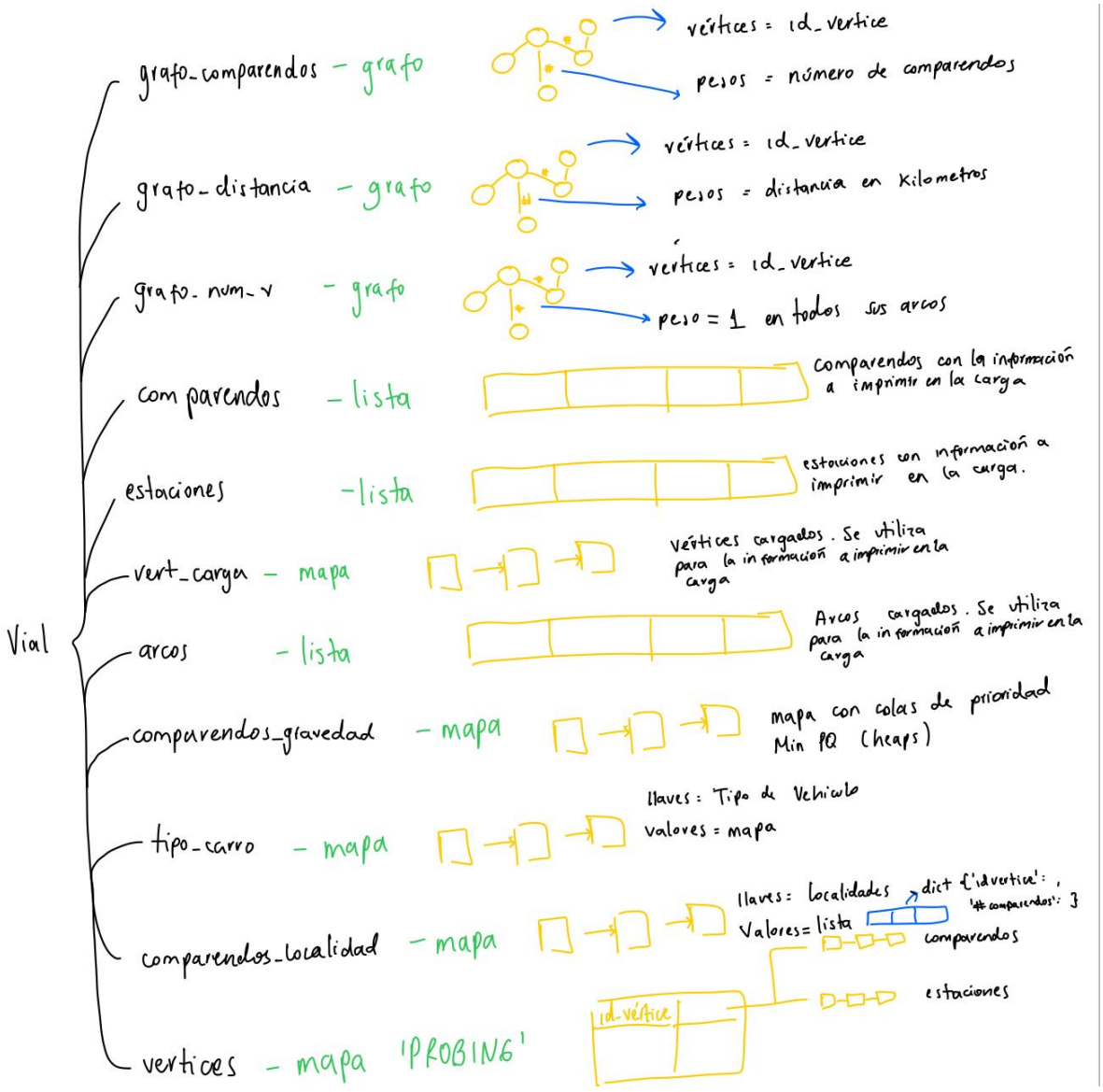
ANÁLISIS DEL RETO 4

María Teresa Duran, 202312780, m.duranr@uniandes.edu.co

Lizeth Johanna Gómez Calderón, 202216352, lj.gomezc1@uniandes.edu.co

Amelia Serrano Arango, 202221556, a.serranoa2@uniandes.edu.co

DIAGRAMA DE CARAG DE DATOS



Requerimiento 1

```python
def req_1(vial, lat_i, long_i, lat_f, long_f):
    """
    Función que soluciona el requerimiento 1
    """
    # TODO: Realizar el requerimiento 1
    lat_long_i = (float(lat_i), float(long_i))
    lat_long_f = (float(lat_f), float(long_f))

    esta_i = esta_en_bogota(lat_long_i)
    esta_f = esta_en_bogota(lat_long_f)


    if esta_i and esta_f:

        vertice_i = vertice_cercano(vial, lat_long_i)
        vertice_f = vertice_cercano(vial, lat_long_f)

        search = dfs.DepthFirstSearch(vial["grafo_distancia"], vertice_i)
        camino = dfs.pathTo(search, vertice_f)

        id_vertices = lt.newList()

        total_km = 0

        vertexa = st.pop(camino)
        lt.addLast(id_vertices, vertexa)
        vertexb = None
        while (not st.isEmpty(camino)):
            vertexb = st.pop(camino)
            arco = gr.getEdge(vial["grafo_distancia"],vertexa, vertexb)
            total_km += arco["weight"]
            lt.addLast(id_vertices, vertexb)
            vertexa = vertexb

    else:
        id_vertices = None
        total_km = 0

    return id_vertices, total_km
```

Descripción

Este requerimiento encuentra un posible camino entre dos puntos de geográficos localizados en los límites de la ciudad de Bogotá.

Entrada	1.Punto de origen (una localización geográfica con latitud y longitud). 2. Punto de destino (una localización geográfica con latitud y longitud).
Salidas	La respuesta contiene la siguiente información: 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)	Si.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: Se llama la funcion esta_en_bogota	O(1)
Paso 2 : verificar si la latitud inicial y final si se encuentran en bogotá	O(1)
Paso 3 : Se llama la funcion Vertice_cercano	O(1)
Paso 4: Se realiza un DFS iniciando en el vertice inicial encontrado en el paso 3	O(V + E) V: número de vértices E: número de arcos
Paso 5: Se encuentra el camino resultado del DFS aplicado con pathTo	O(E) E: número de arcos
Paso 6: se hace pop del camino (pila)	O(1)
Paso 7 : se añade el vertice que se obtuvo del paso 6 y se agrega a la lista id_vertices	O(1)
Paso 8 : iniciar un ciclo que permite ir sacando con pop el primer del camino y lo va eliminando de forma que va buscando el arco entre el primer vértice y el vertice en el que se encuentre el ciclo y saca el peso (los km) de cada arco.	O(n) n: número de vértices adyacentes al V
TOTAL	***O(V+E)***

Pruebas Realizadas

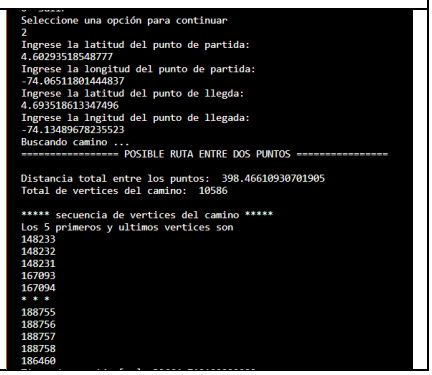
Datos : Origen: Latitud: 4.60293518548777, Longitud: -74.06511801444837 • Destino: Latitud: 4.693518613347496, Longitud: -74.13489678235523

Procesadores	**Procesador11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz, 2803 Mhz, 4 procesadores principales, 8 procesadores lógicos**
Memoria RAM	8 GB
Sistema Operativo	Windows 11

Entradas	**Tiempo (s)**
Origen: Latitud: 4.60293518548777, Longitud: -74.06511801444837 • Destino: Latitud: 4.693518613347496, Longitud: -74.13489678235523	32.7

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Entrada	Tiempo (s)	Salida
Origen: Latitud: 4.60293518548777, Longitud: -74.06511801444837 • Destino: Latitud: 4.693518613347496, Longitud: -74.13489678235523	32.7	

Análisis

El tiempo de procesamiento para el Requerimiento 1 muestra una notable eficiencia al emplear un grafo en comparación con otros resultados.

Requerimiento 2

```python
def req_2(vial, lat_i, long_i, lat_f, long_f):
    """
    Función que soluciona el requerimiento 2
    """
    lat_long_i = (float(lat_i), float(long_i))
    lat_long_f = (float(lat_f), float(long_f))

    esta_i = esta_en_bogota(lat_long_i)
    esta_f = esta_en_bogota(lat_long_f)


    if esta_i and esta_f:

        vertice_i = vertice_cercano(vial, lat_long_i)
        vertice_f = vertice_cercano(vial, lat_long_f)

        search = djk.Dijkstra(vial["grafo_num_v"], vertice_i)
        camino = djk.pathTo(search, vertice_f)

        id_vertices = lt.newList()

        total_km = 0

        vertexa = st.pop(camino)
        lt.addLast(id_vertices, vertexa)
        vertexb = None
        while (not st.isEmpty(camino)):
            vertexb = st.pop(camino)
            arco = gr.getEdge(vial["grafo_distancia"],vertexa, vertexb)
            total_km += arco["weight"]
            lt.addLast(id_vertices, vertexb)
            vertexa = vertexb

    else:

        id_vertices = None
        total_km = 0

    return id_vertices, total_km
```

Descripción

El requerimiento 2 encuentra un posible camino “más corto”, según el número de intersecciones a cruzar, 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 respuesta contiene la siguiente información: 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 (Sí/No)	Sí.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Paso 1: Se llama la funcion esta_en_bogota	O(1)
Paso 2 : verificar si la latitud inicial y final si se encuentran en bogotá	O(1)
Paso 3 : Se llama la funcion Vertice_cercano	O(1)
Paso 4: Se realiza un recorrido con Djistra para encontrar el camino minimo iniciando en el vertice inicial encontrado en el paso 3	$O((V + E) * \log(V))$
Paso 5: Se encuentra el camino resultado del Djistra aplicado con pathTo	O(E) E: número de arcos
Paso 6: se hace pop del camino	O(1)
Paso 7 : se añade el vertice que se obtuvo del paso 6 y se agrega a la lista id_vertices	O(1)
Paso 8 : iniciar un ciclo que permite ir sacando con pop el primer del camino y lo va eliminando de forma que va buscando el arco entre el primer vértice y el vertice en el que se encuentre el ciclo y saca el peso (los km) de cada arco.	O(n) n: número de vértices adyacentes al V
TOTAL	***O(V+E)***

Pruebas Realizadas

Datos : Origen: Latitud: 4.60293518548777, Longitud: -74.06511801444837 • Destino: Latitud: 4.693518613347496, Longitud: -74.1348967823552

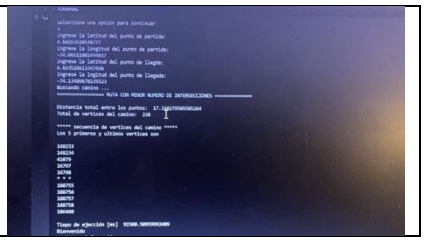
Procesadores	**Procesador11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz, 2803 Mhz, 4 procesadores principales, 8 procesadores lógicos**
Memoria RAM	8 GB
Sistema Operativo	Windows 11

Entrada	Tiempo (s)
Origen: Latitud: 4.60293518548777, Longitud: -74.06511801444837 • Destino: Latitud: 4.693518613347496, Longitud: -74.1348967823552	92.36

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Entrada	Tiempo (s)	Salida

Origen: Latitud: 4.60293518548777, Longitud: -74.06511801444837 • Destino: Latitud: 4.693518613347496, Longitud: -74.1348967823552	92.36	

Análisis

En el algoritmo del requerimiento 2, el resultado del tiempo que tomó el procesamiento de las tareas fue bastante rápido, teniendo el código una complejidad temporal O(N+E).

Requerimiento 3

```
def req_3(vial, localidad , m):
    """
    Función que soluciona el requerimiento 3
    """
    # TODO: Realizar el requerimiento 3


    comp_localidad= vial["comparendos_localidad"]
    vertices= vial["vertices"]
    grafo_prin= gr.newGraph("ADJ_LIST")
    grafo_distancia = vial["grafo_distancias"]

    entry= mp.get(comp_localidad, localidad)
    print(entry)
    value_dic= me.getValue(entry)

    quk.sort(value_dic, compare_req5)

    pre_lista= lt.subList(value_dic, 0, lt.size(value_dic))
    while lt.size(pre_lista) > m:
        lt.removeLast(pre_lista)

    arcos_inclus= lt.newList('ARRAY_LIST')
    print(pre_lista)
    copia=lt.newList('ARRAY_LIST')
    copia= pre_lista
    km_fibra=0
    id_vertex = lt.newList('ARRAY_LIST')
    for vertx in lt.iterator(pre_lista):

        vertexa= vertx["comparendo"]
        lt.addLast(id_vertex, vertexa)
        for f in lt.iterator(copia):
            vertexb= f["comparendo"]

            if vertexa != vertexb :
                print("entró")
                arco= gr.getEdge(grafo_distancia,vertexa, vertexb)
                km_fibra += arco["weight"]
                print(arco)
                if arco == False :
                    arcos_inclus= lt.addLast(arcos_inclus, arco)
```

```
    n_vertex= m

    costo= km_fibra *  1000000
    result= lt.newList('ARRAY_LIST')
    result = lt.addLast(result, crear_resultado(n_vertex , id_vertex, arcos_inclus , km_fibra , costo))

    return result
```

Descripción

El requerimiento 3 encuentra la red más eficiente en términos de distancia para la instalación de M cámaras de seguridad en una localidad específica.

Entrada	La cantidad de cámaras de video que se desean instalar (M). La localidad donde se desean instalar
Salidas	La respuesta contiene la siguiente información: El tiempo que se demora algoritmo en encontrar la solución (en milisegundos). • La siguiente información de la red de comunicaciones propuesta: o El total de vértices de la red. o Los vértices incluidos (identificadores). o Los arcos incluidos (Id vértice inicial e Id vértice final). o La cantidad de kilómetros de fibra óptica extendida. o El costo (monetario) total.
Implementado (Sí/No)	Sí, Lizeth Gómez.

Requerimiento 4

```python
def req_4(vial, n):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4

    n_vertices = mp.newMap()

    diplomaticos = me.getValue(mp.get(vial["comparendos_gravedad"], "Diplomatico"))


    while lt.size(n_vertices) < n and not mpq.isEmpty(diplomaticos):
        if not mp.get(n_vertices, mpq.min(diplomaticos)["id"]):
            mp.put(n_vertices, mpq.delMin(diplomaticos)["id"], 1)


    if lt.size(n_vertices) < n:
        oficial = me.getValue(mp.get(vial["comparendos_gravedad"], "Oficial"))

        while lt.size(n_vertices) < n and not  mpq.isEmpty(oficial):
            if not mp.get(n_vertices, mpq.min(oficial)["id"]):
                mp.put(n_vertices, mpq.delMin(oficial)["id"], 1)

    if lt.size(n_vertices) < n:
        publico = me.getValue(mp.get(vial["comparendos_gravedad"], "Público"))

        while lt.size(n_vertices) < n and not mpq.isEmpty(publico):
            if not mp.get(n_vertices, mpq.min(publico)["id"]):
                mp.put(n_vertices, mpq.delMin(publico)["id"])

    if lt.size(n_vertices) < n:
        particular = me.getValue(mp.get(vial["comparendos_gravedad"], "Particular"))

        while lt.size(n_vertices) < n and not mpq.isEmpty(publico):
            if not mp.get(n_vertices, mpq.min(particular)["id"]):
                mp.put(n_vertices, mpq.delMin(particular)["id"])

    n_vertices = mp.keySet(n_vertices)

    total_vertices, lista_vertices, arcos, total_km, costo = mst_mini_grafo(vial, n_vertices)

    return total_vertices, lista_vertices, arcos, total_km, costo
```

Descripción

Este requerimiento busca instalar una red de comunicaciones que tenga cámaras en los puntos con los comparendos de mayor gravedad, además que cuente con el menor costo posible. Esta red se instalará por medio de una fibra óptica que tiene un costo de 1000000 COP/km.

Entrada	Cantidad de cámaras que se desean instalar en la red de comunicación
Salidas	Total de vértices de la red, vértices incluidos en la instalación, arcos incluidos en la instalación, cantidad de kilómetros totales, de la instalación y el costo total de la instalación
Implementado (Sí/No)	Si María Teresa Duran

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**

Crear un nuevo mapa	O(1)
Se repiten los pasos siguientes varias veces de ser necesario, no más de 4 veces:	
Se saca de mapa de la estructura el minpq de los comparendos según su gravedad (Diplomático, oficial, publico, particular)	O(1)
Se hace un ciclo de While Mientras el minpq no esté vacío y la lista de n_vertices tenga menos de n elementos se hacen los siguientes pasos	Comparaciones de complejidad O(1)
Se verifica que el id del elemento está en el mapa de n_vertices.	O(1)
Se hace delMin del elemento	O(logn) (por el swimm)
Se agrega el elemento a n_vertices	O(1)
Total del while	O(M) en el peor caso siendo M el número de cámaras dado por el usuario. Esto es una constante no muy grande por lo tanto la complejidad se aproxima a O(1)
Se verifica si n_vertices tiene aún menos de n elementos. De ser así se repiten los pasos	O(1)
Total de repetición de pasos ….	Maximo se harán 4 veces. Da un total de complejidad de O(4) ~ O(1)
Se saca el keySet de n_vertices	O(M) con M número de camaras. Esto se aproxima, como se explicó anteriormente, a O(1)
Se llama a la función auxiliar mst_mini_graf() cuya complejidad se analiza más abajo	O(V^2) donde v que son los vértices corresponde como máximo a los M puntos donde se instalarán las cámaras... esto se aproxima, por ser una constante pequeña, a O(1)
TOTAL	***O(1)***

Pruebas Realizadas

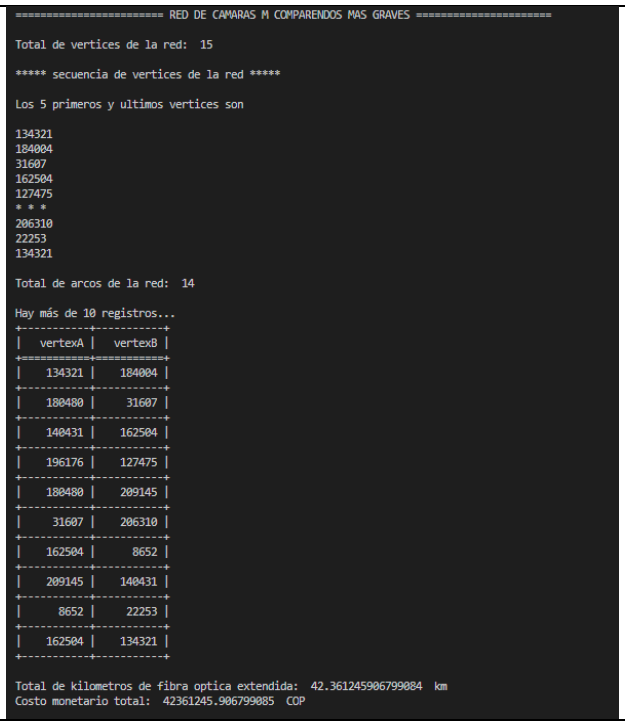
Entradas: 15 camaras.

Entrada	**Tiempo (s)**
15 cámaras	25.105199992656708

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Entrada	**Salida**	**Tiempo (s)**

15 cámaras			25.105199992656708

Análisis

La complejidad de este requerimiento es O(1) esto se debe en primer lugar a la forma en la que se pensó la carga de datos con mapas y colas de prioridad mínimas que son estructuras altamente eficientes. Además, el número de datos con el que se trabaja constantemente es el numero M de cámaras que quiere el usuario lo cual es una constante pequeña.

Por otro lado, este requerimiento es O(1) porque asumimos que la red de comunicaciones puede ser de forma aerea o subterranea y no necesariamente tiene que seguir las calles de Bogotá. Esto lo hicimos así ya que en la instrucción no se mencionaba.

Si se considera esta forma para la red de comunicaciones, entonces su costo de construcción será mucho más barato. Esta es una forma de reducir el problema.

Requerimiento 5

```python
def req_5(vial, clase_carro, m):
    """
    Función que soluciona el requerimiento 5
    """
    # TODO: Realizar el requerimiento1 5
    #Sacar los comparendos por tipo de carro
    #Sacar los M primeros vertices que tengan mas comparendos
    #Recorrer comparendos para encontrar el menos costoso
    comp_tipo_carro= vial["tipo_carro"]
    vertices= vial["vertices"]
    #grafo_prin= gr.newGraph("ADJ_#LIST")
    lista_vertices= lt.newList("ARRAY_LIST")
    carros= mp.get(comp_tipo_carro, clase_carro)
    value_map= me.getValue(carros)
    dics= mp.keySet(value_map)
    if lt.size(dics) < int(m):
        pre_lista= dics
    else:
        pre_lista= lt.subList(dics, 1, int(m))
    for ids in lt.iterator(pre_lista):
        """tupla= mp.get(vertices, ids["comparendos"])
        vertice= me.getKey(tupla)"""
        lt.addLast(lista_vertices, ids)

    final= mst_mini_grafo(vial, lista_vertices)
    total_vertices= final[0]
    lis_vertices= final[1]
    arcos= final[2]
    total_km= final[3]
    costo= final[4]

    return total_vertices, lis_vertices, arcos, total_km, costo
```

Descripción

En este requerimiento se pide encontrar la forma de instalar una red de comunicación de cámaras en donde se determina como puntos de supervisión los puntos con más comparendos según un tipo de vehículo. Esta red debe tener el menor costo posible teniendo en cuenta que la instalación cuesta 1000000 COP/km.

Entrada	La estructura de datos Vial donde se tienen los datos, la clase de vehículo que se quiere buscar y la cantidad de cámaras que se quieran instalar en la red
Salida	Total de vértices, lista con los vértices utilizados para instalar las cámaras, lista de arcos en la instalación de las cámaras, total de kilómetros necesarios para hacer la red y el costo total de la instalación.
Implementado (Sí/No)	Si, Amelia Serrano

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**

Se obtienen las estructuras de datos necesarias para trabajar que estan guardadas en vial y se crea una lista para guardar los vertices que filtrados	O(1)
Se obtiene el mapa de tipo de carros y se busca la clase de vehiculo que entra por parametro. De esta tabla se obtiene el valor que es otra tabla con los ID de los vertices	O(1)
Se obtiene un set de todos los vertices que tienen comparendos con ese tipo de vehiculo	O(V)
Se organizan la lista de los vertices con respecto a la cantidad de comparendos que tenga cada uno con quick sort	O(VlogV)
Se hace una lista previa que tiene unicamente los vertices necesarios para la red de camaras (m)	O(V)
Se realiza un recorrido de la lista previa y se obtienen los vertices de la estructura de datos vial para agragarlos a una lista	O(V)
Esta lista se envia a la funcion auxiliar que se explica a cntinuacion	O(V^2)
TOTAL	***O(V^2)***

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).

Entrada	**Tiempo (s)**

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Entrada	**Salida**	**Tiempo (s)**

Análisis

La complejidad de este requerimiento es de O(V^2) por lo que se comporta cuadráticamente según la cantidad de datos de entrada. Esto se debe principalmente a el sort de la lista y la funcion auxiliar .

Función Auxiliar

```python
def mst_mini_grafo(vial, lista_vertices):
    """ Carga un subgrafo con la distancia haversine entre los vertices proporcionados en un alista de DISClib.
    los pesos son las distancia haversine.
    Retorna lo pedido por cada uno de los reuerimientos indivduales
    """
    mini_graf = gr.newGraph(directed=False)

    existe_arc = mp.newMap()

    for vertice in lt.iterator(lista_vertices):
        gr.insertVertex(mini_graf, vertice)

    for vertice in lt.iterator(lista_vertices):
        lat_long = me.getValue(mp.get(vial["vertices"],vertice))["lat_long"]
        for vertice_adj in lt.iterator(lista_vertices):
            if vertice_adj != vertice:
                lat_long_adj = me.getValue(mp.get(vial["vertices"], vertice_adj))["lat_long"]
                dist = haversine(lat_long, lat_long_adj)
                tup_1 = (vertice, vertice_adj)
                tup_2 = (vertice, vertice_adj)
                if not mp.get(existe_arc(tup_1)) and not mp.get(existe_arc, tup_2):
                    gr.addEdge(mini_graf, vertice, vertice_adj)
                    mp.put(existe_arc, tup_1, ":)")
                    mp.put(existe_arc, tup_2, ":)")

    search = prim.PrimMST(mini_graf)
    prim.weightMST(mini_graf, search)

    #search["mst"] es un apila con los arcos del mst
```

```python
total_vertices = lt.size(lista_vertices)
arcos = lt.newList
total_km = 0
if st.isEmpty(search["mst"]):
    print(":)")
while not st.isEmpty(search["mst"]):
    arc = st.pop(search["mst"])
    arc_sin_peso = new_arc_sin_peso(arc)
    lt.addLast(arcos, arc_sin_peso)
    total_km += arc["weight"]

costo = total_km * 1000000

return total_vertices, lista_vertices, arcos, total_km, costo
```

Descripción

Crea y carga un subgrafo con la distancia haversine entre los vértices proporcionados en un alista de DISClib. Los pesos son las distancias haversine. Retorna lo pedido por cada uno de los requerimientos individuales

Entrada	La estructura de datos vial, una lista con los vértices necesario para el grafo
Salidas	Total, de vértices, lista con los vértices, arcos utilizados, kilómetros totales para las respectivas redes y costo total de la instalación

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**

Se crea un nuevo grafo en donde se meterán los vértices que cumplen con los requisitos. Y Un mapa	O(1)
Se realiza un recorrido de la lista de vértices y se insertan al grafo	O(V)
Se realiza un segundo recorrido de la lista de vertices en donde se busca cada vertice de la estructura general de datos vial. Y se obtiene su latitud y longitud	O(V)
Adentro de ese recorrido se realiza otro recorrido de la lista de vértices en donde se buscan las coordenadas del segundo vértice para la distancia entre ellos con haversine	O(V^2)
Se realiza un condicional para crear los arcos entre los vertices del grafo y a una tabla con todos los vertices como llave.	O(1)
Se utiliza el algoritmo PRIM de busqueda y se calcula el peso del arbol de recubrimiento	O((V+E)logE)
Se saca un size de la lista para determinar el numero total de vertices	O(V)
Se crea una nueva lista para poder guardar los arcos del grafo creado	O(1)
Se realiza un while que añade todos los arcos encontrados en el PRIM MST a la lista y suma sus pesos para determinar los kilometros	O(E)
Finalmente se realiza una multiplicacion para encontrar el costo de realizar la red	O(1)
TOTAL	***O(V^2)***

Requerimiento 6

N.A

Descripción

N.A

Entrada	N.A
Salidas	N.A
Implementado (Sí/No)	No.

Requerimiento 7

```python
def req_7(vial, lat_i, long_i, lat_f, long_f):
    """
    Como conductor deseo encontrar el
    camino "más corto" en términos número de menor cantidad de
    comparendos entre dos puntos de geográficos localizados en los limites de la ciudad de Bogotá.
    El punto de origen y destino son ingresados por el usuario como latitudes y longitudes (debe validarse que
    dichos puntos se encuentren dentro de los límites encontrados de la ciudad). Estas ubicaciones deben
    aproximarse a los vértices más cercanos en la malla vial.
    """
    # TODO: Realizar el requerimiento 7

    lat_long_i = (float(lat_i), float(long_i))
    lat_long_f = (float(lat_f), float(long_f))

    esta_i = esta_en_bogota(lat_long_i)
    esta_f = esta_en_bogota(lat_long_f)


    if esta_i and esta_f:

        vertice_i = vertice_cercano(vial, lat_long_i)
        vertice_f = vertice_cercano(vial, lat_long_f)

        search = djk.Dijkstra(vial["grafo_comparendos"], vertice_i)
        camino = djk.pathTo(search, vertice_f)

        #search = bf.BellmanFord(vial["grafo_comparendos"], vertice_i)
        #camino = bf.pathTo(search, vertice_f)

        if camino == None:
            print(camino)

        id_vertices = lt.newList()
        arcos = lt.newList()
        total_km = 0
        total_comparendos = 0

        arc_1 = st.pop(camino)
        vertexA = arc_1["vertexA"]
        lt.addLast(id_vertices, vertexA)
        vertexB = arc_1["vertexB"]
        lt.addLast(id_vertices, vertexB)
        arc_1_dis = gr.getEdge(vial["grafo_distancia"], vertexA, vertexB)
        arc_1_comp = gr.getEdge(vial["grafo_comparendos"], vertexA, vertexB)
        total_km += arc_1_dis["weight"]
        total_comparendos += arc_1_comp["weight"]
        arc_1 = new_arc_sin_peso(arc_1)
        vertexA = vertexB
```

```python
    while (not st.isEmpty(camino)):
        vertexB = st.pop(camino)["vertexB"]
        arco_km = gr.getEdge(vial["grafo_distancia"],vertexA, vertexB)
        arco_com = gr.getEdge(vial["grafo_comparendos"],vertexA, vertexB)
        arco = new_arc_sin_peso(arco_com)
        lt.addLast(arcos, arco)
        total_km += arco_km["weight"]
        total_comparendos += arco_com["weight"]
        lt.addLast(id_vertices, vertexB)
        vertexA = vertexB
else:
    id_vertices = None
    arcos = None
    total_comparendos = 0
    total_km = 0

return id_vertices, arcos, total_comparendos, total_km
```

Descripción

Se desea encontrar el camino más corto en términos de numero de menor cantidad de comparendos entre dos puntos. Esto con el fin de que los conductores transiten por las vías que tienen la menor cantidad de comparendos.

Entrada	Entra por parámetro la estructura de datos general vial, además de un punto de origen (coordenada con latitud y longitud) y un punto de destino(coordenada con latitud y longitud)

Salidas	La salida del requerimiento es el total de vértices en el camino, los vértices incluidos(identificadores), arcos incluidos, cantidad de comparendos y cantidad de Kilómetros del camino.
Implementado (Sí/No)	Si Grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Se crean dos variables que tengan las coodenadas de los puntos que entraron por parametro	O(1)
Se utiliza una función auxiliar que determina si los puntos se encuentra en Bogotá	O(1)
Se realiza un condicional en donde si si se encuentra en bogota se realiza lo siguiente	O(1)
Por medio de una función auxiliar se determina el vértice mas cercano al vértice de origen y de destino	O(V)
Por medio del algoritmo dijkstra se busca el camino de mínimo de el vértice de origen al vértice final	O((V+E)logV)
Se crea una nueva lista para poder guardar los vértices, otra lista para guardar los arcos y dos contadores para determinar los kilometros y el total de comparendos	O(1)
Se saca el primer elemento de la pila que se devuelve por la busqueda Dijkstra, se obtienen los vertices de de ese primer elemento y se intoducen a la lista de vertices	O(1)
Se obtienen ambos arcos entre los vertices , el que tiene el peso de comparendos y el que tiene el peso de distancia y se añaden estos pesos a los contadores de distancia y numero de comparendos	O(1)
Con una funcion auxiliar se crea un diccionario con ambos vertices	O(1)
Se hace un while en donde se hace pop a la fila del camino y se obtiene el vertice b de los arcos que componen el camino	O(V)
Se obtienen lo arcos que conectan todos los vertices y se agregan a la lista de arcos y se suman sus respectivos pesos a los contadores	O(E)
TOTAL	***O(V)***

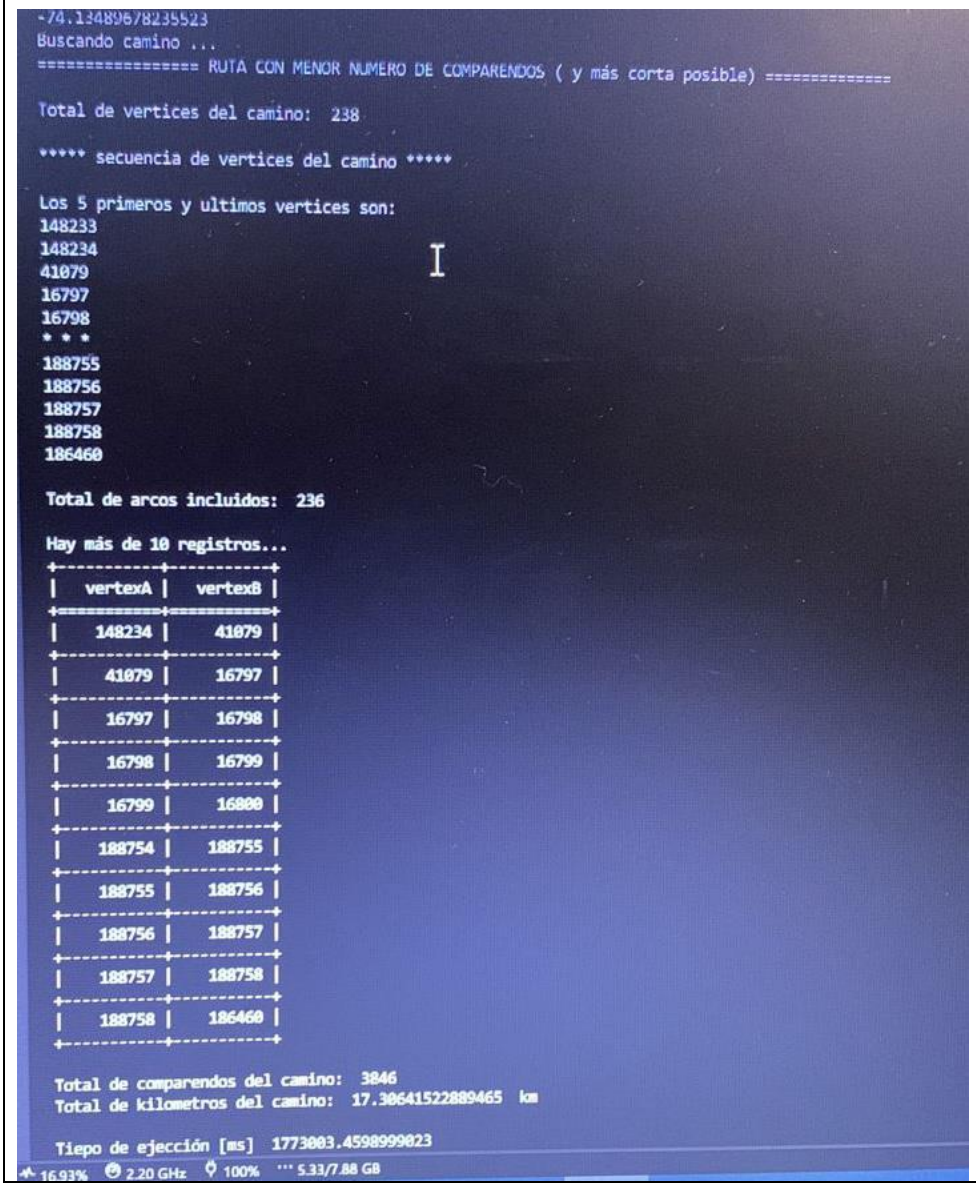
Pruebas Realizadas

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).

Entrada	Tiempo (s)
100% de los datos	1773003.46

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Salida	Tiempo (s)
	1773003.46

Analisis

La complejidad de este requerimiento es de O(V) es decir que su tiempo de ejecución aumenta a medida que mas datos entren a la carga de datos. Esto se debe al algoritmo de Dijkstra pues su recorrido va a ser mas largo entre mas vertices existan.