ANÁLISIS DEL RETO 4

Andres Chaparro Diaz, 202111146, a.chaparrod

Edward Camilo Sánchez Novoa, 202113020, e.sanchezn

Juan Esteban Rojas, 202124797, je.rojasc1

Carga de Datos

Árbol de árboles:

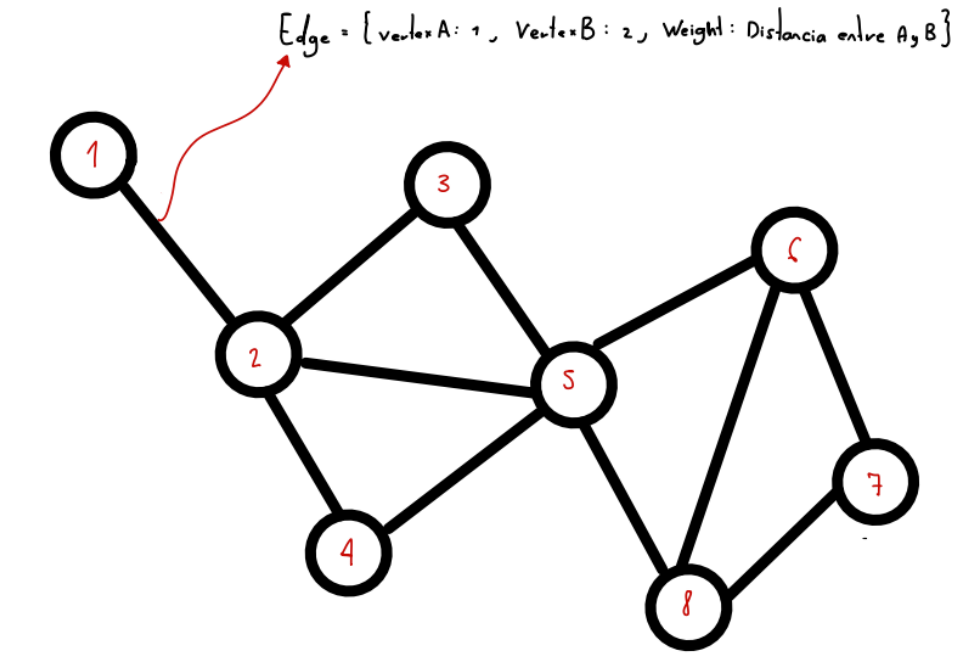
```
def add_vertice_Tree(control, vertice):

    lista = vertice.split(',')
    long = (lista[1])
    lat = (lista[2])
    long = replace(long)
    lat = replace(lat)
    lat = float(lat)
    long = float(long)
    lat = round(lat, 3)
    long = round(long, 1)
    updateDateIndex(control['latTree'], vertice, lat, long)

def updateDateIndex(map, vertice, lat, long):


    entry = om.get(map, lat)
    if entry is None:
        datentry = newDataEntry(vertice)
        om.put(map, lat, datentry)
    else:
        datentry = me.getValue(entry)
    addDateIndex(datentry, vertice, long)

    return map
```

```
def addDateIndex(datentry, vertice, long):

    lst = datentry["lstvertices"]
    lt.addLast(lst, vertice)
    nstIndex = datentry["longIndex"]
    offentry = om.get(nstIndex, long)
    if (offentry is None):
        entry = newNstEntry(long, vertice)
        lt.addLast(entry["lstvertices"], vertice)
        om.put(nstIndex, long , entry)
    else:
        entry = me.getValue(offentry)
        lt.addLast(entry["lstvertices"], vertice)
    return datentry

def newNstEntry(long, vertice):

    ofentry = {"long": None, "lstvertces": None}
    ofentry["long"] = long
    ofentry["lstvertices"] = lt.newList("ARRAY_LIST")

    return ofentry


def newDataEntry(vertice):

    entry = {}
    entry["longIndex"] = om.newMap(omaptype='RBT',
                                   cmpfunction= compareLatLong)

    entry["lstvertices"] = lt.newList("SINGLE_LINKED", compareLatLong)

    return entry
```

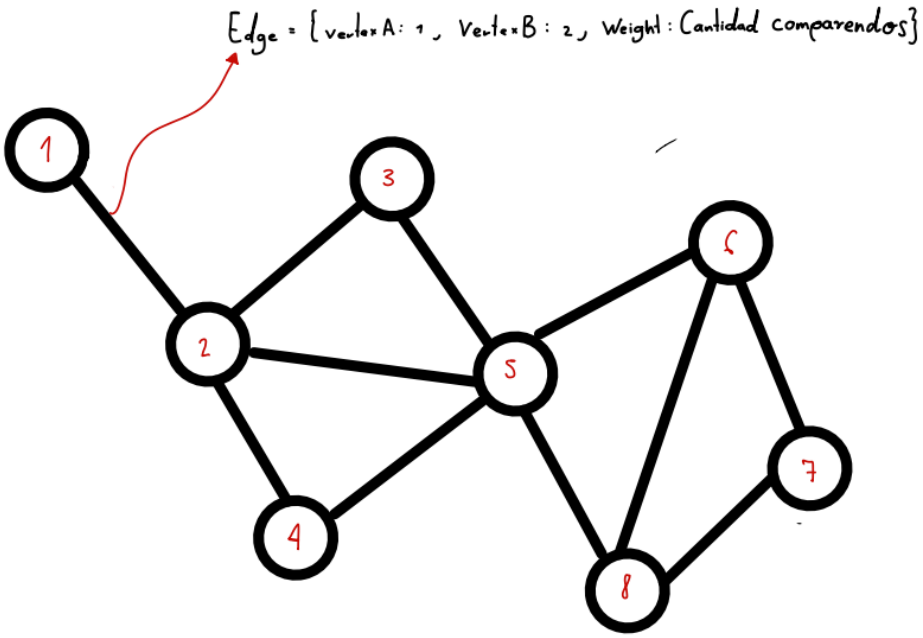
```python
def add_comparendo_tree(model, comparendo):

    arbol = model['latTree']
    lat_comparendo = comparendo['LATITUD']
    long_comparendo = comparendo['LONGITUD']
    lat_comparendo = float(lat_comparendo)
    long_comparendo = float(long_comparendo)
    lat_comparendo = round(lat_comparendo, 3)
    long_comparendo = round(long_comparendo, 1)

    entry = om.get(arbol, lat_comparendo)
    arbollong = me.getValue(entry)['longIndex']
    mayor = om.maxKey(arbollong)
    menor = om.minKey(arbollong)
    if long_comparendo > mayor:
        long_comparendo = mayor
    if long_comparendo < menor:
        long_comparendo = menor
    entry2 = om.get(arbollong, long_comparendo)
    if entry2 == None:
        print('no hay')
        print(om.keySet(arbollong))
        print(lat_comparendo)
        print(long_comparendo)
    lstvertices = me.getValue(entry2)['lstvertices']
    min_distance = 100000000
    for vertice in lt.iterator(lstvertices):
        vertice1 = vertice.split(',')
        distance = haversine(float(comparendo['LATITUD']), float(comparendo['LONGITUD']), float(vertice1[2]), float(vertice1[1]))
        if distance < min_distance:
            min_distance = distance
            nearest_vertex = vertice1[0]
    llave = mp.get(model['Malla_Vial'], nearest_vertex)
    valor = me.getValue(llave)
    lt.addLast(valor['comparendos'], comparendo)
    lt.addLast(model['Comparendos'], comparendo)
```

```python
def load_data(control):
    """
    Carga los datos del reto
    """
    filename_vertices = "tickets/bogota_vertices.txt"
    file=cf.data_dir+filename_vertices
    long_max = -100
    long_min = 100
    lat_max =  -100
    lat_min = 100
    with open(file, 'r') as archivo:
        for vertice in archivo:
            model.add_vertex(control['model'], vertice)
            model.add_vertice_Tree(control['model'], vertice)
            lista = vertice.split(',')
            long = float(lista[1])
            lat = float(lista[2])
            if long > long_max:
                long_max = long
            if long < long_min:
                long_min = long
            if lat > lat_max:
                lat_max = lat
            if lat < lat_min:
                lat_min = lat
    i = 0
    start_time = get_time()
    filename_vertices = "tickets/Comparendos_2019_Bogota_D_C.geojson"
    file=cf.data_dir+filename_vertices
    with open(file, encoding="utf-8") as archivo_json:
        datos = json.load(archivo_json)
        for dict in datos["features"]:
            comparendo = dict["properties"]
            i += 1
            model.add_comparendo_tree(control['model'], comparendo)
    model.sort(control['model'])
    filename_vertices = "tickets/estacionpolicia.json"
    file=cf.data_dir+filename_vertices
    with open(file, encoding="utf-8") as archivo_json:
        datos = json.load(archivo_json)
        for dict in datos["features"]:
            estacion = dict["properties"]
            model.add_estacion(control['model'], estacion)
    filename_vertices = "tickets/bogota_arcos.txt"
    file=cf.data_dir+filename_vertices
    key = 0
    with open(file, 'r') as archivo:
        for lista_adj in archivo:
            key += 1
            if key > 2:
                model.add_arco(control['model'], lista_adj)

    return lat_max, lat_min, long_max, long_min
```

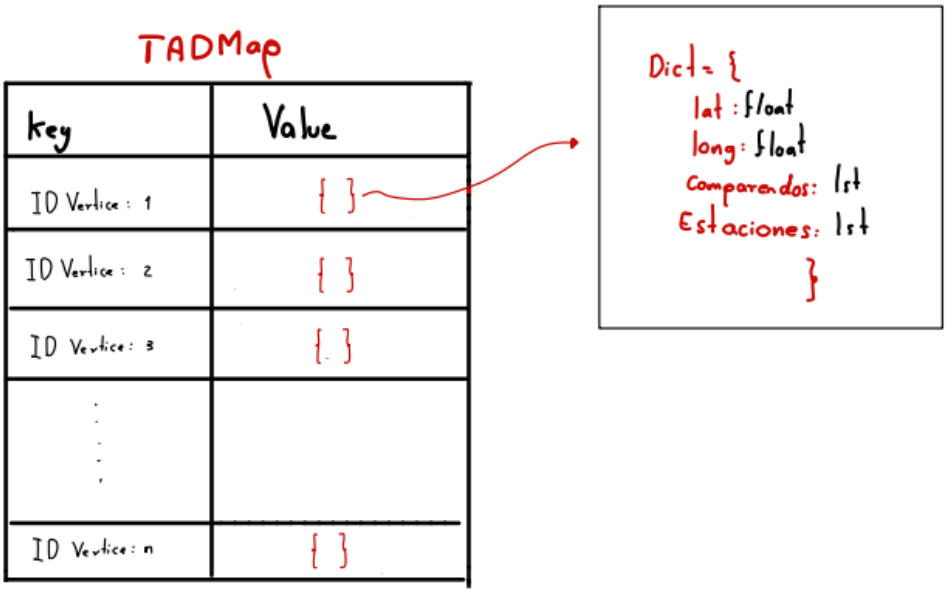
GRAFO_DISTANCIAS:



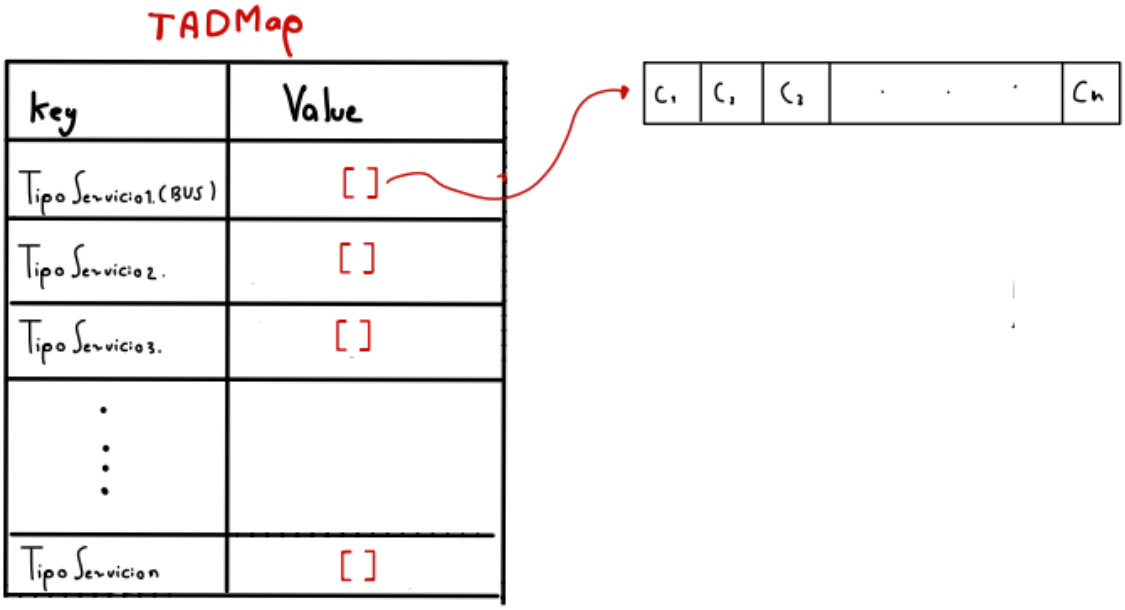
GRAFO COMPARENDOS:



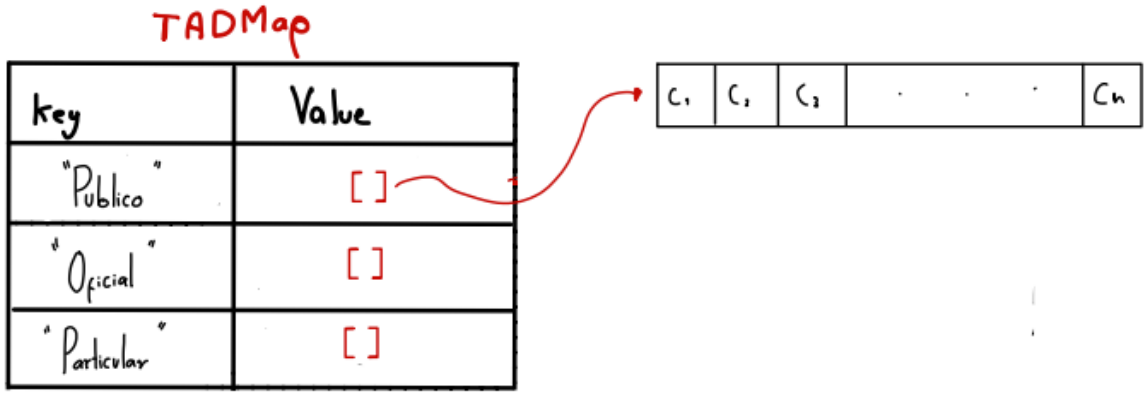
MAPA MALLA VIAL



MAPA TIPO SERVICO



MAPA GRAVEDADES



Requerimiento 1

Descripción

```python
def req_1(model, lat1, long1, lat2, long2)  :
    """
    Función que soluciona el requerimiento 1
    """
    Malla_Vial = model['Malla_Vial']
    Grafo_distancias = model['Grafo_distancias']
    vertices = model['vertices']
    lat1 = float(lat1)
    long1 = float(long1)
    lat2 = float(lat2)
    long2 = float(long2)
    distancia_minima_1 = 1000000
    distancia_minima_2 = 1000000
    vertice1 = None
    vertice2 = None
    vertices = mp.keySet(vertices)
    for vertice in lt.iterator(vertices):
        vertice_i = vertice.split(",")
        long_i = float(vertice_i[1])
        lat_i = float(vertice_i[2])
        distancia1 = haversine(lat1, long1, lat_i, long_i)
        distancia2 = haversine(lat2, long2, lat_i, long_i)
        if distancia1 < distancia_minima_1:
            distancia_minima_1 = distancia1
            vertice1 = vertice_i[0]
        if distancia2 < distancia_minima_2:
            distancia_minima_2 = distancia2
            vertice2 = vertice_i[0]

    search = bfs.BreathFirstSearch(Grafo_distancias, vertice1)

    if bfs.hasPathTo(search, vertice2):
        respuesta = bfs.pathTo(search, vertice2)

    lista = lt.newList(datastructure='ARRAY_LIST')
    for i in lt.iterator(respuesta):
        lt.addFirst(lista, i)
    i = 1
    peso = 0

    for vertice in lt.iterator(lista):
        if i < lt.size(lista):
            prox_vertice = lt.getElement(lista, i+1)
            peso = peso + float(gr.getEdge(Grafo_distancias, vertice, prox_vertice)['weight'])
        i += 1
    totalVertices = lt.size(respuesta)
    return respuesta, totalVertices, peso, vertice1, vertice2
```

Para el desarrollo del requerimiento 1, se utilizó el grafo de distancias y el mapa de “vértices”.

Para este requerimiento se itero sobre los valores del mapa de vértices para obtener los vértices más cercanos a los de la longitud y latitud ingresada como parámetro, posteriormente se usó el algoritmo de bfs usando como nodo inicial alguno de los dos nodos más cercanos a las dos latitudes y longitudes y se obtienen la cantidad de pasos que debe dar para llegar de un nodo al otro y los arcos que utiliza para llegar

Entrada	Control, lat1, lat2, long1, long2
Salidas	Respuesta, totalVertices, peso, vertice1, vertice2
Implementado (Sí/No)	SI

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Iteración por cada vértice	O(v)
Algoritmo bfs	O(v+e)
TOTAL	***O (v+e)***

Requerimiento 2

Descripción

```python
def req_2(data_structs, lat_inicio, long_inicio, lat_destino, long_destino):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    grafo = data_structs["Grafo_distancias"]
    vertices = data_structs['vertices']
    vertices = mp.keySet(vertices)
    distancia_minima_1 = 100000000
    distancia_minima_2 = 100000000
    for vertice in lt.iterator(vertices):
        vertice_i = vertice.split(",")
        long_i = float(vertice_i[1])
        lat_i = float(vertice_i[2])
        distancia1 = haversine(lat_inicio, long_inicio, lat_i, long_i)
        distancia2 = haversine(lat_destino, long_destino, lat_i, long_i)
        if distancia1 < distancia_minima_1:
            distancia_minima_1 = distancia1
            vertice_inicio = vertice_i[0]
        if distancia2 < distancia_minima_2:
            distancia_minima_2 = distancia2
            vertice_final = vertice_i[0]
    busqueda = bfs.BreathFirstSearch(grafo, vertice_inicio)
    if bfs.hasPathTo(busqueda,vertice_final):
        pasos = bfs.pathTo(busqueda,vertice_final)

    lista = lt.newList(datastructure='ARRAY_LIST')
    for i in lt.iterator(pasos):
        lt.addFirst(lista, i)
    i = 1
    peso = 0
    for vertice in lt.iterator(lista):
        if i < lt.size(lista):
            prox_vertice = lt.getElement(lista, i+1)
            peso = peso + float(gr.getEdge(grafo, vertice, prox_vertice)['weight'])
        i += 1
    return peso,lt.size(pasos),lista["elements"]
```

Se busca encontrar el camino más corto entre dos coordenadas (según el menor número de intersecciones), primero se hayan los vértices más cercanos a las coordenadas proporcionadas y uno se realiza una búsqueda del tipo bst ya que usando este método encontraremos como la relación más corta en termino de arcos del vértice de origen con el resto de los vértices.

Entrada	data_structs, lat_inicio, long_inicio, lat_destino, long_destino
Salidas	peso,lt.size(pasos) , lista["elements"]
Implementado (Sí/No)	SI

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Búsqueda de vértices cercanos	O(V)
Búsqueda BST	O (E + V)
Cálculo de la distancia	O(V)
TOTAL	***O (E + V)***

Requerimiento 3

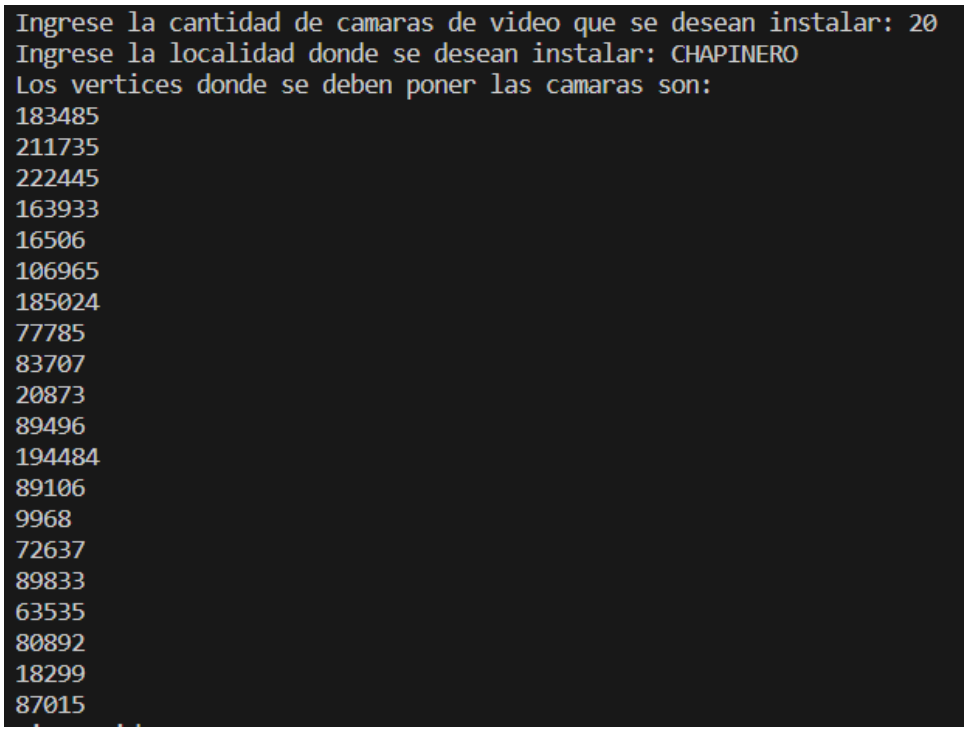
Descripción

```python
def req_3(model, M, localidad):
    """
    Función que soluciona el requerimiento 1
    """
    vertices_localidad = lt.newList(datastructure='ARRAY_LIST')
    Malla_Vial = model['Malla_Vial']
    Grafo_distancias = model['Grafo_distancias']
    lista_vertices = mp.valueSet(Malla_Vial)
    for vertice in lt.iterator(lista_vertices):
        if lt.size(vertice['comparendos']) > 0:
            dict = {}
            dict['comparendos'] = 0
            for comparendo in lt.iterator(vertice['comparendos']):
                if comparendo['LOCALIDAD'] == localidad:
                    dict['id'] = vertice['id']
                    dict['comparendos'] = dict['comparendos'] + 1
            if dict['comparendos'] > 0:
                lt.addLast(vertices_localidad, dict)

    vertices_localidad = sort_comparendos(vertices_localidad)

    nodo_origen = lt.getElement(vertices_localidad, 1)['id']
    search = prim.PrimMST(Grafo_distancias, nodo_origen)
    i = 1
    for vertice in lt.iterator(vertices_localidad):
        if 1 < i < int(M):
            try:
                arcos = prim.edgesMST(search, vertice['id'])
                print(arcos)
            except Exception:
                print('No hay camino')
        i += 1
    return vertices_localidad
```

Para el desarrollo del requerimiento 3, se utilizó el mapa grafo con arcos de las distancias entre los nodos. Asimismo se utilizó el mapa “MallaVial”

Para el desarrollo del requerimiento se itero sobre la lista de valores del mapa de malla vial que contiene todos los vértices y se filtraron los vértices de la localidad ingresada. Posteriormente, se ordeno esta lista de acuerdo a los que tuvieran mayores comparendos. Finalmente, el que tiene mas comparendos se uso como nodo origen para hacer un árbol prim en el cual se busca obtener las distancias y los arcos para llegar a los siguientes 19 vértices con más comparendos.

Entrada	control, M (numero de cámaras), Localidad
Salidas	Vértices donde colocar las cámaras
Implementado (Sí/No)	SI – Andres Felipe Chaparro



Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Iteración por vértice	O(v)
Ordenar por comparendos (Quicksort)	O(v(filtrados)logv(filtrados))
Prim usando el nodo con mas comparendos como nodo inicial	O(E log V)
TOTAL	***O(E log V)***

Análisis

Teniendo en cuenta las pruebas realizadas y el análisis de complejidad, podemos concluir que el uso de estructuras de datos como mapas binarios ordenados mejoro significativamente la eficiencia de la función req_3. Por otro lado, el requerimiento tiene una complejidad logarítmica ya que al buscar los datos lo hace como si fuera una búsqueda binaria debido a su estructura y al recorrer estos datos no tiene que iterar todos para encontrarlos.

Requerimiento 4

Descripción

```python
def req_4(data_structs,M):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4
    grafo_distancias = data_structs["Grafo_distancias"]
    comparendos_graves= data_structs["Gravedad_comparendos"]
    lista_vertices_graves = lt.newList("ARRAY_LIST")
    dato_o = me.getValue(mp.get(comparendos_graves,"Oficial"))
    dato_o = quk.sort(dato_o,comparendos_iguales)
    for i in lt.iterator(dato_o):
        if lt.isPresent(lista_vertices_graves,i["VERTICES"])==0 and lt.size(lista_vertices_graves)<=int(M):
            lt.addLast(lista_vertices_graves,i["VERTICES"])
    if lt.size(lista_vertices_graves)<=int(M):
        dato_pu = me.getValue(mp.get(comparendos_graves,"Público"))
        dato_pu = quk.sort(dato_pu,comparendos_iguales)
        for i in lt.iterator(dato_pu):
            if lt.isPresent(lista_vertices_graves,i["VERTICES"])==0 and lt.size(lista_vertices_graves)<=int(M):
                lt.addLast(lista_vertices_graves,i["VERTICES"])

    if lt.size(lista_vertices_graves)<=int(M):
        dato_pa = me.getValue(mp.get(comparendos_graves,"Particular"))
        dato_pa = quk.sort(dato_pa,comparendos_iguales)
        for i in lt.iterator(dato_pa):
            if lt.isPresent(lista_vertices_graves,i["VERTICES"])==0 and lt.size(lista_vertices_graves)<=int(M):
                lt.addLast(lista_vertices_graves,i["VERTICES"])
```

```python
    grafo_distancia_optima = djk.Dijkstra(grafo_distancias,lt.getElement(lista_vertices_graves,1))
    lt.deleteElement(lista_vertices_graves,1)
    kilometros = 0
    vertices_red = lt.newList("ARRAY_LIST")
    arcos_red = lt.newList("ARRAY_LIST")
    for i in lt.iterator(lista_vertices_graves):
        kilometros += djk.distTo(grafo_distancia_optima, i)
        camino = djk.pathTo(grafo_distancia_optima, i)
        for o in lt.iterator(camino):
            verticeA = o['vertexA']
            verticeB = o['vertexB']
            if lt.isPresent(vertices_red, verticeA) == 0:
                lt.addLast(vertices_red, verticeA)
            if lt.isPresent(vertices_red, verticeB) == 0:
                lt.addLast(vertices_red, verticeB)
            opcion_1 = str(verticeA) + '-' + str(verticeB)
            if lt.isPresent(arcos_red, opcion_1) == 0:
                lt.addLast(arcos_red, opcion_1)
    costo = 1000000*kilometros
    return lt.size(vertices_red),vertices_red["elements"],kilometros,costo
```

En este requerimiento nos solicitan poner cámaras clave según diversos vértices que tengan comparendos muy graves. El algoritmo toma los datos de los comparendos según sus 3 clasificaciones: oficial, publico y particular. De ser necesario los organiza uno a uno según el criterio respectivo a el tipo de infracción, de esta lista solo toma los primeros M vértices de interés correspondientes a las M cámaras que se desean instalar. A continuación, se implementa el algoritmo de Dijkstra partiendo desde el vértice con el peor comparendo, desde este grafo de costo mínimo se calcula la cantidad de kilómetros de fibra óptica, los arcos usados, los vértices presentes, etc.

Entrada	data_structs , M
Salidas	lt.size(vertices_red) , vertices_red["elements"] , kilometros , costo
Implementado (Sí/No)	SI – Esteban

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**

En el peor caso se organizan las 3 listas respectivas a la clasificación del comparendo	O (V log V)
Implementación del algoritmo de Dijkstra	O(V^2)
Recorrido del camino mínimo y cálculos menores de manera simultanea	O(V)
TOTAL	***O(V^2)***

Requerimiento 5

Descripción

```python
def req_5(data_structs, m, clase):

    """
    Función que soluciona el requerimiento 5
    """
    # TODO: Realizar el requerimiento 5

    # obtencion vertices con mayor numero de comparendos
    MapClases = data_structs['MapComparendos_clases']
    entry = mp.get(MapClases, clase)
    listaComparendos = me.getValue(entry)['datos']

    mapVertices = mp.newMap(numelements=228046, maptype='PROBING', cmpfunction=compare)
    for comparendo in lt.iterator(listaComparendos):
        vertice = comparendo['VERTICES']
        vertice_cercano = mp.get(mapVertices, vertice)
        if vertice_cercano == None:
            entry = entryREQ5(vertice)
            mp.put(mapVertices, vertice, entry)
        else:
            entry = me.getValue(vertice_cercano)

        lt.addLast(entry['comparendos'], comparendo)
        entry['Tcomparendos'] += 1
    lstEntrys = lt.newList(datastructure='ARRAY_LIST')
    listaVertices = mp.keySet(mapVertices)

    for vertice in lt.iterator(listaVertices):
        entry = me.getValue(mp.get(mapVertices, vertice))
        lt.addLast(lstEntrys, entry)

    sortedlst = quk.sort(lstEntrys, sort_criteriaREQ5)

    SubListaVertices = lt.subList(sortedlst, 1,int(m))
```

```
origen = lt.getElement(SubListaVertices, 1)['vertice']
search = djk.Dijkstra(data_structs['Grafo_distancias'], origen)
distancia_total = 0
sublistaSinOrigen = lt.subList(SubListaVertices, 2, lt.size(SubListaVertices) - 1)

listaVerticesRed = lt.newList('ARRAY_LIST')
listaArcosRed = lt.newList('ARRAY_LIST')

for vertice in lt.iterator(sublistaSinOrigen):
        verticeD = vertice['vertice']
        comparendos = vertice['comparendos']
        Tcomparendos = vertice['Tcomparendos']
        distancia = djk.distTo(search, verticeD)
        distancia_total += distancia
        camino = djk.pathTo(search, verticeD)
        for verticeID in lt.iterator(camino):
            verticeA = verticeID['vertexA']
            verticeB = verticeID['vertexB']
            if lt.isPresent(listaVerticesRed, verticeA) == 0:
                lt.addLast(listaVerticesRed, verticeA)
            if lt.isPresent(listaVerticesRed, verticeB) == 0:
                lt.addLast(listaVerticesRed, verticeB)
            aeco = str(verticeA) + '-' + str(verticeB)
            if lt.isPresent(listaArcosRed, aeco) == 0:
                lt.addLast(listaArcosRed, aeco)

precio = 1000000 * distancia_total
formato ="${:,.2f}".format(float(precio))
return listaVerticesRed, listaArcosRed, distancia_total, formato, SubListaVertices
```

Para el desarrollo del requerimiento 5, se utilizó un grafo que tiene como vértices las intersecciones de una ciudad, y como arcos las calles, que tienen como peso la distancia entre ambos vértices conectados. Además, se utilizó un mapa que tiene como llaves los diferentes tipos de vehículos, y como valor una lista con los comparendos registrados a este tipo de vehículos.

Para empezar, se obtuvo una lista de comparendos de vehículos de la clase solicitada por el usuario, utilizando esta lista se construyó un nuevo mapa que tiene como llave cada vértice, y como valor los comparendos que ocurrieron cerca de este. Con la ayuda de este mapa se encontraron los vértices con mayor número de comparendos, y que por lo tanto serán en los que se instalen las cámaras. Para hallar el presupuesto y la longitud de fibra óptica se utilizó el algoritmo Dijkstra que calculo la manera más eficiente de conectar estos vértices.

Entrada	data_structs, m, clase
Salidas	listaVerticesRed, listaArcosRed, distancia_total, formato, SubListaVertices
Implementado (Sí/No)	SI – Camilo Sánchez Novoa

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Encontrar los vértices con más comparendos por clase de vehículo.	O (N log N)

Implementación search algoritmo Dijkstra	O (E log V)
Recorrido caminos entre vertices	O(V)
TOTAL	**O (E log V)**

Análisis

En Cuanto al análisis de este requerimiento, concluimos que, a pesar de que se llegó a la respuesta solicitada, y la función cumple con su objetivo. Se tiene un tiempo de ejecución elevado ya que debido al tamaño del grafo se requiere mayor tiempo para la ejecución del algoritmo Dijkstra.

Respuesta en terminal:

```
Ingrese el numero de comparendos: 5
Ingrese la clase de vehiculo: CAMIONETA
los vertices que contaran con camaras som:

163933
183485
227153
184004
101432

la longitud total de fibra optica es:23.963380138372084 [km]
la cantidad de dinero para la implementacion de la red es:$23,963,380.14 [COP]
Los verticen conectados por la red son:


{'elements': ['73554', '183485', '183484', '183483', '13418', '13417', '13416', '10949', '10948', '10947', '17569', '17568', '183477', '10946', '10945', '22327', '
22326', '10955', '10954', '10951', '10950', '23530', '23529', '21802', '21801', '30761', '30760', '22407', '22406', '6358', '6357', '6356', '6355', '6352', '6351',
 '6350', '6349', '20248', '20247', '13017', '13016', '18305', '13423', '13422', '13421', '13420', '13419', '20271', '12999', '214730', '13390', '13389', '13388', '
13387', '13385', '13384', '13383', '23423', '23422', '18593', '18592', '18591', '18590', '218679', '218678', '18585', '18584', '18583', '22013', '22012', '22011',
 '22010', '17022', '17021', '17020', '17019', '37587', '37586', '218666', '19529', '19528', '29638', '21121', '21120', '21993', '6346', '6345', '6341', '6340', '137
95', '13796', '9914', '89124', '88181', '88180', '63360', '63680', '84500', '77216', '77215', '58632', '58633', '58634', '58635', '58636', '10901', '20275', '20276
', '26789', '108606', '95932', '103458', '103459', '96382', '96383', '96384', '196512', '78192', '108588', '108587', '99444', '99443', '1020', '108591', '109376',
 '99714', '103250', '103249', '103248', '103247', '110466', '10936', '106570', '1023', '95815', '106891', '175244', '10297', '10296', '10293', '21326', '100284', '1
69810', '151303', '105129', '105128', '91738', '91739', '91073', '91074', '91075', '99364', '99365', '162645', '91136', '108489', '110450', '110451', '95165', '971
04', '220221', '220222', '18958', '18957', '18956', '18955', '18954', '18953', '43604', '43603', '43602', '22599', '22598', '22597', '19257', '19256', '19255', '19
254', '19253', '19252', '217721', '217720', '217719', '218145', '218144', '24551', '24550', '24549', '24548', '27512', '27511', '24555', '24554', '24553', '24552',
 '27052', '27051', '27050', '27049', '27048', '27047', '27046', '27045', '24686', '32350', '32349', '195211', '20150', '20149', '20148', '20147', '20146', '20145',
 '163934', '163933', '227154', '227153', '87006', '95240', '95241', '95242', '199586', '199585', '26833', '26834', '170302', '19671', '40442', '53754', '19674', '2
2190', '22191', '19365', '177645', '155612', '155613', '155557', '174349', '184003', '184004', '51333', '184019', '184018', '184017', '184022', '184021', '184020',
 '183977', '183976', '183975', '183974', '183973', '56486', '183968', '183967', '215725', '180580', '180579', '180578', '139278', '184526', '184525', '186492', '18
6469', '186468', '186467', '133589', '186458', '186457', '186456', '186455', '186454', '186453', '8516', '11604', '73958', '188558', '188557', '166719', '182191',
 '166710', '215402', '95003', '184641', '184640', '176075', '176074', '176073', '134504', '213689', '176079', '176078', '176077', '176076', '71752', '213175', '1819
65', '181964', '181963', '181962', '181961', '181960', '186154', '186153', '13688', '177535', '177534', '13725', '19737', '187051', '167748', '217913', '225963', '
94638', '186150', '155188', '213128', '191349', '193580', '172488', '172487', '172486', '172485', '172484', '172483', '171420', '171419', '73732', '227478', '18057
7', '180576', '180575', '180574', '196824', '196823', '196822', '171418', '171417', '171416', '30843', '30845', '171415', '171414', '173084', '31825', '172478', '1
72477', '172476', '172475', '172481', '172480', '172479', '22227', '183887', '171413', '171412', '11562', '11555', '171411', '171410', '171409', '172482', '186919'
, '186918', '180597', '222192', '222191', '131351', '178381', '132792', '45656', '186725', '186724', '186723', '186722', '81543', '186726', '186917', '186916', '13
3946', '7838', '7837', '7836', '7835', '17793', '17792', '17789', '12341', '12340', '12336', '29932', '17777', '14615', '14614', '14613', '216591', '216590', '4640
', '9906', '23642', '21588', '21587', '21589', '17695', '17694', '22952', '26410', '26409', '18131', '26418', '28080', '28079', '22008', '22007', '20594', '20593',
 '18970', '21144', '21143', '25168', '25165', '32321', '18143', '18142', '35123', '11008', '11007', '22551', '22550', '18972', '11107', '11106', '21750', '11108',
'21524', '46565', '13863', '189859', '18589', '18588', '18933', '18973', '29319', '21844', '13012', '13006', '20269', '211378', '195354', '195353', '195352', '1953
51', '124020', '22196', '21704', '22200', '22199', '22202', '22201', '19087', '217781', '21760', '16505', '10312', '41750', '41749', '213367', '213366', '213365',
'213364', '215399', '18195', '18194', '97105', '101432', '97106', '97107', '111218', '111512', '111513', '111514', '29038', '217956', '217955', '217954', '31593',
 '31592', '31591', '31590', '31589', '208528', '208527', '208526', '208525', '208524', '208523', '31653', '25057', '87836', '87837', '31527', '192323', '31588', '31
587', '31586', '213856', '18188', '167191', '202639', '160182'], 'size': 510, 'type': 'ARRAY_LIST', 'cmpfunction': <function defaultfunction at 0x0000028259CCBF78>
```

Requerimiento 6

Descripción

```python
def req_6(data_structs, m):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6
    solucion = lt.newList(datastructure='ARRAY_LIST')
    lista_comparendos  = data_structs['Comparendos']
    lista_comisarias = data_structs['Estaciones de policia']
    grafo_distancias = data_structs['Grafo_distancias']
    sorted_comaprendos = sa.sort(lista_comparendos, sort_criteriaREQ6)
    seleccionComparendos = lt.subList(sorted_comaprendos, 1, int(m))
    mapaaux = mp.newMap(numelements=100, maptype='PROBING', cmpfunction=compa

    for comparendo in lt.iterator(seleccionComparendos):
        estacionCercana = comisaria_cercanas(data_structs, comparendo)
        if not(mp.contains(mapaaux, estacionCercana['VERTICES'])):
            entry = entryREQ6(estacionCercana)
            mp.put(mapaaux, estacionCercana['VERTICES'], entry)
        else:
            entry = me.getValue(mp.get(mapaaux, estacionCercana['VERTICES']))
        lt.addLast(entry['comparendos'], comparendo)

    listaEstaciones = mp.keySet(mapaaux)
    for estacion in lt.iterator(listaEstaciones):
        search = djk.Dijkstra(grafo_distancias, estacion)
        entry = mp.get(mapaaux, estacion)
        listaComp = me.getValue(entry)['comparendos']
        for comparendo in lt.iterator(listaComp):
            formato = {}
            vertice = comparendo['VERTICES']
            camino = djk.pathTo(search, vertice)
            vertextotal = lt.size(camino)
            distancia = djk.distTo(search, vertice)
            formato['verticesTotales'] = vertextotal
            listaVerticesCamino = lt.newList('ARRAY_LIST')
            listaArcosCamino = lt.newList('ARRAY_LIST')
            for verticeID in lt.iterator(camino):
                verticeA = verticeID['vertexA']
                verticeB = verticeID['vertexB']
                if lt.isPresent(listaVerticesCamino, verticeA) == 0:
                    lt.addLast(listaVerticesCamino, verticeA)
                if lt.isPresent(listaVerticesCamino, verticeB) == 0:
                    lt.addLast(listaVerticesCamino, verticeB)
                aeco = str(verticeA) + '-' + str(verticeB)
                if lt.isPresent(listaArcosCamino, aeco) == 0:
                    lt.addLast(listaArcosCamino, aeco)
            formato['listaVerticesCamino'] = listaVerticesCamino
            formato['listaArcosCamino'] = listaArcosCamino
            formato['distancia'] = distancia
            formato['comparendo'] = comparendo
            lt.addLast(solucion, formato)
    return solucion
```

Para el desarrollo del requerimiento 5, se utilizó un grafo que tiene como vértices las intersecciones de una ciudad, y como arcos las calles, que tienen como peso la distancia entre ambos vértices conectados. Además, se utilizó una lista que contiene todos los comparendos.

Para empezar la solución, se realizó un sort a la lista de comparendos de tal manera que se priorizara el riesgo de este. De esta forma se obtuvo una sublista con los n comparendos más graves. Luego de esto se asoció cada comparendo con su comisaria más cercana utilizando un mapa del tipo:

‘Estacion’: [comparendo1, comarendo2 comparendo N]

Utilizando este mapa, se realizaron la menor cantidad posible de algoritmos Dijkstra con el fin de hallar el camino más corto y su respectiva distancia.

Entrada	data_structs, m
Salidas	solucion
Implementado (Sí/No)	SI

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Shell Sort	O (N Log N)
Implementación search algoritmo Dijkstra	O (E log V)
Recorridoo de comaprendos y clasificación	O (N*M)
TOTAL	O (E log V)

Análisis

El requerimiento 6 no resulto eficiente debido a que para su solución se necesitaban tantos Dijkstra como comparendos pidiera el usuario. A pesar de que se buscaron alternativas, no se encontró ninguna optima, ya que para reducir el tiempo de ejecución del algoritmo habría que reducir el tamaño del grafo y por lo tanto eliminar algunos vértices. Lo que podría significar eliminar una posible ruta más rápida. Sin embargo, se desarrolló el algoritmo de forma funcional retornando los valores requeridos.

Respuesta en terminal:

```
Ingrese el numero de comparendos: 5


ID comparendo: 20059646

El camino mas corto es:
{'elements': ['186920', '201310', '17513', '17514', '17515', '14059', '14060', '14061', '14062', '12810', '12811', '12812', '12813', '12814', '12815', '12816', '12817', '12818', '227945', '175237
', '175228', '175227', '175241', '175240', '175239', '175238', '87551', '87552', '62780', '62781', '93363', '93362', '202793', '135300', '74823', '74822', '201996'], 'size': 39, 'type': 'ARRAY_LIS
 <function defaultfunction at 0x00000223FB02BD38>, 'key': None, 'datastructure': <module 'DISClib.DataStructures.arraylist' from 'c:\\Users\\camil\\OneDrive - Universidad de los Andes\\Documentos\
S\\RETO 4\\Reto4-G09\\DISClib\\DataStructures\\arraylist.py'>}

La distancia total del camino es: 1.6441199306616372 [km]
```

```
ID comparendo: 4032379

El camino mas corto es:
{'elements': ['102374', '112715', '209187', '41702', '155848', '87219', '29374', '155849', '161400', '161399', '83006', '83005', '83004', '62773', '97349', '88862', '205720', '89003', '89002', '89
1194', '221195', '173557', '173558', '173559', '181884', '182442', '173462', '182452', '176224', '185128', '185127', '185126', '185125', '185124', '113042', '204782', '186999', '93109', '93110', '
204407', '204406', '220557', '54170', '221737', '77199', '77200', '77201', '207519', '94347'], 'size': 53, 'type': 'ARRAY_LIST', 'cmpfunction': <function defaultfunction at 0x00000223FB02BD38>, 'k
ructure': <module 'DISClib.DataStructures.arraylist' from 'c:\\Users\\camil\\OneDrive - Universidad de los Andes\\Documentos\\camilo\\EDA\\RETOS\\RETO 4\\Reto4-G09\\DISClib\\DataStructures\\arrayl

La distancia total del camino es: 1.3462051660212615 [km]
```

```
ID comparendo: 3944469

El camino mas corto es:
{'elements': ['13447', '13448', '8857', '8856', '138012', '138013', '136759', '147934', '137766', '137767', '137768', '137769', '140287', '140288', '140289', '135154', '135153', '126810', '126809
6', '141292', '211366', '211367', '211368', '211369', '141028', '183949', '183950', '183951', '15382', '183952', '93424', '93425', '93426', '100458', '87087', '87088', '87089', '107304', '107305',
 '95644', '143617', '146130'], 'size': 46, 'type': 'ARRAY_LIST', 'cmpfunction': <function defaultfunction at 0x00000223FB02BD38>, 'key': None, 'datastructure': <module 'DISClib.DataStructures.arra
sers\\camil\\OneDrive - Universidad de los Andes\\Documentos\\camilo\\EDA\\RETOS\\RETO 4\\Reto4-G09\\DISClib\\DataStructures\\arraylist.py'>}

La distancia total del camino es: 1.6535487706661574 [km]
```

```
ID comparendo: 20067205

El camino mas corto es:
{'elements': ['161120', '161121', '161119', '146586', '168982', '23269', '23274', '10163', '10164', '10165', '17427', '23275', '21310', '21308', '21309', '12483', '216293', '59749', '216127', '144
840', '19550', '19551', '16900', '16901', '17482', '23667', '222559', '222560', '222561', '222562', '222563', '222564', '222565', '183040', '28081', '28082', '28083', '28084', '28085', '19547', '1
8594', '188552', '188551', '188550', '188549', '188548', '136520', '136519', '136518'], 'size': 53, 'type': 'ARRAY_LIST', 'cmpfunction': <function defaultfunction at 0x00000223FB02BD38>, 'key': No
': <module 'DISClib.DataStructures.arraylist' from 'c:\\Users\\camil\\OneDrive - Universidad de los Andes\\Documentos\\camilo\\EDA\\RETOS\\RETO 4\\Reto4-G09\\DISClib\\DataStructures\\arraylist.py'

La distancia total del camino es: 2.0515388150926404 [km]
```

```
ID comparendo: 3901315

El camino mas corto es:
{'elements': ['174349', '163933', '155557', '155613', '155612', '177645', '19365', '22191', '22190', '19674', '19673', '19670', '19669', '24842', '24841', '23116', '23115', '23114', '26980', '2697
7', '25587', '31615', '28368', '89681', '89680', '89679', '89678', '89677', '175780'], 'size': 31, 'type': 'ARRAY_LIST', 'cmpfunction': <function defaultfunction at 0x00000223FB02BD38>, 'key': Non
: <module 'DISClib.DataStructures.arraylist' from 'c:\\Users\\camil\\OneDrive - Universidad de los Andes\\Documentos\\camilo\\EDA\\RETOS\\RETO 4\\Reto4-G09\\DISClib\\DataStructures\\arraylist.py'>

La distancia total del camino es: 1.0718742403323431 [km]
```

Requerimiento 7

Descripción

```python
def req_7(data_structs,lat_inicio, long_inicio, lat_destino, long_destino):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7
    grafo = data_structs["Grafo_distancias"]
    grafo_busqueda = data_structs["Grafo_comparendos"]
    vertices = data_structs['vertices']
    vertices = mp.keySet(vertices)
    distancia_minima_1 = 100000000
    distancia_minima_2 = 100000000
    for vertice in lt.iterator(vertices):
        vertice_i = vertice.split(",")
        long_i = float(vertice_i[1])
        lat_i = float(vertice_i[2])
        distancia1 = haversine(lat_inicio, long_inicio, lat_i, long_i)
        distancia2 = haversine(lat_destino, long_destino, lat_i, long_i)
        if distancia1 < distancia_minima_1:
            distancia_minima_1 = distancia1
            vertice_inicio = vertice_i[0]
        if distancia2 < distancia_minima_2:
            distancia_minima_2 = distancia2
            vertice_final = vertice_i[0]

    busqueda = djk.Dijkstra(grafo_busqueda, vertice_inicio)
    if djk.hasPathTo(busqueda,vertice_final):
        pasos = djk.pathTo(busqueda,vertice_final)

    vertices = lt.newList("ARRAY_LIST")
    arcos = lt.newList("ARRAY_LIST")
    cantidad_comparendos = djk.distTo(busqueda,vertice_final)
    for o in lt.iterator(pasos):
        verticeA = o['vertexA']
        verticeB = o['vertexB']
        if lt.isPresent(vertices, verticeA) == 0:
            lt.addLast(vertices, verticeA)
        if lt.isPresent(vertices, verticeB) == 0:
            lt.addLast(vertices, verticeB)
        opcion_1 = str(verticeA) + '-' + str(verticeB)
        if lt.isPresent(arcos, opcion_1) == 0:
            lt.addLast(arcos, opcion_1)
    total_vertices = lt.size(vertices)
    cantidad_kilometros = 0
```

```python

    for edge in lt.iterator(arcos):
        arco = gr.getEdge(grafo, edge.split("-")[0], edge.split("-")[1])
        distancia = arco["weight"]
        cantidad_kilometros += distancia

    return total_vertices,vertices["elements"],arcos["elements"],cantidad_comparendos,cantidad_kilometros
```

Se busca hallar el camino más corto entre dos coordenadas en términos de cantidad de comparendos, para esto se usó el grafo cuyo peso de arco era el número de comparendos. Como primer paso se calculan los vértices más cercanos a las coordenadas dadas y se continúa implementando el algoritmo de Dijkstra teniendo como punto base el vértice hallado basado en las coordenadas iniciales. Una vez se completa el grafo se toma el camino entre el vértice de origen y el de destino, en base a este se calcula la distancia en términos de comparendos, los kilómetros, se toman los arcos usados, se toman los vertices considerados, etc.

Entrada	data_structs,lat_inicio, long_inicio, lat_destino, long_destino
Salidas	total_vertices , vertices["elements"] , arcos["elements"] , cantidad_comparendos , cantidad_kilometros
Implementado (Sí/No)	SI

Análisis de complejidad

Pasos	**Complejidad**
Búsqueda de vértices cercanos a las coordenadas dadas	O (V)

Implementación del algoritmo de Dijkstra	O (V^2)
Cálculos de distancia, peso y pasos menores	O (V)
TOTAL	***O (V^2)***

Respuesta en terminal:

```
0- Salir
Seleccione una opción para continuar
8
Ingrese la latitud de inicio: 4.60293518548777
Ingrese la longitud de inicio: -74.06511801444837
Ingrese la latitud de destino: 4.693518613347496
Ingrese la longitud de destino: -74.13489678235523

El total de vertices fue de: 466

Los vertices son:
['186461', '186460', '186462', '186470', '181864', '181865', '141832', '135104', '135103', '181866', '223717', '223716', '223715', '134324', '134323', '1
34322', '134321', '134320', '134319', '186491', '186466', '186465', '186464', '186463', '186052', '186051', '186459', '186458', '186457', '186456', '1864
55', '186454', '186453', '8516', '8515', '8514', '217479', '4420', '11255', '11254', '129105', '129104', '202446', '103971', '128916', '128915', '134611'
, '130929', '130928', '72709', '61502', '61501', '76896', '176376', '176378', '176377', '177333', '222185', '176372', '83008', '68459', '100366', '100365
', '70010', '90052', '90051', '173173', '85293', '200168', '201506', '132410', '129504', '131149', '85661', '82475', '134108', '134109', '133007', '12358
5', '12055', '12054', '12053', '12052', '12051', '12050', '12163', '12162', '134387', '132129', '127767', '127766', '127765', '121578', '152015', '125860
', '199131', '134739', '126800', '158084', '158083', '125195', '125196', '125197', '13728', '32931', '219252', '219251', '219250', '18573', '18572', '185
71', '219249', '13916', '13735', '198704', '129508', '129507', '129506', '135669', '135670', '137463', '70847', '138520', '124771', '130228', '140520', '
94152', '83435', '83434', '133284', '131635', '223308', '223307', '60548', '60547', '131295', '14235', '34675', '53778', '53779', '57240', '57239', '5723
8', '57237', '57236', '57235', '57234', '57233', '57232', '57231', '57230', '57229', '22823', '21312', '20179', '20180', '13569', '10128', '10127', '1187
5', '11874', '11872', '6316', '6317', '43346', '48179', '54631', '54630', '183041', '183040', '23666', '183039', '183038', '183037', '183036', '183035',
'183034', '183033', '132848', '132847', '131666', '18788', '18787', '216363', '216364', '18786', '18789', '12324', '12323', '18336', '214829', '214828',
'214827', '214826', '214825', '21874', '4213', '5941', '12344', '38725', '40108', '36322', '36321', '34461', '34460', '34459', '34458', '199887', '199886
', '188334', '10299', '188338', '10301', '10300', '10302', '154227', '154226', '157807', '579', '580', '175639', '176487', '98252', '147552', '147551', '
187382', '106335', '171269', '166060', '97452', '31824', '17737', '19278', '19763', '18026', '27134', '227438', '227439', '227440', '227441', '227442', '
227443', '227444', '27132', '23799', '8829', '8828', '8827', '8826', '17979', '21113', '21112', '17977', '197498', '19103', '38856', '38855', '32587', '3
2586', '32585', '32584', '19349', '221036', '211378', '195354', '23087', '23086', '29881', '29882', '29883', '29884', '188300', '186759', '186758', '1867
57', '1954', '15301', '29953', '29952', '19289', '19288', '218150', '218149', '19352', '19351', '23050', '23049', '16858', '16857', '33680', '182228', '2
3098', '29888', '224720', '101794', '225957', '225956', '225955', '88910', '178541', '147376', '147377', '147378', '158604', '158605', '183733', '183063'
, '183062', '164684', '164293', '214933', '214932', '188986', '166693', '166692', '166691', '166690', '166689', '166688', '166687', '16897', '16896', '18
3681', '13612', '13611', '92700', '97916', '92805', '92804', '102431', '98464', '98463', '159820', '181055', '27957', '105044', '17998', '98927', '173653
', '170412', '25297', '164271', '164270', '164269', '161488', '164259', '164258', '183250', '183258', '183257', '176331', '176335', '176334', '215196', '
215195', '215194', '174136', '174135', '174134', '174133', '174132', '174131', '174140', '174139', '174138', '174137', '173466', '173465', '173464', '173
463', '173462', '182442', '176245', '182445', '175182', '176243', '182451', '182450', '182449', '182448', '182447', '182446', '176238', '176237', '176236
', '176235', '176234', '176233', '176232', '176242', '176241', '176240', '176239', '217843', '217842', '217841', '177247', '36696', '36695', '204970', '2
04969', '204968', '204967', '204966', '204965', '204964', '204963', '173353', '173352', '173357', '173356', '173355', '173354', '227530', '227529', '2275
28', '227527', '227526', '227525', '173340', '55461', '55460', '55459', '55458', '55457', '215747', '215746', '215745', '215744', '24979', '24978', '2497
7', '24976', '206025', '206024', '206023', '206022', '39945', '45155', '45154', '45153', '45152', '45151', '45150', '37068', '71653', '71652', '89165', '
207025', '207026', '207027', '207028', '207029', '207030', '30531', '30532', '30533', '30534', '32380', '32379', '45064', '25930', '222120', '41079', '14
8234', '148233']
```