

ANÁLISIS DEL RETO 4

1. ***req No.3**, David Elias Forero Cobos, de.foreroc1@uniandes.edu.co, 202310499.*
2. ***req No.4**, Alejandro Garcia Rojas, a.garcia23@uniandes.edu.co, 202122516.*
3. ***req No.5**, Pablo Andrés Sebastián Parra Céspedes, p.parrac@uniandes.edu.co, 202310768.*

Requerimiento 1

Descripción

```
# Funciones de consulta
def req1(dataStructs, startPoint, arrivalPoint):
    filtered = lt.newList('ARRAY_LIST')
    foliumMap = lt.newList()
    drawPath = []
    metaData = {'totalVertex': 0, 'totalDistance': 0}
    if isInside(dataStructs, startPoint) and isInside(dataStructs, arrivalPoint):
        startPoint = getClosestVertex(dataStructs, startPoint)
        arrivalPoint = getClosestVertex(dataStructs, arrivalPoint)
        paths = dfs.DepthFirstSearch(dataStructs['distanceGraph'], startPoint['id'])
        path = dfs.pathTo(paths, arrivalPoint['id'])
        if path:
            metaData['totalVertex'] = st.size(path)
            datum = {'Starting Point': 'Satarting point', 'Arrival Point': st.top(path), 'Distance': startPoint['distance']}
            lt.addLast(filtered, datum)
            while st.size(path) > 1:
                startingVertex = st.pop(path)
                arrivalVertex = st.top(path)
                distance = gr.getEdge(dataStructs['distanceGraph'], startingVertex, arrivalVertex)['weight']
                metaData['totalDistance'] += distance
                datum = {'Starting Point': startingVertex, 'Arrival Point': arrivalVertex, 'Distance': distance}
                lt.addLast(filtered, datum)
                startingVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], startingVertex))
                arrivalVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], arrivalVertex))
                lt.addLast(foliumMap, startingVertexDetails)
                lt.addLast(foliumMap, arrivalVertexDetails)
                drawPath.append((startingVertexDetails['coordinates'], arrivalVertexDetails['coordinates']))
            datum = {'Starting Point': st.top(path), 'Arrival Point': 'Arrival point', 'Distance': arrivalPoint['distance']}
            lt.addLast(filtered, datum)
            metaData['totalDistance'] += startPoint['distance']
            metaData['totalDistance'] += arrivalPoint['distance']
        metaData["map"] = lt.size(foliumMap) > 0
        metaData["path"] = "req1"
        req8(foliumMap, "req1", drawPath)
    return filtered, metaData
```

Entrada

Las entradas esperadas en el requerimiento 1 son las siguientes:

- Una estructura de datos con los datos a consultar.
- Un punto de origen (una localización geográfica con latitud y longitud).

	<ul style="list-style-type: none"> - Un punto de destino (una localización geográfica con latitud y longitud).
Salida	<p>Las salidas esperadas en el requerimiento 1 son las siguientes:</p> <ul style="list-style-type: none"> - El tiempo que se demora el algoritmo en encontrar la solución (en milisegundos). - La distancia total que tomará el camino entre el punto de origen y el de destino. - El total de vértices que contiene el camino encontrado. - La secuencia de vértices (sus identificadores) que componen el camino encontrado.
Implementado	Sí, el requerimiento fue implementado, desarrollado y completado por el grupo No.1 de la sección No.3 del curso de Estructuras de Datos y Algoritmos.

Análisis de complejidad

Pasos	Complejidad
def req1(dataStructs, startPoint, arrivalPoint):	$O(1)$
filtered = lt.newList('ARRAY_LIST')	$O(1)$
foliumMap = lt.newList()	$O(1)$
drawPath = []	$O(1)$
metaData = {'totalVertex': 0, 'totalDistance': 0}	$O(1)$
if isInside(dataStructs, startPoint) and isInside(dataStructs, arrivalPoint):	$O(1)$
startPoint = getClosestVertex(dataStructs, startPoint)	$O(V)$
arrivalPoint = getClosestVertex(dataStructs, arrivalPoint)	$O(V)$
paths = dfs.DepthFirstSearch(dataStructs['distanceGr aph'], startPoint['id'])	$O(V + E)$
path = dfs.pathTo(paths, arrivalPoint['id'])	$O(M)$ (donde $M \leq V$)
if path:	$O(1)$
metaData['totalVertex'] = st.size(path)	$O(1)$
datum = {'Starting Point': 'Satarting point',	$O(1)$

'Arrival Point': st.top(path), 'Distance': startPoint['distance']}]}	
lt.addLast(filtered, datum)	O(1)
while st.size(path) > 1:	O(1)
startingVertex = st.pop(path)	O(1)
arrivalVertex = st.top(path)	O(1)
distance = gr.getEdge(dataStructs['distanceGraph'], startingVertex, arrivalVertex)['weight']	O(1)
metaData['totalDistance'] += distance	O(1)
datum = {'Starting Point': startingVertex, 'Arrival Point': arrivalVertex, 'Distance': distance}	O(1)
lt.addLast(filtered, datum)	O(1)
startingVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], startingVertex))	O(1)
arrivalVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], arrivalVertex))	O(1)
lt.addLast(foliumMap, startingVertexDetails)	O(1)
lt.addLast(foliumMap, arrivalVertexDetails)	O(1)
drawPath.append((startingVertexDetails['coor dinates'], arrivalVertexDetails['coordinates']))	O(1)
datum = {'Starting Point': st.top(path), 'Arrival Point': 'Arrival point', 'Distance': arrivalPoint['distance']}	O(1)
lt.addLast(filtered, datum)	O(1)
metaData['totalDistance'] += startPoint['distance']	O(1)
metaData['totalDistance'] += arrivalPoint['distance']	O(1)
metaData["map"] = lt.size(foliumMap) > 0	O(1)

metaData["path"] = "req1"	$O(1)$
req8(foliumMap, "req1", drawPath)	$O(M + O)$ (donde M y $O \leq V$)
return filtered, metaData	$O(1)$
Total	$O(V+E)$

Pruebas Realizadas

Procesador	Apple M1
Memoria RAM	8 GB
Sistema Operativo	macOS Ventura Versión 13.5.2

Entrada	Tiempo (s)	Memoria (MB)
large	1964.817	452.911

Análisis

La solución para el requerimiento 1 tiene una complejidad en el peor caso de $V+E$, donde V representa los vértices y E los arcos. Esta complejidad se debe a la utilización del algoritmo de Búsqueda en Profundidad (Depth First Search, DFS).

El programa primero asegura que los puntos ingresados estén dentro de los límites de la ciudad. Luego, aproxima estos puntos a los vértices más cercanos en la malla vial, que es representada como un grafo. Utilizando el algoritmo DFS, el sistema busca un camino entre estos vértices. Como resultado, el programa proporciona información detallada que incluye la distancia total del camino, el número total de vértices que lo componen, y una secuencia específica de vértices que forman el camino encontrado.

Requerimiento 2

Descripción

```
def req2(dataStructs, startPoint, arrivalPoint):
    filtered = lt.newList('ARRAY_LIST')
    metaData = {'totalVertex': 0, 'totalDistance': 0}
    foliumMap = lt.newList()
    drawPath = []
    if isInside(dataStructs, startPoint) and isInside(dataStructs, arrivalPoint):
        startPoint = getClosestVertex(dataStructs, startPoint)
        arrivalPoint = getClosestVertex(dataStructs, arrivalPoint)
        paths = bfs.BreathFirstSearch(dataStructs['distanceGraph'], startPoint['id'])
        path = bfs.pathTo(paths, arrivalPoint['id'])
        if path:
            metaData['totalVertex'] = st.size(path)
            datum = {'Starting Point': 'Starting point', 'Arrival Point': st.top(path), 'Distance': startPoint['distance']}
            lt.addLast(filtered, datum)
            while st.size(path) > 1:
                startingVertex = st.pop(path)
                arrivalVertex = st.top(path)
                distance = gr.getEdge(dataStructs['distanceGraph'], startingVertex, arrivalVertex)['weight']
                metaData['totalDistance'] += distance
                datum = {'Starting Point': startingVertex, 'Arrival Point': arrivalVertex, 'Distance': distance}
                lt.addLast(filtered, datum)
                startingVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], startingVertex))
                arrivalVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], arrivalVertex))
                lt.addLast(foliumMap, startingVertexDetails)
                lt.addLast(foliumMap, arrivalVertexDetails)
                drawPath.append((startingVertexDetails['coordinates'], arrivalVertexDetails['coordinates']))
            datum = {'Starting Point': st.top(path), 'Arrival Point': 'Arrival point', 'Distance': arrivalPoint['distance']}
            lt.addLast(filtered, datum)
            metaData['totalDistance'] += startPoint['distance']
            metaData['totalDistance'] += arrivalPoint['distance']
        metaData["map"] = lt.size(foliumMap) > 0
        metaData["path"] = "req2"
        req8(foliumMap, "req2", drawPath)
    return filtered, metaData
```

Entrada	<p>Las entradas esperadas en el requerimiento 2 son las siguientes:</p> <ul style="list-style-type: none"> - Una estructura de datos con los datos a consultar. - Un punto de origen (una localización geográfica con latitud y longitud). - Un punto de destino (una localización geográfica con latitud y longitud).
Salida	<p>Las salidas esperadas en el requerimiento 2 son las siguientes:</p> <ul style="list-style-type: none"> - El tiempo que se demora el algoritmo en encontrar la solución (en milisegundos). - La distancia total que tomará el camino entre el punto de encuentro de origen y el de destino. - El total de vértices que contiene el camino encontrado. - La secuencia de vértices (sus identificadores) que componen el camino encontrado.
Implementado	<p>Sí, el requerimiento fue implementado, desarrollado y completado por el grupo No.1 de la sección No.3 del curso de Estructuras de Datos y Algoritmos.</p>

Análisis de complejidad

Pasos	Complejidad
-------	-------------

def req2(dataStructs, startPoint, arrivalPoint):	$O(1)$
filtered = lt.newList('ARRAY_LIST')	$O(1)$
metaData = {'totalVertex': 0, 'totalDistance': 0}	$O(1)$
foliumMap = lt.newList()	$O(1)$
drawPath = []	$O(1)$
if isInside(dataStructs, startPoint) and isInside(dataStructs, arrivalPoint):	$O(1)$
startPoint = getClosestVertex(dataStructs, startPoint)	$O(V)$
arrivalPoint = getClosestVertex(dataStructs, arrivalPoint)	$O(V)$
paths = bfs.BreadhtFirstSearch(dataStructs['distance Graph'], startPoint['id'])	$O(V + E)$
path = bfs.pathTo(paths, arrivalPoint['id'])	$O(M)$ (donde $M \leq V$)
if path:	$O(1)$
metaData['totalVertex'] = st.size(path)	$O(1)$
datum = {'Starting Point': 'Starting point', 'Arrival Point': st.top(path), 'Distance': startPoint['distance']}	$O(1)$
lt.addLast(filtered, datum)	$O(1)$
while st.size(path) > 1:	$O(N)$
startingVertex = st.pop(path)	$O(1)$
arrivalVertex = st.top(path)	$O(1)$
distance = gr.getEdge(dataStructs['distanceGraph'], startingVertex, arrivalVertex)['weight']	$O(1)$
metaData['totalDistance'] += distance	$O(1)$
datum = {'Starting Point': startingVertex, 'Arrival Point': arrivalVertex, 'Distance': distance}	$O(1)$
lt.addLast(filtered, datum)	$O(1)$

startingVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], startingVertex))	O(1)
arrivalVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], arrivalVertex))	O(1)
lt.addLast(foliumMap, startingVertexDetails)	O(1)
lt.addLast(foliumMap, arrivalVertexDetails)	O(1)
drawPath.append((startingVertexDetails['coor dinates'], arrivalVertexDetails['coordinates']))	O(1)
datum = {'Starting Point': st.top(path), 'Arrival Point': 'Arrival point', 'Distance': arrivalPoint['distance']}	O(1)
lt.addLast(filtered, datum)	O(1)
metaData['totalDistance'] += startPoint['distance']	O(1)
metaData['totalDistance'] += arrivalPoint['distance']	O(1)
metaData["map"] = lt.size(foliumMap) > 0	O(1)
metaData["path"] = "req2"	O(1)
req8(foliumMap, "req2", drawPath)	O(M + O) (donde M y O <= V)
return filtered, metaData	O(1)
Total	O(V + E)

Pruebas Realizadas

Procesador	Apple M1
Memoria RAM	8 GB
Sistema Operativo	macOS Ventura Versión 13.5.2

Entrada	Tiempo (s)	Memoria (MB)
large	13.118	15.106

Análisis

La complejidad de este algoritmo es $O(V+E)$, donde V representa los vértices y E los arcos. Esta complejidad se origina del uso de la técnica de Búsqueda en Anchura (Breadth-First Search) implementada para encontrar el camino que cruza el menor número de intersecciones.

El propósito de este requerimiento es hallar una ruta entre dos puntos geográficos en Bogotá que minimice las intersecciones a cruzar. Los usuarios introducen los puntos de origen y destino en forma de coordenadas de latitud y longitud. El sistema verifica que estos puntos se encuentren dentro de los límites urbanos de la ciudad y los aproxima a los vértices más cercanos en la red vial. Como resultado, el algoritmo proporciona la distancia total del camino, el número total de vértices que lo componen y la secuencia específica de estos vértices.

Requerimiento 3

Descripción

```

229
230 def req3(dataStructs, locality, M):
231     filtered = lt.newList('ARRAY_LIST')
232     metaData = {'totalDistance': 0}
233     foliumMap = lt.newList()
234     drawPath = []
235     entry = mp.get(dataStructs['localities'], locality)
236     if entry:
237         if not dataStructs['mstGraph']:
238             mst = prim.PrimMST(dataStructs['distanceGraph'])
239             prim.edgesMST(dataStructs['distanceGraph'], mst)
240             mstGraph = createMstGraph(mst)
241             dataStructs['mstGraph'] = mstGraph
242             mstGraph = dataStructs['mstGraph']
243             localityVertices = sortData(mp.valueSet(me.getValue(entry)), compareByCount)
244             paths = bfs.BreadthFirstSearch(mstGraph, lt.firstElement(localityVertices)['vertex'])
245             arrivalVertexIndex = 2
246             while lt.size(filtered) < M and arrivalVertexIndex <= lt.size(localityVertices):
247                 arrivalVertex = lt.getElement(localityVertices, arrivalVertexIndex)['vertex']
248                 if bfs.hasPathTo(paths, arrivalVertex):
249                     path = bfs.pathTo(paths, arrivalVertex)
250                     while st.size(path) > 1:
251                         startingVertex = st.pop(path)
252                         arrivalVertex = st.top(path)
253                         distance = gr.getEdge(mstGraph, startingVertex, arrivalVertex)['weight']
254                         datum = {'Starting Point': startingVertex, 'Arrival Point': arrivalVertex, 'Distance': round(distance, 4)}
255                         if not existsStep(filtered, datum):
256                             lt.addLast(filtered, datum)
257                             metaData['totalDistance'] += distance
258                             startingVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], startingVertex))
259                             arrivalVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], arrivalVertex))
260                             lt.addLast(foliumMap, startingVertexDetails)
261                             lt.addLast(foliumMap, arrivalVertexDetails)
262                             drawPath.append((startingVertexDetails['coordinates'], arrivalVertexDetails['coordinates']))
263                 arrivalVertexIndex += 1
264             metaData['totalVertices'] = lt.size(filtered) + 1 if lt.size(filtered) > 0 else 0
265             metaData['map'] = lt.size(foliumMap) > 0
266             metaData['path'] = "req3"
267             req8(foliumMap, "req3", drawPath)
268             return filtered, metaData
269
270 > def req4(dataStructs, M):
271
309

```

Entrada	Las entradas esperadas en el requerimiento 3 son las siguientes: <ul style="list-style-type: none"> - Una estructura de datos con los datos a consultar.
Salida	Las salidas esperadas en el requerimiento 3 son las siguientes:

	<ul style="list-style-type: none"> - El tiempo que se demora el algoritmo en encontrar la solución (en milisegundos). - El total de vértices de la red. - Los vértices incluidos (identificadores). - Los arcos incluidos (Id vértice inicial e Id vértice final). - La cantidad de kilómetros de fibra óptica extendida. - El costo (monetario) final.
Implementado	Sí, el requerimiento fue implementado, desarrollado y completado por David Elias Forero Cobos, código 202310499, del grupo No.1 de la sección No.3 del curso de Estructuras de Datos y Algoritmos.

Análisis de complejidad

Pasos	Complejidad
def req3(dataStructs, locality, M):	$O(1)$
filtered = lt.newList('ARRAY_LIST')	$O(1)$
metaData = {'totalDistance': 0}	$O(1)$
foliumMap = lt.newList()	$O(1)$
drawPath = []	$O(1)$
entry = mp.get(dataStructs['localities'], locality)	$O(1)$
if entry:	$O(1)$
if not dataStructs['mstGraph']:	$O(1)$
mst = prim.PrimMST(dataStructs['distanceGraph'])	$O(E \log V)$
prim.edgesMST(dataStructs['distanceGraph'], mst)	$O(V)$
mstGraph = createMstGraph(mst)	$O(V)$
dataStructs['mstGraph'] = mstGraph	$O(1)$
mstGraph = dataStructs['mstGraph']	$O(1)$
localityVertices = sortData(mp.valueSet(me.getValue(entry)), compareByCount)	$O(M^{3/2})$ (donde $M \leq V$)
paths = bfs.BreadhtFirstSearch(mstGraph,	$O(V + E)$

lt.firstElement(localityVertices)['vertex']	
arrivalVertexIndex = 2	O(1)
while lt.size(filtered) < M and arrivalVertexIndex <= lt.size(localityVertices):	O(1)
arrivalVertex = lt.getElement(localityVertices, arrivalVertexIndex)['vertex']	O(1)
if bfs.hasPathTo(paths, arrivalVertex):	O(1)
path = bfs.pathTo(paths, arrivalVertex)	O(M) (donde $M \leq V$)
while st.size(path) > 1:	O(1)
startingVertex = st.pop(path)	O(1)
arrivalVertex = st.top(path)	O(1)
distance = gr.getEdge(mstGraph, startingVertex, arrivalVertex)['weight']	O(1)
datum = {'Starting Point': startingVertex, 'Arrival Point': arrivalVertex, 'Distance': round(distance, 4)}	O(1)
if not existsStep(filtered, datum):	O(M) (donde $M \leq V$)
lt.addLast(filtered, datum)	O(1)
metaData['totalDistance'] += distance	O(1)
startingVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], startingVertex))	O(1)
arrivalVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], arrivalVertex))	O(1)
lt.addLast(foliumMap, startingVertexDetails)	O(1)
lt.addLast(foliumMap, arrivalVertexDetails)	O(1)
drawPath.append((startingVertexDetails['coor dinates'], arrivalVertexDetails['coordinates']))	O(1)
arrivalVertexIndex += 1	O(1)
metaData['totalVertices'] = lt.size(filtered) + 1 if lt.size(filtered) > 0 else 0	O(1)

<code>metaData["map"] = lt.size(foliumMap) > 0</code>	$O(1)$
<code>metaData["path"] = "req3"</code>	$O(1)$
<code>req8(foliumMap, "req3", drawPath)</code>	$O(M + O)$ (donde M y $O \leq V$)
<code>return filtered, metaData</code>	$O(1)$
Total	$O(E \log V)$

Pruebas Realizadas

Procesador	Apple M1
Memoria RAM	8 GB
Sistema Operativo	macOS Ventura Versión 13.5.2

Entrada	Tiempo (s)	Memoria (MB)
large	4287.925	404.757

Análisis

La complejidad de este algoritmo es $O(E \log V)$, que surge de la creación y uso de un árbol de expansión mínima de Prim, cuyo propósito es encontrar el costo mínimo para conectar una red de cámaras en la ciudad.

El objetivo del algoritmo es establecer una red de comunicaciones de fibra óptica para instalar cámaras de vídeo en M ubicaciones específicas dentro de una localidad determinada. El usuario proporciona el número M de cámaras a instalar y la localidad objetivo. El sistema, en primer lugar, genera la red de comunicaciones utilizando un árbol de recubrimiento mínimo. A continuación, lleva a cabo una búsqueda por anchura para recorrer los vértices, comenzando por el que tiene el mayor número de comparendos y avanzando hacia otros vértices. Finalmente, identifica y selecciona los vértices de mayor relevancia, es decir, aquellos con el mayor número de comparendos. La salida del algoritmo incluye: el total de vértices de la red, los vértices y arcos que conforman la red, la longitud total de fibra óptica que se instalará, y el costo monetario total de la instalación.

Requerimiento 4

Descripción

```

230 > def req3(dataStructs, locality, M):
269
270 def req4(dataStructs, M):
271     filtered = lt.newList('ARRAY_LIST')
272     metaData = {'totalDistance': 0}
273     foliumMap = lt.newList()
274     drawPath = []
275     if not dataStructs['mstGraph']:
276         mst = prim.PrimMST(dataStructs['distanceGraph'])
277         prim.edgesMST(dataStructs['distanceGraph'], mst)
278         mstGraph = createMstGraph(mst)
279         dataStructs['mstGraph'] = mstGraph
280     mstGraph = dataStructs['mstGraph']
281     infractions = dataStructs['infractions']
282     paths = bfs.BreadthFirstSearch(mstGraph, lt.firstElement(infractions)['VERTICES'])
283     visitedVertices = set()
284     arrivalVertexIndex = 2
285     while lt.size(filtered) < M and arrivalVertexIndex <= lt.size(infractions):
286         arrivalVertex = lt.getElement(infractions, arrivalVertexIndex)['VERTICES']
287         if arrivalVertex not in visitedVertices and bfs.hasPathTo(paths, arrivalVertex):
288             visitedVertices.add(arrivalVertex)
289             path = bfs.pathTo(paths, arrivalVertex)
290             while st.size(path) > 1:
291                 startingVertex = st.pop(path)
292                 arrivalVertex = st.top(path)
293                 distance = gr.getEdge(mstGraph, startingVertex, arrivalVertex)['weight']
294                 datum = {'Starting Point': startingVertex, 'Arrival Point': arrivalVertex, 'Distance': round(distance, 4)}
295                 if not existsStep(filtered, datum):
296                     lt.addLast(filtered, datum)
297                     metaData['totalDistance'] += distance
298                     startingVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], startingVertex))
299                     arrivalVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], arrivalVertex))
300                     lt.addLast(foliumMap, startingVertexDetails)
301                     lt.addLast(foliumMap, arrivalVertexDetails)
302                     drawPath.append((startingVertexDetails['coordinates'], arrivalVertexDetails['coordinates']))
303             arrivalVertexIndex += 1
304     metaData['totalVertices'] = lt.size(filtered) + 1 if lt.size(filtered) > 0 else 0
305     metaData['map'] = lt.size(foliumMap) > 0
306     metaData['path'] = "req4"
307     req8(foliumMap, "req4", drawPath)
308     return filtered, metaData
309
310 > def req5(dataStructs, vehicle, M):
349

```

Entrada	Las entradas esperadas en el requerimiento 4 son las siguientes: <ul style="list-style-type: none"> - Una estructura de datos con los datos a consultar.
Salida	Las salidas esperadas en el requerimiento 4 son las siguientes: <ul style="list-style-type: none"> - El tiempo que se demora el algoritmo en encontrar la solución (en milisegundos). - El total de vértices de la red. - Los vértices incluidos (identificadores). - Los arcos incluidos (Id vértice inicial e Id vértice final). - La cantidad de kilómetros de fibra óptica extendida. - El costo (monetario) final.
Implementado	Sí, el requerimiento fue implementado, desarrollado y completado por Alejandro García Rojas, código 202122516, del grupo No.1 de la sección No.3 del curso de Estructuras de Datos y Algoritmos.

Análisis de complejidad

Pasos	Complejidad
def req4(dataStructs, M):	O(1)
filtered = lt.newList('ARRAY_LIST')	O(1)
metaData = {'totalDistance': 0}	O(1)

foliumMap = lt.newList()	$O(1)$
drawPath = []	$O(1)$
if not dataStructs['mstGraph']:	$O(1)$
mst = prim.PrimMST(dataStructs['distanceGraph'])	$O(E \log V)$
prim.edgesMST(dataStructs['distanceGraph'], mst)	$O(V)$
mstGraph = createMstGraph(mst)	$O(V)$
dataStructs['mstGraph'] = mstGraph	$O(1)$
mstGraph = dataStructs['mstGraph']	$O(1)$
infractions = dataStructs['infractions']	$O(1)$
paths = bfs.BreadthFirstSearch(mstGraph, lt.firstElement(infractions)['VERTICES'])	$O(V + E)$
visitedVertices = set()	$O(1)$
arrivalVertexIndex = 2	$O(1)$
while lt.size(filtered) < M and arrivalVertexIndex <= lt.size(infractions):	$O(1)$
arrivalVertex = lt.getElement(infractions, arrivalVertexIndex)['VERTICES']	$O(1)$
if arrivalVertex not in visitedVertices and bfs.hasPathTo(paths, arrivalVertex):	$O(1)$
visitedVertices.add(arrivalVertex)	$O(1)$
path = bfs.pathTo(paths, arrivalVertex)	$O(M)$ (donde $M \leq V$)
while st.size(path) > 1:	$O(1)$
startingVertex = st.pop(path)	$O(1)$
arrivalVertex = st.top(path)	$O(1)$
distance = gr.getEdge(mstGraph, startingVertex, arrivalVertex)['weight']	$O(1)$
datum = {'Starting Point': startingVertex, 'Arrival Point': arrivalVertex, 'Distance': round(distance, 4)}	$O(1)$

if not existsStep(filtered, datum):	$O(M)$ (donde $M \leq V$)
lt.addLast(filtered, datum)	$O(1)$
metaData['totalDistance'] += distance	$O(1)$
startingVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], startingVertex))	$O(1)$
arrivalVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], arrivalVertex))	$O(1)$
lt.addLast(foliumMap, startingVertexDetails)	$O(1)$
lt.addLast(foliumMap, arrivalVertexDetails)	$O(1)$
drawPath.append((startingVertexDetails['coor dinates'], arrivalVertexDetails['coordinates']))	$O(1)$
arrivalVertexIndex += 1	$O(1)$
metaData['totalVertices'] = lt.size(filtered) + 1 if lt.size(filtered) > 0 else 0	$O(1)$
metaData["map"] = lt.size(foliumMap) > 0	$O(1)$
metaData["path"] = "req4"	$O(1)$
req8(foliumMap, "req4", drawPath)	$O(M + O)$ (donde M y $O \leq V$)
return filtered, metaData	$O(1)$
Total	$O(E \log V)$

Pruebas Realizadas

Procesador	Apple M1
Memoria RAM	8 GB
Sistema Operativo	macOS Ventura Versión 13.5.2

Entrada	Tiempo	Memoria
large	761078.93	407642

Análisis

La complejidad de este algoritmo es $O(E \log V)$, derivada de la creación y empleo del árbol de expansión mínima de Prim para determinar el costo mínimo de conexión en la localidad.

El propósito del algoritmo es establecer una red de comunicaciones de fibra óptica para instalar cámaras de video en M ubicaciones, dando prioridad a los sitios con comparendos de mayor gravedad. Se utiliza un Árbol de Recubrimiento Mínimo (MST) para conectar los vértices de manera eficiente y económica. El usuario especifica el número M de cámaras a instalar. Inicialmente, el algoritmo genera una red de comunicaciones de costo mínimo a partir del MST. Posteriormente, aplica un algoritmo de búsqueda por anchura para recorrer los vértices, enfocándose en la gravedad de los comparendos, considerando tanto el tipo de servicio como el código de infracción. Durante este proceso, se seleccionan los M vértices más relevantes. Como resultado, el algoritmo proporciona un desglose que incluye el total de vértices de la red, los vértices y arcos incluidos, la longitud en kilómetros de la fibra óptica a instalar, y el costo monetario de la instalación.

Requerimiento 5

Descripción

```

App > model.py > ...
278 > def req5(dataStructs, M):
309
310 def req5(dataStructs, vehicle, M):
311     filtered = lt.newList('ARRAY_LIST')
312     metaData = {'totalDistance': 0}
313     foliumMap = lt.newList()
314     drawPath = []
315     entry = mp.get(dataStructs['localities'], vehicle)
316     if entry:
317         if not dataStructs['mstGraph']:
318             mst = prim.PrimMST(dataStructs['distanceGraph'])
319             prim.edgesMST(dataStructs['distanceGraph'], mst)
320             mstGraph = createMstGraph(mst)
321             dataStructs['mstGraph'] = mstGraph
322             mstGraph = dataStructs['mstGraph']
323             vehicleVertices = sortData(mp.valueSet(me.getValue(entry)), compareByCount)
324             paths = bfs.BreadthFirstSearch(mstGraph, lt.firstElement(vehicleVertices)['vertex'])
325             arrivalVertexIndex = 2
326             while lt.size(filtered) < M and arrivalVertexIndex <= lt.size(vehicleVertices):
327                 arrivalVertex = lt.getElement(vehicleVertices, arrivalVertexIndex)['vertex']
328                 if bfs.hasPathTo(paths, arrivalVertex):
329                     path = bfs.pathTo(paths, arrivalVertex)
330                     while st.size(path) > 1:
331                         startingVertex = st.pop(path)
332                         arrivalVertex = st.top(path)
333                         distance = gr.getEdge(mstGraph, startingVertex, arrivalVertex)['weight']
334                         datum = {'Starting Point': startingVertex, 'Arrival Point': arrivalVertex, 'Distance': round(distance, 4)}
335                         if not existsStops(filtered, datum):
336                             lt.addLast(filtered, datum)
337                             metaData['totalDistance'] += distance
338                             startingVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], startingVertex))
339                             arrivalVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], arrivalVertex))
340                             lt.addLast(foliumMap, startingVertexDetails)
341                             lt.addLast(foliumMap, arrivalVertexDetails)
342                             drawPath.append([startingVertexDetails['coordinates'], arrivalVertexDetails['coordinates']])
343                 arrivalVertexIndex += 1
344             metaData['totalVertices'] = lt.size(filtered) + 1 if lt.size(filtered) > 0 else 0
345             metaData['map'] = lt.size(foliumMap) > 0
346             metaData['path'] = "req5"
347             req8(foliumMap, "req5", drawPath)
348             return filtered, metaData
349
350 > def req6(dataStructs, M):
386

```

Entrada

Las entradas esperadas en el requerimiento 5 son las siguientes:

	- Una estructura de datos con los datos a consultar.
Salida	<p>Las salidas esperadas en el requerimiento 5 son las siguientes:</p> <ul style="list-style-type: none"> - El tiempo que se demora el algoritmo en encontrar la solución (en milisegundos). - El total de vértices de la red. - Los vértices incluidos (identificadores). - Los arcos incluidos (Id vértice inicial e Id vértice final). - La cantidad de kilómetros de fibra óptica extendida. - El costo (monetario) final.
Implementado	Sí, el requerimiento fue implementado, desarrollado y completado por Pablo Andrés Sebastián Parra Céspedes, código 202310768, del grupo No.1 de la sección No.3 del curso de Estructuras de Datos y Algoritmos.

Análisis de complejidad

Pasos	Complejidad
def req5(dataStructs, vehicle, M):	$O(1)$
filtered = lt.newList('ARRAY_LIST')	$O(1)$
metaData = {'totalDistance': 0}	$O(1)$
foliumMap = lt.newList()	$O(1)$
drawPath = []	$O(1)$
entry = mp.get(dataStructs['localities'], vehicle)	$O(1)$
if entry:	$O(1)$
if not dataStructs['mstGraph']:	$O(1)$
mst = prim.PrimMST(dataStructs['distanceGraph'])	$O(E \log V)$
prim.edgesMST(dataStructs['distanceGraph'], mst)	$O(V)$
mstGraph = createMstGraph(mst)	$O(V)$
dataStructs['mstGraph'] = mstGraph	$O(1)$
mstGraph = dataStructs['mstGraph']	$O(1)$
vehicleVertices = sortData(mp.valueSet(me.getValue(entry)),	$O(M^{3/2})$ (donde $M \leq V$)

compareByCount)	
paths = bfs.BreadthFirstSearch(mstGraph, lt.firstElement(vehicleVertices)['vertex'])	$O(V + E)$
arrivalVertexIndex = 2	$O(1)$
while lt.size(filtered) < M and arrivalVertexIndex <= lt.size(vehicleVertices):	$O(1)$
arrivalVertex = lt.getElement(vehicleVertices, arrivalVertexIndex)['vertex']	$O(1)$
if bfs.hasPathTo(paths, arrivalVertex):	$O(1)$
path = bfs.pathTo(paths, arrivalVertex)	$O(M)$ (donde $M \leq V$)
while st.size(path) > 1:	$O(1)$
startingVertex = st.pop(path)	$O(1)$
arrivalVertex = st.top(path)	$O(1)$
distance = gr.getEdge(mstGraph, startingVertex, arrivalVertex)['weight']	$O(1)$
datum = {'Starting Point': startingVertex, 'Arrival Point': arrivalVertex, 'Distance': round(distance, 4)}	$O(1)$
if not existsStep(filtered, datum):	$O(1)$
lt.addLast(filtered, datum)	$O(1)$
metaData['totalDistance'] += distance	$O(1)$
startingVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], startingVertex))	$O(1)$
arrivalVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], arrivalVertex))	$O(1)$
lt.addLast(foliumMap, startingVertexDetails)	$O(1)$
lt.addLast(foliumMap, arrivalVertexDetails)	$O(1)$
drawPath.append((startingVertexDetails['coordinates'], arrivalVertexDetails['coordinates']))	$O(1)$
arrivalVertexIndex += 1	$O(1)$

metaData['totalVertices'] = lt.size(filtered) + 1 if lt.size(filtered) > 0 else 0	$O(1)$
metaData["map"] = lt.size(foliumMap) > 0	$O(1)$
metaData["path"] = "req5"	$O(1)$
req8(foliumMap, "req5", drawPath)	$O(M + O)$ (donde M y $O \leq V$)
return filtered, metaData	$O(1)$
Total	$O(E \log V)$

Pruebas Realizadas

Procesador	Apple M1
Memoria RAM	8 GB
Sistema Operativo	macOS Ventura Versión 13.5.2

Entrada	Tiempo (s)	Memoria (MB)
large	19.908	27.707

Análisis

Este algoritmo tiene una complejidad de $O(E \log V)$, resultante del uso del algoritmo de Prim para crear un árbol de expansión mínima. El propósito es minimizar el costo de conectar las zonas con más comparendos por vehículo mediante la instalación de cámaras de video.

El algoritmo busca establecer una red eficiente de comunicaciones instalando cámaras en M puntos clave, seleccionados por tener el mayor número de comparendos para un tipo específico de vehículo, y asegurando que la instalación de la fibra óptica sea lo más económica posible. El usuario especifica el número M de cámaras y el tipo de vehículo. Inicialmente, se determina el costo mínimo para conectar la ciudad usando un árbol de recubrimiento mínimo. A continuación, se realiza una búsqueda por anchura para identificar los vértices relacionados con el tipo de vehículo seleccionado, y de estos se escogen los M con más comparendos. Finalmente, se genera una lista con los vértices y arcos de la red propuesta. El algoritmo proporciona como resultado el total de vértices de la red, los vértices y arcos involucrados, la distancia total cubierta por la fibra óptica y el costo monetario total de la instalación.

Requerimiento 6

Descripción

```
def req6(dataStructs, M):
    filtered = lt.newList('ARRAY_LIST')
    metaData = {}
    foliumMap = lt.newList()
    drawPath = []
    infractions = dataStructs['infractions']
    mostSeverityIndex = 1
    while lt.size(filtered) < M and mostSeverityIndex <= lt.size(infractions):
        pathDetails = lt.newList('ARRAY_LIST')
        mostSeverity = lt.getElement(infractions, mostSeverityIndex)
        metaData = {'totalDistance': 0, 'totalVertices': 0, 'infraction': lt.newList()}
        lt.addLast(metaData['infraction'], mostSeverity)
        paths = djik.Dijkstra(dataStructs['distanceGraph'], mostSeverity['VERTICES'])
        closestStation = getClosestStation(dataStructs, paths)
        if closestStation:
            path = djik.pathTo(paths, closestStation['VERTICES'])
            lt.addLast(pathDetails, {'Starting Point': 'Severitiest Infraction', 'Arrival Point': st.top(path)['vertexA'] if st.size(path) > 0 else closestStation['VERTICES'], 'Distance': 0})
            metaData['totalVertices'] = lt.size(path) + 1 if lt.size(path) > 0 else 0
            while not st.isEmpty(path):
                step = st.pop(path)
                datum = {'Starting Point': step['vertexA'], 'Arrival Point': step['vertexB'], 'Distance': round(step['weight'], 3)}
                lt.addLast(pathDetails, datum)
                startingVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], step['vertexA']))
                arrivalVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], step['vertexB']))
                lt.addLast(foliumMap, startingVertexDetails)
                lt.addLast(foliumMap, arrivalVertexDetails)
                drawPath.append((startingVertexDetails['coordinates'], arrivalVertexDetails['coordinates']))
            lt.addLast(pathDetails, {'Starting Point': closestStation['VERTICES'], 'Arrival Point': 'Closest Police Station', 'Distance': 0})
            metaData['totalDistance'] = djik.distTo(paths, closestStation['VERTICES'])
            pathDetails['metaData'] = metaData
            lt.addLast(filtered, pathDetails)
            mostSeverityIndex += 1
        metaData['map'] = lt.size(foliumMap) > 0
        metaData['path'] = "req6"
        req8(foliumMap, "req6", drawPath)
    return filtered, metaData
```

Entrada	Las entradas esperadas en el requerimiento 6 son las siguientes: <ul style="list-style-type: none"> - Una estructura de datos con los datos a consultar.
Salida	Las salidas esperadas en el requerimiento 6 son las siguientes: <ul style="list-style-type: none"> - El tiempo que se demora el algoritmo en encontrar la solución (en milisegundos). - El total de vértices del camino. - Los vértices incluidos (identificadores). - Los arcos incluidos (Id vértice inicial e Id vértice final). - La cantidad de kilómetros del camino.
Implementado	Sí, el requerimiento fue implementado, desarrollado y completado por el grupo No.1 de la sección No.3 del curso de Estructuras de Datos y Algoritmos.

Análisis de complejidad

Pasos	Complejidad
def req6(dataStructs, M):	O(1)
filtered = lt.newList('ARRAY_LIST')	O(1)
metaData = {}	O(1)
foliumMap = lt.newList()	O(1)
drawPath = []	O(1)

infractions = dataStructs['infractions']	$O(1)$
mostSeverityIndex = 1	$O(1)$
while lt.size(filtered) < M and mostSeverityIndex <= lt.size(infractions):	$O(1)$
pathDetails = lt.newList('ARRAY_LIST')	$O(1)$
mostSeverity = lt.getElement(infractions, mostSeverityIndex)	$O(1)$
metaData = {'totalDistance': 0, 'totalVertices': 0, 'infracton': lt.newList()}	$O(1)$
lt.addLast(metaData['infracton'], mostSeverity)	$O(1)$
paths = djk.Dijkstra(dataStructs['distanceGraph'], mostSeverity['VERTICES'])	$O(V \log V + E \log V)$
closestStation = getClosestStation(dataStructs, paths)	$O(V)$
if closestStation:	$O(1)$
path = djk.pathTo(paths, closestStation['VERTICES'])	$O(1)$
lt.addLast(pathDetails, {'Starting Point': 'Severitiest Infracton', 'Arrival Point': st.top(path)['vertexA'] if st.size(path) > 0 else closestStation['VERTICES'], 'Distance': 0})	$O(1)$
metaData['totalVertices'] = lt.size(path) + 1 if lt.size(path) > 0 else 0	$O(1)$
while not st.isEmpty(path):	$O(1)$
step = st.pop(path)	$O(1)$
datum = {'Starting Point': step['vertexA'], 'Arrival Point': step['vertexB'], 'Distance': round(step['weight'], 3)}	$O(1)$
lt.addLast(pathDetails, datum)	$O(1)$
startingVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], step['vertexA']))	$O(1)$

arrivalVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], step['vertexB']))	$O(1)$
lt.addLast(foliumMap, startingVertexDetails)	$O(1)$
lt.addLast(foliumMap, arrivalVertexDetails)	$O(1)$
drawPath.append((startingVertexDetails['coordinates'], arrivalVertexDetails['coordinates']))	$O(1)$
lt.addLast(pathDetails, {'Starting Point': closestStation['VERTICES'], 'Arrival Point': 'Closest Police Station', 'Distance': 0})	$O(1)$
metaData['totalDistance'] = dj.k.distTo(paths, closestStation['VERTICES'])	$O(1)$
pathDetails['metaData'] = metaData	$O(1)$
lt.addLast(filtered, pathDetails)	$O(1)$
mostSeverityIndex += 1	$O(1)$
metaData["map"] = lt.size(foliumMap) > 0	$O(1)$
metaData["path"] = "req6"	$O(1)$
req8(foliumMap, "req6", drawPath)	$O(M + O)$ (donde $M + O \leq V$)
return filtered, metaData	$O(1)$
Total	$O((V+E)\log V)$

Pruebas Realizadas

Procesador	Apple M1
Memoria RAM	8 GB
Sistema Operativo	macOS Ventura

Entrada	Tiempo (s)	Memoria (MB)
large	1296.644	8.423

Análisis

La complejidad de este algoritmo es $O((V+E)\log V)$, debido a la aplicación del algoritmo de Dijkstra para determinar la ruta más corta desde un punto específico hasta la estación de policía más cercana.

El objetivo del algoritmo es optimizar los tiempos de respuesta de la policía ante los comparendos más graves. Funciona permitiendo que cada comparendo sea atendido por la estación de policía más próxima. El usuario introduce un número M de comparendos a responder y especifica la estación de policía más cercana al comparendo de mayor gravedad. Primero, el algoritmo clasifica los comparendos en función de su gravedad, considerando el tipo de servicio y el código de la infracción. Luego, utiliza el algoritmo de Dijkstra para encontrar el camino más corto desde la estación de policía más cercana hasta el comparendo más grave. Posteriormente, el algoritmo recorre los comparendos graves para identificar la estación de policía más cercana a cada uno. Como resultado, el programa ofrece un desglose detallado que incluye el total de vértices del camino, los vértices y arcos involucrados, y la distancia total en kilómetros del camino.

Requerimiento 7

Descripción

```

318 > def req5(dataStructs, vehicle, M):-
319
358 > def req6(dataStructs, M):-
359
387 def req7(dataStructs, startPoint, arrivalPoint):
388     filtered = lt.newList('ARRAY_LIST')
389     metaData = {'totalDistance': 0, 'totalVertices': 0, 'totalInfractions': 0}
390     foliumMap = lt.newList()
391     drawPath = []
392     if isInside(dataStructs, startPoint) and isInside(dataStructs, arrivalPoint):
393         startPoint = getClosestVertex(dataStructs, startPoint)
394         arrivalPoint = getClosestVertex(dataStructs, arrivalPoint)
395         paths = dj.k.Dijkstra(dataStructs['infracionGraph'], startPoint['id'])
396         path = dj.k.pathTo(paths, arrivalPoint['id'])
397         if path:
398             metaData['totalVertices'] = st.size(path) + 1 if st.size(path) > 0 else 0
399             metaData['totalDistance'] += startPoint['distance']
400             datum = {'Starting Point': 'Starting point', 'Arrival Point': st.top(path)['vertexA'], 'Num Infractions': 0, 'Distance': startPoint['distance']}
401             lt.addLast(filtered, datum)
402             while not st.isEmpty(path):
403                 step = st.pop(path)
404                 distance = gr.getEdge(dataStructs['distanceGraph'], step['vertexA'], step['vertexB'])['weight']
405                 datum = {'Starting Point': step['vertexA'], 'Arrival Point': step['vertexB'], 'Num Infractions': step['weight'], 'Distance': round(distance, 4)}
406                 lt.addLast(filtered, datum)
407                 metaData['totalDistance'] += distance
408                 startingVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], step['vertexA']))
409                 arrivalVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], step['vertexB']))
410                 lt.addLast(foliumMap, startingVertexDetails)
411                 lt.addLast(foliumMap, arrivalVertexDetails)
412                 drawPath.append((startingVertexDetails['coordinates'], arrivalVertexDetails['coordinates']))
413                 datum = {'Starting Point': arrivalPoint['id'], 'Arrival Point': 'Arrival point', 'Num Infractions': 0, 'Distance': arrivalPoint['distance']}
414                 metaData['totalDistance'] += arrivalPoint['distance']
415                 metaData['totalInfractions'] = dj.k.distTo(paths, arrivalPoint['id'])
416                 lt.addLast(filtered, datum)
417             metaData["map"] = lt.size(foliumMap) > 0
418             metaData["path"] = "req7"
419             req8(foliumMap, "req7", drawPath)
420             return filtered, metaData
421
422 > def req8(events, path, drawPath):-
423
432 > def getClosestVertex(dataStructs, coordinates):-
433

```

Entrada

Las entradas esperadas en el requerimiento 7 son las siguientes:

- Una estructura de datos con los datos a consultar.
- Un punto de origen (una localización geográfica con latitud y longitud).

	<ul style="list-style-type: none"> - Un punto de destino (una localización geográfica con latitud y longitud).
Salida	<p>Las salidas esperadas en el requerimiento 7 son las siguientes:</p> <ul style="list-style-type: none"> - El tiempo que se demora el algoritmo en encontrar la solución (en milisegundos). - El total de vértices del camino. - Los vértices incluidos (identificadores). - Los arcos incluidos (Id vértice inicial e Id vértice final). - La cantidad de comparendos del camino. - La cantidad de kilómetros del camino.
Implementado	Sí, el requerimiento fue implementado, desarrollado y completado por el grupo No.1 de la sección No.3 del curso de Estructuras de Datos y Algoritmos.

Análisis de complejidad

Pasos	Complejidad
def req7(dataStructs, startPoint, arrivalPoint):	$O(1)$
filtered = lt.newList('ARRAY_LIST')	$O(1)$
metaData = {'totalDistance': 0, 'totalVertices': 0, 'totalInfractions': 0}	$O(1)$
foliumMap = lt.newList()	$O(1)$
drawPath = []	$O(1)$
if isInside(dataStructs, startPoint) and isInside(dataStructs, arrivalPoint):	$O(1)$
startPoint = getClosestVertex(dataStructs, startPoint)	$O(V)$
arrivalPoint = getClosestVertex(dataStructs, arrivalPoint)	$O(V)$
paths = djk.Dijkstra(dataStructs['infracionGraph'], startPoint['id'])	$O(V \log V + E \log V)$
path = dj.k.pathTo(paths, arrivalPoint['id'])	$O(M)$ (donde $M \leq V$)
if path:	$O(1)$
metaData['totalVertices'] = st.size(path) + 1 if st.size(path) > 0 else 0	$O(1)$

metaData['totalDistance'] += startPoint['distance']	O(1)
datum = {'Starting Point': 'Starting point', 'Arrival Point': st.top(path)['vertexA'], 'Num Infractions': 0, 'Distance': startPoint['distance']}	O(1)
lt.addLast(filtered, datum)	O(1)
while not st.isEmpty(path):	O(1)
step = st.pop(path)	O(1)
distance = gr.getEdge(dataStructs['distanceGraph'], step['vertexA'], step['vertexB'])['weight']	O(1)
datum = {'Starting Point': step['vertexA'], 'Arrival Point': step['vertexB'], 'Num Infractions': step['weight'], 'Distance': round(distance, 4)}	O(1)
lt.addLast(filtered, datum)	O(1)
metaData['totalDistance'] += distance	O(1)
startingVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], step['vertexA']))	O(1)
arrivalVertexDetails = me.getValue(mp.get(dataStructs['vertexMap'], step['vertexB']))	O(1)
lt.addLast(foliumMap, startingVertexDetails)	O(1)
lt.addLast(foliumMap, arrivalVertexDetails)	O(1)
drawPath.append((startingVertexDetails['coor dinates'], arrivalVertexDetails['coordinates']))	O(1)
datum = {'Starting Point': arrivalPoint['id'], 'Arrival Point': 'Arrival point', 'Num Infractions': 0, 'Distance': arrivalPoint['distance']}	O(1)
metaData['totalDistance'] += arrivalPoint['distance']	O(1)
metaData['totalInfractions'] = djikstra.distTo(paths, arrivalPoint['id'])	O(1)

lt.addLast(filtered, datum)	$O(1)$
metaData["map"] = lt.size(foliumMap) > 0	$O(1)$
metaData["path"] = "req7"	$O(1)$
req8(foliumMap, "req7", drawPath)	$O(M + O)$ (donde M y $O \leq V$)
return filtered, metaData	$O(1)$
Total	$O((V+E)\log V)$

Pruebas Realizadas

Procesador	Apple M1
Memoria RAM	8 GB
Sistema Operativo	macOS Ventura Versión 13.5.2

Entrada	Tiempo	Memoria
large	5346614.1	18908.36

Análisis

La complejidad del algoritmo es $O((V+E)\log V)$, resultado de implementar el algoritmo de Dijkstra, que se utiliza para determinar la ruta más corta basada en la menor cantidad de comparendos entre dos puntos específicos en Bogotá.

El propósito del algoritmo es identificar la ruta entre dos ubicaciones geográficas en Bogotá con la mínima cantidad de comparendos. Los usuarios introducen los puntos de origen y destino en términos de coordenadas de latitud y longitud. Estos puntos se validan para asegurar que se encuentren dentro de los límites de la ciudad y luego se aproximan a los vértices más cercanos de la red vial. La salida del algoritmo proporciona información detallada que incluye: el total de vértices en la ruta, los identificadores de los vértices y arcos implicados, la cantidad total de comparendos en la ruta, y la distancia total recorrida.

Requerimiento 8

Descripción

```
def req8(events, path, drawPath):
    eventsMap = folium.Map((4.7110, -74.0721), zoom_start= 11)
    mark = MarkerCluster()
    for event in lt.iterator(events):
        mark.add_child(folium.Marker(event['coordinates'], popup= createPopUp(event)))
    eventsMap.add_child(mark)
    for draw in drawPath:
        folium.PolyLine(draw, color="red", weight=2.5, opacity=1).add_to(eventsMap)
    eventsMap.save(f'{path}.html')
```

Entrada	La función acepta una lista de diccionarios, cada uno detallando las coordenadas de los vértices iniciales y finales. Adicionalmente, requiere un parámetro para denominar el elemento resultante. Finalmente, necesita una lista especificando las conexiones esenciales para la construcción de las rutas que interconectan los vértices dentro de la estructura del grafo.
Salida	La función retorna
Implementado	Sí, el requerimiento fue implementado, desarrollado y completado por el grupo No.1 de la sección No.3 del curso de Estructuras de Datos y Algoritmos.

Análisis de complejidad

Pasos	Complejidad
def req8(events, path, drawPath):	O(1)
eventsMap = folium.Map((4.7110, -74.0721), zoom_start= 11)	O(1)
mark = MarkerCluster()	O(1)
for event in lt.iterator(events):	O(M) (donde $M \leq V$)
mark.add_child(folium.Marker(event['coordina tes'], popup= createPopUp(event)))	O(1)
eventsMap.add_child(mark	O(1)
for draw in drawPath:	O(O) (donde $O \leq V$)
folium.PolyLine(draw, color="red", weight=2.5, opacity=1).add_to(eventsMap)	O(1)
eventsMap.save(f'{path}.html')	O(1)
Total	O(M + O)

Pruebas Realizadas

Procesador	Apple M1
Memoria RAM	8 GB
Sistema Operativo	macOS Ventura Versión 13.5.2

Análisis

La complejidad del algoritmo es $O(M+N)$, derivada de la presencia de dos bucles en el código. Esta complejidad se refleja en el objetivo de visualizar los resultados de los requerimientos anteriores (del 1 al 7) a través de recursos multimedia e interfaces gráficas, mejorando así la comprensión y el análisis de los datos. Se alcanza este objetivo mediante el uso de mapas interactivos que no solo destacan rutas y puntos de interés, sino que también presentan otros datos geográficos de relevancia. Este enfoque facilita una interpretación más intuitiva y detallada de la información geográfica procesada.

Inciso de aclaraciones

1. En nuestros experimentos, observamos que la implementación del algoritmo de Búsqueda en Profundidad (DFS) consumió más memoria que la búsqueda en anchura (BFS), lo cual se explica por la naturaleza de la implementación recursiva de DFS. Al aumentar el límite de recursividad para DFS, permitimos que el algoritmo explore caminos más profundos en el grafo, incrementando así la pila de llamadas recursivas. Esto resulta en un mayor uso de memoria, especialmente en grafos con caminos largos y profundos. En contraste, BFS, que se basa en una estructura de cola y no en llamadas recursivas, no experimenta un incremento significativo en el uso de memoria en función de la profundidad del grafo, lo que explica por qué, en este caso específico, DFS utiliza más memoria que BFS.
2. Para abordar el reto, se emplearon dos grafos distintos, cada uno con sus propias características. Los vértices de ambos grafos representan intersecciones entre dos calles de una red vial. La principal diferencia entre ellos reside en los pesos asignados a los arcos. En el primer grafo, el peso de cada arco corresponde a la distancia Haversine en kilómetros entre dos vértices. Por otro lado, en el segundo grafo, el peso se define como la suma total de infracciones de tránsito (comparendos) registradas entre dos vértices. Adicionalmente, se destaca el uso de un subgrafo en forma de un árbol de recubrimiento mínimo. Este árbol, una vez generado, se almacena para utilidades futuras; este enfoque resulta beneficioso, ya que permite ahorrar tiempo y memoria en procesos subsiguientes. Este método eficiente de almacenamiento y reutilización del árbol de recubrimiento mínimo contribuye significativamente a la optimización del análisis y la gestión de los datos en los grafos.
3. Para la implementación del algoritmo en nuestro proyecto, utilizamos estructuras adicionales a los grafos, específicamente mapas y listas, debido a las limitaciones de la librería y como parte de una demostración práctica de los conceptos aprendidos en

clase. Los mapas fueron esenciales para referenciar a los grafos, ya que la librería con la que trabajamos no permitía una manipulación directa de estos. Esta elección no solo se alineó con las restricciones técnicas, sino que también sirvió para demostrar nuestra comprensión de las estructuras de datos vistas en el curso. Además, empleamos listas para organizar y almacenar los datos de manera ordenada e iterable. Esta estructura facilitó la impresión y el manejo comprensible de los datos, permitiéndonos una manipulación eficiente y clara de la información durante la ejecución del algoritmo.