ANÁLISIS DEL RETO 4

1. Rodrigo Paz Londoño <r.pazl@uniandes.edu.co>, 202225425

1. Juan Diego Rodríguez Barragán <jd.rodriguezb12@uniandes.edu.co>, 202221822

1. Samuel Escobar Pineda <sa.escobarp1@uniandes.edu.co>, 202310474

Carga de datos

```python
def new_data_structs():
    """
    Inicializa las estructuras de datos del modelo. Las crea de
    manera vacía para posteriormente almacenar la información.
    """
    #TODO: Inicializar las estructuras de datos
    analyzer = {}

    analyzer['intersections'] = mp.newMap(numelements=500000,
                                          maptype='PROBING',
                                          cmpfunction=compareIntersectionIds)

    analyzer['coordinates'] = mp.newMap(numelements=500000,
                                        maptype='PROBING',
                                        cmpfunction=compareIntersectionIds)

    analyzer['localities'] = mp.newMap(numelements=500000,
                                       maptype='PROBING',
                                       cmpfunction=compareIntersectionIds)

    analyzer['connectionsDistance'] = gr.newGraph(datastructure='ADJ_LIST',
                                                  directed=False,
                                                  size=500000)

    analyzer['connectionsComparendos'] = gr.newGraph(datastructure='ADJ_LIST',
                                                     directed=False,
                                                     size=500000)

    analyzer['policeStations'] = lt.newList("ARRAY_LIST")

    analyzer['infractions'] = lt.newList("ARRAY_LIST")

    analyzer['vertices'] = lt.newList("ARRAY_LIST")

    analyzer['edges'] = set()

    analyzer['cityLimits'] = {
                            "minLongitude": 1000000000,
                            "maxLongitude": -100000000,
                            "minLatitude": 1000000000,
                            "maxLatitude": -100000000
                           }

    return analyzer
```

Descripción

El objetivo final de la carga era completar dos grafos, donde sus vértices eran las intersecciones y los arcos eran las calles, pero uno tenía de peso la distancia entre vértices y el otro la cantidad de comparendos entre dos vértices. Para ello al momento de cargar la información se creaban:

- Mapa de vértices, donde cada vértice su valor era un dict con toda su información.
- Mapa de longitudes, para poder asociar los parámetros de entrada de los requerimientos por longitud y latitud.
- Mapa de localidades, donde cada localidad su valor era una lista con todos los comparendos de esa localidad.
- Dos grafos con distintos pesos en sus arcos.
- Tres listas con la información cargada de cada vértice, estación de policía y comparendos en diccionarios.
- Mapa de arcos(set) con todos los vértices, que tienen arcos, sin repetir.
- Diccionario con los límites de la ciudad.

Entrada	Inicialización de la estructura de datos
Salidas	Estructura de datos, cargada con los vértices (intersecciones), estaciones de policía, comparendos y arcos (calles).
Implementado (Sí/No)	Si, todos la implementaron.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
load_vertices(V): - add_vertex(1):	O(V)
load_police_stations(P): - add_police_station(1)	O(P)
load_comparendos(C): - add_comparendo(1)	O(C)
load_edges(E): - add_edge(1):	O(E)
TOTAL	***O(V+P+C+E)***

Análisis

La complejidad temporal de la carga de datos es ***O(V+P+C+E),*** porque todas las funciones de adicción en el **model** tienen complejidad **O(1)**, sin embargo, estas se tienen que ejecutar para cada línea de los archivos cargados. Por ende, es **O(1)*N (tomando N como el tamaño de cualquier archivo).** En este caso como tenemos 4 archivos con distintos tamaños, es la suma de los 4 tamaños.

Requerimiento 1

```
def req_1(analyzer, longitud1, latitud1, longitud2, latitud2):
    """
    Función que soluciona el requerimiento 1
    """
    # TODO: Realizar el requerimiento 1
    longitud1 = round(longitud1,5)
    latitud1 = round(latitud1, 5)
    longitud2 = round(longitud2, 5)
    latitud2 = round(latitud2, 5)
    verticein = ""
    verticeto = ""
    menor1 = 9999999
    menor2 = 9999999
    if mp.contains(analyzer["coordinates"], longitud1):
        entryin = mp.get(analyzer["coordinates"], longitud1)
        verticeins = me.getValue(entryin)
        for i in lt.iterator(verticeins):
                resta = abs(latitud2-i["latitudeVertex"])
                if resta < menor1:
                    menor1 = resta
                    verticein = i["ID"]
    if mp.contains(analyzer["coordinates"], longitud2):
        entryto = mp.get(analyzer["coordinates"], longitud2)
        verticetos = me.getValue(entryto)
        for o in lt.iterator(verticetos):
                resta = abs(latitud2-o["latitudeVertex"])
                if resta < menor2:
                    menor2= resta
                    verticeto = o["ID"]
    search= dfs.DepthFirstSearch(analyzer['connectionsDistance'], verticein)
    camino = dfs.pathTo(search, verticeto)
    distancia = 0
```

```
peso = None
while i < lt.size(camino):
    verticea = lt.getElement(camino, i)
    verticeb = lt.getElement(camino, i+1)
    arco = gr.getEdge(analyzer['connectionsDistance'], verticea, verticeb)
    if arco != None:
        peso = arco["weight"]
    if peso != None:
        distancia += peso
    i += 1
mapa = req_8(analyzer['intersections'], camino, 1)
return distancia, lt.size(camino), camino
```

Descripción

El requerimiento uno busca un posible camino de un punto de origen a un punto de destino, no necesariamente el camino más corto, retornando en si la distancia total que tomará el camino, el número de vértices que tiene el camino y todos sus vértices que lo componen.

Entrada	Longitud y Latitud de el punto de origen y de el punto de destino
Salidas	La distancia total que tomará el camino, el número de vértices que tiene el camino y todos sus vértices que lo componen.
Implementado (Sí/No)	Si se implementó por todos

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicialización de variables	O(1)
Primer condicional= om.contains	O(1)
Om.get y me.getvalues()	O(1)
Primer ciclo for	***O(V)***
comparación	***O(1)***
Segundo condicional= om.contains	***O(v)***
Om.get y me.getvalues()	***O(1)***
Segundo ciclo for	***O(v)***
comparación	***O(1)***
Algoritmo dfs	***O(E+V)***
Prmer ciclo while	***O(V)***
Llamado a la funcion req 8	***O(V)***
Total	***O(E+V)***

.

Análisis

La complejidad de este requerimiento es de O(E+V) ya que todos los ciclos se suman por lo que sería solo V, sin embargo, gracias a que se realiza un algoritmo de dfs su complejidad es O(E+V) y al sumarlo con los ciclos nos da dicha complejidad.

Requerimiento 2

```python
def req_2(analyzer, originLatiude, originLongitude, destinationLatitude, destinationLongitude):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    originCoordinate = round(originLongitude, 5)
    destinationCoordinate = round(destinationLongitude, 5)

    if mp.contains(analyzer['coordinates'], originCoordinate):
        originCoordinateEntry = mp.get(analyzer['coordinates'], originCoordinate)
        originCoordinatesID = me.getValue(originCoordinateEntry)

        minOriginDistance = 1000000
        originID = None

        for originCoordinateID in lt.iterator(originCoordinatesID):
            originDistance = abs(originLatiude - originCoordinateID["latitudeVertex"])

            if originDistance < minOriginDistance:
                minOriginDistance = originDistance
                originID = originCoordinateID["ID"]

    BFSearch = bfs.BreathFirstSearch(analyzer['connectionsDistance'], originID)

    if mp.contains(analyzer['coordinates'], destinationCoordinate):
        destinationCoordinateEntry = mp.get(analyzer['coordinates'], destinationCoordinate)
        destinationCoordinatesID = me.getValue(destinationCoordinateEntry)

        minDestinationDistance = 1000000
        destinationID = None

        for destinationCoordinateID in lt.iterator(destinationCoordinatesID):
            destinationDistance = abs(destinationLatitude - destinationCoordinateID["latitudeVertex"])

            if destinationDistance < minDestinationDistance:
                minDestinationDistance = destinationDistance
                destinationID = destinationCoordinateID["ID"]

    path = bfs.pathTo(BFSearch, destinationID)
    verticesPath = lt.size(path)
```

Descripción

El requerimiento busca un posible camino “más corto”, según el número de intersecciones a cruzar, entre dos puntos geográficos localizados dentro límites de la ciudad de Bogotá. El punto de origen y destino son ingresados por el usuario como latitudes y longitudes (debe validarse que dichos puntos se encuentren dentro de los límites encontrados de la ciudad). Estas ubicaciones deben aproximarse a los vértices más cercanos en la malla vial.

Entrada	- Punto de origen (una localización geográfica con latitud y longitud) - Punto de destino (una localización geográfica con latitud y longitud)
Salidas	- Distancia total que tomará el camino entre el punto de encuentro de origen y el de destino. - Total de vértices que contiene el camino encontrado. - Secuencia de vértices que componen el camino encontrado
Implementado (Sí/No)	Si, todos lo implementaron.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicialización de variables y operaciones matemáticas	O(1)
om.contains(), om.get y me.getvalues()	O(1)
Primer ciclo FOR (Búsqueda de vértice más cercano)	**O(V)**
- Condicionales y operaciones matemáticas	O(1)
Algoritmo BFS	**O(E+V)**
Segundo ciclo FOR (Búsqueda de vértice más cercano)	**O(V)**
- Condicionales y operaciones matemáticas	O(1)
BFS.pathTo (Reconstrucción camino)	**O(V)**
Tercer ciclo FOR (Sumatorias distancias del camino)	**O(V)**
- lt.getElement() y gr.getEdge()	O(1)
Total	***O(E+V)***

Análisis

La complejidad temporal del requerimiento es ***O(E+V),*** porque con respecto a todas las demás funciones que tienen peso en el requerimiento el algoritmo BFS es mayor, todas las demás son **O(V)** en el peor caso y están por debajo de ***O(E+V).***

Requerimiento 3

```python
def req_3(analyzer, cameras, locality):
    """
    Función que soluciona el requerimiento 3
    """
    # TODO: Realizar el requerimiento 3
    verticesLocality = set()

    localityEntry = mp.get(analyzer['localities'], locality)
    localityInfractions = me.getValue(localityEntry)

    for infraction in lt.iterator(localityInfractions):
        verticesLocality.add(infraction["vertex"])

    condiHeap = mpq.newMinPQ(sortComparendos)

    for vertexLocality in verticesLocality:
        vertexEntry = mp.get(analyzer['intersections'], vertexLocality)
        infoVertex = me.getValue(vertexEntry)
        mpq.insert(condiHeap, infoVertex)

    camerasVertices = lt.newList("ARRAY_LIST")

    for _ in range(cameras):

        if not mpq.isEmpty(condiHeap):
            maxInfractions = mpq.delMin(condiHeap)
            lt.addLast(camerasVertices, maxInfractions["ID"])

    print(camerasVertices["elements"])
    originVertex = lt.firstElement(camerasVertices)
    communicationRed = prim.PrimMST(analyzer['connectionsDistance'], originVertex)

    DistanceRed = 0
    routesRed = mp.newMap(numelements=500000,
                                    maptype='PROBING',
                                    cmpfunction=compareIntersectionIds)
```

Descripción

El requerimiento busca instalar una red de comunicaciones por fibra óptica de cámaras de video en M sitios. Sin embargo, se requiere que esta red tenga el menor costo de instalación posible. Para que la red sea eficiente se seleccionan como puntos de supervisión los M vértices con el mayor número de comparendos para localidad definida por el usuario.

Entrada	- Cantidad de cámaras de video que se desean instalar (M) - Localidad donde se desean instalar
Salidas	- Vértices incluidos (identificadores) - Rutas de los identificadores entre si - Cantidad de kilómetros de fibra óptica extendida - Costo (monetario) total.
Implementado (Sí/No)	Si, lo implemento Juan Diego Rodríguez.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

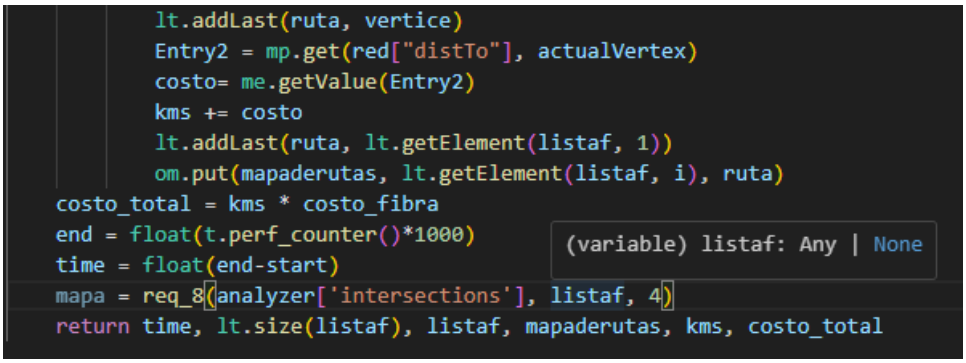
Pasos	Complejidad
Inicialización de variables y operaciones matemáticas	O(1)
om.get y me.getvalues()	O(1)
Primer ciclo FOR (Agregar vertices no repetidos de la localidad)	**O(V)**
- Insertar set.add()	O(1)
Segundo ciclo FOR (Agregar dict del vertice al heap)	**O(V)**
- om.get y me.getvalues()	O(1)
- **mpq.insert()**	**O(logV)**
mpq.delMin()	**O(logV)**
Algoritmo MST (prim)	**O(E*logE)**
Tercer ciclo FOR (Calculo distancias MST)	**O(1)**
- **Ciclo While**	**O(V)**
Total	***O(E*logE)***

Análisis

La complejidad temporal del requerimiento es ***O(E*logE),*** porque con respecto a todas las demás funciones que tienen peso en el requerimiento el algoritmo Prim es mayor, todas las demás son **O(V)** y en el segundo ciclo FOR seria **O(V*logV)** en el peor caso y están por debajo de ***O(E*logE).***

Requerimiento 4

```
def req_4(analyzer, m):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4
    start = float(t.perf_counter()*1000)
    costo_fibra = 1000000
    kms = 0
    lista_ordenada = mpq.newMinPQ(cmp_gravedad_2)
    listaf = lt.newList("ARRAY_LIST")
    for infraccion in lt.iterator(analyzer["infractions"]):
        mpq.insert(lista_ordenada, infraccion)
    while lt.size(listaf) < m:
        elemento = mpq.delMin(lista_ordenada)
        lt.addLast(listaf, elemento["ID"])
    red = prim.PrimMST(analyzer['connectionsDistance'], lt.getElement(listaf, 1))
    mapaderutas = mp.newMap(numelements=500000,
                                        maptype='PROBING',
                                        cmpfunction=compareIntersectionIds)
    for i in range(2, lt.size(listaf)):
        ruta = lt.newList("ARRAY_LIST")
        actualVertex = lt.getElement(listaf, i)
        lt.addLast(ruta, actualVertex)
        while actualVertex != lt.getElement(listaf, 1):
            Entry = mp.get(red["edgeTo"], actualVertex)
            arco = me.getValue(Entry)
            vertice = ed.other(arco, actualVertex)
```



Descripción

El requerimiento 4 busca los m puntos que tienen conparedos de más gravedad para instalar una red de camaras en estos puntos con el menor costo, para esto devuelve: el tiempo en el que este se demora en solucionar el problema, el numero de vertices que hay, los vertices, las rutas, la distancia total que tiene la red y el costo total

Entrada	El numero de camaras que se quieren instalar.
Salidas	el tiempo en el que este se demora en solucionar el problema, el numero de vertices que hay, los vertices, las rutas, la distancia total que tiene la red y el costo total
Implementado (Sí/No)	Si se implementó por Samuel Escobar

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicialización de variables	O(1)
Primer ciclo for = mpq.insert	O(VlogV)
Segundo ciclo while = mpq.delmin y lt.addlast	O(logV)
MST algoritmo prim	O(E*log(E)***)***
Creación de mapa	O(1)
Tercer ciclo for	O(V)
Cuarto ciclo While	O(M)
Total	O(V)

.

Análisis

La complejidad del requerimiento 4 es de O(V), ya que los dos primero ciclos se suman y como el cuarto está dentro del tercero pero este depende de el tercero, entonce también se suman y al sumar el mayor es V, por eso da dicha complejidad.

Requerimiento 5

```python
def req_5(grafo_d, data, m, clase_v):
    """
    Función que soluciona el requerimiento 5
    """
    costo_fibra = 1000000
    ids = mp.keySet(data)
    lista = lt.newList("ARRAY_LIST")
    max_comparendos = mpq.newMinPQ(cmp_n_comparendos)
    vertices_red = lt.newList("ARRAY_LIST")
    kms = 0

    for i in lt.iterator(ids):
        dict_c = {}
        entry = mp.get(data, i)
        info = me.getValue(entry)
        if len(info) >= 5:
            comparendos = info["Infractions"]
            contador = 0

            for j in lt.iterator(comparendos):
                if j["Vehicle Class"] == clase_v:
                    contador += 1
            dict_c["id"] = i
            dict_c["n"] = contador
            mpq.insert(max_comparendos, dict_c)


    for _ in range(m):
        max = mpq.delMin(max_comparendos)
        lt.addLast(lista, max)

    red = prim.PrimMST(grafo_d, lt.getElement(lista, 1)["id"])
    lt.addLast(vertices_red, lt.getElement(lista, 1)["id"])
    n_c = mp.newMap(numelements=200, maptype='PROBING')

    for i in range(2, m+1):
        valor, vertice_1, vertice_2 = distancia_total(red, lt.getElement(lista, i)["id"])
        kms += valor
        mp.put(n_c, vertice_1, 1)
        mp.put(n_c, vertice_2, 1)


    vertices_red = mp.keySet(n_c)


    costo_total = kms * costo_fibra

    mapa = req_8(data, vertices_red, 5)

    return m, vertices_red, kms, costo_total
```

Descripción

Se quiere conocer la red más eficiente que conecte los m puntos con mayor cantidad de comparendos según un tipo de vehículo. Para esto, se van a clasificar los m puntos de mayor a menor según la cantidad de comparendos y posteriormente se va a utilizar el algoritmo de Prim, el cual permitirá conocer la ruta más eficiente entre todos estos puntos.

Entrada	Mapa con la información de los vértices, grafo de distancias, los m puntos de la red y el tipo de vehículo.
Salidas	La red más eficiente para instalar las cámaras, el costo total de la red y la extensión total de esta y el número de vértices incluidos.
Implementado (Sí/No)	Si, por Rodrigo Paz Londoño

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

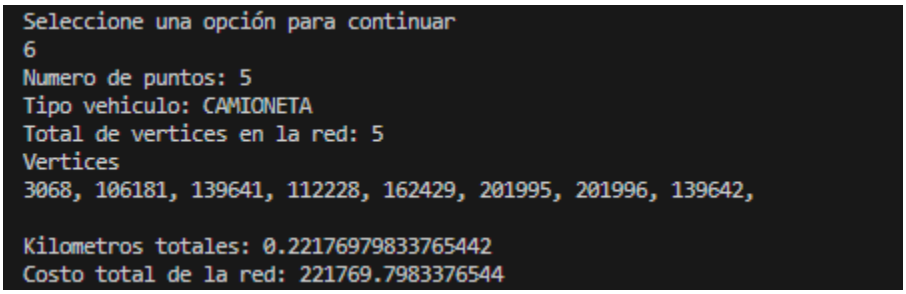
Pasos	**Complejidad**
Insertar elemento a la cola	O(V*log(v))
Recorrer Ids	O(V*C)
Comparaciones, sumas	O(1)
delMin	O(log(V))
MST(prim)	O(E*log(E))
mp.put	O(V)
mp.get	O(V)
Total	**O(V*C)**

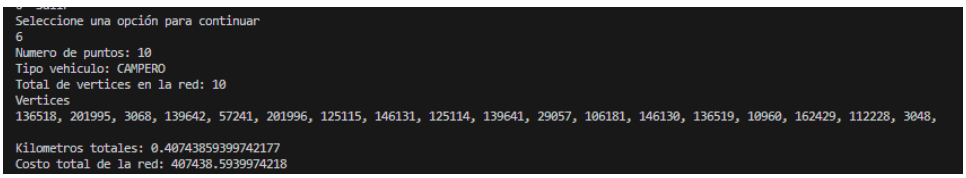
.

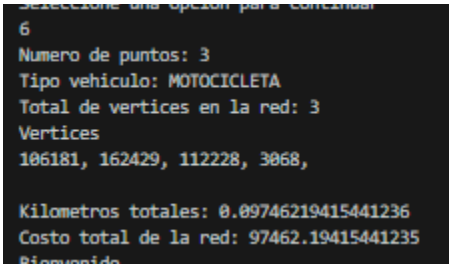
Análisis

La complejidad es de O(V*C), porque para cada vértice se tienen que recorrer todos los comparendos de este para comprobar cuales comparendos cumplen con el tipo de vehículo que entra como parámetro. La ejecución del requerimiento ronda entre los 30 y 45 minutos, por lo cual anexo pruebas con algunos casos aleatorios y su respectiva respuesta.

Prueba Funcionamiento







Requerimiento 6

```python
def req_6(grafo, data, m, estaciones, estacion):
    """
    Función que soluciona el requerimiento 6
    """
    partida = ""
    for s in lt.iterator(estaciones):
        if s["ID"] == estacion:
            partida = s["vertex"]

    ids = mp.keySet(data)
    lista = lt.newList("ARRAY_LIST")
    for i in lt.iterator(ids):
        dict_c = {}
        entry = mp.get(data, i)
        info = me.getValue(entry)
        if len(info) >= 5:
            comparendos = info["Infractions"]
            for j in lt.iterator(comparendos):
                dict_c["id"] = i
                dict_c["Service Type"] = j["Service Type"]
                dict_c["Infraction"] = j["Infraction"]
                lt.addLast(lista, dict_c)

    sorted_list = merg.sort(lista, cmp_gravedad)
    sorted_list = lt.subList(sorted_list, 1, m)
    coordenadas = lt.newList("ARRAY_LIST")
    red = djk.Dijkstra(grafo, partida)
    rta = lt.newList("ARRAY_LIST")
    for j in lt.iterator(lista):
        if djk.hasPathTo(red, j["id"]):
            camino = djk.pathTo(red, j["id"])
            n_c = mp.newMap(numelements=200, maptype='PROBING')
            for c in lt.iterator(camino):
                mp.put(n_c, c["vertexA"], 1)
                mp.put(n_c, c["vertexB"], 1)
                lt.addLast(coordenadas, c["vertexA"])
                lt.addLast(coordenadas, c["vertexB"])
            vertices = mp.keySet(n_c)
            cadena = ""
            for v in lt.iterator(vertices):
                cadena += v + ", "
            cadena.strip(" ,")
            data_camino = {}
            data_camino["vertices totales"] = mp.size(n_c)
            data_camino["vertices"] = cadena
            data_camino["kilometros"] = djk.distTo(red, j["id"])
            lt.addLast(rta, data_camino)
    mapa = req_8(data, coordenadas, 6)
    return rta
```

Descripción

Se desean conocer los caminos más eficientes, desde una estación de policía dada, a los m comparendos más graves. Para esto, se van a organizar los vértices según su gravedad, posteriormente se filtran por los m puntos. Después, mediante el algoritmo de Dijsktra, se van a obtener los caminos con menor costos desde la estación de policía a los demás puntos del grafo. Finalmente, se obtiene todos los vértices para llegar a cada uno de los m puntos, su costo y la cantidad de kilómetros que hay entre el punto y la estación.

Entrada	Mapa con la información, grafo de distancias, numero m de comparendos y el id de la estación de policía.
Salidas	Diccionario, que contiene la información del camino más corto desde la estación a los m comparendos más graves.
Implementado (Sí/No)	Si, por todos

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

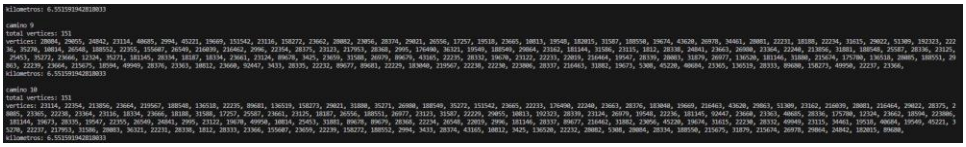
Pasos	**Complejidad**
Insertar elemento a la cola	O(Vlog(v))
delMin	O(log(V))
Dijsktra	O(E*log(V))
Recorrer Ids	O(V*C)
Recorrer comparendos mas graves	O(V*X)
Comparaciones, sumas	O(1)
mp.get	O(V)
mp.put	O(V)
Total	**O(V*X)**

.

Análisis

La complejidad es O(V*X) porque en el último ciclo, para cada vértice, se obtiene el camino desde la estación de policía y, posteriormente, se recorren todos los caminos para crear una cadena de caracteres con la red de caminos para llegar al vértice.

Prueba Funcionamiento



Requerimiento 7

```python
def req_7(analyzer, originLatiude, originLongitude, destinationLatitude, destinationLongitude):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7
    originCoordinate = round(originLongitude, 5)
    destinationCoordinate = round(destinationLongitude, 5)

    if mp.contains(analyzer['coordinates'], originCoordinate):
        originCoordinateEntry = mp.get(analyzer['coordinates'], originCoordinate)
        originCoordinatesID = me.getValue(originCoordinateEntry)

        minOriginDistance = 1000000
        originID = None

        for originCoordinateID in lt.iterator(originCoordinatesID):
            originDistance = abs(originLatiude - originCoordinateID["latitudeVertex"])

            if originDistance < minOriginDistance:
                minOriginDistance = originDistance
                originID = originCoordinateID["ID"]

    print(originID)
    DJKsearch = djk.Dijkstra(analyzer['connectionsComparendos'], originID)

    if mp.contains(analyzer['coordinates'], destinationCoordinate):
        destinationCoordinateEntry = mp.get(analyzer['coordinates'], destinationCoordinate)
        destinationCoordinatesID = me.getValue(destinationCoordinateEntry)

        minDestinationDistance = 1000000
        destinationID = None

        for destinationCoordinateID in lt.iterator(destinationCoordinatesID):
            destinationDistance = abs(destinationLatitude - destinationCoordinateID["latitudeVertex"])

            if destinationDistance < minDestinationDistance:
                minDestinationDistance = destinationDistance
                destinationID = destinationCoordinateID["ID"]

    path = djk.pathTo(DJKsearch, destinationID)
    verticesPath = lt.size(path)
```

Descripción

El requerimiento encuentra el camino “más corto” en términos número de menor cantidad de comparendos entre dos puntos de geográficos localizados en los límites de la ciudad de Bogotá. El punto de origen y destino son ingresados por el usuario como latitudes y longitudes (debe validarse que dichos puntos se encuentren dentro de los límites encontrados de la ciudad). Estas ubicaciones deben aproximarse a los vértices más cercanos en la malla vial.

Entrada	- Punto de origen (una localización geográfica con latitud y longitud) - Punto de destino (una localización geográfica con latitud y longitud)
Salidas	- Distancia total que tomará el camino entre el punto de encuentro de origen y el de destino. - Total de comparendos que contiene el camino encontrado. - Total de vértices que contiene el camino encontrado. - Secuencia de vértices que componen el camino encontrado
Implementado (Sí/No)	Si, todos lo implementaron.

Análisis de complejidad

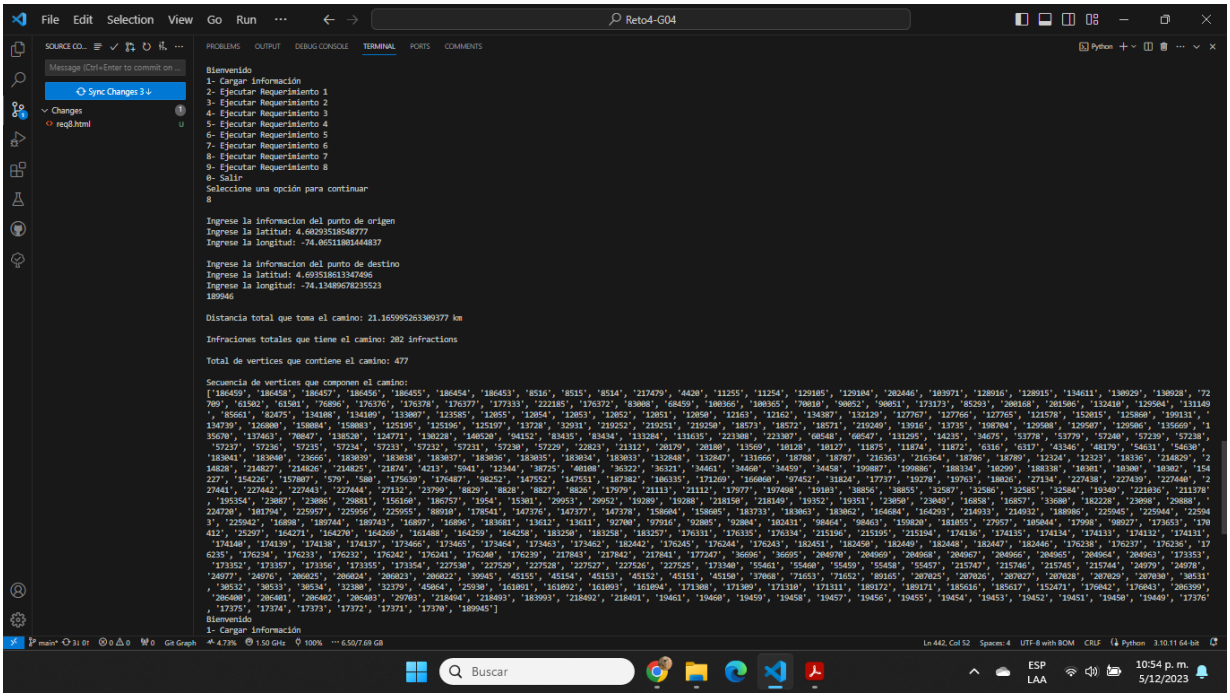
Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicialización de variables y operaciones matemáticas	O(1)
om.contains(), om.get y me.getvalues()	O(1)
Primer ciclo FOR (Búsqueda de vértice más cercano)	**O(V)**
- Condicionales y operaciones matemáticas	O(1)
Algoritmo DIJKSTRA	**O(E*logV)**
Segundo ciclo FOR (Búsqueda de vértice más cercano)	**O(V)**
- Condicionales y operaciones matemáticas	O(1)
DJK.pathTo y DJK.distTo (Reconstrucción camino y su peso total)	**O(V)**
Tercer ciclo FOR (Sumatorias distancias del camino)	**O(V)**
- lt.getElement() y gr.getEdge()	O(1)
Total	***O(E*logV)***

Análisis

La complejidad temporal del requerimiento es ***O(E*logV),*** porque con respecto a todas las demás funciones que tienen peso en el requerimiento el algoritmo DIJKSTRA es mayor, todas las demás son **O(V)** en el peor caso y están por debajo de ***O(E*logV).***

Prueba Funcionamiento



Requerimiento 8

```python
def req_8(data, vertices, req):
    """
    Función que soluciona el requerimiento 8
    """
    mapa = folium.Map(location=[4.6097, -74.0817], zoom_start=12)

    coordenadas = []
    for v in lt.iterator(vertices):
        entry = mp.get(data, v)
        info = me.getValue(entry)
        punto = [info["latitudeVertex"], info["longitudeVertex"]]
        coordenadas.append(punto)

    folium.PolyLine(
    locations=coordenadas,
    color='red',
    weight=3,
    opacity=1
    ).add_to(mapa)
    mapa.save("req"+ str(req) +".html")
```

Descripción

Se van a representar gráficamente todos los requerimientos anteriores mediante la librería folium.

Entrada	Mapa con la información de los vértices, la lista de vértices que se tienen que graficar y el número del requerimiento que se va a graficar.
Salidas	No retorna nada, aunque guarda un mapa con archivo html en la carpeta que se encuentre guardad el reto.
Implementado (Sí/No)	Si, por todos.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Obtener coordenadas	O(V)
Acceder a llaves	O(1)
Agregar elementos a una lista	O(1)
mp.get	O(V)
TOTAL	***O(V)***

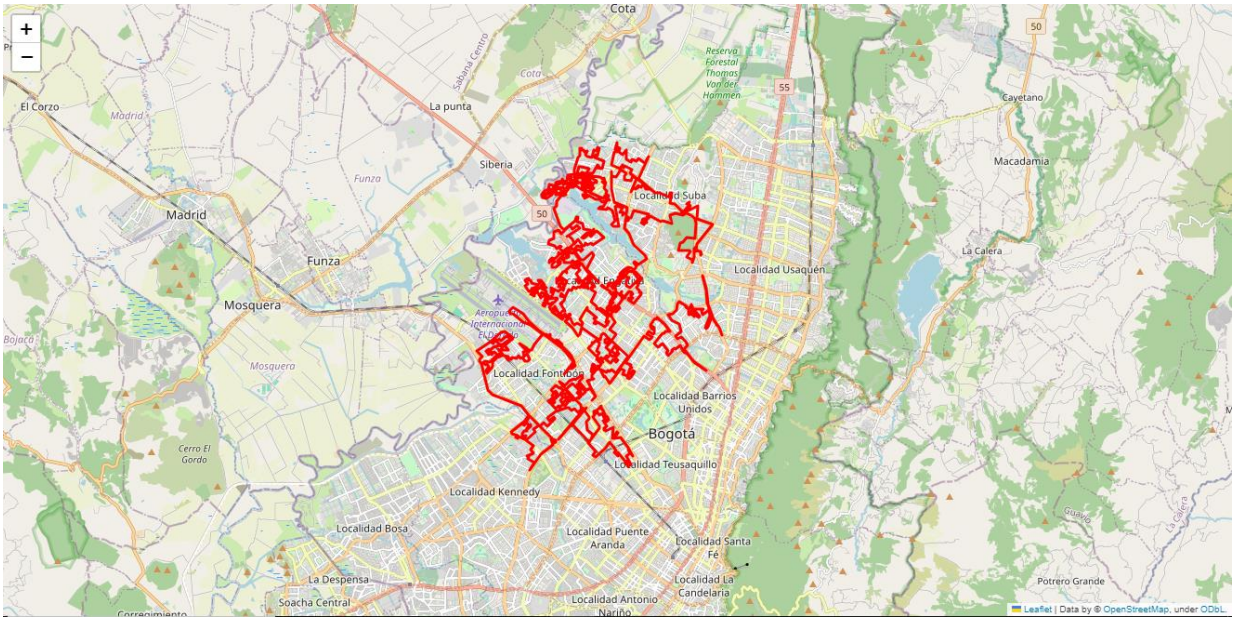
.

Análisis

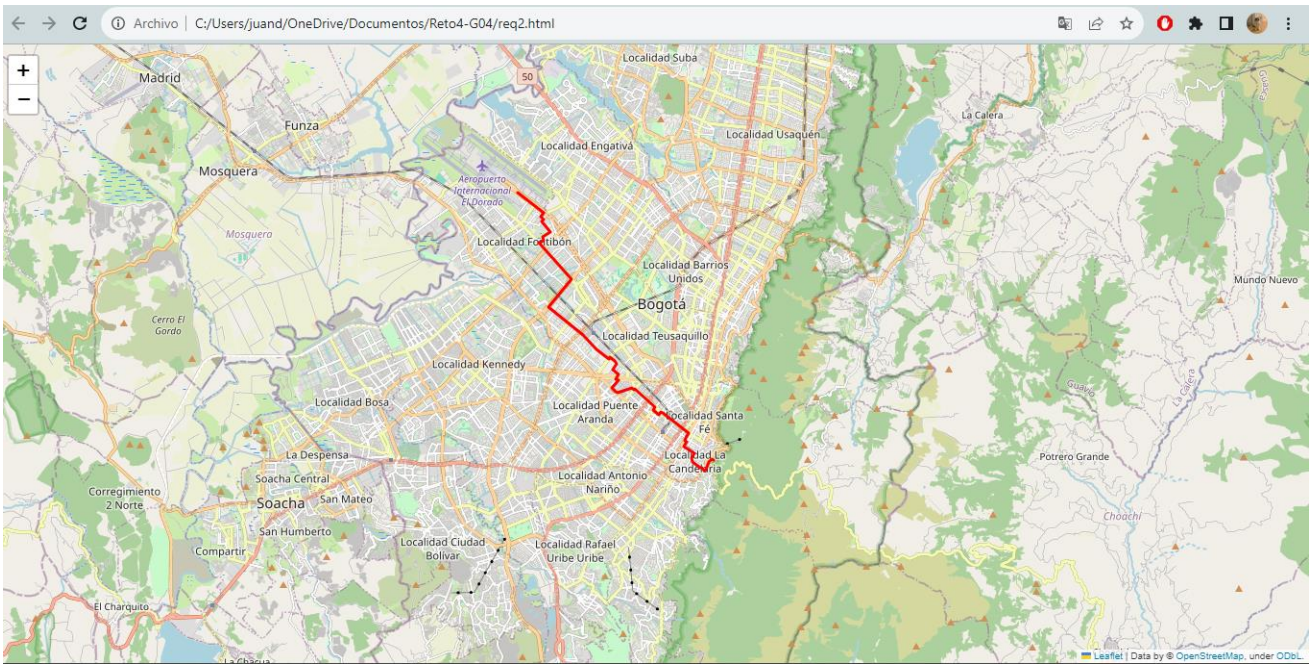
La complejidad es O(V) porque para cada vértice es necesario acceder a su valor en el mapa para obtener la información respecto a su longitud y latitud, para poner todas las coordenadas en una lista y ejecutar la función de folium, la cual recibe como parámetro una lista de listas.

Prueba Funcionamiento

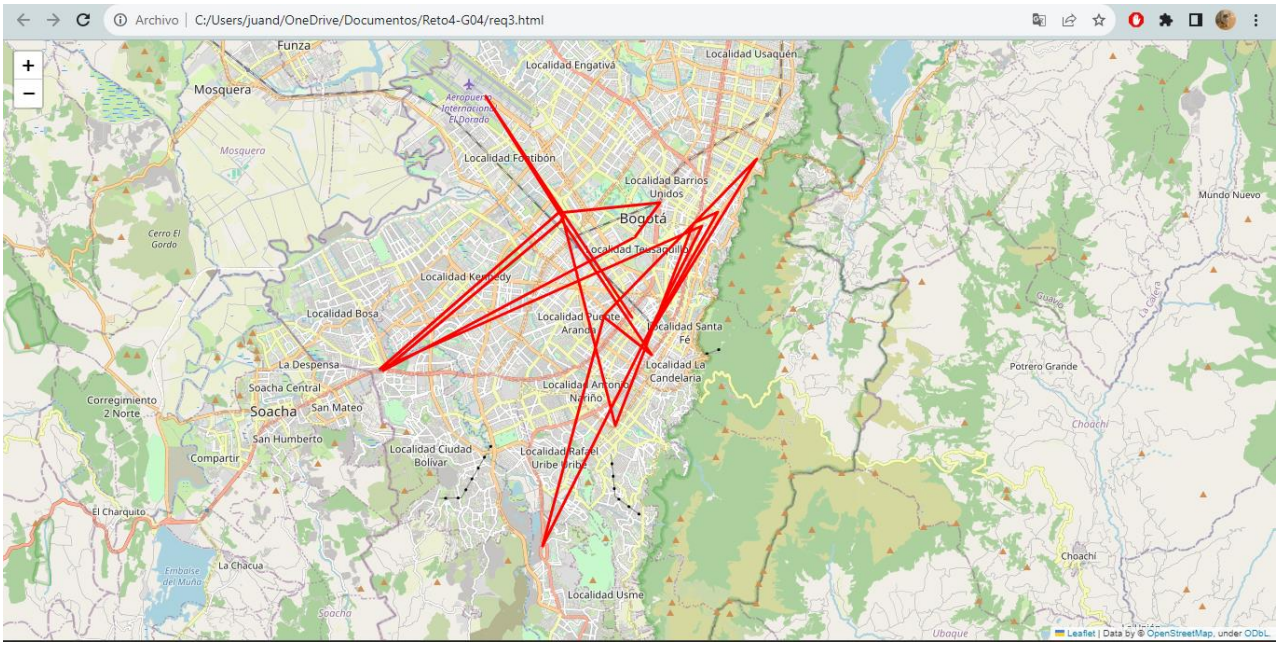
Requerimiento 1:



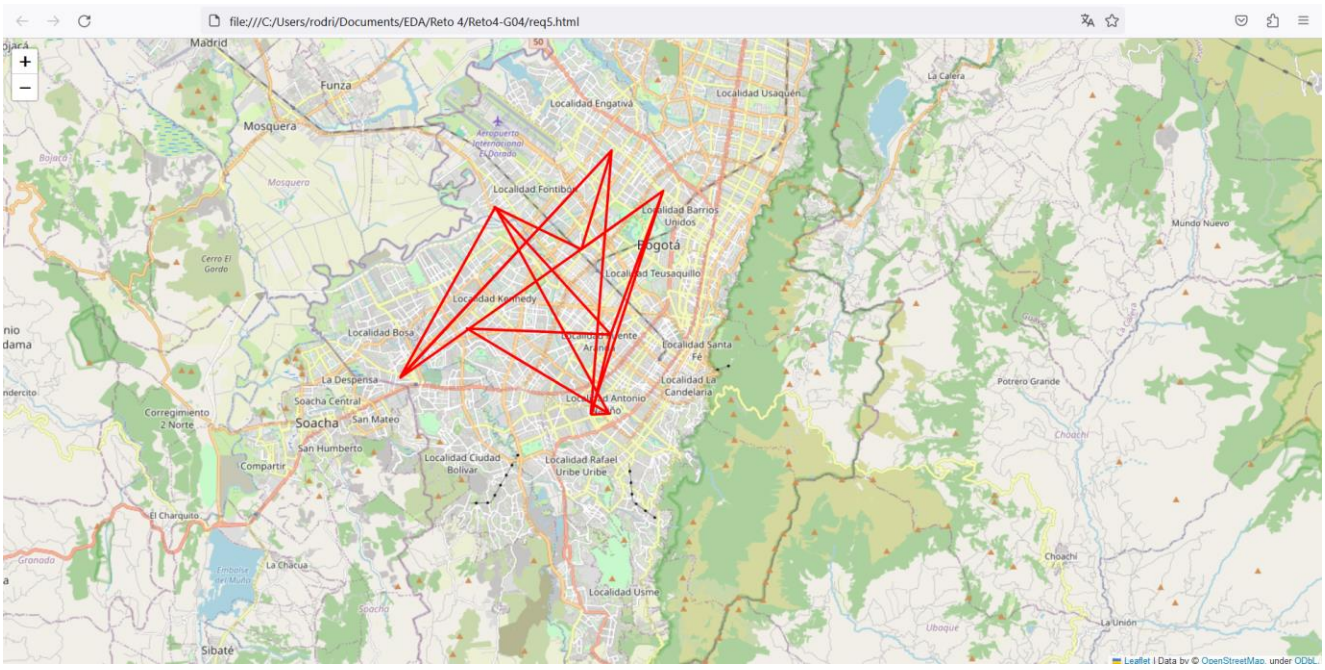
Requerimiento 2:



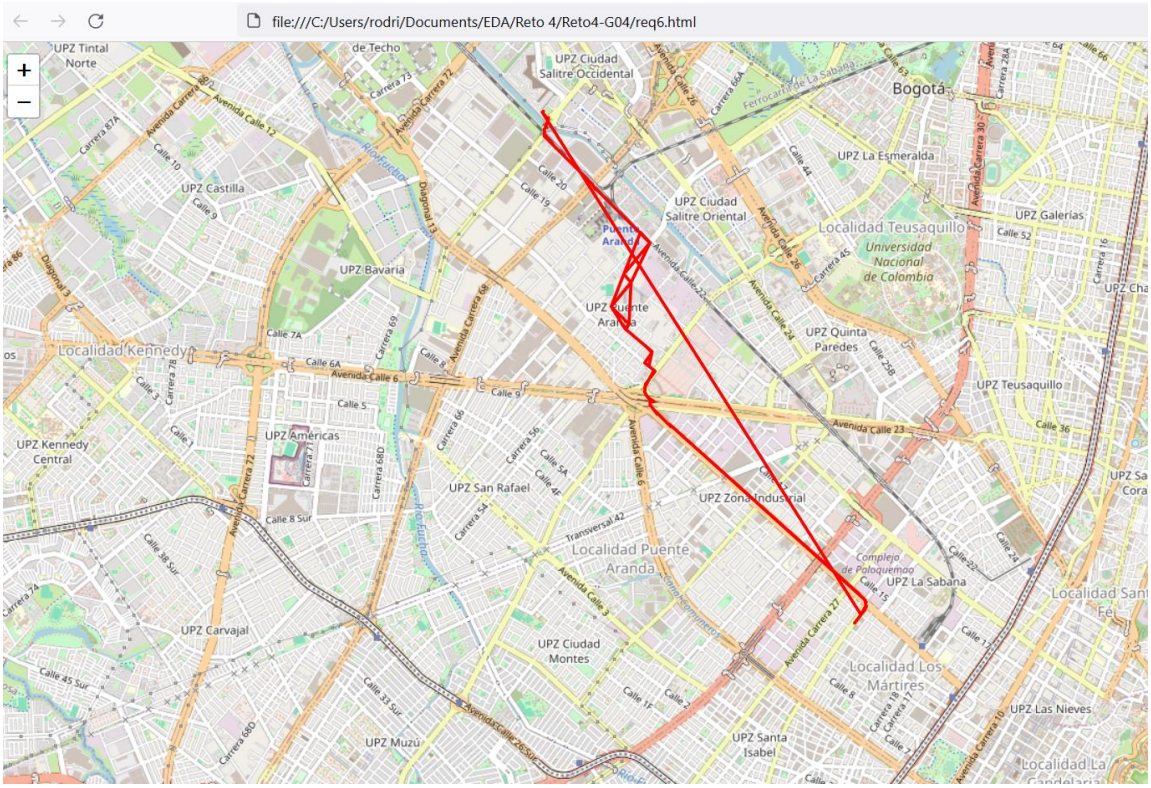
Requerimiento 3:



Requerimiento 5:



Requerimiento 6:



Requerimiento 7: