ANÁLISIS DEL RETO

Yohan Felipe Gaitan Carvajal 202312115
y.gaitan@uniandes.edu.co

Requerimiento <<1>>

Descripción

```python
def req_1(catalog, vertices, lat1, long1, lat2, long2):
    """
    Función que soluciona el requerimiento 1
    """
    try:
        mapa_ids = catalog["hash_ids"]
        malla_vial = catalog["malla_vial"]
        distancia_total = 0
        total_vertx = 0
        camino_total = lt.newList("ARRAY_LIST")
        lista_bono = []
        vertice_destino = closest_vertx(vertices, lat1, long1)
        vertice_origen = closest_vertx(vertices, lat2, long2)
        search = dfs.DepthFirstSearch(malla_vial, vertice_origen)
        haspath = dfs.hasPathTo(search, vertice_destino)

        if haspath:
            camino = dfs.pathTo(search, vertice_destino)
            prev = None
            for vertice in lt.iterator(camino):
                lat, long = lat_long_bono(mapa_ids,vertice)
                lista_bono.append((lat,long))
                total_vertx +=1
                lt.addLast(camino_total,vertice)
                if prev is not None:
                    arco = gr.getEdge(malla_vial, vertice, prev)
                    peso = arco["weight"]
                    distancia_total += peso
                prev = vertice

        req_8(lista_bono, 1)
    except:
        print("Error, revise nuevamente los datos")
    return camino_total,distancia_total,total_vertx
```

La función del req_1 recibe por parámetro la estructura de datos previamente creada, identificador del

punto de encuentro de origen e identificador del punto de encuentro de destino, donde debemos encontrar si existe un camino entre estos dos puntos en la estructura de datos con toda la información del archivo csv.

Entrada	Estructura de datos, array_list de vertices, latitud y longitud 1 y latitud y longitud 2
Salidas	Una lista con la ruta que conecta los dos puntos, la distancia total y el total de losvértices
Implementado (Sí/No)	Si, Yohan Felipe Gaitan Carvajal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Closest_vertx(vértices,lat1, long1)	O(n)
Closest_vertx(vértices,lat2, long2)	O(n)
dfs.DepthFirstSearch(malla_vial, vertice_origen)	O(V + E)
Dfs.pathTo(search, vertice_destino)	O(m)= siendo el tamaño de la variable “recorrido”
For vertice in lt.iterator(camino)	O(m)
TOTAL	O(V + E + n + m)

Requerimiento <<2>>

Descripción

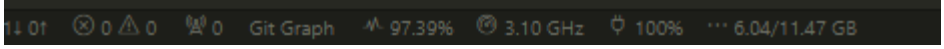
```python
def req_2(catalog, vertices, lat1, long1, lat2, long2):
    """
    Función que soluciona el requerimiento 2
    """

    mapa_ids = catalog["hash_ids"]
    malla_vial = catalog["malla_vial"]
    distancia_total = 0
    total_vertx = 0
    camino_total = lt.newList("ARRAY_LIST")
    lista_bono = []

    vertice_origen = closest_vertx(vertices, lat1, long1)
    vertice_destino = closest_vertx(vertices, lat2, long2)
    search = bfs.BreathFirstSearch(malla_vial, vertice_origen)
    haspath = bfs.hasPathTo(search, vertice_destino)

    if haspath:
        camino = bfs.pathTo(search, vertice_destino)
        prev = None
        for vertice in lt.iterator(camino):
            lat, long = lat_long_bono(mapa_ids,vertice)
            lista_bono.append((lat,long))
            total_vertx +=1
            lt.addLast(camino_total,vertice)
            if prev is not None:
                arco = gr.getEdge(malla_vial, vertice, prev)
                peso = arco["weight"]
                distancia_total += peso
            prev = vertice

    req_8(lista_bono, 2)
    return camino_total,distancia_total,total_vertx
```



Breve descripción de como abordaron la implementación del requerimiento

La función del req_2 recibe por parámetro la estructura de datos previamente creada, identificador del punto de encuentro de origen e identificador del punto de encuentro de destino, donde debemos encontrar el camino con menor puntos de seguimiento entre los dos puntos dados.

Entrada	Estructura de datos, vertices, latitud y longitud 1 y latitud y longitud 2
Salidas	La ruta con menos interesecciones entre los 2 putos definidos
Implementado (Sí/No)	Si, Yohan Felipe Gaitan

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Closest_vertx(vértices,lat1, long1)	O(n)
Closest_vertx(vértices,lat2, long2)	O(n)
bfs.breathFirstSearch(malla_vial, vertice_origen)	O(V + E)
Dfs.pathTo(search, vertice_destino)	O(m)= siendo el tamaño de la variable "recorrido"
For vertice in lt.italratos(camino)	O(m)
TOTAL	O(V + E + n + m)

Requerimiento <<4>>

Descripción

```python
def req_4(catalog, comparendos_ordenados, camaras, bono):
    """
    Función que soluciona el requerimiento 4
    """
    lista_bono = []
    arcos = lt.newList("ARRAY_LIST")
    vertices = lt.newList("ARRAY_LIST")
    i = 2
    g_distancia = catalog["malla_vial"]
    mayor_gravedad = lt.firstElement(comparendos_ordenados)
    print("Cargando MST...")
    search = prim.PrimMST(g_distancia, mayor_gravedad["VERTICES"])
    peso = prim.weightMST(g_distancia, search)

    subgrafo = gr.newGraph(datastructure="ADJ_LIST", directed = False, cmpfuntion = compare_id)
    mst = search["mst"]
    print("Reconstruyendo el grafo...")
    for minicamino in lt.iterator(mst):
        add_vertx(subgrafo, minicamino["vertexA"])
        add_vertx(subgrafo, minicamino["vertexB"])
        add_edge(subgrafo, minicamino["vertexA"], minicamino["vertexB"], minicamino["weight"])

    sub_bfs = bfs.BreathFirstSearch(subgrafo, mayor_gravedad["VERTICES"])
    grafo_camaras = gr.newGraph(datastructure="ADJ_LIST", directed = False, cmpfuntion = compare_id)
    print("Filtrando las  M camaras...")

    while i <= camaras:
        info_c = lt.getElement(comparendos_ordenados, i)
        camino = bfs.pathTo(sub_bfs, info_c["VERTICES"])
        prev = None
        for vertice in lt.iterator(camino):
            if prev == None:
                lt.addLast(vertices, vertice)
                prev = vertice
            else:
                minipath = {"vertexA" : prev, "vertexB": vertice,"weight" : 0 }
                prev = vertice
```

```
        for vertice in lt.iterator(camino):
            if prev == None:
                lt.addLast(vertices, vertice)
                prev = vertice
            else:
                minipath = {"vertexA" : prev, "vertexB": vertice,"weight" : 0 }
                prev = vertice
                v1 = minipath["vertexA"]
                v2 = minipath["vertexB"]
                v1_info = me.getValue(mp.get(catalog["hash_ids"], v1))["info"]
                v2_info = me.getValue(mp.get(catalog["hash_ids"], v2))["info"]
                minipath["weight"] = distance(v1_info[1], v1_info[0],v2_info[1], v2_info[0])

                if not gr.getEdge(grafo_camaras, minipath["vertexA"], minipath["vertexB"]):
                    lt.addLast(arcos, minipath)
                    weight += minipath["weight"]
                if not gr.containsVertex(grafo_camaras, minipath["vertexA"]):
                    lt.addLast(vertices, minipath["vertexA"])
                if not gr.containsVertex(grafo_camaras, minipath["vertexB"]):
                    lt.addLast(vertices, minipath["vertexB"])
                add_vertx(grafo_camaras, minipath["vertexA"])
                add_vertx(grafo_camaras, minipath["vertexB"])
                add_edge(grafo_camaras, minipath["vertexA"], minipath["vertexB"], minipath["weight"])
                if bono:
                    lat1_bono = v1_info[1]
                    long1_bono = v1_info[0]
                    lat2_bono = v2_info[1]
                    long2_bono = v2_info[0]
                    lista_bono.append([[lat1_bono,long1_bono], [lat2_bono,long2_bono]])
        i +=1
    if bono:
        req_8(lista_bono, 4)
    return vertices, arcos, weight
```

En esta función se utilizó el algoritmo de prim y también la ecuación de haversine para poder realizar la busque de la ruta más corta entre 2 puntos mediante el grafo

Entrada	Estructura de datos, Longitud y latitud punto inicial, Longitud y latitud punto final
Salidas	La red de comunicación con las M cámaras en los puntos con mayor gravedad
Implementado (Sí/No)	Si, implementado por Felipe Gaitan

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
PRIM	O(E log V)
Crear un grafo	O(V + E)
Recorrer el MST	O(V)
Recorrer las cámaras	O(M)
BFS	O(V + E)
Bfs.pathTo	O (V + E)
Recorrer el camino del BFS	O (V)
TOTAL	***O(E LOG V + (V + E) + V + M)***

Requerimiento <<5>>

Descripción

No implementado - Carlos llain

Requerimiento <<6>>

Descripción

```python
def req_6(catalog, numero_comparendos, comparendos_ordenados, bono):
    """
    Función que soluciona el requerimiento 6
    """
    map_stations = catalog["djk_estaciones"]
    i = 0
    lista_bono = []
    camino_total = lt.newList("ARRAY_LIST")
    while i <= numero_comparendos:
        print("Obteniendo  camino de comparendo #" + str(i))
        info_comparendo = lt.getElement(comparendos_ordenados, i)

        vertx_comparendo = info_comparendo["VERTICES"]
        info_vertx_comparendo = me.getValue(mp.get(catalog["hash_ids"], vertx_comparendo))
        closest_estacion_info= info_vertx_comparendo["closest_estacion"]
        entry = mp.get(map_stations, closest_estacion_info["EPONOMBRE"])
        if entry:
            search = me.getValue(entry)
        else:
            subgrafo = me.getValue(mp.get(catalog["grafo_estaciones"], closest_estacion_info["OBJECTID"]))
            search = djk.Dijkstra(subgrafo, closest_estacion_info["VERTICES"])
            mp.put(map_stations, closest_estacion_info["OBJECTID"], search)
        camino = djk.pathTo(search, info_comparendo["VERTICES"])
        info_camino = {
            "estacion": closest_estacion_info["EPONOMBRE"],
            "comparendo" : info_comparendo,
            "vertice_comparendo" : vertx_comparendo,
            "total_vertx" : 0,
            "identificadores" : [],
            "arcos" : [],
            "km" : 0
        }
        lt.addLast(camino_total, info_camino)
```

```
    lt.addLast(camino_total, info_camino)
    for minipath in lt.iterator(camino):
        info_camino["total_vertx"] += 1
        if len(info_camino["identificadores"]) == 0:
            info_camino["identificadores"].append(minipath["vertexA"])
        info_camino["identificadores"].append(minipath["vertexB"])
        info_camino["km"] += minipath["weight"]
        info_camino["arcos"].append(minipath)
        if bono:
            v1 = minipath["vertexA"]
            v2 = minipath["vertexB"]
            v1_info = me.getValue(mp.get(catalog["hash_ids"], v1))["info"]
            v2_info = me.getValue(mp.get(catalog["hash_ids"], v2))["info"]
            lat1_bono = v1_info[1]
            long1_bono = v1_info[0]
            lat2_bono = v2_info[1]
            long2_bono = v2_info[0]
            lista_bono.append([[lat1_bono,long1_bono], [lat2_bono,long2_bono]])

    i +=1
return camino_total
```

Entrada	catalog, numero_comparendos, estacion más cercana y comparendos_ordenados
Salidas	EL camino más corto para que los policías atiendas los M comparendos mas graves.
Implementado (Sí/No)	Si, Yohan Felipe Gaitan Carvajal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
while i <= numero_comparendos:	O(M) M= número de comparendo
DJKSTRA	O((V + E) * log(V))
djk.pathTo	O(V)
for minipath in lt.iterator(camino):	O(C) C = camino
TOTAL	**O(M + V + C + ((V + E) * log(V))**

Requerimiento <<7>>

Descripción

```
def req_7(catalog, vertices,  lat1, long1, lat2, long2, bono):
    """
    Función que soluciona el requerimiento 7
    """
    bono = bono_check(bono)
    total_multas = 0
    distancia_total = 0
    vertices_totales = 0
    camino_total = lt.newList("ARRAY_LIST")
    lista_bono = []

    g_comparendos = catalog["grafo_comparendos"]
    vertx_origen = closest_vertx(vertices, lat1, long1)
    vertx_destino = closest_vertx(vertices, lat2, long2)
    search = djk.Dijkstra(g_comparendos, vertx_origen)
    haspath = djk.hasPathTo(search, vertx_destino)

    if haspath:
        camino = djk.pathTo(search, vertx_destino )
        prev = None
        for vertice in lt.iterator(camino):
            vertices_totales += 1
            if prev == None:
                lt.addLast(camino_total, vertice["vertexA"])
            lt.addLast(camino_total, vertice["vertexB"])
            total_multas += vertice["weight"]
            v1 = vertice["vertexA"]
            v2 = vertice["vertexB"]
            v1_info = me.getValue(mp.get(catalog["hash_ids"], v1))["info"]
            v2_info = me.getValue(mp.get(catalog["hash_ids"], v2))["info"]
            distancia = distance(v1_info[1], v1_info[0],v2_info[1], v2_info[0])
            distancia_total += distancia
            prev = vertice
            if bono:
                lat1_bono = v1_info[1]
                long1_bono = v1_info[0]
                lat2_bono = v2_info[1]
                long2_bono = v2_info[0]
                lista_bono.append([[lat1_bono,long1_bono], [lat2_bono,long2_bono]])
        if bono:
            req_8(lista_bono, 7)
    else:
        print("No hay camino")

    return distancia_total, vertices_totales, camino_total, total_multas
```

Entrada	catalog, vertices, lat1, long1, lat2, long2, bono
Salidas	EL camino mas corto para que los conductores transiten con la ruta con menor cantidad de comparendos
Implementado (Sí/No)	Si, Yohan Felipe Gaitan Carvajal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Closest_vertx	O(V)
DIJKSTRA	O((V+E) LOG V)
Djk.hasPathTo	O (1)
Djk.pathTo	O(V)
Lt.iterator(camino)	O(C)
TOTAL	**O(V + C + ((V+E) LOG V)**