Universidad de los Andes
Facultad de Ingeniería

Departamento de Ingeniería de Sistemas y Computación
Estructuras de Datos y Algoritmos
ISIS-1225

ABET Engineering Accreditation Commission

ANÁLISIS DEL RETO

Gabriel Estiven Borrero Silva, 202212129, g.borreros@uniandes.edu.co

Sebastian Quevedo, 202223735, s.quevedo1@uniandes.edu.co

Juan Pablo Delgado, 202212285, jp.delgadom1@uniandes.edu.co

Requerimiento 1

Descripción

El algoritmo tiene como objetivo encontrar un posible camino entre dos puntos de geográficos localizados en los límites de la ciudad de Bogotá

Entrada	• Punto de origen (una localización geográfica con latitud y longitud). • Punto de destino (una localización geográfica con latitud y longitud)
Salidas	• 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/No)	Sí, Gabriel Borrero.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
if (float(min_lon)<=float(lon_initial)<=float(max_lon)) and (float(min_lat)<=float(lat_initial)<=float(max_lat)) and (float(min_lon)<=float(lon_destiny)<=float(max_lon)) and (float(min_lat)<=float(lat_destiny)<=float(max_lat)):	O(1)
llaves=mp.keySet(data_structs["vertexInfo"])	O(V)
for llave in lt.iterator(llaves):	O(V)
pareja_llave_valor=mp.get(data_structs["vertexInfo"], llave)	O(1)
lat_= me.getValue(pareja_llave_valor)["lat"]	O(1)
lon_= me.getValue(pareja_llave_valor)["lon"]	O(1)

distancia_initial = math.sqrt((float(lat_) - float(lat_initial))**2 + (float(lon_) - float((lon_initial)))**2)	O(V)
if distancia_initial < distancia_minima: distancia_minima = distancia_initial vertice_initial = llave	O(V)
data_structs['search'] = bfs.BreathFirstSearch(data_structs['connections'], vertice_initial)	O(E+V)
existe_camino = bfs.hasPathTo(data_structs['search'], vertice_destiny)	O(E+V)
path = bfs.pathTo(data_structs['search'], vertice_destiny)	O(E+V)
for i in lt.iterator(path):	O(V) —> V>M (M representa la cantidad de vértices que componen el camino encontrado por BFS)
if existe_camino==True or existe_camino=="True":	O(1)
distancia=gr.getEdge(data_structs['connections'], vertice_initial, i)	O(M)
distancia_arco=float(distancia["weight"])	O(M)
lt.addLast(camino, i)	O(1)
TOTAL	***Como E>V (El número de arcos es mayor que el número de vértices) → Grafo no dirigido O(E+V) —> Como sobrevive la mayor complejidad, entonces: O(E)***

Pruebas Realizadas y Tablas de datos.

Debido a que no existe un archivo small, o menor al archivo que se está utilizando, sólo se realizó una prueba para tiempo y espacio utilizado.

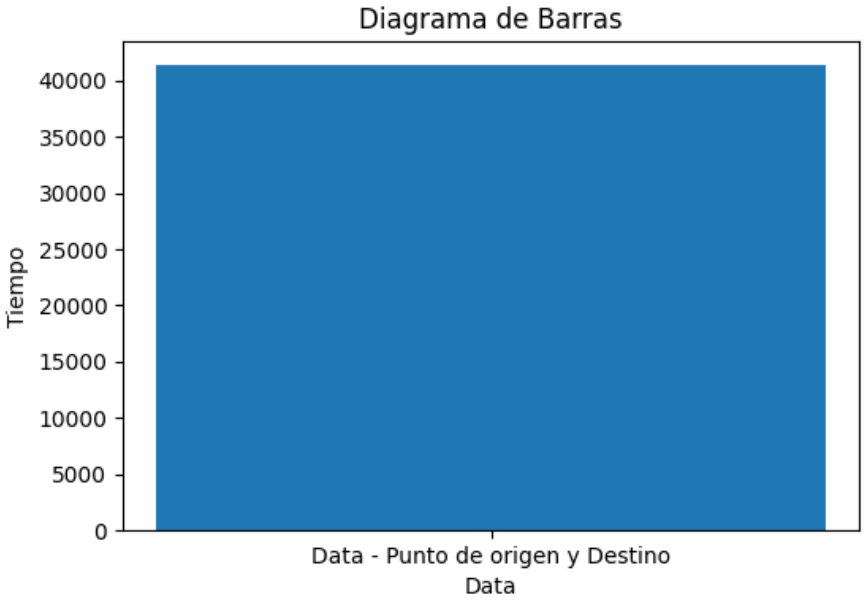
Procesadores	**AMD Ryzen 5 4500U 6 x 2.3 - 4 GHz, Renoir-U (Zen 2)**
Memoria RAM	8 GB
Sistema Operativo	Windows 10

Entrada	Tiempo (ms)	Espacio (kB)
Origen: • Latitud: 4.60293518548777, • Longitud: -74.06511801444837 Destino: • Latitud: 4.693518613347496 • Longitud: -74.13489678235523	41447.38	166305.3

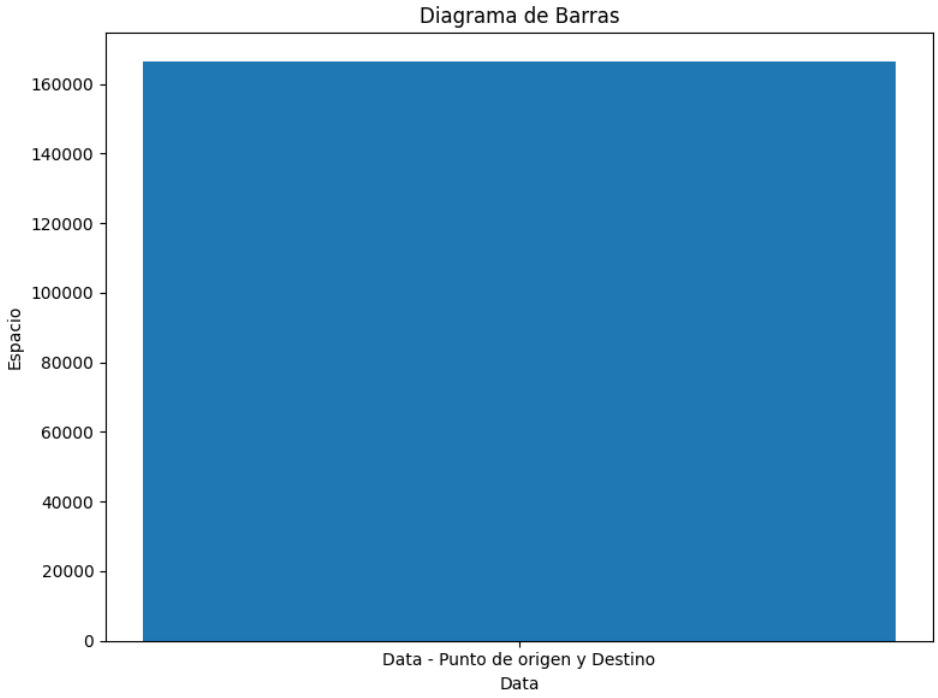
Graficas

Debido a que no existe un archivo small, o menor al archivo que se está utilizando, sólo se realizó una prueba para tiempo y espacio utilizado. Por ello, es imposible generar gráficas para estos 2 casos, pues sólo hay un punto (tanto para tiempo, como para espacio).

Tiempo:



Espacio:



Análisis

Para este requerimiento, se ha utilizado el algoritmo BFS para en contrar el camino más corto en cuanto a saltos entre un punto A (origen) y un punto B (Destino) , apesar de que sólo es necesario encontrar un posible camino. Se tiene que, la mayor complejidad del requerimiento es implementar el algoritmo BFS, el cual tiene una complejidad proporcional a O(E+V) para encontrar el camino más corto en cuanto a saltos de arcos entre dos vértices. Sin embargo, como se trata de un grafo completo no dirigido, entonces necesariamente el número de arcos es mayor al número de vértices, y como en notación O sólo sobrevive la complejidad mayor, podemos afirmar que el requerimiento 1 tiene una complejidad de O(E), donde E>V.

Para el algoritmo, primero encontramos los vértices más cercanos a los puntos de origen y destino proporcionados por el usuario como parámetros. Luego, aplicamos BFS a los vértices encontrados, de esta manera finalizando de manera exitosa el req 1.

Requerimiento 2

Plantilla para el documentar y analizar cada uno de los requerimientos.

```python
def req_2(data_structs, initial_point, final_point):
    #initial_point = "4.60293518548777, -74.06511801444837"
    #final_point =  "4.693518613347496, -74.13489678235523"

    """Preparacion para encontrar vertices en el grafo"""
    initial_point=initial_point.split(",")
    lat_initial=initial_point[0]
    lon_initial=initial_point[1].strip()
    final_point=final_point.split(",")
    lat_destiny=final_point[0]
    lon_destiny=final_point[1].strip()

    #Hallamos el vértice de origen más cercano
    llaves=mp.keySet(data_structs["vertexInfo"])

    vertice_i = None
    vertice_f = None
    distancia_minima = float('inf')
    distancia_minima_2 = float('inf')

    for llave in lt.iterator(llaves):

        pareja_llave_valor=mp.get(data_structs["vertexInfo"], llave)
        lat_= me.getValue(pareja_llave_valor)["lat"]
        lon_= me.getValue(pareja_llave_valor)["lon"]

        distancia_initial = math.sqrt((float(lat_) - float(lat_initial))**2 + (float(lon_) - float((lon_initial)))**2)
        distancia_destiny = math.sqrt((float(lat_) - float(lat_destiny))**2 + (float(lon_) - float((lon_destiny)))**2)

        # Actualizar el vértice más cercano si la distancia actual es menor que la mínima registrada
        if distancia_initial < distancia_minima:
            distancia_minima = distancia_initial
            vertice_i = llave

        if distancia_destiny < distancia_minima_2:
            distancia_minima_2 = distancia_destiny
            vertice_f = llave
```

```
    data_structs['search'] = bfs.BreathFirstSearch(data_structs['connections'], vertice_i)
    camino = bfs.pathTo(data_structs['search'], vertice_f)
    lista = lt.newList('SINGLE_LINKED')
    retorno = recur(camino['first'], lista)

    anterior = None
    distancia = 0
    for vertice in lt.iterator(retorno):
        if anterior == None:
            pass
        else:
            h = gr.getEdge(data_structs['connections'], anterior, vertice[0])
            distancia += float(h['weight'])
        anterior = vertice[0]


    return retorno, distancia


def recur(camino, lista):
    if camino['next'] == None:
        return lista
    else:
        lt.addLast(lista, [camino['info']])
        recur(camino['next'], lista)
    return lista
```

Descripción

Breve descripción de cómo abordaron la implementación del requerimiento

Entrada	Latitud y longitud del inicio al final de la ruta
Salidas	Distancia total, cuántos vértices fueron recorridos, camino recorrido
Implementado (Sí/No)	Si, por Sebastian Quevedo

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Recorrido de los vértices para saber cuales son los más cercanos a la ubicación pasada por parámetro	O(V)
bfs.BreathFirstSearch	O(V + E)
bfs.pathTo	O(V + E)
recur()	O(V)
Recorrido de los vertices del camino para calcular la distancia	O(V)
TOTAL	***O(V + E)***

Análisis

Análisis de resultados de la implementación, teniendo en cuenta las pruebas realizadas y el análisis de complejidad.

El requerimiento dura relativamente poco tiempo (por lo menos con los parámetros del ejemplo). Halla de buena manera un camino corto entre los vértices más cercanos a las ubicaciones pasadas. El crecimiento de la demora temporal es de manera lineal y es proporcional a ***O(V + E)***.

Requerimiento 3

Plantilla para documentar y analizar cada uno de los requerimientos.

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	Parámetros necesarios para resolver el requerimiento.
Salidas	Respuesta esperada del algoritmo.
Implementado (Sí/No)	No, Sebastian Quevedo

Requerimiento 4

Este requerimiento se encarga de instalar una red de comunicaciones por fibra óptica de cámaras de video en M sitios teniendo en cuenta el menor costo de instalación posible.

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	• La cantidad de cámaras de video que se desean instalar (M).
Salidas	• El tiempo que se demora algoritmo en encontrar la solución (en milisegundos). • La siguiente información de la red de comunicaciones propuesta: - 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. o El costo (monetario) total.
Implementado (Sí/No)	Sí, Gabriel Borrero.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
llaves=mp.keySet(data_structs["comparendos"])	O(V)
pareja_llave_valor=mp.get(data_structs["comparendos"], llave)	O(1)
vertice=me.getValue(pareja_llave_valor)	O(1)
if vertice["Tipo de servicio"] == "Diplomatico": vertice["Tipo de servicio"]=4	O(V)
mp.put(data_structs["Req_4_comparendos"], llave, vertice)	O(1)
llaves=mp.keySet(data_structs["Req_4_comparendos"])	O(V)
for llave in lt.iterator(llaves):	O(V)
if i<=M: lt.addLast(lista_comparendos, llave) i+=1	O(M) —> M es el # de comparendos más graves. Por supuesto, V>M
llaves=mp.keySet(data_structs["vertexInfo"])	O(V)
for comparendos in lt.iterator(lista_comparendos):	O(M)
inicial=lt.firstElement(vertices_mayor_gravedad)	O(1)
lt.removeFirst(vertices_mayor_gravedad)	O(1)
for i in lt.iterator(vertices_mayor_gravedad):	O(M)

data_structs['search']=djk.Dijkstra(data_structs['connections'], vertice_initial)	O(M*E*log(V))
existe_camino =djk.hasPathTo(data_structs['search'], vertice_destiny)	O((M-1)*E*log(V))
path =djk.pathTo(data_structs['search'], vertice_destiny)	O((M-1)*E*log(V))
TOTAL	**O((M-1)*E*log(V)) —> El algoritmo djk tiene una complejidad de O(E*log(V)). Sin embargo, este algoritmo se implementa para cada par de vértices (en el peor de los casos). Es decir, el algoritmo djk se implementa (M-1) veces.**

Pruebas Realizadas y Tabla de datos

Debido a que no existe un archivo small, o menor al archivo que se está utilizando, sólo se realizó una prueba para tiempo y espacio utilizado.

Procesadores	**AMD Ryzen 5 4500U 6 x 2.3 - 4 GHz, Renoir-U (Zen 2)**
Memoria RAM	8 GB
Sistema Operativo	Windows 10

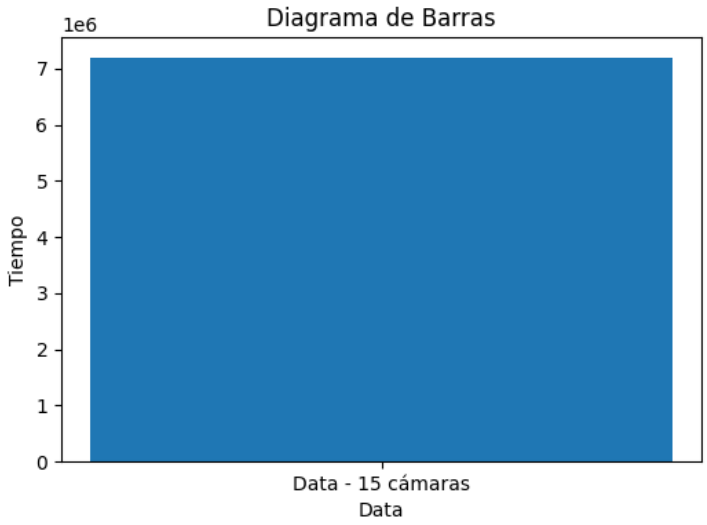
Tabla de datos:

Entrada	Tiempo (ms)	Espacio (kB)
15 cámaras	7.200.000 (2 horas/120 min)	1566305.3

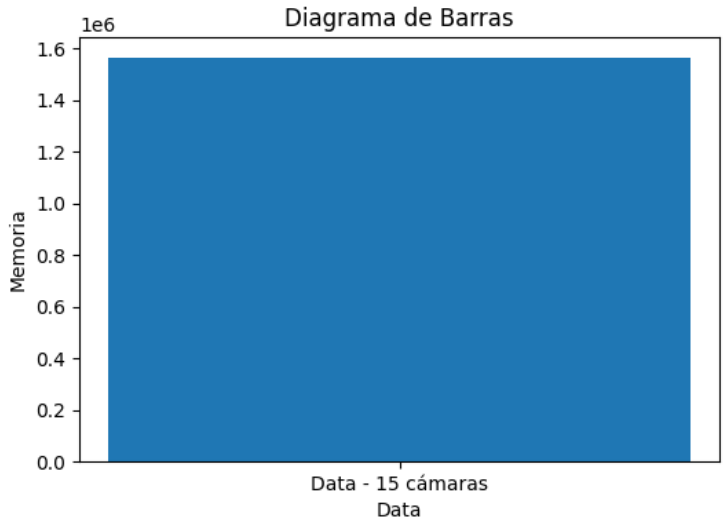
Graficas

Debido a que no existe un archivo small, o menor al archivo que se está utilizando, sólo se realizó una prueba para tiempo y espacio utilizado. Por ello, es imposible generar gráficas para estos 2 casos, pues sólo hay un punto (tanto para tiempo, como para espacio).

Tiempo:



Espacio:



Análisis

Para este requerimiento, se ha utilizado el algoritmo *Dijkstra* para en contrar el camino más corto en cuanto a costos (menor número de comparendos) entre un punto A (Vértice de mayor gravedad) y un punto B (Vertice de menor gravedad dentro de los mayores), pasando por varios vértices. Se tiene que, la mayor complejidad del requerimiento es implementar el algoritmo *Dijkstra*, el cual tiene una complejidad proporcional a O(E*log(V)) para encontrar el camino más corto en cuanto a costos de arcos entre dos vértices. Es necesario aclarar que E representa los arcos y V los vértices, donde, además, el número de arcos (E) es mayor al número de vértices (V). Sin embargo, en este caso, es necesario implementar este algoritmo (M-1) veces en el peor caso, ya que lo que se quiere es encontrar el camino mínimo en costos entre el primer vertice con el segundo, el segundo con el tercero, el tercero con el cuarto… (n-1 y n), lo cual da una cantidad de repeticiones de M-1, donde M es el número de vértices de mayor gravedad.

Para el algoritmo, primero encontramos los vértices más cercanos a los vértices de mayor gravedad según el número de camaras proporcionadas por el usuario como parámetros. Luego, aplicamos *Dijkstra* a los vértices encontrados, de esta manera finalizando de manera exitosa el req 4.

Requerimiento 5

Este requerimiento se encarga de instalar una red de comunicaciones por fibra óptica de cámaras de video en M sitios teniendo en cuenta el menor costo de instalación posible.

```
def req_5(data_structs, M, V):
    """
    Función que soluciona el requerimiento 5
    """
    # TODO: Realizar el requerimiento 5
    llaves_C = mp.keySet(data_structs['comparendos'])
    req_vertex = data_structs['vertex_reqs']
    m = int(M)
    map_req = mp.newMap(numelements=50000,
                        maptype='CHAINING', loadfactor=4,
                        cmpfunction=None)
    map_vertex_req = mp.newMap(numelements=50000,
                        maptype='CHAINING', loadfactor=4,
                        cmpfunction=None)
    gr_req_5 = gr.newGraph(datastructure='ADJ_LIST',
                                directed=False,
                                size=m,
                                cmpfunction=None)
    for llave in lt.iterator(llaves_C):
        entry = mp.get(data_structs['comparendos'],llave)
        value = me.getValue(entry)
        if value['Clase vehiculo'] == V:
            Values = {"ID": value['ID'], "lat": value['Lat'],
                  "lon": value["Lon"], "Clase vehiculo": value['Clase vehiculo']}
            mp.put(map_req, llave,Values)
    for vertex in lt.iterator(req_vertex):
        if vertex["comparendos"]!= "None":
            comparendos = vertex["comparendos"].split(';')
            if len(comparendos) > 1:
                for comparendo in comparendos:
                    if mp.contains(map_req, comparendo):
                        addBogotaVertexReq5(map_vertex_req, vertex, comparendo, gr_req_5)
            else:
                if mp.contains(map_req, comparendos[0]):
                        addBogotaVertexReq5(map_vertex_req, vertex, comparendos[0], gr_req_5)
    vertex_req = mp.valueSet(map_vertex_req)
    merg.sort(vertex_req, compare_req_5)
    origin = lt.getElement(vertex_req, 1)
    id_origin = origin["id"]
    gr.insertVertex(gr_req_5, id_origin)
    distance = 0
    vertices_incluidos = 0
    while int(gr.numVertices(gr_req_5)) < m:
        listaV = gr.vertices(gr_req_5)
        for vertex in lt.iterator(vertex_req):
            if (lt.isPresent(listaV, vertex["id"]) == False) or (lt.isPresent(listaV, vertex["id"]) == "False"):
                id_vertex = vertex["id"]
                if (id_vertex != id_origin):
                    exist = bfs.BreathFirstSearch(data_structs['connections'], id_origin)
                    existe_camino = bfs.hasPathTo(exist, id_vertex)
                    if (existe_camino == True) or (existe_camino == "True"):
                        camino = bfs.pathTo(exist, id_vertex)
                        vertices_incluidos += lt.size(camino)
                        gr.insertVertex(gr_req_5, id_vertex)
                        weight = haversine(float(origin['lon']), float(origin['lat']), float(vertex['lon']), float(vertex['lat']))
                        distance += weight
                        gr.addEdge(gr_req_5, id_origin, id_vertex, weight)
                        id_origin = id_vertex
                        break
    costo = distance * 1000000
    vertices_identicadores = gr.numVertices(gr_req_5)
    arcos = gr.edges(gr_req_5)
    total_vertices = vertices_incluidos
    v = gr.vertices(gr_req_5)

    return distance, costo, total_vertices,vertices_identicadores, arcos, v
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	• La cantidad de cámaras de video que se desean instalar (M) • El tipo de vehiculo
Salidas	• Numero total de vertices • Numero de arcos • Cantidad de kilometros • Costo total
Implementado (Sí/No)	Si, por Juan Pablo Delgado.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
gr_req_5 = gr.newGraph(datastructure='ADJ_LIST', directed=False, size=m, cmpfunction=None)	**O(1)**
for llave in lt.iterator(llaves_C):	O(C)
for vertex in lt.iterator(req_vertex):	O(O)
merg.sort(vertex_req, compare_req_5)	O(OlogO)
while int(gr.numVertices(gr_req_5)) < m:	O(M)
for vertex in lt.iterator(vertex_req):	O(M * O)
existe_camino = bfs.hasPathTo(exist, id_vertex)	O(O * M * (V + E))
TOTAL	O(O * M * (V + E))

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Procesadores	**11th gen intel(r) core(tm) i5-11300h**
Memoria RAM	8 GB
Sistema Operativo	Windows 11

Entrada	Tiempo (s)	Espacio (kB)

M:5 ; Vehiculo: Automovil	458492.57	345607.3

Análisis

Análisis de resultados de la implementación, teniendo en cuenta las pruebas realizadas y el análisis de complejidad.

En este requerimiento usa el algoritmo bfs, Sin embargo recorre los vértices por prioridad de número de comparendos y que cumpla la condición de vehículo, al hacer sort por número de comparendos se obtiene el vértice origen y busca un vértice que cumpla la condición de vehículo si lo encuentra se añade el camino a un sub grafo con pesos de distancias, de esa manera se optimiza el tiempo de ejecución.

Requerimiento 6

Plantilla para el documentar y analizar cada uno de los requerimientos.

```python
def req_6(data_structs):
    M = 2
    """
    Función que soluciona el requerimiento 6
    """
    # Organizacion de los comparendos de mas grave a menos grave
    lista = mp.valueSet(data_structs['comparendos'])
    #lista = quk.sort(lista, sort_crit_req_6)
    lista = lt.subList(lista, 1, M)

    # Validacion de estacion mas cercana por comparendo
    estaciones = mp.valueSet(data_structs['stations'])
    llaves = mp.keySet(data_structs["vertexInfo"])
    vertices_estaciones = lt.newList('SINGLE_LINKED')

    for estacion in lt.iterator(estaciones):
        lat = float(estacion['Lat'])
        lon = float(estacion['Lon'])
        vertice_i = None
        distancia_minima = float('inf')

        for llave in lt.iterator(llaves):

            pareja_llave_valor=mp.get(data_structs["vertexInfo"], llave)
            lat_= float(me.getValue(pareja_llave_valor)["lat"])
            lon_= float(me.getValue(pareja_llave_valor)["lon"])

            distancia_initial = abs(abs(abs(lat_) - abs(lat)) - abs(abs(lon_) - abs(lon)))
            # Actualizar el vértice más cercano si la distancia actual es menor que la mínima registrada
            if distancia_initial < distancia_minima:
                distancia_minima = distancia_initial
                vertice_i = llave
        lt.addLast(vertices_estaciones, {estacion['Name']: vertice_i})
    #vertices_estaciones es mi lista de estaciones con su respectivo codigo de vertice

    #hallar la calle del comparendo para saber en que vertice se encuentra
    vertices_comparendos = lt.newList('SINGLE_LINKED')
    for comparendo in lt.iterator(lista):
```

```
vertices_comparendos = lt.newList('SINGLE_LINKED')
for comparendo in lt.iterator(lista):
    distancia = 100000000
    lat = float(comparendo['Lat'])
    lon = float(comparendo['Lon'])
    for llave in lt.iterator(llaves):
        pareja_llave_valor=mp.get(data_structs["vertexInfo"], llave)
        lat_= float(me.getValue(pareja_llave_valor)["lat"])
        lon_= float(me.getValue(pareja_llave_valor)["lon"])

        distancia_initial = abs(abs(abs(lat_) - abs(lat)) - abs(abs(lon_) - abs(lon)))

        if distancia_initial < distancia:
            distancia = distancia_initial
            vertice = llave
    lt.addLast(vertices_comparendos, vertice)
#vertices_comparendos me dice en que vertice ocurre cada comparendo

#algoritmo correcto tomando en cuenta 'distancias' de los arcos con dijkstra
"""
retorno_prime = lt.newList('SINGLE_LINKED')
for comparendo in lt.iterator(vertices_comparendos):
    closest = 10000000
    recorrido = djk.Dijkstra(data_structs['connections'], str(comparendo))
    for estacion in lt.iterator(vertices_estaciones):
        for nombre in estacion:
            codigo = estacion[nombre]
            validacion = djk.hasPathTo(recorrido, codigo)
            camino = djk.pathTo(recorrido, codigo)
            if validacion == True:
                retorno = lt.newList('SINGLE_LINKED')
                retorno = recur(camino['first'], retorno)
            if lt.size(retorno) < closest:
                cercana = retorno
                closest = lt.size(retorno)
                retorno['nombre'] = nombre
    lt.addLast(retorno_prime, cercana)
"""
```

```
    #algoritmo provisional usando bfs para una menor demora
    retorno_prime = lt.newList('SINGLE_LINKED')
    for comparendo in lt.iterator(vertices_comparendos):
        closest = 10000000
        recorrido = bfs.BreathFirstSearch(data_structs['connections'], str(comparendo))
        for estacion in lt.iterator(vertices_estaciones):
            for nombre in estacion:
                codigo = estacion[nombre]
                validacion = bfs.hasPathTo(recorrido, codigo)
                camino = bfs.pathTo(recorrido, codigo)
                if validacion == True:
                    retorno = lt.newList('SINGLE_LINKED')
                    retorno = recur(camino['first'], retorno)
                if lt.size(retorno) < closest:
                    cercana = retorno
                    closest = lt.size(retorno)
                    retorno['nombre'] = nombre
        lt.addLast(retorno_prime, cercana)

    return retorno_prime



def sort_crit_req_6(data1, data2):
    if data1['Tipo de servicio'] == 'Publico' and data2['Tipo de servicio'] != 'Publico':
        return 1
    elif data2['Tipo de servicio'] == 'Publico' and data1['Tipo de servicio'] != 'Publico':
        return 0
    elif data1['Tipo de servicio'] != 'Publico' and data2['Tipo de servicio'] != 'Publico':
        if data1['Tipo de servicio'] == 'Oficial' and data2['Tipo de servicio'] != 'Oficial':
            return 1
        elif data2['Tipo de servicio'] == 'Oficial' and data1['Tipo de servicio'] != 'Oficial':
            return 0
        elif data1['Tipo de servicio'] != 'Particular' and data2['Tipo de servicio'] != 'Particular':
            if data1['Infraccion'] > data2['Infraccion']:
                return 1
```

Descripción

Breve descripción de cómo abordaron la implementación del requerimiento

Entrada	Cantidad M de comparendos importantes
Salidas	La estación más cercana, el camino que recorre, la distancia/vértices que recorre
Implementado (Sí/No)	Si, por Sebastian Quevedo.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Organización de lista con comparendos para saber cuales son los más graves (no va incluido en el código)	O(n logn)

Recorrido para analizar los vertices mas cercanos a los comparendos	O(V*M)
Recorrido para analizar los vertices mas cercanos a las estaciones.	O(V*estaciones)
Hallar recorrido dijkstra (bfs en la prueba para mayor eficacia temporal)	O(V + E)
Para cada comparendo, se halla la distancia bfs de las estaciones para saber cual es la mas cercana	O(M * estaciones * (V + E))
TOTAL	***O(M * estaciones * (V + E)***

Análisis

Análisis de resultados de la implementación, tener en cuenta las pruebas realizadas y el análisis de complejidad.

En este requerimiento se reemplaza el algoritmo de Dijkstra por uno de bfs, esto para que sea más rápido encontrar los caminos más cercanos de comparendos a las estaciones de policía. Del mismo modo, no se puede organizar de manera óptima la importancia de los comparendos sin antes tener que pasar por un largo proceso de un quick sort, el cual también se omite en la ejecución práctica del requerimiento.

Requerimiento 7

Descripción

Este algoritmo se encarga de encontrar el camino “más corto” en términos número de menor cantidad de comparendos entre dos puntos de geográficos localizados en los límites de la ciudad de Bogotá.

Entrada	• Punto de origen (una localización geográfica con latitud y longitud). • Punto de destino (una localización geográfica con latitud y longitud)
Salidas	La siguiente información del camino seleccionado: • 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/No)	Sí, Gabriel Borrero.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
if (float(min_lon)<=float(lon_initial)<=float(max_lon)) and (float(min_lat)<=float(lat_initial)<=float(max_lat)) and (float(min_lon)<=float(lon_destiny)<=float(max_lon)) and (float(min_lat)<=float(lat_destiny)<=float(max_lat)):	O(1)
llaves=mp.keySet(data_structs["vertexInfo"])	O(V)
for llave in lt.iterator(llaves):	O(V)
pareja_llave_valor=mp.get(data_structs["vertexInfo"], llave)	O(1)
lat_ = me.getValue(pareja_llave_valor)["lat"]	O(1)
lon_ = me.getValue(pareja_llave_valor)["lon"]	O(V)
distancia_initial = math.sqrt((float(lat_) - float(lat_initial))**2 + (float(lon_) - float((lon_initial)))**2)	O(V)
if distancia_initial < distancia_minima: distancia_minima = distancia_initial vertice_initial = llave	O(V)
data_structs['search']=djk.Dijkstra(data_structs['connections'], vertice_initial)	O(E*log(V))
existe_camino =djk.hasPathTo(data_structs['search'], vertice_destiny)	O(E*log(V))
path =djk.pathTo(data_structs['search'], vertice_destiny)	O(E*log(V))

for i in lt.iterator(path):	O(M) —> V>M (M representa la cantidad de vértices que componen el camino encontrado por BFS)
if existe_camino==True or existe_camino=="True":	O(1)
distancia=gr.getEdge(data_structs['connections'], vertice_initial, i)	O(M)
distancia_arco=float(distancia["weight"])	O(M)
lt.addLast(camino, i)	O(1)
TOTAL	**O(E*log(V))**

Pruebas Realizadas y Tablas de datos.

Debido a que no existe un archivo small, o menor al archivo que se está utilizando, sólo se realizó una prueba para tiempo (ms) y espacio utilizado (kB).

Procesadores	**AMD Ryzen 5 4500U 6 x 2.3 - 4 GHz, Renoir-U (Zen 2)**
Memoria RAM	8 GB
Sistema Operativo	Windows 10

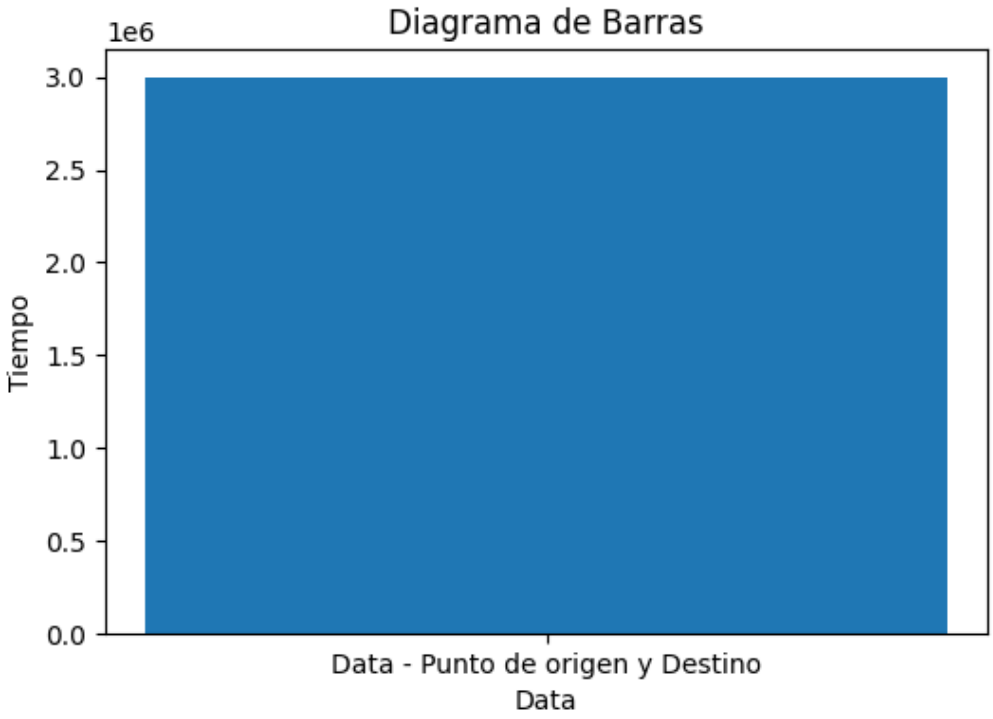
Tabla de datos:

Entrada	Tiempo (ms)	Memoria (kB)
Origen: • Latitud: 4.60293518548777, • Longitud: -74.06511801444837 Destino: • Latitud: 4.693518613347496 • Longitud: -74.13489678235523	3,000,000 (50 minutos)	966305.3

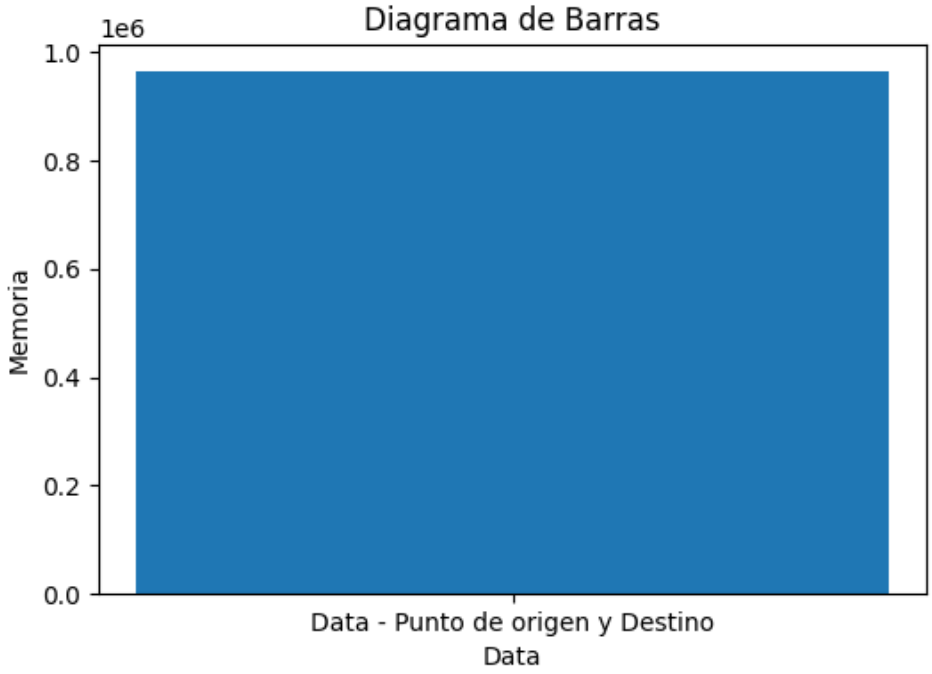
Graficas

Debido a que no existe un archivo small, o menor al archivo que se está utilizando, sólo se realizó una prueba para tiempo y espacio utilizado. Por ello, es imposible generar gráficas para estos 2 casos, pues sólo hay un punto (tanto para tiempo, como para espacio).

Tiempo:



Distancia:



Análisis

Para este requerimiento, se ha utilizado el algoritmo *Dijkstra* para en contrar el camino más corto en cuanto a costos (menor número de comparendos) entre un punto A (origen) y un punto B (Destino). Se tiene que, la mayor complejidad del requerimiento es implementar el algoritmo *Dijkstra*, el cual tiene una complejidad proporcional a O(E*log(V)) para encontrar el camino más corto en cuanto a costos de arcos entre dos vértices. Es necesario aclarar que E representa los arcos y V los vértices, donde, además, el número de arcos (E) es mayor al número de vértices (V).

Para el algoritmo, primero encontramos los vértices más cercanos a los puntos de origen y destino proporcionados por el usuario como parámetros. Luego, aplicamos *Dijkstra* a los vértices encontrados, de esta manera finalizando de manera exitosa el req 7.