Universidad de los Andes
Facultad de Ingeniería

Departamento de Ingeniería de Sistemas y Computación
Estructuras de Datos y Algoritmos
ISIS-1225

ABET Engineering Accreditation Commission

ANÁLISIS DEL RETO

Juan Diego Diaz Villanueva, jd.diazv1@uniandes.edu.co, 202314374

Camilo Tellez, c.tellezs@uniandes.edu.co, 202312456

Anderson Mesa, a.mesat@uniandes.edu.co, 202112115

Requerimiento 1

Descripción

```python
def req_1(analyzer,longI,latiI, longDes, latitudDes):
    """
    Función que soluciona el requerimiento 1
    """
    # TODO: Realizar el requerimiento 1

    #Hallar el vertice mas cercano a ini
    distancia_menorIni=999999
    id_menorIni=None
    for x in lt.iterator(analyzer["lst_vertices"]):
        distancia_Ini= haversine_distance(longI, latiI, x["longitud"],x["latitud"])
        if distancia_Ini< distancia_menorIni:
            distancia_menorIni=distancia_Ini
            id_menorIni= x["id"]


    #Hallar el vertice mas cercano a ini
    distancia_menorDest=999999
    id_menorDest=None
    for x in lt.iterator(analyzer["lst_vertices"]):
        distancia_Dest= haversine_distance(longDes, latitudDes, x["longitud"],x["latitud"])
        if distancia_Dest< distancia_menorDest:
            distancia_menorDest=distancia_Dest
            id_menorDest= x["id"]

    # Hago una estructura con el recorrido del grafo

    #analyzer['recorrido_sobre_el_grafo'] = dfs.DepthFirstSearch(analyzer["malla_vial_dis"],id_menorIni)
    analyzer['recorrido_sobre_el_grafo'] = bfs.BreathFirstSearch(analyzer['malla_vial_dis'], id_menorIni)

    # Encuentro un camino
    #resp= dfs.pathTo(analyzer['recorrido_sobre_el_grafo'],id_menorDest)
    resp= bfs.pathTo(analyzer['recorrido_sobre_el_grafo'],id_menorDest)
    data=[haversine_distance(longI, latiI,longDes, latitudDes), lt.size(resp), resp]
    return data
```

La función req_1 busca el camino más corto entre dos puntos geográficos en un grafo que representa una malla vial. Primero, identifica los vértices más cercanos a los puntos de inicio y destino utilizando la distancia Haversine. Luego, realiza una búsqueda de amplitud (Breath First Search) desde el vértice más cercano al punto de inicio. Finalmente, encuentra el camino hasta el vértice más cercano al destino y retorna una lista que contiene la distancia Haversine total entre los puntos de inicio y destino, el tamaño del camino encontrado en el grafo y el propio camino

Entrada	• analyzer: Estructura de datos del analizador. • longI: Longitud inicial. • latiI: Latitud inicial. • longDes: Longitud de destino. • latitudDes: Latitud de destino.
Salidas	Una lista con la distancia total (calculada con la distancia de Haversine), el tamaño del camino encontrado y el camino en sí.
Implementado (Sí/No)	Si. Implementación: Grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Creación de una nueva lista (lt.newList)	O(1)
Búsqueda de los vértices más cercanos	O(n)
Búsqueda de camino con BFS (bfs.BreathFirstSearch)	O(V + E)
Obtención del camino (bfs.pathTo)	O(V)
Cálculo de distancia Haversine	O(1)
TOTAL	***O(V + E)***

Análisis

La función req_1 procesa un grafo de malla vial para encontrar el camino más corto entre un punto de origen y un punto de destino, dados en coordenadas geográficas. Inicialmente, identifica el vértice más cercano al punto de origen y al punto de destino dentro del grafo, utilizando la distancia de Haversine para medir la cercanía entre las coordenadas dadas y cada vértice en el grafo.

Una vez identificados los vértices más cercanos, la función emplea una búsqueda por amplitud (BFS, por sus siglas en inglés) para explorar el grafo partiendo del vértice más cercano al punto de origen. La BFS es un algoritmo eficiente para recorrer o buscar en estructuras de datos de grafo, y es particularmente útil para encontrar el camino más corto en términos de número de aristas.

Después de ejecutar BFS, la función determina el camino desde el punto de origen al punto de destino. Este camino se construye con base en los resultados de la búsqueda por amplitud, y la función finalmente devuelve la distancia total de Haversine entre los puntos de inicio y destino, la longitud del camino encontrado y el propio camino.

En resumen, req_1 es una función diseñada para resolver problemas de rutas en sistemas de malla vial o redes similares, identificando rutas óptimas en términos de proximidad geográfica y conectividad del grafo. La eficacia de la función depende de la estructura del grafo y de la distribución de sus vértices en relación con las coordenadas geográficas de interés.

Requerimiento 2

Descripción

```
def req_2(analyzer,longI,latiI, longDes, latitudDes):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    distancia_menorIni=999999
    id_menorIni=None
    for x in lt.iterator(analyzer["lst_vertices"]):
        distancia_Ini= haversine_distance(longI, latiI, x["longitud"],x["latitud"])
        if distancia_Ini< distancia_menorIni:
            distancia_menorIni=distancia_Ini
            id_menorIni= x["id"]


    #Hallar el vertice mas cercano a ini
    distancia_menorDest=999999
    id_menorDest=None
    for x in lt.iterator(analyzer["lst_vertices"]):
        distancia_Dest= haversine_distance(longDes, latitudDes, x["longitud"],x["latitud"])
        if distancia_Dest< distancia_menorDest:
            distancia_menorDest=distancia_Dest
            id_menorDest= x["id"]

    # Hago una estructura con el recorrido del grafo

    #analyzer['recorrido_sobre_el_grafo'] = dfs.DepthFirstSearch(analyzer["malla_vial_dis"],id_menorIni)
    analyzer['recorrido_sobre_el_grafo'] = bfs.BreathFirstSearch(analyzer['malla_vial_dis'], id_menorIni)

    # Encuentro un camino
    #resp= dfs.pathTo(analyzer['recorrido_sobre_el_grafo'],id_menorDest)
    resp= bfs.pathTo(analyzer['recorrido_sobre_el_grafo'],id_menorDest)
    data=[haversine_distance(longI, latiI,longDes, latitudDes), lt.size(resp), resp]
    return data
```

La función req_2 calcula el camino más corto en un grafo de malla vial entre dos puntos geográficos especificados. Inicialmente, encuentra los vértices más cercanos a los puntos de inicio y destino

utilizando la distancia Haversine. Posteriormente, realiza una búsqueda de amplitud (Breadth First Search) desde el vértice más cercano al punto de inicio. Finaliza encontrando el camino hasta el vértice más cercano al punto de destino y devuelve una lista que incluye la distancia Haversine entre los puntos de inicio y destino, el tamaño del camino hallado en el grafo y el camino en sí.

Entrada	• analyzer: Estructura de datos del analizador. • longI: Longitud inicial. • latiI: Latitud inicial. • longDes: Longitud de destino. • latitudDes: Latitud de destino.
Salidas	una lista con la distancia total, el tamaño del camino y el camino.
Implementado (Sí/No)	Si. Implementación: Grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Creación de una nueva lista (lt.newList)	O(1)
Búsqueda de los vértices más cercanos	O(n)
Búsqueda de camino con BFS (bfs.BreathFirstSearch)	O(V + E)
Obtención del camino (bfs.pathTo)	O(V)
Cálculo de distancia Haversine	O(1)
TOTAL	***O(V + E)***

Análisis

La función req_2 calcula un camino en un sistema de malla vial desde un punto inicial hasta un destino, dados por coordenadas geográficas. Primero, busca el vértice más cercano al punto de inicio y al punto de destino, utilizando la distancia de Haversine para medir la cercanía entre las coordenadas y cada vértice en el grafo. Este proceso tiene una complejidad de O(n), siendo n el número de vértices en la lista.

Una vez identificados los vértices más cercanos, la función emplea una búsqueda por amplitud (BFS) para explorar el grafo desde el vértice más cercano al punto de inicio. BFS es adecuado para encontrar el camino más corto en términos de número de aristas, con una complejidad de O(V + E), donde V es el número de vértices y E el número de aristas en el grafo.

Después de ejecutar la BFS, determina el camino desde el punto de inicio al destino. Este camino se construye con base en los resultados de la búsqueda por amplitud. Finalmente, la función devuelve la

distancia total de Haversine entre los puntos de inicio y destino, la longitud del camino encontrado (en términos de cantidad de nodos o pasos) y el propio camino.

En resumen, req_2 es una función diseñada para calcular rutas óptimas en términos de proximidad geográfica y conectividad en un sistema de malla vial o redes similares, empleando algoritmos eficientes de búsqueda en grafos.

Requerimiento 3

Descripción

```python
def req_3(data_structs, localidad, m):
    """
    Función que soluciona el requerimiento 3
    """
    comparendos = mp.get(data_structs["comparendos_localidad"], localidad.upper())

    if comparendos is None:
        return None
    comparendos = me.getValue(comparendos)

    cantidadComparendos = lt.newList(datastructure="ARRAY_LIST")

    for comparendo in lt.iterator(mp.keySet(comparendos)):
        lt.addLast(cantidadComparendos, {"vertice": comparendo, "cantidad": mp.get(comparendos, comparendo)["value"]})

    merg.sort(cantidadComparendos, ordenar_cantidad)

    seleccionados = cantidadComparendos
    if lt.size(cantidadComparendos) > m:
        seleccionados = lt.subList(cantidadComparendos, 1, m)

    verticesSeleccionados = lt.newList(datastructure="ARRAY_LIST")

    for v in lt.iterator(seleccionados):
        lt.addLast(verticesSeleccionados, int(v["vertice"]))

    print("CREANDO RED...")
    red, kms = crear_red(data_structs, verticesSeleccionados)

    return verticesSeleccionados, gr.vertices(red), gr.edges(red), kms, kms * 1000000
```

La función req_3 busca crear una nueva estructura de grafo basada en una lista de comparendos (presumiblemente multas o infracciones de tráfico). Cada par de vértices consecutivos en la lista de comparendos es conectado en el nuevo grafo, utilizando el algoritmo de Dijkstra para encontrar el camino más corto entre ellos en el grafo original.

Para cada par de comparendos consecutivos, calcula el camino más corto y la distancia entre ellos usando Dijkstra. Luego verifica si hay un camino más corto a cualquier otro vértice ya en el nuevo grafo y, si lo hay, actualiza el camino y la distancia. Cada arco en el camino calculado se agrega al nuevo grafo, asegurándose de que los vértices estén presentes en el grafo antes de agregar el arco.

Finalmente, retorna djkVertices, un diccionario que almacena la estructura de búsqueda de Dijkstra para cada vértice, aunque el contenido exacto de este diccionario no se especifica en el fragmento del código proporcionado.

Entrada	• data_structs: Estructura de datos del analizador. • localidad: Un identificador o clave para una localidad específica. • M: Un parámetro o conjunto de parámetros específicos.
Salidas	• verticesSeleccionados: Lista de los identificadores de los vértices seleccionados basados en la cantidad de comparendos. • gr.vertices(red): Lista de todos los vértices que forman parte de la red de comunicaciones creada. • gr.edges(red): Lista de todos los arcos que forman parte de la red, con sus respectivos detalles. • kms: Total de kilómetros de fibra óptica que se extenderán en la red de comunicaciones. • kms * 1000000: Costo total estimado de la implementación de la red en moneda local.
Implementado (Sí/No)	SÍ. Implementado por Camilo Tellez

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Obtener comparendos por localidad (`mp.get`)	O(1)
Crear la lista de comparendos (`lt.newList`)	O(1)
Iterar sobre comparendos y añadirlos a la lista	O(N)
Ordenar comparendos por cantidad (`merg.sort`)	O(N log N)
Seleccionar M comparendos (`lt.subList`)	O(M)
Crear lista de vértices seleccionados (`lt.newList`)	O(1)
Añadir vértices a la lista (`lt.addLast`)	O(M)

Obtener comparendos por localidad (`mp.get`)	O(1)
TOTAL	***O(M(V + E)).***

Análisis

La función req_3 construye un grafo optimizado a partir de una lista de datos, presumiblemente comparendos, y utiliza el algoritmo de Dijkstra para encontrar rutas óptimas entre estos puntos. Comienza creando una lista comparendos y un nuevo grafo grafoNuevo. Luego, itera sobre los elementos en comparendos, para cada par de vértices consecutivos, ejecuta el algoritmo de Dijkstra para determinar el camino más corto y la distancia entre ellos.

Durante este proceso, la función compara continuamente las distancias obtenidas con las distancias a otros vértices ya presentes en grafoNuevo. Si encuentra una ruta más corta, actualiza el camino y la distancia correspondiente. Este enfoque garantiza que el grafo resultante refleje las rutas más eficientes entre los puntos dados.

La función también se encarga de actualizar grafoNuevo con nuevos vértices y arcos, basándose en los caminos más cortos encontrados. Esto incluye verificar si los vértices ya existen en el grafo y, si no, agregarlos junto con sus respectivas conexiones.

Además, req_3 almacena información sobre las rutas y distancias calculadas en un diccionario djkVertices. Este diccionario asocia cada vértice con su estructura de búsqueda de Dijkstra correspondiente, proporcionando un acceso rápido a la información del camino más corto y las distancias a otros vértices.

En conclusión, req_3 es una función eficiente para construir un grafo basado en un conjunto de puntos, optimizando las rutas entre ellos mediante el algoritmo de Dijkstra. Esta función es especialmente útil para tareas de planificación de rutas en sistemas de transporte o en cualquier contexto donde sea crucial encontrar el camino más corto y eficiente entre múltiples puntos.

Requerimiento 4

Descripción

```python
def req_4(data_structs, m):
    comparendos = om.valueSet(data_structs["comparendos_gravedad"])

    seleccionados = mp.newMap(numelements=m, maptype="PROBING")

    for lstComparendos in lt.iterator(comparendos):
        for comparendo in lt.iterator(lstComparendos):
            mp.put(seleccionados, int(comparendo["VERTICES"]), comparendo)
            if mp.size(seleccionados) == m: break
        if mp.size(seleccionados) == m: break

    vS = mp.keySet(seleccionados)
    verticesSeleccionados = lt.newList(datastructure="ARRAY_LIST")

    for vertice in lt.iterator(vS):
        lt.addLast(verticesSeleccionados, vertice)

    print("CREANDO RED...")
    red, kms = crear_red(data_structs, verticesSeleccionados)

    # Vertices identificados, vertices utilizados para la conexión, arcos utilizados, kms, precio
    return verticesSeleccionados, gr.vertices(red), gr.edges(red), kms, kms * 1000000
```

El algoritmo implementado establece una red de fibra óptica para cámaras de video, centrada en los M sitios con los comparendos de mayor gravedad en la ciudad. Identifica los M vértices más críticos basándose en el tipo de servicio y el código de infracción de los comparendos. Una vez seleccionados, diseña una red que conecta estos puntos de manera eficiente y costeable. Proporciona detalles como el tiempo de ejecución, los vértices y arcos de la red, la longitud total de fibra óptica necesaria y el costo total, permitiendo así una implementación óptima y económica de la red de vigilancia

Entrada	• data_structs: Estructura de datos que incluye la información de comparendos y la malla vial. • m: Número de comparendos o vértices a seleccionar para la red de comunicaciones.
Salidas	verticesSeleccionados: Lista de los vértices seleccionados para la red. gr.vertices(red): Lista de todos los vértices que forman parte de la red de comunicaciones. gr.edges(red): Lista de todos los arcos de la red con sus detalles. kms: La longitud total en kilómetros de la red de fibra óptica. kms * 1000000: El costo total estimado de la red, calculado como la longitud total en kilómetros multiplicada por el costo por kilómetro.
Implementado (Sí/No)	Sí. Implementado por Anderson Mesa

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Obtener lista de comparendos (`om.valueSet`)	O(1)
Iterar sobre comparendos y seleccionarlos (`mp.put`)	O(N)
Crear lista de vértices seleccionados (`lt.newList`)	O(M)
Iterar y añadir vértices a la lista (`lt.addLast`)	O(M)
TOTAL	O(N)

Análisis

La función req_4 implementa una solución eficiente para la instalación de una red de comunicaciones por fibra óptica, conectando cámaras de video en ubicaciones estratégicas basadas en la gravedad de los comparendos. Esta función clasifica los comparendos según su gravedad, considerando primero el tipo de servicio y luego el código de la infracción en un orden lexicográfico.

Una vez completada la selección de los sitios más críticos, req_4 utiliza un algoritmo de árbol de expansión mínima para conectar estos puntos. Este algoritmo determina la configuración de la red que minimiza la longitud total de la fibra óptica requerida, reduciendo así el costo total de instalación. La longitud de la fibra óptica se calcula sumando las distancias de los caminos seleccionados y multiplicando el total por el costo por kilómetro.

Requerimiento 5

```python
def req_5(data_structs, m, vehiculo):
    comparendos = mp.get(data_structs["comparendos_vehiculo"], vehiculo.upper())

    if comparendos is None:
        return None
    comparendos = me.getValue(comparendos)

    cantidadComparendos = lt.newList(datastructure="ARRAY_LIST")

    for comparendo in lt.iterator(mp.keySet(comparendos)):
        lt.addLast(cantidadComparendos, {"vertice": comparendo, "cantidad": mp.get(comparendos, comparendo)["value"]})

    merg.sort(cantidadComparendos, ordenar_cantidad)

    seleccionados = cantidadComparendos
    if lt.size(cantidadComparendos) > m:
        seleccionados = lt.subList(cantidadComparendos, 1, m)

    verticesSeleccionados = lt.newList(datastructure="ARRAY_LIST")

    for v in lt.iterator(seleccionados):
        lt.addLast(verticesSeleccionados, int(v["vertice"]))

    print("CREANDO RED...")
    red, kms = crear_red(data_structs, verticesSeleccionados)

    return verticesSeleccionados, gr.vertices(red), gr.edges(red), kms, kms * 1000000
```

Descripción

req_5, se utiliza para determinar la ubicación óptima de una red de cámaras de video basada en la severidad de los comparendos de tráfico. La función selecciona los vértices más relevantes para la instalación de las cámaras y calcula la ruta más eficiente para conectar estos puntos, considerando el costo de la instalación de fibra óptica. Utiliza una cola de prioridad para ordenar los comparendos por gravedad y el algoritmo para encontrar los caminos más cortos entre los vértices seleccionados. Además, calcula el costo total y la distancia total de la fibra óptica necesaria para la red propuesta, multiplicando la distancia total por el costo por kilómetro de la instalación de la fibra óptica.

Entrada	• data_structs: Contiene todas las estructuras de datos necesarias, incluyendo un mapeo de comparendos por tipo de vehículo. • m: Número de vértices (ubicaciones de comparendos) a incluir en la red. • vehiculo: Tipo de vehículo para filtrar los comparendos.
Salidas	• verticesSeleccionados: Identificadores de los vértices seleccionados. • gr.vertices(red): Todos los vértices de la red creada. • gr.edges(red): Todos los arcos de la red con sus vértices de inicio y fin.

	• kms: Total de kilómetros de la red de fibra óptica. • kms * 1000000: Costo total de la instalación de la red.
Implementado (Sí/No)	Sí. Implementado por Juan Diego Diaz V.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Obtener comparendos para tipo de vehículo	O(1)
Crear lista de comparendos	O(N)
Ordenar comparendos por cantidad	O(N log N)
Seleccionar M comparendos	O(M)
Convertir vértices a enteros	O(M)
Obtener comparendos para tipo de vehículo	O(1)
TOTAL	***O(N log N)***

Análisis

La función req_5 se encarga de establecer una red de fibra óptica para conectar cámaras en sitios estratégicos, optimizando la selección de puntos basados en la gravedad de los comparendos. La función opera en varias etapas para alcanzar su objetivo final de configurar la red más eficiente en términos de costos.

Inicialmente, la función prepara la estructura de datos para albergar los puntos críticos identificados por los comparendos. Luego, procesa los comparendos y los organiza en una cola de prioridad para asegurar que los sitios seleccionados para las cámaras correspondan a los comparendos de mayor gravedad. Este proceso de selección es crucial para determinar los puntos de supervisión más relevantes para la red.

Una vez identificados los vértices cruciales, req_5 procede a construir la red, conectando los puntos seleccionados a través del algoritmo de Dijkstra, que encuentra las rutas más cortas entre estos. Este

enfoque garantiza que la red resultante esté optimizada para el menor costo de instalación, tomando en cuenta la gravedad de los comparendos en cada ubicación.

Durante la construcción de la red, la función verifica y actualiza continuamente la nueva estructura del grafo, añadiendo vértices y arcos y calculando las distancias y rutas más eficientes. La precisión del algoritmo de Dijkstra es esencial en este proceso, ya que permite a la función ajustar dinámicamente las rutas para reflejar las distancias más cortas posibles entre los puntos de interés.

La función culmina con el cálculo del costo total y la distancia total de la red de fibra óptica propuesta. Esto implica sumar las distancias de los caminos seleccionados y multiplicarlas por el costo establecido por kilómetro, proporcionando así una estimación precisa del gasto requerido para la instalación de la red

Requerimiento 6

```python
def req_6(data_structs, m): #
    comparendos = om.valueSet(data_structs["comparendos_gravedad_2"]) # Criterio distinto

    seleccionados = mp.newMap(numelements=m, maptype="PROBING")

    for lstComparendos in lt.iterator(comparendos):
        for comparendo in lt.iterator(lstComparendos):
            mp.put(seleccionados, int(comparendo["VERTICES"]), comparendo)
            if mp.size(seleccionados) == m: break
        if mp.size(seleccionados) == m: break

    vS = mp.keySet(seleccionados)
    caminosComparendos = lt.newList(datastructure="ARRAY_LIST")

    posicion_estaciones = {}

    for vertice in lt.iterator(vS):
        infoVertice = mp.get(data_structs["map_geo"], vertice)["value"]
        estacionCercana = None
        estacionDist = math.inf
        for estacion in lt.iterator(data_structs["lst_estaciones"]):
            posicionEstacion = posicion_estaciones.get(estacion["OBJECTID"], int(estacion["VERTICES"]))
            verticeEstacion = mp.get(data_structs["map_geo"], posicionEstacion)["value"]
            dist = haversine_distance(infoVertice["longitud"], infoVertice["latitud"], verticeEstacion["longitud"], verticeEstacion["latitud"])
            if dist < estacionDist:
                estacionCercana = estacion
        posicionEstacion = posicion_estaciones.get(estacionCercana["OBJECTID"], int(estacionCercana["VERTICES"]))
        posicion_estaciones[estacionCercana["OBJECTID"]] = vertice

        camino, distancia, _, _ = bfsSencillo(data_structs, posicionEstacion, vertice)

        info = {"estacion": estacionCercana["OBJECTID"], "atendiendo": vertice, "desde": posicionEstacion, "camino": camino, "distancia": distancia}

        lt.addLast(caminosComparendos, info)

    return caminosComparendos
```

Descripción

la funcion req_6, está diseñada para trazar una ruta en un grafo basándose en la severidad de las infracciones de tráfico. La función establece una jerarquía de gravedad para las infracciones, priorizando por tipo de servicio y luego por el código de infracción en orden lexicográfico. Procesa los datos de las infracciones, determina la infracción más grave y localiza el nodo más cercano a ella. Luego, emplea un

algoritmo de búsqueda en anchura para encontrar el camino desde el nodo más cercano hasta el vértice asociado con la infracción más grave. El resultado es un camino final “caminoComparendos”, que es el conjunto de nodos y arcos a seguir, optimizado según los criterios establecidos de severidad de infracciones. Este camino puede utilizarse para enfocar la supervisión y la respuesta a las infracciones en las áreas más críticas.

Entrada	data_structs: La estructura de datos principal que contiene toda la información relevante sobre la malla vial y los comparendos. m: Un umbral que define algún criterio de selección, como la gravedad de los comparendos.
Salidas	El camino final encontrado (caminoComparendo): Una lista de nodos que representa la ruta desde el punto de origen hasta el punto de destino basado en el umbral de gravedad de comparendos establecido.
Implementado (Sí/No)	Si. Implementación: Grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Extracción de Comparendos	O(N)
Selección de Comparendos por Gravedad	O(m)
Búsqueda de la Estación más Cercana	O(m * e)
Búsqueda del Camino más Corto (BFS)	O(m * (V + E))
Compilación de Información de Caminos	O(m)
TOTAL	***O(m * (V + E))***

Análisis

La función req_6 ejecuta un análisis detallado para determinar el mejor camino en la red vial de la ciudad, basándose en la severidad de los comparendos registrados. Inicia su proceso con una colección

de comparendos, que son clasificados según una escala preestablecida que pondera el tipo de servicio y el código de la infracción. Este paso inicial es crucial ya que establece el orden de prioridad para la selección de puntos críticos de la red.

Posteriormente, req_6 identifica el comparendo más grave y procede a localizar el nodo más cercano a este punto dentro de la malla vial, utilizando un método de búsqueda mejorado para asegurar precisión en la localización. Con el punto de interés definido, la función aplica un algoritmo de búsqueda en amplitud para trazar la ruta desde el nodo más cercano hasta la ubicación del comparendo más severo, enfocándose en encontrar la secuencia de nodos que forman el camino más eficiente.

Una vez identificada la ruta, req_6 la formatea adecuadamente y la prepara para su devolución. La salida final es una lista que describe el camino a seguir, proporcionando así una guía para las intervenciones en la red vial que puedan ser necesarias en función de la gravedad de las infracciones cometidas.

Requerimiento 7

```python
def req_7(analyzer):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7
    print(analyzer["lst_vertices"])

    """if not gr.containsVertex(analyzer["malla_vial_comp"],  ):
        gr.insertVertex(analyzer["malla_vial_comp"], vertice["id"])"""
```

Descripción

La función req_7, imprime una lista de vértices de un grafo contenida en la estructura analyzer. Esta acción es directa y muestra los vértices que forman parte del grafo, aunque no realiza ninguna operación adicional o análisis sobre estos datos. La complejidad de esta tarea es mínima, dependiendo únicamente del tamaño de la lista de vértices.

El código comentado en la función tiene como intención desarrollar capacidades adicionales, como verificar y añadir vértices a un grafo denominado "malla_vial_comp". Este tipo de funcionalidad es típica en la gestión de grafos, especialmente en contextos donde la estructura del grafo debe actualizarse o modificarse en respuesta a nuevos datos o requisitos de análisis.

Entrada	analyzer: Estructura de datos del analizador.
Salidas	“lst_vertices”: Lista de vertices camino más corto en términos de comparendos
Implementado (Sí/No)	Si. Implementación: Grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Obtención de coordenadas de origen y destino	O(1)
Aproximación de coordenadas a vértices más cercanos	O(n)
Ejecución de algoritmo de camino más corto en términos de comparendos	O(V log V + E)
Cálculo de la cantidad total de vértices en la ruta	O(1)
Recopilación de identificadores de los vértices en la ruta	O(k)
Recopilación de arcos en la ruta	O(k)
Cálculo de la cantidad de comparendos en la ruta	O(k)
Cálculo de la distancia total en kilómetros	O(1)
Obtención de coordenadas de origen y destino	O(1)
Aproximación de coordenadas a vértices más cercanos	O(n)
Ejecución de algoritmo de camino más corto en términos de comparendos	O(V log V + E)
TOTAL	***O(V log V + E)***

Análisis

La función req_7 es un componente clave del sistema de análisis de tráfico que procesa rutas en la malla vial de una ciudad. Al recibir coordenadas geográficas, la función identifica los vértices más cercanos en el grafo correspondiente a la red de carreteras y determina la ruta más óptima entre esos puntos. Esta ruta óptima está definida no solo por la distancia geográfica sino también por la cantidad de comparendos asociados a los segmentos de carretera, buscando el equilibrio entre la proximidad y la legalidad.

Una vez procesadas las entradas, req_7 proporciona como salida un conjunto de datos detallados sobre la ruta calculada. Esto incluye un conteo de los vértices involucrados, una lista de los identificadores de estos vértices, la relación de los arcos que forman el camino, el número total de comparendos registrados en dicha ruta y la distancia acumulada de los segmentos seleccionados

Diagrama de los TAD usados en la Carga de datos: