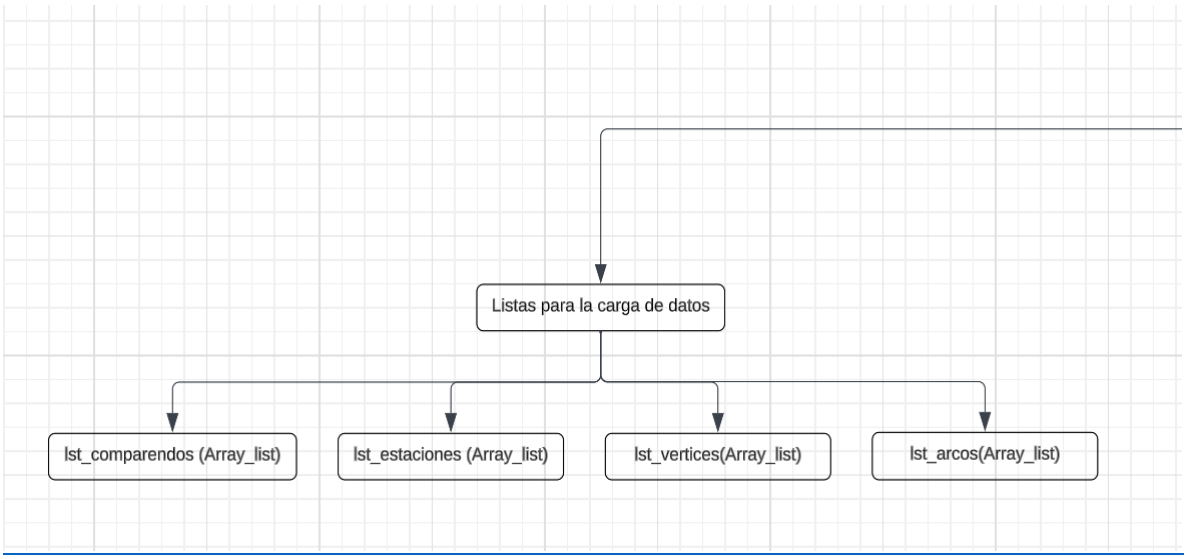
ANÁLISIS DEL RETO

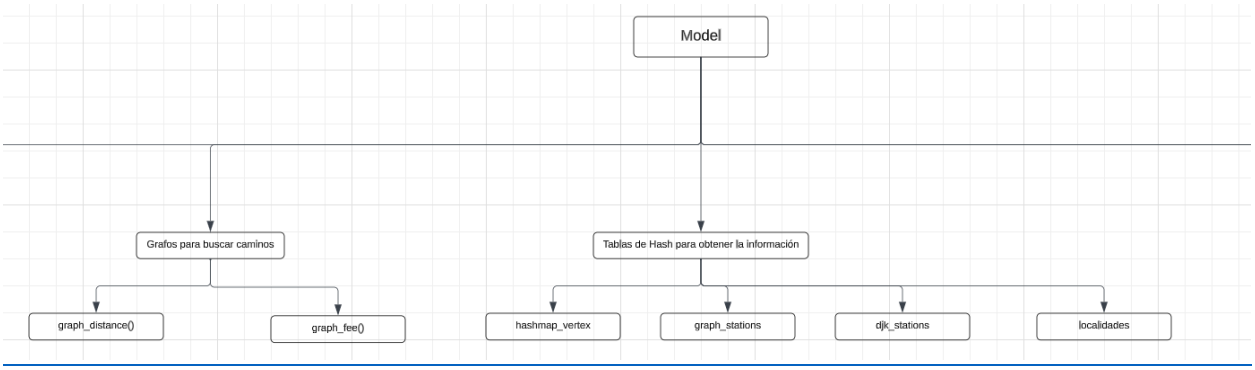
Daniel Camilo Quimbay Velásquez, 202313861, d.quimbay@uniandes.edu.co

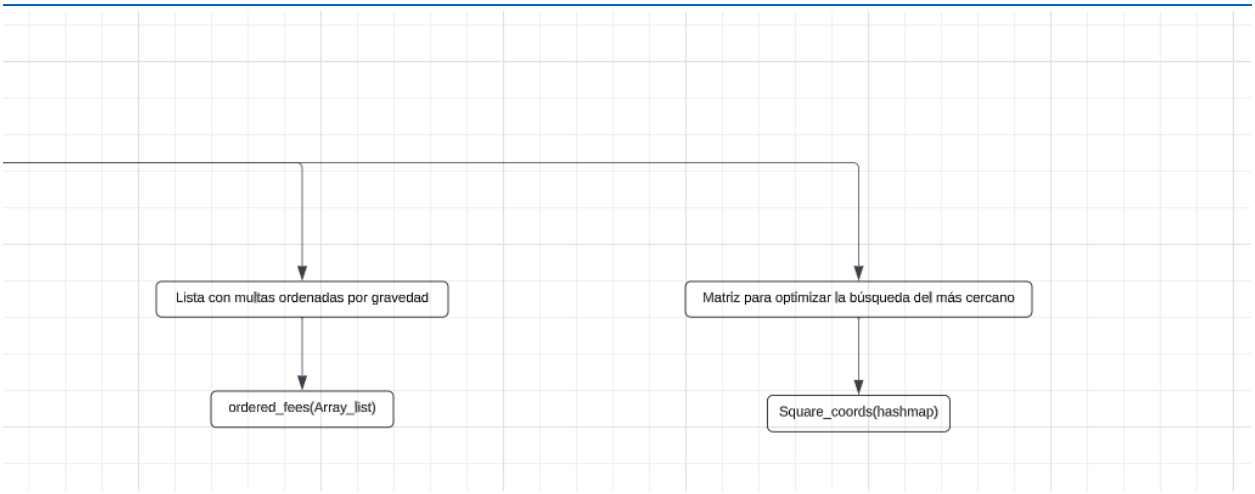
Julián David Contreras Pinilla, 202223394, j.contrerasp@uniandes.edu.co

Carga de datos

Desde el model saldrían doce flechas para “lst_comparendos”, “lst_estaciones”, “lst_vertices” y “lst_arcos” que serían las listas para la carga de datos. Luego tendríamos “graph_distance” y “graph_fee” que son grafos que buscan caminos. Después tenemos “hashmap_vertex”, “graph_stations”, “djk_stations” y “localidades” que son Tablas de hash de donde se obtiene la información. A continuación, tenemos “ordered_fees” que es una lista con multas ordenadas por gravedad. Por último, “square_coords” que es una matriz para optimizar la búsqueda del más cercano.


Listas para la carga de datos
lst_comparendos (Array_list)
lst_estaciones (Array_list)
lst_vertices(Array_list)
lst_arcos(Array_list)


Model
Grafos para buscar caminos
Tablas de Hash para obtener la información
graph_distance()
graph_fee()
hashmap_vertex
graph_stations
djk_stations
localidades


Lista con multas ordenadas por gravedad
Matriz para optimizar la búsqueda del más cercano
ordered_fees(Array_list)
Square_coords(hashmap)

Requerimiento 01

Descripción

```python
def req_1(model, lat_origin, long_origin, lat_dest, long_dest, bono):
    """
    Función que soluciona el requerimiento 1
    """
    # TODO: Realizar el requerimiento 1
    graph = model['graph_distance']
    vertex_origin = closest_vertice(model, lat_origin, long_origin)
    vertex_dest = closest_vertice(model, lat_dest, long_dest)
    model['search'] = dfs.DepthFirstSearch(graph, vertex_origin['id'])
    haspath = dfs.hasPathTo(model['search'], vertex_dest['id'])
    total_distancy = 0
    total_vertex = 0
    path = lt.newList('ARRAY_LIST')
    list_bono = []
    prev_coords = []
    if haspath:
        model['search'] = dfs.pathTo(model['search'], vertex_dest['id'])
        prev = None
        for vertex in lt.iterator(model['search']):
            total_vertex += 1
            lt.addLast(path, vertex)
            if prev is not None:
                edge = gr.getEdge(graph, vertex, prev)
                weight = edge['weight']
                total_distancy += weight
            if bono:
                v_info = me.getValue(mp.get(model['hashmap_vertex'], vertex))
                lat = v_info['lat']
                long = v_info['long']
                info = [lat, long]
                if prev_coords != []:
                    double_coords = [prev_coords, info]
                    list_bono.append(double_coords)
                prev_coords = info
            prev = vertex
        if bono:
            req_8(model, list_bono, 1)
    return total_distancy, total_vertex, path
```

Se utilizan varias variables y estructuras de datos para almacenar y manipular información. Después, se hace uso de un grafo, algoritmo de búsquedas en grafos y se utiliza un DFS para encontrar un camino entre dos vértices, donde calcula la distancia total recorrida a lo largo del camino encontrado en el grafo, sumando los pesos de los bordes entre los vértices adyacentes en el camino. En general, parece ser una función compleja que encuentra un camino en un grafo entre dos puntos dados, calcula la distancia total recorrida a lo largo de ese camino, y realiza alguna operación relacionada con coordenadas geográficas si se activa la bandera bono. Además, invoca la función req_8 con ciertos parámetros si se cumple cierta condición.

Entrada	- Estructuras de datos del modelo - Punto de origen (una localización geográfica con latitud y longitud). - Punto de destino (una localización geográfica con latitud y longitud).
Salidas	- Retorna la distancia total que tomará el camino entre el punto de origen y el de destino.

	- Retorna el total de vértices que contiene el camino encontrado. - Retorna la secuencia de vértices (sus identificadores) que componen el camino encontrado.
Implementado (Sí/No)	Sí. Implementado por Daniel Camilo Quimbay Velásquez

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Encontrar el vértice más cercano al punto de origen y destino.	O(V)
Ejecutar DFS en el grafo de distancia	O(V + E)
Encontrar el camino hasta el destino	O(E)
TOTAL	***O(V+E)***

Requerimiento 02

Descripción

```python
def req_2(model, lat_origin, long_origin, lat_dest, long_dest, bono):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    graph = model['graph_distance']
    vertex_origin = closest_vertice(model, lat_origin, long_origin)
    vertex_dest = closest_vertice(model, lat_dest, long_dest)
    model['search'] = bfs.BreathFirstSearch(graph, vertex_origin['id'])
    haspath = bfs.hasPathTo(model['search'], vertex_dest['id'])
    total_distancy = 0
    total_vertex = 0
    path = lt.newList('ARRAY_LIST')
    list_bono = []
    prev_coords = []
    if haspath:
        model['search'] = bfs.pathTo(model['search'], vertex_dest['id'])
        prev = None
        for vertex in lt.iterator(model['search']):
            total_vertex += 1
            lt.addLast(path, vertex)
            if prev is not None:
                edge = gr.getEdge(graph, vertex, prev)
                weight = edge['weight']
                total_distancy += weight
            if bono:
                v_info = me.getValue(mp.get(model['hashmap_vertex'], vertex))
                lat = v_info['lat']
                long = v_info['long']
                info = [lat, long]
                if prev_coords != []:
                    double_coords = [prev_coords, info]
                    list_bono.append(double_coords)
                prev_coords = info
            prev = vertex
        if bono:
            req_8(model, list_bono, 2)
    return total_distancy, total_vertex, path
```

Se utiliza un grafo y emplea algoritmos de búsqueda en grafos que sería un BFS. En este requerimiento calcula la distancia total recorrida a lo largo del camino encontrado en el grafo, manipulando coordenadas geográficas y almacenando información relevante en list_bono. En resumen, req_2 parece ser una variante de la función req_1 diseñada para resolver un requerimiento diferente utilizando un algoritmo de búsqueda en amplitud (BFS) en lugar de un algoritmo de búsqueda en profundidad (DFS), pero manteniendo una estructura y lógica de funcionamiento muy similar.

Entrada	- Estructuras de datos del modelo - Punto de origen (una localización geográfica con latitud y longitud). - Punto de destino (una localización geográfica con latitud y longitud).
Salidas	- Retorna la distancia total que tomará el camino entre el punto de origen y el de destino. - Retorna el total de vértices que contiene el camino encontrado. - Retorna la secuencia de vértices (sus identificadores) que componen el camino encontrado

Implementado (Sí/No)	Sí. Implementado por Daniel Camilo Quimbay Velásquez

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Encontrar el vértice más cercano al punto de origen y destino.	O(V)
Ejecutar BFS en el grafo de distancia	O(V + E)
Encontrar el camino hasta el destino	O(E)
TOTAL	***O(V+E)***

Requerimiento 03

Descripción

```python
def req_3(model, localidad, num_cam):
    """
    Función que soluciona el requerimiento 3
    """
    map_localidad = model["localidades"]
    parejas_localidad = mp.get(map_localidad, localidad)
    values_localidad = me.getValue(parejas_localidad)
    mapa_vertices = model["hashmap_vertex"]

    lista_vertices = lt.newList("ARRAY_LIST")
    for multa in lt.iterator(values_localidad):
        vertice = multa["VERTICES"]
        if not lt.isPresent(lista_vertices, vertice):
            lt.addLast(lista_vertices, vertice)

    lista_fees = lt.newList("ARRAY_LIST")
    mapa_vertices_info = mp.newMap(numelements=230000, maptype='PROBING', loadfactor=0.5, cmpfunction=compare_id)
    for vertice in lt.iterator(lista_vertices):
        num_fees = lt.size(me.getValue(mp.get(mapa_vertices, vertice))["fees"])
        info  = me.getValue(mp.get(mapa_vertices, vertice))
        lt.addLast(lista_fees, (vertice, num_fees))
        mp.put(mapa_vertices_info, vertice, info)

    merg.sort(lista_fees, cmp_fees)
    sub_n = lt.subList(lista_fees, 1, num_cam)
    origen = lt.getElement(sub_n, 1)

    dist = 0
    for i in range(1, lt.size(sub_n)):
        vertexA = lt.getElement(sub_n, i)[0]
        info_vertexA = me.getValue(mp.get(mapa_vertices_info, vertexA))
        lata = info_vertexA["lat"]
        longa = info_vertexA["long"]
        vertexB = lt.getElement(sub_n, i+1)[0]
        info_vertexB = me.getValue(mp.get(mapa_vertices_info, vertexB))
        latb = info_vertexB["lat"]
        longb = info_vertexB["long"]

        dist += calculate_distancy(lata, longa, latb, longb)

    return sub_n, dist
```

Para este requerimiento no se puede usar el grafo a la hora de sacar los vértices con mayor número de fees y es por esto que se utiliza el map que contiene todos los vértices agruppados por localidad, para ir

filtrando los datos de una vez. Una vez tenemos todos los vértices de la localidad, hacemos un sort para saber cúales son los top N con mayor número de multas. Se intentó usar Djkstra para determinar el camino más corto entre los vértices pero se demoraba mucho tiempo saber si contenía a todos los N vértices que se necesitaban en el camino, por lo que se optó directamente en tomar la distancia entre estos vértices e irla sumando.

Entrada	- Estructuras de datos del modelo - La cantidad de cámaras de video que se desean instalar (M). - La localidad donde se desean instalar.
Salidas	- Retorna el tiempo que se demora algoritmo en encontrar la solución (en milisegundos). - Retorna la siguiente información de la red de comunicaciones: 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. Implementado por Julián David Contreras Pinilla

Análisis de complejidad

Para este requerimiento utilizaremos los siguientes nombres: E(total de arcos en el grafo), V (total de vértices en el grafo), E' (arcos incluidos en el MST). E'' (arcos hasta el destino).

Pasos	**Complejidad**
Encontrar el vértice más cercano al punto de origen y destino.	O(V)
Ejecutar PrimMST del grafo de distancias	O(E + V(log(V)))
Reconstruir el grafo	O(E' + V)
Aplicar BFS para obtener los caminos a las M cámaras	O(E' + V)
Encontrar el camino a los M comparendos	O(M * E'')
TOTAL	***O(E + V(log(V)))***

Requerimiento 04

Descripción

```python
def req_4(model, camaras, bono):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4
    i = 2
    graph = model['graph_distance']
    sorted_list = model['ordered_fees']
    most_important = lt.firstElement(sorted_list)
    arcos = lt.newList('ARRAY_LIST')
    vertices = lt.newList('ARRAY_LIST')
    print('Cargando MST...')
    model['search'] = prim.PrimMST(graph, most_important['VERTICES'])
    weight = prim.weightMST(graph, model['search'])
    lst_bono = []
    cluster_bono = []
    infobono = {
        'lat': most_important['LATITUD'],
        'long': most_important['LONGITUD'],
        'info': most_important
    }
    cluster_bono.append(infobono)
    subgraph = gr.newGraph(datastructure='ADJ_LIST', directed=False, cmpfunction=compare_id)
    mst = model['search']['mst']
    print('Reconstruyendo grafo...')
    for minipath in lt.iterator(mst):
        add_vertex(subgraph, {'id': minipath['vertexA']})
        add_vertex(subgraph, {'id': minipath['vertexB']})
        add_edge(subgraph, minipath['vertexA'], minipath['vertexB'], minipath['weight'])
    subdjk = bfs.BreathFirstSearch(subgraph, most_important['VERTICES'])
    graph_cameras = gr.newGraph(datastructure='ADJ_LIST', directed=False, cmpfunction=compare_id)
    print('Filtrando las M cámaras')
    while i <= camaras:
        info_v = lt.getElement(sorted_list, i)
        infobono = {
            'lat': info_v['LATITUD'],
            'long': info_v['LONGITUD'],
            'info': info_v
        }
        cluster_bono.append(infobono)
        pathTo = bfs.pathTo(subdjk, info_v['VERTICES'])
        prev =None
```

```python
        for vertex in lt.iterator(pathTo):
            if prev == None:
                lt.addLast(vertices, vertex)
                prev = vertex
            else:
                minipath = {
                    'vertexA': prev,
                    'vertexB': vertex,
                    'weight': 0
                }
                prev = vertex
                v1 = minipath['vertexA']
                v2 = minipath['vertexB']
                v_info1 = me.getValue(mp.get(model['hashmap_vertex'], v1))
                v_info2 = me.getValue(mp.get(model['hashmap_vertex'], v2))
                lat1 = v_info1['lat']
                long1 = v_info1['long']
                lat2 = v_info2['lat']
                long2 = v_info2['long']
                minipath['weight'] = calculate_distancy(lat1, long1, lat2,  long2)
                if not gr.getEdge(graph_cameras, minipath['vertexA'], minipath['vertexB']):
                    lt.addLast(arcos, minipath)
                    weight += minipath['weight']
                if not gr.containsVertex(graph_cameras, minipath['vertexA']):
                    lt.addLast(vertices, minipath['vertexA'])
                if not gr.containsVertex(graph_cameras, minipath['vertexB']):
                    lt.addLast(vertices, minipath['vertexB'])
                add_vertex(graph_cameras, {'id': minipath['vertexA']})
                add_vertex(graph_cameras, {'id': minipath['vertexB']})
                add_edge(graph_cameras, minipath['vertexA'], minipath['vertexB'], minipath['weight'])
                if bono:
                    lst_bono.append([[lat1, long1], [lat2, long2]])
        i += 1
    print('Cálculos finales...')
    if bono:
        req_8(model, lst_bono, 4, cluster_bono)
    return vertices, arcos, weight
```

Con un grafo cuyo peso es la distancia entre los vértices se ejecuta un recorrido MST con Prim. Después se reconstruye el grafo para poder tener el recorrido entre las M cámaras que se quieren instalar. Esto se logró convirtiendo el grafo a un recorrido con BFS (que retornará el mismo grafo pero permitirá usar la función PathTo). Con estos recorridos se logrará reducir el árbol solo a las M cámaras que desea el usuario.

Entrada	- Estructuras de datos del modelo - La cantidad de cámaras de video que se desean instalar (M).
Salidas	- Retorna el tiempo que se demora algoritmo en encontrar la solución (en milisegundos). - Retorna la siguiente información de la red de comunicaciones.
Implementado (Sí/No)	Sí. Implementado por Daniel Camilo Quimbay Velásquez

Análisis de complejidad

Para este requerimiento utilizaremos los siguientes nombres: E(total de arcos en el grafo), V (total de vértices en el grafo), E' (arcos incluidos en el MST). E'' (arcos hasta el destino).

Pasos	**Complejidad**
Encontrar el vértice más cercano al punto de origen y destino.	O(V)
Ejecutar PrimMST del grafo de distancias	O(E + V(log(V)))
Reconstruir el grafo	O(E' + V)
Aplicar BFS para obtener los caminos a las M cámaras	O(E' + V)
Encontrar el camino a los M comparendos	O(M * E'')
TOTAL	***O(E + V(log(V)))***

Requerimiento 06

Descripción

```python
def req_6(model, comparendos, bono):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6
    i = 1
    lst_bono = []
    cluster_bono = []
    lst_pathTo = lt.newList('ARRAY_LIST')
    while i <= comparendos:
        print('Obteniendo camino de comparendo #' + str(i))
        info_comparendo = lt.getElement(model['ordered_fees'], i)
        if bono:
            infobono = {
                'lat': info_comparendo['LATITUD'],
                'long': info_comparendo['LONGITUD'],
                'info': info_comparendo
            }
        cluster_bono.append(infobono)
        vertex_comparendo = info_comparendo['VERTICES']
        info_vertex_comparendo = me.getValue(mp.get(model['hashmap_vertex'], vertex_comparendo))
        closest_station = info_vertex_comparendo['closest_station']
        entry = mp.get(model['djk_stations'], closest_station['EPONOMBRE'])
        if entry:
            search = me.getValue(entry)
        else:
            if bono:
                infobono = {
                    'lat': closest_station['EPOLATITUD'],
                    'long': closest_station['EPOLONGITU'],
                    'info': closest_station
                }
                cluster_bono.append(infobono)
            subgraph = me.getValue(mp.get(model['graph_stations'], closest_station['EPONOMBRE']))
            search = djk.Dijkstra(subgraph, closest_station['VERTICES'])
            mp.put(model['djk_stations'], closest_station['EPONOMBRE'], search)
        pathTo = djk.pathTo(search, info_comparendo['VERTICES'])
```

```python
        info_path = {
            'station': closest_station['EPONOMBRE'],
            'fee': info_comparendo,
            'vertex_fee': vertex_comparendo,
            'total_vertex': 0,
            'identificadores': [],
            'arcos': [],
            'km': 0
        }
        lt.addLast(lst_pathTo, info_path)
        for minipath in lt.iterator(pathTo):
            info_path['total_vertex'] += 1
            if len(info_path['identificadores']) == 0:
                info_path['identificadores'].append(minipath['vertexA'])
            info_path['identificadores'].append(minipath['vertexB'])
            info_path['km'] += minipath['weight']
            info_path['arcos'].append(minipath)
            if bono:
                v1 = minipath['vertexA']
                v2 = minipath['vertexB']
                v_info1 = me.getValue(mp.get(model['hashmap_vertex'], v1))
                v_info2 = me.getValue(mp.get(model['hashmap_vertex'], v2))
                lat1 = v_info1['lat']
                long1 = v_info1['long']
                lat2 = v_info2['lat']
                long2 = v_info2['long']
                lst_bono.append([[lat1, long1], [lat2, long2]])
        i += 1
    if bono:
        req_8(model, lst_bono, 6, cluster_bono)
    return lst_pathTo
```

Primero se inicializa un contador i en 1, se crea una lista vacía lst_bono que posiblemente almacene información adicional para el bono, se crea otra lista vacía cluster_bono que parece ser utilizada para almacenar información específica relacionada con el bono y se inicializa una lista vacía lst_pathTo que contendrá información de los caminos. Después, se inicia un bucle while que itera sobre los comparendos, se accede a la información del comparendo correspondiente según el contador i, si la bandera bono está activada, se agrega información de ubicación geográfica del comparendo a cluster_bono. Luego, se identifica la estación de policía más cercana al comparendo. Si se encuentra información sobre esta estación en una estructura de datos previamente definida. De lo contrario, se realiza un cálculo para encontrar la ruta más corta hasta esta estación utilizando el algoritmo de Dijkstra. Luego, se calcula la ruta desde la estación de policía más cercana hasta el lugar del comparendo utilizando el algoritmo de Dijkstra y se crea un registro de información relacionada con la ruta obtenida, incluyendo la estación, el comparendo, los vértices involucrados, el total de vértices en la ruta, la distancia recorrida (en kilómetros) y los arcos que conforman la ruta. Finalmente, se devuelve la lista lst_pathTo que contiene información detallada de los caminos relacionados con los comparendos.

Entrada	- Estructuras de datos del modelo - La cantidad de comparendos que se desea responder (M). - La estación de policía más cercana al comparendo más grave.
Salidas	- Retorna el tiempo que se demora algoritmo en encontrar la solución (en milisegundos). - Retorna la siguiente información de cada uno de los caminos seleccionado
Implementado (Sí/No)	Sí. Implementado por Daniel Camilo Quimbay Velásquez

Análisis de complejidad

Para este requerimiento utilizaremos los siguientes nombres: E(total de arcos en el subgrafo), V (total de vértices en el subgrafo), N (cantidad de estaciones en la ciudad).

Pasos	**Complejidad**
Obtener subgrafo	O(1)
Ejecutar Dijkstra	O(E*log(V))
Dijkstra por cada estación diferente	O(N*E*log(V))
Encontrar el camino al comparendo	O(E)
TOTAL	***O(N*E*(log(V)))***

Requerimiento 07

Descripción

```python
def req_7(model, lat_origin, long_origin, lat_dest, long_dest, bono):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7
    graph = model['graph_fee']
    vertex_origin = closest_vertice(model, lat_origin, long_origin)
    vertex_dest = closest_vertice(model, lat_dest, long_dest)
    model['search'] = djk.Dijkstra(graph, vertex_origin['id'])
    print('Ya se completó la búsqueda con BellmanFord')
    print('Ciclos negativos:')
    haspath = djk.hasPathTo(model['search'], vertex_dest['id'])
    print('Camino', haspath)
    total_fees = 0
    total_distancy = 0
    total_vertex = 0
    path = lt.newList('ARRAY_LIST')
    list_bono = []
    prev_coords = []
    if total_distancy==0:
        pathTo = djk.pathTo(model['search'], vertex_dest['id'])
        if pathTo == None:
            print('NONE MAMAWEBO')
        prev = None
        for vertex in lt.iterator(pathTo):
            total_vertex += 1
            if prev == None:
                lt.addLast(path, vertex['vertexA'])
            lt.addLast(path, vertex['vertexB'])
            total_fees += vertex['weight']
            v1 = vertex['vertexA']
            v2 = vertex['vertexB']
            v_info1 = me.getValue(mp.get(model['hashmap_vertex'], v1))
            v_info2 = me.getValue(mp.get(model['hashmap_vertex'], v2))
            distance = calculate_distancy(v_info1['lat'], v_info1['long'], v_info2['lat'], v_info2['long'])
            total_distancy += distance
            if bono:
                lat1 = v_info1['lat']
                long1 = v_info1['long']
                lat2 = v_info2['lat']
                long2 = v_info2['long']
```

```python
                list_bono.append([[lat1, long1], [lat2, long2]])
            prev = vertex
        if bono:
            req_8(model, list_bono, 7)
    else:
        print('No hay camino')
    return total_distancy, total_vertex, path
```

Primero se accede a un grafo graph desde el modelo proporcionado y encuentra los vértices más cercanos a las ubicaciones de origen y destino. Luego, hace la búsqueda del Camino más Corto donde se utiliza el algoritmo de Dijkstra para encontrar el camino más corto desde el vértice de origen hasta el vértice de destino en el grafo y se imprime mensajes sobre la finalización de la búsqueda y verifica si hay ciclos negativos. Después, se calcula la distancia total y la suma de las tarifas a lo largo del camino encontrado. Para cada vértice en el camino, calcula la distancia entre vértices adyacentes y suma las tarifas de los arcos. Si la bandera bono está activada, almacena información de coordenadas geográficas en list_bono. A continuación, se maneja el escenario donde no hay un camino válido entre los vértices de origen y destino, imprimiendo un mensaje indicando la ausencia de un camino. Si la bandera bono

está activada y se ha encontrado un camino válido, llama a la función req_8 con ciertos parámetros. Por último, devuelve la distancia total recorrida, el número total de vértices en el camino y el camino encontrado.

Entrada	- Estructuras de datos del modelo - Punto de origen (una localización geográfica con latitud y longitud). - Punto de destino (una localización geográfica con latitud y longitud).
Salidas	- Retorna la siguiente información del camino seleccionado: o El total de vértices del camino. o Los vértices incluidos (identificadores). o Los arcos incluidos (Id vértice inicial e Id vértice final). o La cantidad de comparendos del camino. o La cantidad de kilómetros del camino.
Implementado (Sí/No)	Sí. Implementado por Daniel Camilo Quimbay Velásquez

Análisis de complejidad

Para este requerimiento utilizaremos los siguientes nombres: E(total de arcos en el grafo), V (total de vértices en el grafo), E' (Arcos hasta el destino).

Pasos	**Complejidad**
Encontrar el vértice más cercano	O(V)
Ejecutar Dijkstra	O(E*log(V))
Recorrer el camino	O(E')
TOTAL	***O(E*log(V))***

Requerimiento 08 BONO

Descripción

```python
def req_8(data_structs, list, req, cluster=None):
    """
    Función que soluciona el requerimiento 8
    """
    # TODO: Realizar el requerimiento 8
    mapObj = folium.Map()
    str_name = 'Req ' + str(req) + ' Map.html'
    if req in [1, 2, 4, 6, 7]:
        for pair in list:
            folium.PolyLine(
                locations=pair,
                color='blue',
                opacity=0.8
            ).add_to(mapObj)
    if cluster != None:
        mCluster = MarkerCluster(name='Markers').add_to(mapObj)
        for info in cluster:
            lat = info['lat']
            long = info['long']
            folium.Marker(location=[lat, long], popup=info['info']).add_to(mCluster)
    folium.LayerControl().add_to(mapObj)
    mapObj.save(str_name)
```

Visualizar gráficamente en un mapa interactivo TODOS los requerimientos previamente implementados

Análisis de complejidad

Pasos	**Complejidad**
Añadir los E arcos al mapa	O(E)
Añadir los P puntos al Cluster	O(P)
TOTAL	***O(E + P)***