ANÁLISIS DEL RETO

Tomas Acosta Romero. 202310218, t.acostar@uniandes.edu.co

Jose Rojas,202313911, j.rojasz@uniandes.edu.co

Leonardo Beltrán, al.acostar@uniandes.edu.co

Carga de datos

```
def load_data(control):

    tracemalloc.start()
    tiempo_carga = get_time()
    memoria_carga = get_memory()

    vertices = cf.data_dir + f"tickets/bogota_vertices.txt"
    arcos = cf.data_dir + f"tickets/bogota_arcos.txt"
    comparendos = cf.data_dir + f"tickets/comparendos_2019_bogota_vertices.csv"

    archivo_vertices = list(csv.reader(open(vertices, encoding="utf-8"), delimiter=","))
    archivo_arcos = list(csv.reader(open(arcos, encoding="utf-8"), delimiter=" "))
    archivo_comparendos = list(csv.reader(open(comparendos, encoding="utf-8"), delimiter=","))



    for linea in archivo_vertices:

        identificador, longitud, latitud = linea
        model.añadir_vertice(control, identificador)


    for linea in archivo_arcos:

        if len(linea) == 1:
            pass
        else:

            vertice_principal = linea[0]
            info_principal = archivo_vertices[int(vertice_principal)]

            vertices_a_conectar = linea[1:]

            for vertice in vertices_a_conectar:

                info_a_conectar = archivo_vertices[int(vertice)]

                lat1 = info_a_conectar[2]
                lon1 = info_a_conectar[1]
                lat2 = info_principal[2]
                lon2 = info_principal[1]

                distancia = model.haversine_function(lat1, lon1, lat2, lon2)
```

```
            model.añadir_arco_distancia(control, vertice, vertice_principal, distancia)

    control["vertices"] = archivo_vertices
    control["comparendos"] = archivo_comparendos


    tiempo_carga = delta_time(tiempo_carga, get_time())
    memoria_carga = delta_memory(get_memory(), memoria_carga)
    numero_vertices = model.gr.numVertices(control["malla_vial"])
    numero_arcos = model.gr.numEdges(control["malla_vial"])

    tracemalloc.stop()

    return tiempo_carga, memoria_carga, numero_vertices, numero_arcos
```

Análisis

La carga de datos inicializa los vértices dados por el archivo y los arcos. El costo de los arcos esta dado por la distancia entre dos vértices que se calcula usando la formula de haversine. Como la librería no permite guardar información dentro de los grafos, se crea una lista en la que el índice representa el vértice y ahí se guarda información importante como la latitud y la longitud del punto. También se crea una lista con información importante sobre los comparendos.

Requerimiento 1

Descripción

```
def req_1(control, latitud_origen, longitud_origen, latitud_destino, longitud_destino):
    """
    Función que soluciona el requerimiento 1
    """

    origen, destino = obtener_vertices_cercanos(control, latitud_inicial, longitud_inicial, latitud_final, longitud_final)

    grafo_a_recorrer = dfs.DepthFirstSearch(control["malla_vial"], origen)

    informacion = dfs.pathTo(grafo_a_recorrer, destino)

    return informacion
```

Entrada	Punto de origen y punto de destino
Salidas	Distancia entre el camino de los dos puntos, total de vértices de ese camino y la secuencia de ellos.
Implementado (Sí/No)	Grupalmente

Análisis de complejidad

Pasos	**Complejidad**
Determinar el vértice más cercano al punto de origen	O(V)
Determinar el vértice más cercano al punto de destino	O(V)
Implementar algoritmo de DFS	O(V + E)
TOTAL	O(V + E)

Análisis

Lo primero que hace este algoritmo es encontrar los vértices más cercanos al origen y al destino. Para esto inicializa el vértice más cercano al origen y el más cercano al destino y les almacena el primer vértice junto a su respectiva distancia calculada con la fórmula de haversine. Luego itera sobre la lista de vértices calculando la distancia respecto a las coordenadas de inicio y destino, y si es menor actualiza ya sea el vértice de origen o de destino. Todo esto trae una complejidad de O(V), donde V es el número de vértices del grafo. Luego el algoritmo implementa DFS que tiene una complejidad de O (V + E), dando una complejidad final de O(V+E)

Requerimiento 2

Descripción

Entrada	Punto de origen y punto de destino
Salidas	• La distancia total que tomará el camino entre el punto de encuentro de origen y el de destino. • El total de vértices que contiene el camino encontrado. • La secuencia de vértices (sus identificadores) que componen el camino encontrado
Implementado (Sí/No)	Grupalmente

Análisis de complejidad

Pasos	Complejidad
Determinar el vértice más cercano al punto de origen	O(V)
Determinar el vértice más cercano al punto de destino	O(V)
Implementar algoritmo de DFS	O(V + E)
TOTAL	O(V + E)

Análisis

El algoritmo es igual al primero pues implementa BFS, lo que garantiza que siempre se encuentre el camino con menor cantidad de niveles recorridos. La complejidad total es O(V+E), pues es lo que tarda en realizarse el algoritmo de BFS.

Requerimiento 3

Descripción

```python
def req_3(control, localidad, n_camaras):
    """
    Función que soluciona el requerimiento 3
    """

    comparendos = control["comparendos"]

    lista_comparendos_localidad = []
    indices_comparendos_localidad = {}

    for comparendo in comparendos:

        if comparendo["18"] not in indices_comparendos_localidad:

            lista_comparendos_localidad.append({"vertice": comparendo["18"], "comparendos": 1})
            indices_comparendos_localidad[comparendo["18"]] = len(lista_comparendos_localidad) - 1

        else:

            lista_comparendos_localidad[indices_comparendos_localidad[comparendo["18"]]]["comparendos"] += 1


    lista_comparendos_localidad.sort(key = lambda x: x["comparendos"], reverse = True)

    vertices_subgrafo = lista_comparendos_localidad[:n_camaras]

    grafo_subgrafo = gr.newGraph()

    for vertice in vertices_subgrafo:

        gr.insertVertex(grafo_subgrafo, vertice["vertice"])

    for i in range(len(vertices_subgrafo)):

        for j in range(i + 1, len(vertices_subgrafo)):

            distancia = haversine_function(vertices_subgrafo[i]["vertice"][2], vertices_subgrafo[i]["vertice"][1], vertices_subgrafo[j]["vertice"][2], vertices_subgrafo[j]["vertice

            gr.addEdge(grafo_subgrafo, vertices_subgrafo[i]["vertice"], vertices_subgrafo[j]["vertice"], distancia)

    prim_mst = prim.PrimMST(grafo_subgrafo)

    return prim_mst
```

Entrada	• Cantidad de cámaras de video a instalar • Localidad donde se desean instalar
Salidas	o El total de vértices de la red. o Los vértices incluidos (identificadores). o Los arcos incluidos (Id vértice inicial e Id vértice final). o La cantidad de kilómetros de fibra óptica extendida. o El costo (monetario) total
Implementado (Sí/No)	Tomas Acosta Romero

Análisis de complejidad

Pasos	**Complejidad**
Crear lista con los vértices que contienen comparendos y son de la localidad dada	O(V)
Ordenar la lista de más a menos comparendos	O(nlogn), donde n es el numero de vértices con comparendos de la localidad
Crear subgrafo con los 5 vértices y las distancias subterráneas asociadas a ellos	O(v + e)
Implementar algoritmo de Prim	O(v^2)
Complejidad Total	***O(V)***

Análisis

El algoritmo itera sobre la lista de comparendos para hallar cuales fueron hechos en la localidad, después de tenerlos los añade a los vértices y los ordena de acuerdo a cuales tienen más comparendos, después de esto toma los que diga el numero de cámaras y crea un subgrafo completo en el cual la distancia es la formula de haversine y con esto puede implementar el algoritmo de prim que retorna el menor MST.

Requerimiento 4

Descripción

Para realizar el requerimiento, se modela la ciudad como un grado, donde los nodos son las posibles ubicaciones para las cámaras y los arcos la conexión entre esos lugares, luego a través de las especificaciones respecto a la gravedad de las infracciones modelada en el reto, se toma en cuenta el más grave y luego los demás. Si surge un empate se analizará según su categoría para determinar la gravedad

Entrada	• La cantidad de cámaras de video que se desean instalar (M)
Salidas	o El total de vértices de la red. o Los vértices incluidos (identificadores). o Los arcos incluidos (Id vértice inicial e Id vértice final). o La cantidad de kilómetros de fibra óptica extendida. o El costo (monetario) total.
Implementado (Sí/No)	

Análisis de complejidad

Pasos	**Complejidad**
Determinar el número de cámaras a instalar	O(V)
Implementar algoritmo de PRIM	O(ElogV)
Busqueda por amplitud	O(V)
TOTAL	O(ElogV)

Análisis

Para este requerimiento se utiliza un Árbol de Cobertura mínima para unir los nodos de manera efectiva, luego de recibir el número de cámaras por el usuario, se genera el MST de una red de comunicaciones y luego se hace una búsqueda en la amplitud tomando en cuenta la gravedad de las infracciones. La complejidad es la del algoritmo PRIM, debido a que todo el algoritmo se basa en el MST

Requerimiento 5

Descripción

Para poder realizar una instalación eficaz a través de fibra óptica de cámaras de video en M sitios con el menor costo posible, se propone modela a la ciudad como un grafo, donde los vértices son las ubicaciones de las cámaras y las aristas son las conexiones entre ellos. Así podemos usar la información de comparendos, luego de solicitar una clase de vehículo definida por el usuario, para hayar los M vértices con más infracciones

Entrada	• La cantidad de cámaras de video que se desean instalar (M), y • Clase de vehículo
Salidas	o El total de vértices de la red. o Los vértices incluidos (identificadores). o Los arcos incluidos (Id vértice inicial e Id vértice final). o La cantidad de kilómetros de fibra óptica extendida. o El costo (monetario) total.
Implementado (Sí/No)	

Análisis de complejidad

Pasos	**Complejidad**
Encontrar las M ubicaciones críticas	O(V)
Implementar PRIM	O(ElogV)
Lista con los nodos y aristas	O(V)
TOTAL	O(ElogV)

Análisis

El algoritmo crea una red de comunicaciones mediante la instalación de M cámaras en M ubicaciones críticas, el usuario proporciona el número M y el vehículo a considerar, primero se ejecuta el PRIM que es el árbol de cobertura mínima, y con él se buscan los nodos asociados a la clase de vehículo y cuáles son los M con mayores infractores, luego en un listado se incluyen los nodos y las aristas

Requerimiento 6

Descripción

Para resolver la tarea de M policías, se propone modelar la infraestructura de la ciudad en un grafo, las estaciones como nodos, y los arcos como son las vías entre ellos. La gravedad de cada comparendo se determina mediante la asignación de recursos mencionados en el reto.

Entrada	• La cantidad de comparendos que se desea responder (M) • La estación de policía más cercana al comparendo más grave.
Salidas	o El total de vértices del camino. o Los vértices incluidos (identificadores). o Los arcos incluidos (Id vértice inicial e Id vértice final). o La cantidad de kilómetros del camino
Implementado (Sí/No)	Grupalmente

Análisis de complejidad

Pasos	**Complejidad**
Creación y asignación de variables	O(V)
Encontrar el vértice de origen y la distancia hacia el siguiente	O(V)
Implementar algoritmo de Dijkstra	O((V+E)*log(V))
Retorno	O(1)
TOTAL	O((V+E)*log(V))

Análisis

Se implementa el dijkstra para saber cuál es la ruta más rapida desde un punto M, hasta la estación más cercana

Requerimiento 7

Descripción

Para implementar el requerimento, se toman los datos de longitud y latitud deseados por el conductor, se modela la ciudad como un grafo con nodos y arcos según locaciones y su distancia. Se aplica dijkstra, con el se encuentra la ruta más corta y la distancia por recorrer.

Entrada	• Punto de origen (una localización geográfica con latitud y longitud). • Punto de destino (una localización geográfica con latitud y longitud).
Salidas	o El total de vértices del camino. o Los vértices incluidos (identificadores). o Los arcos incluidos (Id vértice inicial e Id vértice final). o La cantidad de comparendos del camino. o La cantidad de kilómetros del camino.

Implementado (Sí/No)	Grupalmente

Análisis de complejidad

Pasos	**Complejidad**
Buscar vertices	O(V)
Creación de variables	O(1)
Dijkstra	O((V + E)*log(V))
TOTAL	O((V+E)*log(V))

Análisis

Se usa la estructura de datos, y buscar vertice para encontrar y relacionar los vertices ingresados por el usuario según longitud y latitud, se inicializan estructuras de datos a las cuales se les aplica dijkstra, para calcular el camino más corto, y finalmente se construye el camino contando los vetices recorridos y la distancia en KM