complejidad del sistema aumentó progresivamente.
</ Se evidenció cómo la estructura elegida afecta el rendimiento.
</ Se comprobó que las soluciones más sofisticadas permiten escalar el sistema sin perder eficiencia.

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</ Conclusiones por Entrega

</ Entrega 1 – Listas Enlazadas

</ Estructura simple pero limitada.

</ Búsqueda por fuerza bruta ($\mathcal{O}(n!)$).

</ No recomendable para redes grandes.

</ Entrega 3 – Grafos

</ Uso de algoritmo de Dijkstra con diccionarios.

</ Complejidad: $\mathcal{O}((V + E) \log V)$.

</ Alta eficiencia, flexibilidad y escalabilidad.

</ Entrega 2 – Árboles

</ Mejora en eficiencia usando distancias precalculadas.

</ Complejidad baja: $\mathcal{O}(k)$, donde k = rutas posibles.

</ Ideal para rutas jerárquicas.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Tabla Comparativa

Criterio	Listas	Árboles	Grafos
Eficiencia	Muy baja ($\mathcal{O}(n!)$)	Media ($\mathcal{O}(k)$)	Alta ($\mathcal{O}((V+E)\log V)$)
Escalabilidad	Baja	Media	Alta
Flexibilidad	Limitada	Moderada	Alta
Visualización	No	Parcial	Completa

V = número de nodos (ubicaciones), E = número de aristas(ubicaciones), K = rutas cargadas

1011 011 01 1011001 10 11011 011 01 110110 110111 1101