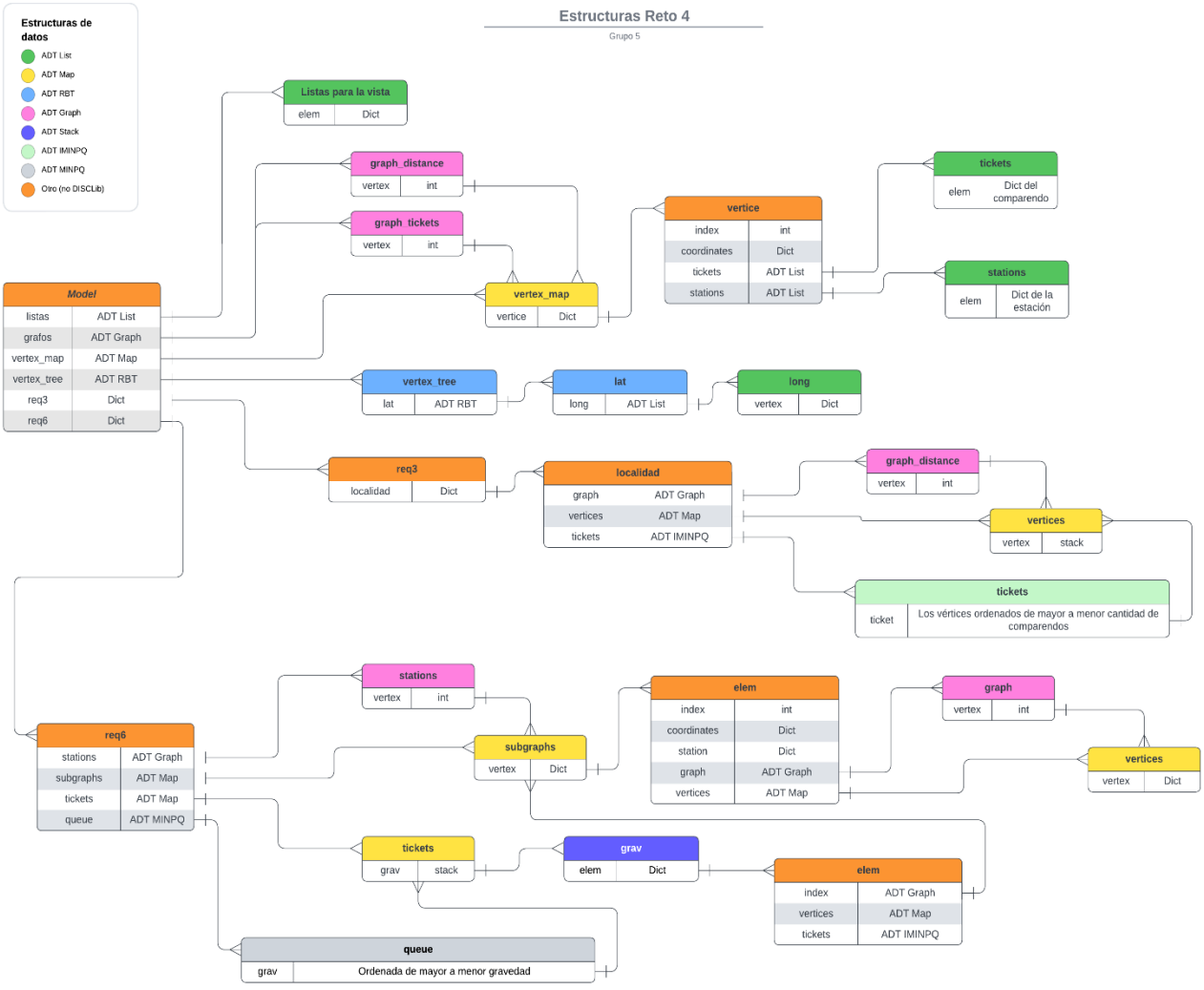
ANÁLISIS DEL RETO

Jhonny Armando Hortua Oyola, 202111749, j.hortuao@uniandes.edu.co

Gabriel Esteban González Carrillo, 202014375, g.gonzalezc@uniandes.edu.co

Adrian Esteban Velasquez Solano - 202222737, a.velasquezs@uniandes.edu.co

Estructuras de Datos



Para el reto 4, utilizamos una gran variedad de estructuras de datos con el fin de acceder a toda la información necesaria para resolver los requerimientos. No solo se utilizaron grafos, sino también colas de prioridad y pilas para garantizar que la información necesaria estuviera disponible de manera adecuada.

Funciones auxiliares

```python
def searchClosestVertex(graph, lat, long):
    # buscar vértice más cercano
    # primero se recorre el arbol de latitudes
    lat_entry = om.get(graph, round(float(lat), 2))
    if lat_entry is None:
        ceiling = om.ceiling(graph, round(float(lat), 2))
        floor = om.floor(graph, round(float(lat), 2))
        if ceiling is None:
            lat_entry = om.get(graph, floor)
        elif floor is None:
            lat_entry = om.get(graph, ceiling)
        else:
            ceiling_distance = abs(float(lat) - float(ceiling))
            floor_distance = abs(float(lat) - float(floor))
            if ceiling_distance < floor_distance:
                lat_entry = om.get(graph, ceiling)
            else:
                lat_entry = om.get(graph, floor)
    # se obtiene la latitud mas cercana, y se accede al arbol de longitudes
    long_map = me.getValue(lat_entry)
    # se recorre el arbol para buscar la longitud más cercana
    long_entry = om.get(long_map, round(float(long), 2))
    if long_entry is None:
        ceiling = om.ceiling(long_map, round(float(long), 2))
        floor = om.floor(long_map, round(float(long), 2))
        if ceiling is None:
            long_entry = om.get(long_map, floor)
        elif floor is None:
            long_entry = om.get(long_map, ceiling)
        else:
            ceiling_distance = abs(float(long) - float(ceiling))
            floor_distance = abs(float(long) - float(floor))
            if ceiling_distance < floor_distance:
                long_entry = om.get(long_map, ceiling)
            else:
                long_entry = om.get(long_map, floor)
    # se obtiene la lista de la longitud más cercana y se recorre
    lst = me.getValue(long_entry)
    # se recorre la lista para determinar cual es el vertice mas cercano
    closest_vertex = -1
    distance = float('inf')
    for vertex in lt.iterator(lst):
        vertex_lat = vertex['coordinates']['lat']
        vertex_long = vertex['coordinates']['long']
        temp_distance = haversineDistance(lat, vertex_lat, long, vertex_long)
        if temp_distance < distance:
            distance = temp_distance
            closest_vertex = vertex['index']
    # retorna el indice del vertice más cercano
    return closest_vertex
```

A1: searchClosestVertex() – Complejidad O(Lat/Long/v)

```python
def getClosestStation(latref, longref, reqó_stations, reqó_subgraphs):
    # se busca la estación más cercana
    stations = gr.vertices(reqó_stations)
    # se comparan todas las distancias del vertice a todas las estaciones
    mindist = float('inf')
    closest_station = -1
    for station_vertex in lt.iterator(stations):
        entry = mp.get(reqó_subgraphs, station_vertex)
        station = me.getValue(entry)
        lat = station['coordinates']['lat']
        long = station['coordinates']['long']
        distance = haversineDistance(latref, lat, longref, long)
        if distance < mindist:
            mindist = distance
            closest_station = station_vertex
    # retorna la estacion más cercana
    return closest_station
```

A2: getClosestStation() – Complejidad O(s)

Requerimiento 1

```
def req1(data_structs, vi, vf, count):
    """
    Función que soluciona el requerimiento 1
    """
    # Realizar el requerimiento 1
    distance_graph = data_structs['graph_distance']
    vertices_map = data_structs['vertex_map']
    vertices_tree = data_structs['vertex_tree']

    vi_index = searchClosestVertex(vertices_tree, vi['lat'], vi['long'])
    vf_index = searchClosestVertex(vertices_tree, vf['lat'], vf['long'])
    paths = dfs.DepthFirstSearch(distance_graph, vi_index)
    if not dfs.hasPathTo(paths, vf_index):
        temp_stack =  st.newStack()
    else:
        temp_stack = dfs.pathTo(paths, vf_index)

    path_q = st.newStack()
    i = 0
    prev = None
    while not st.isEmpty(temp_stack):
        vi = st.pop(temp_stack)
        qu.enqueue(path_q, vi)
        if i == 0:
            entry = mp.get(vertices_map, vi)
            vi_info = me.getValue(entry)
            vilat = vi_info['coordinates']['lat']
            vilong = vi_info['coordinates']['long']

            if prev is None:
                prev = {'index':vi,
                        'coordinates':{'lat':vilat,
                                       'long':vilong}}
            else:
                vflat = prev['coordinates']['lat']
                vflong = prev['coordinates']['long']

                count['distance'] += abs(haversineDistance(vilat, vflat, vilong, vflong))

                prev = {'index':vi,
                        'coordinates':{'lat':vilat,
                                       'long':vilong}}
            i+=1
        else:
            i = 0

    return path_q
```

F1.1: req1()

Descripción

El requerimiento 1 pretende encontrar un posible camino desde un punto A hasta un punto B. Para esto se hace uso del grafo de distancias, al igual que una función auxiliar para encontrar el vértice más cercano de las latitudes y longitudes de entrada. Posteriormente se usa el algoritmo DFS para encontrar un posible camino entre los vértices.

Entrada	Estructuras de datos del modelo, latitudes y longitudes para la búsqueda
Salidas	Pila de un camino encontrado
Implementado (Sí/No)	Si, Adrian Velasquez 202222737

Análisis de complejidad

Función	Paso	Complejidad		
F1.1	Acceder a todas las estructuras	O(	1	)
F1.1	Usar A1	O(	Lat/Long/v	)
F1.1	Usar A1	O(	Lat/Long/v	)
F1.1	DFS	O(	RM	)
F1.1	pathTo	O(	1	)
F1.1	Recorrer stack	O(	s	)
Total		O(	Lat/Long/v + V + E+ s	)

Pruebas Realizadas

Las pruebas fueron realizadas con una máquina de las siguientes especificaciones. Los datos de entrada fueron: **4.60293518548777**, **-74.06511801444837**, **4.693518613347496**, **-74.13489678235523**

Procesadores	**11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz**
Memoria RAM	16 GB
Sistema Operativo	Windows 11

Tablas de datos

Tiempo [ms]	Memoria [kb]
9445.252	N/A

Análisis

En este requerimiento se hizo un DFS con el fin de obtener un camino distinto al del requerimiento 2 entre el punto inicial y el punto final. En el peor caso, el tiempo de carga podría ser mucho mayor al BFS, y la respuesta también se espera que sea muy distinta. Esto se debe a la forma en la que funciona el recorrido DFS, que por la naturaleza es recursivo, y recorre el grafo rama por rama en vez de todas las ramas simultáneamente.

Requerimiento 2

```
def req2(data_structs, vi, vf, count):
    """
    Función que soluciona el requerimiento 2
    """
    # Realizar el requerimiento 2
    distance_graph = data_structs['graph_distance']
    vertices_map = data_structs['vertex_map']
    vertices_tree = data_structs['vertex_tree']

    vi_index = searchClosestVertex(vertices_tree, vi['lat'], vi['long'])
    vf_index = searchClosestVertex(vertices_tree, vf['lat'], vf['long'])

    paths = bfs.BreathFirstSearch(distance_graph, vi_index)
    if not bfs.hasPathTo(paths, vf_index):
        temp_stack =  st.newStack()
    else:
        temp_stack = bfs.pathTo(paths, vf_index)

    path_q = st.newStack()
    i = 0
    prev = None
    while not st.isEmpty(temp_stack):
        vi = st.pop(temp_stack)
        qu.enqueue(path_q, vi)
        if i == 0:
            entry = mp.get(vertices_map, vi)
            vi_info = me.getValue(entry)
            vilat = vi_info['coordinates']['lat']
            vilong = vi_info['coordinates']['long']

            if prev is None:
                prev = {'index':vi,
                        'coordinates':{'lat':vilat,
                                       'long':vilong}}
            else:
                vflat = prev['coordinates']['lat']
                vflong = prev['coordinates']['long']

                count['distance'] += abs(haversineDistance(vilat, vflat, vilong, vflong))

                prev = {'index':vi,
                        'coordinates':{'lat':vilat,
                                       'long':vilong}}
            i+=1
        else:
            i = 0

    return path_q
```

F2.1: req1()

Descripción

El requerimiento 2 pretende encontrar el camino más corto en términos de intersecciones entre el punto A y el punto B, los cuales son los vértices más cercanos a las coordenadas de entrada. Para esto se implementa un grafo cuyos arcos son distancias entre puntos de la malla vial, y se hace un recorrido BFS para encontrar el camino con menos vértices entre ambos puntos.

Entrada	Estructuras de datos del modelo, latitudes y longitudes de búsqueda
Salidas	Pila con el camino más corto en términos de intersecciones entre ambos vértices
Implementado (Sí/No)	Si, Adrian Velasquez 202222737

Análisis de complejidad

Función	Paso	Complejidad		
F1.1	Acceder a todas las estructuras	O(	1	)
F1.1	Usar A1	O(	Lat/Long/v	)
F1.1	Usar A1	O(	Lat/Long/v	)
F1.1	BFS	O(	N	)
F1.1	pathTo	O(	1	)
F1.1	Recorrer stack	O(	s	)
Total		O(	Lat/Long/v + V + E+ s	)

Pruebas Realizadas

Las pruebas fueron realizadas con una máquina de las siguientes especificaciones. Los datos de entrada fueron: **4.60293518548777**, **-74.06511801444837**, **4.693518613347496**, **-74.13489678235523**

Procesadores	**11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz**
Memoria RAM	16 GB
Sistema Operativo	Windows 11

Tablas de datos

Tiempo [ms]	Memoria [kb]
6661.945	N/A

Análisis

Para encontrar el camino del vértice A al vértice B más corto se debe hacer una búsqueda BFS. Esto significa que la complejidad del recorrido en sí no supera el número de vértices y de arcos del grafo. Por otro lado, si en vez de la ruta con menos intersecciones se buscara la ruta más corta, se tendría que implementar otro algoritmo, particularmente el algoritmo de Dijkstra, con el fin de encontrar la ruta más corta en términos de distancia.

Requerimiento 3

```python
def req3(data_structs, cameras, localidad, cost, count):
    """
    Función que soluciona el requerimiento 3
    """
    # Realizar el requerimiento 3
    distance_graph = data_structs['graph_distance']
    req3 = data_structs['req3']

    count['included_vertices'] = st.newStack()

    localidad_vertices = req3[localidad]['vertices']
    localidad_tickets = req3[localidad]['tickets']

    most_tickets_queue = req3GetMostTickets(localidad_vertices, localidad_tickets, cameras)

    # version 1: Prim sencillo
    graph = gr.newGraph()
    req3MakeGraph(most_tickets_queue, graph, localidad_vertices)
    vertices = gr.vertices(graph)
    origin = lt.firstElement(vertices)
    mst = prim.PrimMST(graph)
    edges = prim.edgesMST(graph, mst)
    count['distance'] = prim.weightMST(graph, mst)
    count['cost'] = count['distance']*cost

    return edges, vertices, origin
```

F3.1: req3()

```python
def req3GetMostTickets(localidad_vertices, localidad_tickets, cameras):
    vertices_keylist = mp.keySet(localidad_vertices)
    for key in lt.iterator(vertices_keylist):
        entry = mp.get(localidad_vertices, key)
        vertex = me.getValue(entry)
        size = lt.size(vertex['tickets'])
        impq.insert(localidad_tickets, key, -size)

    most_tickets_queue = qu.newQueue()

    if impq.size(localidad_tickets) < cameras:
        for i in range(impq.size(localidad_tickets)):
            key = impq.min(localidad_tickets)
            qu.enqueue(most_tickets_queue, key)
            impq.delMin(localidad_tickets)
    else:
        for i in range(cameras):
            key = impq.min(localidad_tickets)
            qu.enqueue(most_tickets_queue, key)
            impq.delMin(localidad_tickets)

    return most_tickets_queue
```

F3.2: req3getMostTickets()

```python
def req3MakeGraph(most_tickets_queue, localidad_graph, localidad_vertices):
    while not qu.isEmpty(most_tickets_queue):
        vertex = int(qu.dequeue(most_tickets_queue))
        gr.insertVertex(localidad_graph, vertex)

    vertex_list = gr.vertices(localidad_graph)

    for vi_index in lt.iterator(vertex_list):
        entry = mp.get(localidad_vertices, vi_index)
        vi = me.getValue(entry)
        vi_lat = vi['coordinates']['lat']
        vi_long = vi['coordinates']['long']

        for vf_index in lt.iterator(vertex_list):
            if vi_index != vf_index:
                entry = mp.get(localidad_vertices, vf_index)
                vf = me.getValue(entry)
                vf_lat = vf['coordinates']['lat']
                vf_long = vf['coordinates']['long']

                distance = abs(haversineDistance(vi_lat, vf_lat, vi_long, vf_long))
                gr.addEdge(localidad_graph, vi_index, vf_index, distance)
```

F3.3: req3MakeGraph

Descripción

El requerimiento 3 pretende encontrar la red más corta entre M puntos en una localidad, los cuales se encuentran en los M vértices de dicha localidad con mayor cantidad de comparendos. Para esto se

utilizan muchas subestructuras, como un mapa de vértices separado por localidad, al igual que listas de comparendos por cada uno de lo vértices, y una cola de prioridad indexada para identificar los M puntos con más comparendos.

Entrada	Estructuras de datos del modelo, cantidad de cámaras a instalar, y una localidad a evaluar. (También se ingresa el costo como posible variable).
Salidas	Un árbol de recubrimiento de los M vértices con más comparendos de la localidad, los cuales fueron previamente conectados hipotéticamente con cables subterráneos de fibra óptica.
Implementado (Sí/No)	Sí, Adrian Velasquez 202222737

Análisis de complejidad

Función	Paso	Complejidad		
3.1	Consultar modelo y otros	O(	1	)
3.1	Invocar 3.2	O(	1	)
3.2	Recorrer las llaves del mapa de vertices y añadirlos al iminpq	O(	k	)
3.2	Sacar los M primeros vértices del iminpq	O(	M	)
3.1	Invocar 3.3	O(	1	)
3.3	Insertar M vértices al grafo	O(	M	)
3.3	Para cada vértice, añadir un arco de distancia a todos los demás	O(	M	)
3.1	Prim	O(	ElogV	)
3.1	Retornar estructuras	O(	1	)
Total		O(	k + 3M + ElogM	)

Pruebas Realizadas

Las pruebas realizadas fueron realizadas con las siguientes especificaciones. Los datos de entrada fueron **20** cámaras en la localidad de **Chapinero.**

Procesadores	**11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz**
Memoria RAM	16 GB
Sistema Operativo	Windows 11

Tablas de datos

Tiempo [ms]	Memoria [kb]
211.444	N/A

Análisis

En este requerimiento se necesita la red con menor costo posible, razón por la cual se implementa el algoritmo Prim. Debido a la cantidad relativamente pequeña de vértices, siendo una red de fibra óptica subterránea, la forma más directa de obtener la red mínima es conectando todos los M vértices con más comparendos entre sí usando la distancia entre ellos como el peso de los arcos, y haciendo un árbol de recubrimiento con el menor kilometraje posible.

Requerimiento 4

```python
def req_4(data_structures, cameras, cost, complete, counter):

    req4_data = data_structures['req4'][cameras]
    vertices_location, tickets_location = req4_data['vertices'], req4_data['tickets']
    tickets_queue = req4_get_best_selling_tickets(vertices_location, tickets_location, cameras)

    graph_location = gr.newGraph() if complete else req4_create_graph(tickets_queue, vertices_location)

    vertices = gr.vertices(graph_location)
    vertex_counter = req4_evaluate_vertices(graph_location, vertices)

    min_vertex = min(vertex_counter, key=lambda vertex: vertex_counter[vertex]['distancia'])
    paths = req4_compute_paths(graph_location, vertices, min_vertex, vertex_counter, complete, counter, cost)

    return paths, vertices, min_vertex


def req4_get_best_selling_tickets(vertices_location,cameras):
    sorted_vertices = sorted(vertices_location, key=lambda key: -lt.size(vertices_location[key]['tickets']))
    return qu.newQueue(sorted_vertices[:cameras])
```

```python
def req4_create_graph(tickets_queue, vertices_location):
    graph_location = gr.newGraph()
    for vertex in qu.iterator(tickets_queue):
        gr.insert_vertex(graph_location, vertex)

    for vi_index in gr.vertices(graph_location):
        vi_coordinates = vertices_location[vi_index]['coordinates']
        for vf_index in gr.vertices(graph_location):
            if vi_index != vf_index:
                vf_coordinates = vertices_location[vf_index]['coordinates']
                gr.add_edge(graph_location, vi_index, vf_index)

    return graph_location


def req4_evaluate_vertices(graph_location,):
    vertex_counter = {}
    for vertex in gr.vertices(graph_location):
        vertex_counter[vertex] = {'busqueda': djk.Dijkstra(graph_location, vertex),
                                  'distancia': sum(djk.distance_to(vertex_counter[vertex]['busqueda'], idx)
                                                   for idx in gr.vertices(graph_location) if idx != vertex)}
    return vertex_counter
```

```python
def req4_compute_paths(graph_location, vertices, min_vertex, vertex_counter, complete, counter, cost):
    if complete:
        paths = mp.newMap(loadfactor=4)
        search = vertex_counter[min_vertex]['busqueda']
        for vertex in gr.vertices(graph_location):
            if vertex != min_vertex:
                paths[vertex] = djk.path_to(search, vertex)
                counter['distancia'] += djk.distance_to(search, vertex)

        counter['costo'] = counter['distancia'] * cost
        return paths

    else:
        origin = lt.first_element(vertices)
        mst = prim.PrimMST(graph_location, origin)
        edges = prim.edges_MST(graph_location, mst)
        counter['distancia'] = prim.weight_MST(graph_location, mst)
        counter['costo'] = counter['distancia'] * cost
        return edges
```

Descripción

El requerimiento 4 quiere determinar la red de comunicaciones que soporte instalar cámaras de ciertos puntos haciendo que tengan el menor costo de instalación posible, para que la red sea eficiente se seleccionan como puntos de supervisión los vértices con comparendos de mayor gravedad.

Entrada	Numero de cámaras
Salidas	Respuesta con vertices,arcos,kilometros, costo
Implementado (Sí/No)	Si, Jhonny Hortua

Análisis de complejidad

Función	Paso	Complejidad		
5.1	Consultar modelo y otros	O(	1	)
5.1	Invocar 5.2	O(	1	)
5.2	Recorrer llaves del mapa de vértices	O(	k	)
5.2	Sacar los M primeros vértices del iminpq	O(	M	)
5.1	Insertar M vértices al grafo	O(	1	)
5.3	Para cada vértice, añadir arco a otros	O(	M	)
5.3	Retornar estructuras	O(	M	)
Total		O(	k + 3M + ElogM	)

Pruebas Realizadas

Procesadores	**AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11

Tablas de datos

Tiempo [ms]	Memoria [kb]
930.462	N/A

Requerimiento 5

```python
def req5(data_structs, cameras, vehicle, cost, count):
    """Función que soluciona el requerimiento 3"""
    # Realizar el requerimiento 3
    distance_graph = data_structs['graph_distance']
    req5 = data_structs['req5']

    count['included_vertices'] = st.newStack()

    vehiculo_vertices = req5[vehicle]['vertices']
    vehiculo_tickets = req5[vehicle]['tickets']

    most_tickets_queue = req5GetMostTickets(vehiculo_vertices, vehiculo_tickets, cameras)

    # version 1: Prim sencillo
    graph = gr.newGraph()
    req5MakeGraph(most_tickets_queue, graph, vehiculo_vertices)
    vertices = gr.vertices(graph)
    origin = lt.firstElement(vertices)
    mst = prim.PrimMST(graph)
    edges = prim.edgesMST(graph, mst)
    count['distance'] = prim.weightMST(graph, mst)
    count['cost'] = count['distance']*cost

    return edges, vertices, origin
```

F5.1 req5 ()

```python
def req5GetMostTickets(vehiculo_vertices, vehiculo_tickets, cameras):
    vertices_keylist = mp.keySet(vehiculo_vertices)
    for key in lt.iterator(vertices_keylist):
        entry = mp.get(vehiculo_vertices, key)
        vertex = me.getValue(entry)
        size = lt.size(vertex['tickets'])
        impq.insert(vehiculo_tickets, key, -size)

    most_tickets_queue = qu.newQueue()

    if impq.size(vehiculo_tickets) < cameras:
        for i in range(impq.size(vehiculo_tickets)):
            key = impq.min(vehiculo_tickets)
            qu.enqueue(most_tickets_queue, key)
            impq.delMin(vehiculo_tickets)
    else:
        for i in range(cameras):
            key = impq.min(vehiculo_tickets)
            qu.enqueue(most_tickets_queue, key)
            impq.delMin(vehiculo_tickets)

    return most_tickets_queue
```

F5.2 req5 get most tickets

```python
def req5MakeGraph(most_tickets_queue, vehiculo_graph, vehiculo_vertices):
    while not qu.isEmpty(most_tickets_queue):
        vertex = int(qu.dequeue(most_tickets_queue))
        gr.insertVertex(vehiculo_graph, vertex)

    vertex_list = gr.vertices(vehiculo_graph)

    for vi_index in lt.iterator(vertex_list):
        entry = mp.get(vehiculo_vertices, vi_index)
        vi = me.getValue(entry)
        vi_lat = vi['coordinates']['lat']
        vi_long = vi['coordinates']['long']

        for vf_index in lt.iterator(vertex_list):
            if vi_index != vf_index:
                entry = mp.get(vehiculo_vertices, vf_index)
                vf = me.getValue(entry)
                vf_lat = vf['coordinates']['lat']
                vf_long = vf['coordinates']['long']

                distance = abs(haversineDistance(vi_lat, vf_lat, vi_long, vf_long))
                gr.addEdge(vehiculo_graph, vi_index, vf_index, distance)
```

F5.3 makegraph

Descripción

Entrada	Cantidad de cámaras a instalar, tipo de vehículo.
Salidas	Tiempo del algoritmo, vértices incluidos, arcos incluidos, cantidad de kilómetros de red extendida, costo monetario de la instalación.
Implementado (Sí/No)	Si – Gabriel Esteban González Carrillo

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Función	Paso	Complejidad		
5.1	Consultar modelo y otros	O(	1	)
5.1	Invocar 5.2	O(	1	)
5.2	Recorrer las llaves del mapa de vértices y añadirlos al iminpq	O(	k	)
5.2	Sacar los M primeros vértices del iminpq	O(	M	)
5.1	Invocar 5.3	O(	1	)
5.3	Insertar M vértices al grafo	O(	M	)
5.3	Para cada vértice, añadir un arco de distancia a todos los demás	O(	M	)
5.1	Prim	O(	ElogV	)
5.1	Retornar estructuras	O(	1	)
Total		O(	k + 3M + ElogM	)

Pruebas Realizadas

Las pruebas realizadas se implementaron con una cantidad de cámaras de 10 y la clase de vehículo 'AUTOMÓVIL'. Al ingresar estos datos de entrada al código realizado para el requerimiento 5 se obtuvieron los siguientes tiempos al varias el porcentaje de cantidad de datos a revisar.

Procesadores	Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz 1.20 GHz
Memoria RAM	8 GB
Sistema Operativo	Windows 11

Tablas de datos

Tiempo [ms]	Memoria [kb]
670.832	N/A

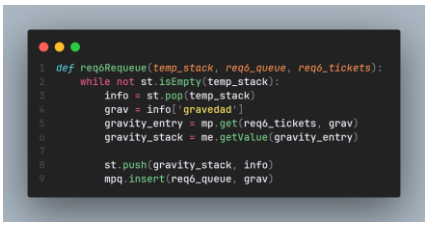
Análisis

Para solucionar el requerimiento se usó el algoritmo de prim con el fin de crear un árbol de esparcimiento de costo mínimo. La estructura de solución es igual que la del requerimiento 3, con la diferencia de que el mapa de infracciones tiene como llaves las clases de autos.

Requerimiento 6

```python
def req6(data_structs, tickets, count):
    """
    Función que soluciona el requerimiento 6
    """
    # Realizar el requerimiento 6
    req6 = data_structs['req6']
    req6_stations = req6['stations']
    req6_subgraphs = req6['subgraphs']
    req6_tickets = req6['tickets']
    req6_queue = req6['queue']

    count['included_vertices'] = st.newStack()

    paths = mp.newMap(loadfactor=4)
    dijkstra_set = mp.newMap(loadfactor=4)

    temp_grav_stack = st.newStack()
    ticket_q = qu.newQueue()
    ticket_count = 0
    while ticket_count < tickets:
        current_gravity = mpq.min(req6_queue)

        gravity_entry = mp.get(req6_tickets, current_gravity)
        gravity_stack = me.getValue(gravity_entry)

        while not st.isEmpty(gravity_stack) and ticket_count < tickets:
            info = st.pop(gravity_stack)
            st.push(temp_grav_stack, info)
            info_index = info['index']
            lat = info['coordinates']['lat']
            long = info['coordinates']['long']
            gravedad = info['gravedad']
            closest_station = info['closest_station']

            entry = mp.get(req6_subgraphs, closest_station)
            station = me.getValue(entry)

            station_graph = station['graph']
            station_index = station['index']
            station_info = station['station']
            name = station_info['EPONOMBRE']


            if not mp.contains(dijkstra_set, station_index):
                search = djk.Dijkstra(station_graph, station_index)
                mp.put(dijkstra_set, station_index, search)

            entry = mp.get(dijkstra_set, station_index)
            search = me.getValue(entry)

            path_to = djk.pathTo(search, info_index)
            dist_to = djk.distTo(search, info_index)

            v = {'index':info_index,
                 'path':path_to,
                 'dist':dist_to,
                 'gravedad':gravedad,
                 'station':name}

            mp.put(paths, info_index, v)
            qu.enqueue(ticket_q, info_index)
            ticket_count += 1

        mpq.delMin(req6_queue)

    req6Requeue(temp_grav_stack, req6_queue, req6_tickets)

    return paths, ticket_q
```

F6.1: req6()



```python
def req6Requeue(temp_stack, req6_queue, req6_tickets):
    while not st.isEmpty(temp_stack):
        info = st.pop(temp_stack)
        grav = info['gravedad']
        gravity_entry = mp.get(req6_tickets, grav)
        gravity_stack = me.getValue(gravity_entry)

        st.push(gravity_stack, info)
        mpq.insert(req6_queue, grav)
```

F6.2: req6Requeue()

Descripción

Este requerimiento pretende encontrar la ruta más corta desde cualquier estación de policía hasta los M comparendos más graves de toda la ciudad. Para esto se utiliza una cola de prioridad con todos los grados de gravedad presentes en el grafo. Asimismo, se crean 21 subgrafos del grafo original, cada uno correspondiendo a una estación de policía, en los cuales solamente se encuentran los vértices más cercanos a dichas estaciones. Para evitar la perdida de información de las colas de prioridad y las pilas de comparendos en el mapa de grados de gravedad, se utiliza una pila temporal con la cual al finalizar los procedimientos se vuelven a agregar los datos en el orden necesario para el buen funcionamiento del programa.

Entrada	Estructuras de datos del modelo, cantidad comparendos a evaluar
Salidas	Las pilas de los caminos más cortos hacia los comparendos más graves de la ciudad.
Implementado (Sí/No)	Sí. Adrian Velasquez 202222737

Análisis de complejidad

Función	Paso	Complejidad		
6.1	Acceder a estructuras y demás operaciones	O(	1	)
6.1	Recorrer minpq de grados de gravedad, y por cada uno recorrer su pila de comparendos	O(	gs	)
6.1	Por cada estación necesaria, hacer un Dijkstra	O(	S(v + elogv)	)
6.1	Demás operaciones	O(	1	)
6.1	Invocar 6.2	O(	1	)
6.2	Recorrer toda la pila temporal y demás operaciones	O(	p	)
Total		O(	gs + S(v + elogv) +p	)

Pruebas Realizadas

Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron los **20** comparendos más graves.

Procesadores	**11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz**
Memoria RAM	16 GB
Sistema Operativo	Windows 11

Tablas de datos

Tiempo [ms]	Memoria [kb]
95622.972	N/A

Análisis

En este requerimiento fue donde más estructuras de datos fueron implementadas, y es gracias a esto que los recorridos, pese a no ser los más rápidos, son ideales y eficientes sin importar cuántos comparendos se busquen. Esto se debe a la manera en la que los subgrafos permiten que la búsqueda por cada estación con el algoritmo de Dijkstra se haga tan solo una vez. Se podría mejorar el requerimiento incluyendo dicha búsqueda durante la carga para cada una de las estaciones, pero por temas del tiempo de carga, uso de memoria, y las pruebas de los requerimientos, al igual que la sustentación, tomamos la decisión de hacer la búsqueda durante el requerimiento. Sin embargo, si se quisiera implementar la búsqueda durante la carga, sería posible mediante una función que, una vez cargados todos los subgrafos, los recorra uno por uno y ejecute el algoritmo de Dijkstra desde todas las estaciones. Esto reduciría la complejidad temporal al máximo, y el requerimiento solo se encargaría de retornar los caminos más cortos desde cada estación hasta los respectivos M comparendos.

Requerimiento 7

```
def req7(data_structs, vi, vf, count):
    """
    Función que soluciona el requerimiento 7
    """
    # Realizar el requerimiento 7
    tickets_graph = data_structs['graph_tickets']
    vertices_tree = data_structs['vertex_tree']
    vertices_map = data_structs['vertex_map']

    vi_index = searchClosestVertex(vertices_tree, vi['lat'], vi['long'])
    vf_index = searchClosestVertex(vertices_tree, vf['lat'], vf['long'])

    paths = djk.Dijkstra(tickets_graph, vi_index)
    if not djk.hasPathTo(paths, vf_index):
        temp_stack = st.newStack()
    else:
        temp_stack = djk.pathTo(paths, vf_index)
        count['tickets'] = djk.distTo(paths, vf_index)

    path_size = st.size(temp_stack)
    path_q = st.newStack()
    i = 0
    prev = None
    while not st.isEmpty(temp_stack):
        ai = st.pop(temp_stack)
        qu.enqueue(path_q, ai)
        vi = ai['vertexA']
        if i == 0:
            entry = mp.get(vertices_map, vi)
            vi_info = me.getValue(entry)
            vilat = vi_info['coordinates']['lat']
            vilong = vi_info['coordinates']['long']

            if prev is None:
                prev = {'index':vi,
                        'coordinates':{'lat':vilat,
                                       'long':vilong}}
            else:
                vflat = prev['coordinates']['lat']
                vflong = prev['coordinates']['long']

                count['distance'] += abs(haversineDistance(vilat, vflat, vilong, vflong))

                prev = {'index':vi,
                        'coordinates':{'lat':vilat,
                                       'long':vilong}}
            i+=1
        elif i == path_size-1:
            vf = ai['vertexB']

            entry = mp.get(vertices_map, vi)
            vi_info = me.getValue(entry)
            vilat = vi_info['coordinates']['lat']
            vilong = vi_info['coordinates']['long']

            entry = mp.get(vertices_map, vf)
            vf_info = me.getValue(entry)
            vflat = vi_info['coordinates']['lat']
            vflong = vi_info['coordinates']['long']

            count['distance'] += abs(haversineDistance(vilat, vflat, vilong, vflong))

        else:
            i = 0

    return path_q
```

F7.1: req7

Descripción

Este requerimiento pretende encontrar la ruta con menos comparendos posible entre un punto A y un punto B. Para esto se utiliza el algoritmo de Dijkstra, al igual que un grafo de toda la ciudad cuyos arcos tienen un peso equivalente a la suma de la cantidad de comparendos entre ambos vértices.

Entrada	Estructuras de datos del modelo, latitudes y longitudes para la búsqueda
Salidas	Pila con el camino de menor cantidad de comparendos entre el punto A y el punto B
Implementado (Sí/No)	Sí. Adrian Velasquez 202222737

Análisis de complejidad

Función	Paso	Complejidad		
7.1	Acceder a todas las estructuras	O(	1	)
7.1	Usar A1	O(	Lat/Long/v	)
7.1	Usar A1	O(	Lat/Long/v	)
7.1	BFS	O(	V + ElogV	)
7.1	pathTo	O(	1	)
7.1	Recorrer stack	O(	s	)
Total		O(	Lat/Long/v + V + ElogV + s	)

Pruebas Realizadas

Las pruebas fueron realizadas con una máquina de las siguientes especificaciones. Los datos de entrada fueron: **4.60293518548777**, **-74.06511801444837**, **4.693518613347496**, **-74.13489678235523**

Procesadores	**11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz**
Memoria RAM	16 GB
Sistema Operativo	Windows 11

Tablas de datos

Tiempo [ms]	Memoria [kb]
2438674	N/A

Análisis

En este requerimiento, idealmente duraría menos. Sin embargo, por dificultades que tuvimos a la hora de utilizar la librería, decidimos utilizar el algoritmo de Dijkstra, y la complejidad temporal se ve afectada. Debido a esto, el tiempo de ejecución es mucho más alto de lo deseado, y reconocemos que puede ser substancialmente mejorado a la hora de implementar los algoritmos adecuados.

Requerimiento 8

Plantilla para el 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.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1	O(...)
Paso 2	O(...)
Paso ….	O(...)
TOTAL	***O(...)***

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.

Graficas

Las gráficas con la representación de las pruebas realizadas.

Análisis

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