ANÁLISIS DEL RETO

Tomas Diaz, 202220658, t.diazv@

Samuel Peña, 202028273, ss.pena@

Manuel Pinzon, 202125748, mma.pinzonpi@

Carga de datos:

Diagrama:

En términos generales el data structs consta de 9 elementos:

1. lista total
2. lista archivo lobos
3. mapa lobos
4. mapa archivo lobos
5. mapa localización
6. mapa nodos de seguimiento
7. mapa nodos de encuentro
8. grafo
9. grafo NO dirigido

La explicación a detalle se encuentra en el diagrama gráfico, adjunto en los anexos.

Nota: Pregunté por discord varias veces y nunca nadie me dijo que no se podía hacer a mano.

Nota: para los reqs 6 y 7, se filtra el mapa lobos por fechas y/o temperaturas y a partir de este se carga de nuevo el data structs filtrado, creándose grafos reducidos.

Descripción

La carga general se divide en 2 partes: 1.carga de archivo tracks y 2.carga de archivo individuals.

CARGA TRACKS:

Primero se crea el data structs vacío:

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

```
    data_structs['lista total']=lt.newList(datastructure='ARRAY_LIST')
#### mapa cuya llave es un lobo en id y el valor es el array de los eventos
seguidos por el dicho lobo (ordenado)
    data_structs['mapa lobos']= None
###mapa con llave localización (latLong compuesto como se indica) y valor array
de eventos con esa localización
    data_structs['mapa localizacion']=None
##mapa cuya llave es el indicador del nodo(coordenada) y el valor una lista de
nodos de seguimiento asociados a dicho nodo
    data_structs['mapa nodos de encuentro']=None
    data_structs['grafo']=gr.newGraph(directed=True)

    data_structs['lista archivo lobos']=lt.newList(datastructure='ARRAY_LIST')

    data_structs['grafo no dirigido']=gr.newGraph(directed=False)
    return(data_structs)
```

Luego cargamos la lista total de eventos en 'lista total':

```
def add_data(data_structs, data):
    """
    Función para agregar nuevos elementos a la lista
    """
    lt.addLast(data_structs['lista total'],data)
```

Ahora, creamos el resto de las estructuras como se indica en la imagen:

```
######                      CREAR GRAFO CON LAS ESPECIFICACIONES CORRESPONDIENTES



def crear_grafo(data_structs):

    ###a. Redondear:
    ### recoge lista total y redondea las coordenadas a 3 decimales
    redondear_lista_total(data_structs)
    ### Añade a cada evento su coordenada asociada, su nodo de seguimiento
asociado y el id-individual
    poner_coordenada_en_formato_a_evento_Y_asociarlo_con_nodo_de_seguimiento(data
_structs)

    ###b. mapa lobos
    ## Crea mapa con llave animal-id y valor array ordenado por fecha de los
eventos asociados a ese lobo
    crear_mapa_lobos(data_structs)
```

```
    ##de cada array un elemento si el anterior tiene su misma coordenada.
    filtrar_mapa_lobos(data_structs)

    ###C. mapa coordenadas
    ## mediante iteración del mapa lobos, crea mapa con llave coordenada y valor
array eventos en dicha coordenada
    crear_mapa_coordenadas(data_structs)

    #####D. mapa nodos de seguimiento
    ## Iterando mapa lobos, crea mapa de nodos de seguimiento cuyo valor es la
coordenada.
    crear_nodos_de_seguimiento(data_structs)

    ###E. Crear nodos de encuentro
    ### Iterando mapa de coordenadas, crea mapa con llave nodo-id y valor array
de nodos de seguimiento adyacentes.
    crear_nodos_de_encuentro(data_structs)

    ####F. Poner nodos  en grafo
    ##Itera los mapas y pone los nodos en el grafo
    poner_nodos__en_grafo(data_structs)

    ###G. Crear arcos entre nodos de seguimiento

    ## Itera mapa lobos para crear arcos de nodos de seguimiento
    crear_arcos_nodos_seguimiento(data_structs)

    ####H. Crear arcos para los nodos de encuentro
    ###Itera mapa nodos de encuentro para poner arcos adyacentes.
    poner_arcos_encuentro(data_structs)


    ##I. rectangulo de area requerido para view
    anadir_menor_mayor_lat_log(data_structs)
    return data_structs
```

Nota: Todas las funciones reciben y devuelven el data_structs.

Nota: Para reqs 6 y 7 se filtra el data_structs. Primero se filtra el mapa lobos y a partir de este se vuelve a crear el data structs. Las funciones de filtrado filtran cada uno de los arrays ordenados del mapa lobos como sigue:

```
##### Funciones para 6 y 7 de filtrar array_ordenado de eventos por rango de
fechas y temperaturas, devuelven el arrray
### ordenadp por tiempo con los eventos dentro del rango
def array_ordenado_filtrado_por_rango_fechas(array,fecha1,fecha2):
```

```
    ### Devuelve un array filtrado ordenado de los eventos en ese rango
    fecha_in=float(fecha1.replace(':','').replace('-','').replace(' ',''))
    fecha_fin=float(fecha2.replace(':','').replace('-','').replace(' ',''))
    array_filt=lt.newList(datastructure='ARRAY_LIST')
    size=lt.size(array)

    i=1
    while i<=size:
        evento= lt.getElement(array,i)
        fecha=float(evento["timestamp"].replace(':','').replace('-','').replace('
','',''))
        if fecha >=fecha_in and fecha<=fecha_fin:
            lt.addLast(array_filt,evento)

        if fecha>fecha_fin:
            break

        i+=1

    return array_filt

## Filtra array por rango de temperatura
def filtrar_array_por_temp(array,temp1,temp2):
    temp_in=float(temp1)
    temp_fin=float(temp2)
    array_filt=lt.newList(datastructure='ARRAY_LIST')
    size=lt.size(array)

    i=1
    while i<=size:
        evento= lt.getElement(array,i)
        temp=float(evento["external-temperature"])
        if temp >=temp_in and temp <=temp_fin:
            lt.addLast(array_filt,evento)
        i+=1
    return array_filt
```

Carga individuals:

Al igual que en el caso de los tracks, primero cargamos los archivos a un array en lista total archivo y luego creamos un mapa archivo lobos con llave el individual-id y volar un dic con los datos del lobo:

```
def cargar_archivo_lobos(data_strucst):
    lista_lobos =data_strucst['lista archivo lobos']
```

```
    anadir_individual_id_a_lobo(data_strucst)
    data_strucst['mapa archivo
lobos']=crear_mapa_de_columna_a_partir_de_ARRAy(lista_lobos,'individual-id')

    return data_strucst
```

Requerimiento 1

Plantilla para el documentar y analizar cada uno de los requerimientos.

```
def req_1(data_structs , origen, destino):
    """
    Función que soluciona el requerimiento 1
    """
    # TODO: Realizar el requerimiento 1
    respuesta = lt.newList("ARRAY_LIST")
    grafo = data_structs["grafo"]
    result = dfs.DepthFirstSearch(grafo, origen)
    tiene = dfs.hasPathTo(result, destino)
    if tiene == False:
        respuesta = None
    else:
        pila = dfs.pathTo(result, destino)
        dist = aux_tam(grafo, pila)
        lt.addLast(respuesta, dist)
        tam = lt.size(pila)
        lt.addLast(respuesta, tam)
        t5 = aux_t5(pila, 5)
        lt.addLast(respuesta, t5)

    return respuesta
```

Descripción

Para este requerimiento usamos el algoritmo DFS para ver si hay un camino entre los dos puntos y en el caso de que si, se usa este mismo algoritmo para ver el recorrido

Entrada	Recibe los datos, el punto de origen y el punto de destino
Salidas	Retorna una lista en donde se encuentra el camino con la información que se pidio
Implementado (Sí/No)	Si, Manuel Pinzon

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Paso 1: Hace el recorrido DFS	O(v+A)
Paso 2 : Se busca si hay camino en entre los puntos	O(V+A)
Paso 3: Se recorre el camino entre los puntos	O(V)
Paso 4: Se saca el peso de todo el recorrido	O(V)
Paso 5: Se sacan los 5 primero y 5 ultimos vertices	O(1)
TOTAL	***O(V+A)***

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Entrada	Tiempo (s)

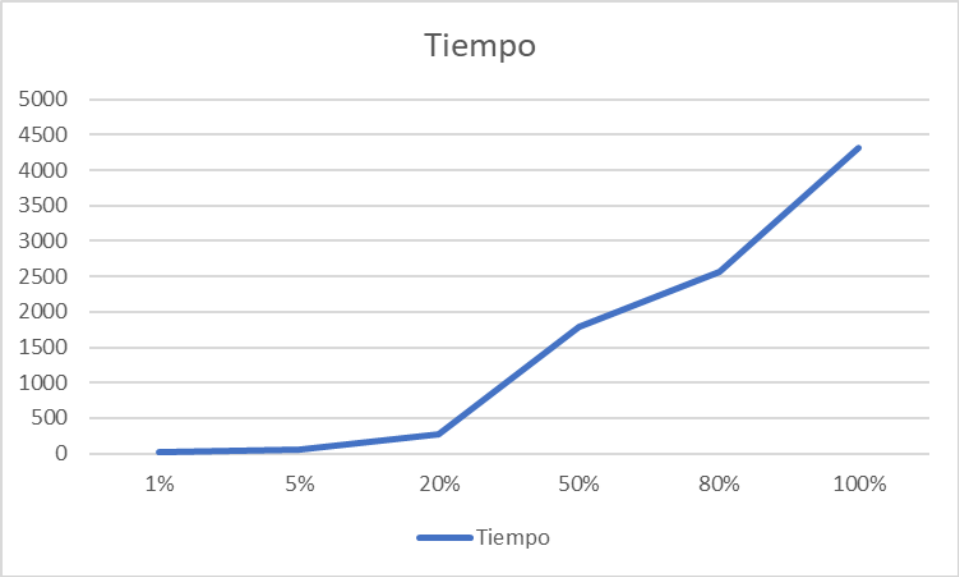
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Tamaño	Tiempo
1%	15.39
5%	61.34
20%	278.69
50%	1784.51
80%	2564.89
100%	4325.35

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Gracias a las funciones que hay pertenecientes a los grafos se facilitó mucho la complejidad y el codigo necesario para esta función.

Requerimiento 2

Plantilla para el documentar y analizar cada uno de los requerimientos.

```
def req_2(data_structs, nodo1, nodo2):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    busqueda = dfs.DepthFirstSearch(data_structs["model"]["grafo"], nodo1)
    hay = dfs.hasPathTo(busqueda,nodo2)
    if hay == True:
        como = dfs.pathTo(busqueda,nodo2)
    lista = []

    i = 1
    lista = lt.newList()
    while i <= 5:
        pos = lt.getElement(como,i)
        lt.addFirst(lista, pos)
        i +=1

    a = 4
    while a >= 0:
        size = lt.size(como) - a
        pos = lt.getElement(como, size)
        lt.addFirst(lista,pos)
        a -=1

    res = []
    vez = 1
    menos = 0
    for valor in lt.iterator(lista):
        dic = {}
        cada_una = separar(valor)
        dic["Location long-aprox"] = cada_una[0]
        dic["location lat-aprox"] = cada_una[1]
        dic["node-id"] = valor
```

```
    siguiente = adelante(como, vez- menos)
    if len(cada_una) == 2:
        cuantos = devolver_value(data_structs["model"]["mapa nodos de encuentro"], valor)
        size = lt.size(cuantos)

        indvid = []
        for individual in lt.iterator(cuantos):
            indvid.append(individual)

        dic["individual-id"] = indvid
        dic["individual-count"] = size

    else:
        dic["individual-id"] = cada_una[2] + "_" + cada_una[3]
        dic["individual-count"] = 1

    dic["edge-to"] = siguiente
    if vez <= 5:
        edge = gr.getEdge(data_structs["model"]["grafo"], valor, siguiente)
        if edge != None:

            dist = edge["weight"]
        else:
            dist = 0
    elif vez < 10:
        edge = gr.getEdge(data_structs["model"]["grafo"], valor, siguiente)
        if edge != None:

            dist = edge["weight"]
        else:
            dist = 0
    else:
        dist = "unknown"

    dic["edge distance- km"] = dist
    vez +=1
    res.append(dic)
return res, lista
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	Nodo 1(donde se quiere iniciar la busqueda), nodo 2 (donde quiere llegar la busqueda)
Salidas	Los 5 primeros nodos con los que se cruza y los ultimos 5
Implementado (Sí/No)	Tomas diaz

Para abordar este problema se invoco a la funcion dfs para saber cual es la conexión con menor numero de punto entre dos dados, se usa el menor como el principal y el segundo se usa con la funcion get path to para saber como se llega, despues se organiza como se pide en la instrucción y se le agregan las cosas que se necesitan

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1 : crear un dfs como nodo principal el dado	O(V+E)
Paso 2: hay es para saber si hay un camino entre los arcos	O(n) cuantos hayan en la pila
Paso 3: devolver ese camino	O(N) cuantos hayan en la pila
Paso4: agregar los primeros 5 a la respuesta	O(5)

Paso 5. agregar los ultimos 5 a la respuesta	O(5)
Paso 6: se convierte el nodo en latitud y longitud	O(1)
Paso 7: se encuentra de donde viene el anterior	O(n) se encuentra donde viene el anterior invocando la lista en según la posicion que se busca
Paso 8: se encuentra el peso de los arcos	O(v+ e) se busca las dos cordenadas y se devuelve su peso
Paso 9: se pone todos los datos para poderlos tabulear y se le agregan todas las cosas que se le piden	
Paso ….	O(...)
TOTAL	***O(V+E)***

Pruebas Realizadas

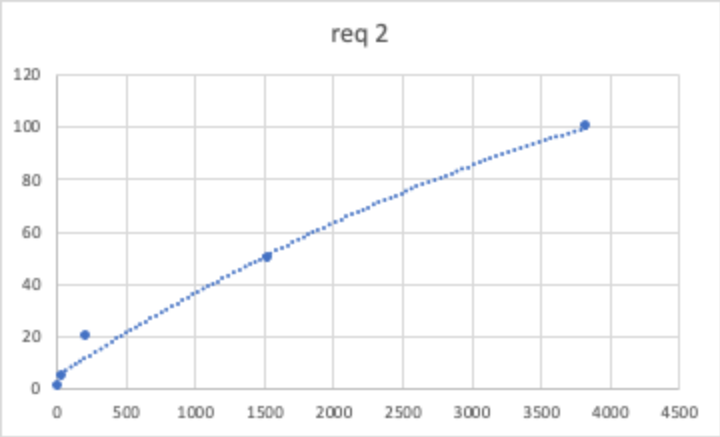
Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

“m111p862_57p449” y “m111p908_57p427”. Los datos utilizados

porcentaje	tiempo
1	13,959
5	46,64
20	219,94
50	1528,61
100	3824,056

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.

En este caso se obtiene una de las mejores complejidades, porque la mayor complejidad que hay en el caso es la de la implementacion de la funcion dfs y pues como es clase de estructura de datos, es la que mejor se nos puede brindar en la libreria

Requerimiento 3

Plantilla para el documentar y analizar cada uno de los requerimientos.

```
def req_3(data_structs):
    """
    Función que soluciona el requerimiento 3
    """
    # TODO: Realizar el requerimiento 3
    grafo=data_structs["model"]['grafo']
    kosaraju = scc.KosarajuSCC(grafo)
    "los puntos conectados "
    total = scc.connectedComponents(kosaraju)
    keys = mp.keySet(kosaraju["idscc"])
    mapa = mp.newMap()

    for manada in lt.iterator(keys):
        "invertir las llaves como valores dentro de una lista y el valor se volvio la llave"
        actual = devolver_value(kosaraju["idscc"],manada)
        esta = mp.contains(mapa, actual)
        if esta == False:
            lista = lt.newList()
            lt.addFirst(lista,manada)
            mp.put(mapa,actual,lista)
        else:
            agregar = devolver_value(mapa, actual)
            lt.addLast(agregar, manada)


    llaves_scc = mp.keySet(mapa)
    i = 1
    final = lt.newList()
```

```
    while i <= 5:
        mayor = 0
        sccc = 0
        a = 1

        while a <= lt.size(llaves_scc):
            sccdid = lt.getElement(llaves_scc,a)
            cantidad_list = devolver_value(mapa,sccdid)
            if lt.size(cantidad_list) > mayor:
                mayor = lt.size(cantidad_list)
                sccc = sccdid
                pos = a
            a += 1
        lt.addLast(final,sccc)
        lt.deleteElement(llaves_scc, pos)
        i +=1

    ' ir poniendo requerimiento por requerimiento'
    valor = pedido(data_structs["model"],mapa,final)

    return total, valor

def pedido( data_structs, mapa, lista_mejores):
    valor = []
    for ultima in lt.iterator(lista_mejores):
        respuesta = {}
        respuesta["SCCID"] = ultima
        respuesta["NODEIDS"] = node_ids(mapa,ultima)
        lista = devolver_value(mapa,ultima)
        respuesta["SCC Size "] = lt.size(lista)
        max_mins = encontrar(lista, data_structs["mapa archivo lobos"])
        respuesta ["min-lat"] = max_mins[0]
        respuesta ["max-lat"] = max_mins[1]
        respuesta ["min-lon"] = max_mins[2]
        respuesta ["max-lon"] = max_mins[3]
        respuesta["wolf Count"] = max_mins[4]
        respuesta["Wolf details"] = max_mins[5]
        valor.append(respuesta)
    return valor
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	Este requerimiento no tiene entradas
Salidas	Las 5 manadas con mayor dominio sobre el territorio
Implementado (Sí/No)	Tomas diaz

En esta funcion se invoca a kosaraju para encontrar los componentens fuertenementes conectados, despues se crea un diccionario con el proposito de agruparlo por el sccid y de valor dejar una lista de los nodos que hacen parte, despues se encuetra el top 5 de los componentes y por ultimo se busca todo lo que se le desea indicar de los lobos que se estan buscando

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: se invoca kosaraju al grafo	O(V+E)
Paso 2 : el mapa de los connectados	O(n) los vertices que se conectan entre si
Paso 3: agrupo según el sccid y lo pongo como llave de un diccionario y el valor son todos los nodos en una lista	O(n) los vertices coneectados
Paso 4: según el sccid encuentro el top 5 sccid con mayor cantidad de nodos	O(n)

Paso 5 Encuentro todo lo que me piden con sccid, encuentro mayores y menos latitudes y longitudes, cuantos lobos por manda y los detalles de los primeros 3 lobos de la manada	O(N) de los nodos según su id
TOTAL	***O(V+E)***

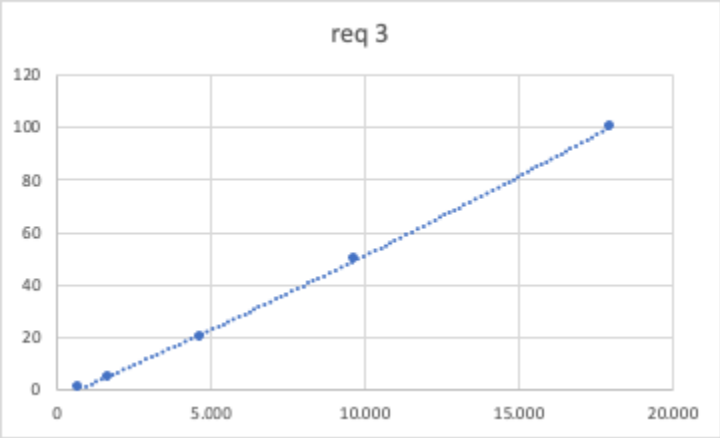
Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

porcentaje	tiempo
1	735
5	1.654,62
20	4638,34
50	9669,98
100	17975,397

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.

En este requerimiento lo que mas tiempo tarde es en general el algoritmo de kosaraju porque tiene que ir por todas los vertices y arcos de los nodos, despues se hacen funciones basicas para agrupar los sccid y encontrar los mayores con las caracteristicas que se piden. Tienen una buena complejidad porque el n que se termina manejando es solo de los componentes conectados.

Requerimiento 4

Plantilla para el documentar y analizar cada uno de los requerimientos.

Descripción

El requerimiento se propone identificar el corredor migratorio entre dos puntos específicos dentro de la región arenosa petrolífera de Athabasca (AOSR) para planear mejor las inspecciones del habitat, y se soluciona mediante un establecimiento de los puntos de encuentro más cercanos a cada una de las coordenadas dadas para luego encontrar la ruta minima mediante bellman-ford. La función que lo soluciona hace lo siguiente:

```
def req_4(data_structs,lat_1,long_1,lat_2,long_2):
    """
    Función que soluciona el requerimiento 4
    """
    grafo=data_structs['grafo']

    ### retorna encuentro más cerca y distancia a este
    nodo_dist_inicio=encontrar_nodo_encuentro_mas_cercano(data_structs,lat_1,long
_1)
    nodo_inicio=nodo_dist_inicio[0]
    distancia_entre_punto_inicio_nodo=nodo_dist_inicio[1]
    ###

    nodo_dist_fin=encontrar_nodo_encuentro_mas_cercano(data_structs,lat_2,long_2)
    nodo_fin=nodo_dist_fin[0]
    distancia_entre_punto_fin_nodo=nodo_dist_fin[1]

    ###recorridos minimos del nodo de incio a todos los demás
    recorridos_inicio=bf.BellmanFord(grafo,nodo_inicio)

    recorrido_min=bf.pathToArray(recorridos_inicio,nodo_fin)
```

```
    #print(nodo_inicio)
    #print(nodo_fin)

    total_arcos=lt.size(recorrido_min)
    total_nodos=total_arcos+1

    #print(total_arcos)
    it=lt.iterator(recorrido_min)
    dist_total=0
    for i in it:
        #print(i)
        dist_total+=i['weight']

    #print(dist_total)
    #print(recorrido_min)
    prim=tres_primeros_nodos(recorrido_min,data_structs['mapa nodos de
encuentro'])
    ult=tres_ultimos_nodos(recorrido_min,data_structs['mapa nodos de encuentro'])
    lista_a_devolver=[]
    lista_a_devolver.append(distancia_entre_punto_inicio_nodo)
    lista_a_devolver.append(distancia_entre_punto_fin_nodo)
    lista_a_devolver.append(dist_total)
    lista_a_devolver.append(total_nodos)
    lista_a_devolver.append(total_arcos)
    lista_a_devolver.append(prim)
    lista_a_devolver.append(ult)
    lista_a_devolver.append(recorrido_min)

    return lista_a_devolver
```

El bono por su parte, se soluciona poniendo como puntos en el mapa los puntos iniciales y finales, los dos nodos siguientes e iterando los arcos de la ruta mínima dibujándolos en el mapa, como muestra el siguiente código:

```
    mapa=folium.Map(location=[lat_c,long_c],zoom_start=5)

    folium.Marker(location=[plat1,plong1],icon=folium.Icon(color='darkblue',icon=
'fire')).add_to(mapa)
    folium.Marker(location=[plat2,plong2],icon=folium.Icon(color='red',icon='fire
')).add_to(mapa)
    for dic in res[5]:
            folium.Marker(location=[dic['lat'],dic['long']],icon=folium.Icon(colo
r='green',icon='fire')).add_to(mapa)
```

```
    for dic in res[6]:
        folium.Marker(location=[dic['lat'],dic['long']],icon=folium.Icon(color='o
range',icon='fire')).add_to(mapa)

    for arco in lt.iterator(res[7]):
        vertexA=arco['vertexA'].replace('m','-').replace('p','.').split('_')
        vertexB=arco['vertexB'].replace('m','-').replace('p','.').split('_')
        latA=float(vertexA[1])
        latB=float(vertexB[1])
        longA=float(vertexA[0])
        longB=float(vertexB[0])
        locs=[(latA,longA),(latB,longB)]
        folium.PolyLine(locs,color='pink',weight=5,opacity=0.8).add_to(mapa)
    mapa.save("C:/Users/samis/Downloads/mapa.html")
```

Entrada	Data strucst, dos localizaciones (latitud-longitud)
Salidas	Una lista con: • La distancia entre el punto GPS de origen y el punto de encuentro más cercano. • La distancia el punto de encuentro de destino más cercano y el punto GPS de destino. • La distancia total que tomará el recorrido entre los puntos de encuentro de origen y destino. • El total de puntos de encuentro que pertenecen al camino identificado (nodos). • El total de individuos/lobos distintos que utilizan el corredor identificado. • El total de segmentos que conforman la ruta identificada (arcos) • Lista de 3 primeros nodos en el camino • Lista de 3 últimos nodos en el camino • Array ordenado de arcos
Implementado (Sí/No)	Si, Samuel Peña

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Encontrar puntos de encuentro más cerca a las 2 coordenadas y la distancia a estos iterando las llaves del mapa de nodos de encuentro	O(V)(iteración)
Ejecutar Bellman-Ford para recorrido mínimo	O(VE)

Iterar recorrido mínimo para encontrar distancia total sumando los arcos y obtener primeros y últimos 3 nodos	O(E)
Encontrar metadatos como el número de los arcos y nodos en el recorrido	O(1)
Encontrar lobos asociados a cada nodo	O(vn) (donde n es el numero de lobos asociados a cada nodo, usualmente es 1 y suele ser muy pequeño)
TOTAL	***O(VE)***
BONO Iterar arcos y nodos de mínima ruta y añadirlos al mapa	O(n) (iteración)

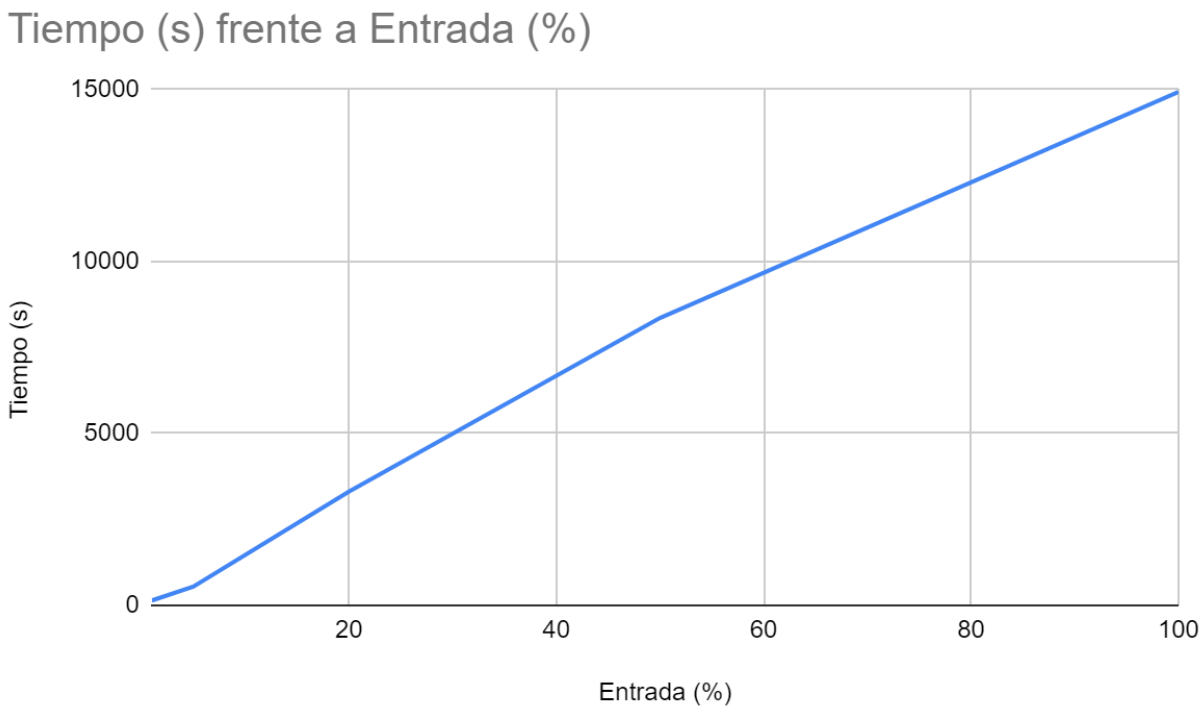
Pruebas Realizadas

Utilizando como entrada las coordenadas:

(-111.911, 57.431) y (-111.865, 57.435):

Entrada (%)	**Tiempo (s)**
1	140,33550000190735
5	541,657900005579
20	3306,9165000021458
50	8355,650299996138
100	14933,337799996138

Grafica



Análisis

Parece ser que el algoritmo tiene complejidad temporal lineal. Lo mas seguro es que esto se deba a que el producto de arcos y nodos sea directamente proporcional al número cargado de eventos.

Requerimiento 5

Plantilla para el documentar y analizar cada uno de los requerimientos.

```python
def req_5(data_structs, min, nodo, dist):
    """
    Función que soluciona el requerimiento 5
    """
    grafo = data_structs["grafo no dirigido"]
    respuesta = lt.newList("ARRAY_LIST")
    dist_f = dist / 2
    re = prim.PrimMST(grafo, nodo)
    mst = re["mst"]
    num = 0
    peso = 0

    edges = gr.adjacentEdges(grafo, nodo)
    tam = len(edges)
    lt.addLast(respuesta, tam)

    r2 = {}
    lt_nodos = lt.newList("ARRAYLIST")
    lt.addLast(lt_nodos,nodo)
    animales = lt.newList("ARRAY_LIST")

    if mst > (min-1):
        while num < lt.size(mst):
            dicco = mst[num]
            pes_dicco = dicco["weight"]
            peso += pes_dicco
            vecto_b = dicco["vertexB"]
            edges_B = data_structs["mapa nodos de encuentro"]
            tam_b = len(edges_B[vecto_b])
            if peso <= dist_f and vecto_b not in lt_nodos:
                r2["Path distance [km]"] = peso
                lt.addLast(lt_nodos, vecto_b)
                lt.addLast(animales, tam_b)

            num += 1
    else:
        respuesta = None

    return respuesta
```

Descripción

En este requerimiento usamos el algoritmo de Prim, de esta forma encontramos los recorridos con MST desde el punto seleccionado y a partir de ahí vamos yendo de arco en arco del MST para ir sumando el peso y para que no se pase del maximo

Entrada	Los datos, un minimo de vertices en el recorrido, el vertice inicial, y la distancia seleccionada
Salidas	Una lista, cn la primera entrada datos del recorrigo y la segunda un diccionario que contiene la informacion relevante para responder el requerimiento
Implementado (Sí/No)	Si, Manuel Pinzon

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: Se hace el algoritmo de Prim para sacar los recorridos	O(AlogA)
Paso 2 : Se sacan los nodos adyacentes al punto	O(V)

Paso 3: Se entra en un ciclo para ir recorriendo los caminos	O(N)
Paso 4: Se busca los elementos del mapa con el vertice	O()
Paso 5: Se compara si ekl vertice cumple con los requisitos para juntarlo al recorrido	O(1)
TOTAL	***O(AlogA)***

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Entrada	**Tiempo (s)**

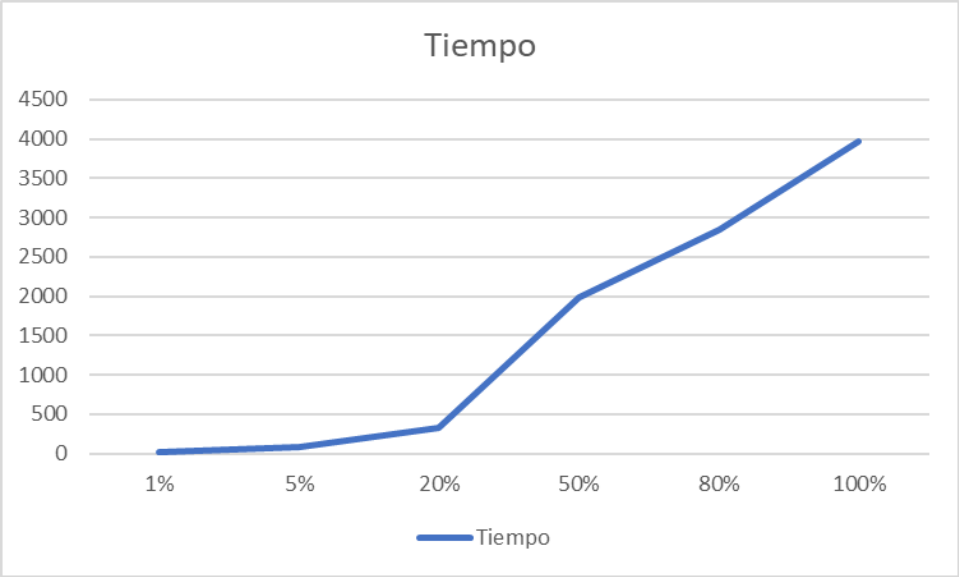
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Tamaño	Tiempo
1%	21.54
5%	78.65
20%	320.45
50%	1982.56
80%	2851.52
100%	3965.21

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

El algoritmo de Prim y el uso de MST simplifican como se hace la busqueda y para mantener la operación dentro del parametro establecido.

Requerimiento 6

Plantilla para el documentar y analizar cada uno de los requerimientos.

```python
def req_6(data_structs, fecha_1, fecha_2, genero):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6
    respuesta = lt.newList("ARRAY_LIST")
    grafo = data_structs["grafo no dirigido"]
    array = data_structs["lista total"]

    filtro = array_ordenado_filtrado_por_rango_fechas(array, fecha_1, fecha_2)
    lobos = aux_gen(filtro, genero)
    max_camin = 0
    min_camin = 100
    r1 = lt.newlist("ARRAY_LIST")
    r2 = lt.newlist("ARRAY_LIST")
    lobo_1 = str()
    r1_1 = str()
    r1_2 = str()
    lobo_2 = str()
    r2_1 = str()
    r2_2 = str()
    for id in lobos:
        eventos = lobos[id]
        nodos = aux_lt_nodo(eventos)
        nodo_m = (grafo, nodos, "max")
        dist = nodo_m[0]
        if dist >= max_camin:
            max_camin = dist
            lobo_1 = id
            r1_1 = dist[1]
            r1_2 = dist[2]

    lt.addLast(r1, lobo_1)
    lt.addLast(r1, r1_1)
    lt.addLast(r1, r1_2)

    for id in lobos:
        eventos = lobos[id]
        nodos = aux_lt_nodo(eventos)
        nodo_m = (grafo, nodos, "min")
        dist = nodo_m[0]
        if dist <= min_camin:
            min_camin = dist
            lobo_2 = id
            r2_1 = dist[1]
            r2_2 = dist[2]
    lt.addLast(r1, lobo_2)
    lt.addLast(r1, r2_1)
    lt.addLast(r1, r2_2)

    lt.addLast(respuesta, r1)
    lt.addLast(respuesta, r2)

    resultado_final = aux_resp6(respuesta, data_structs)

    return resultado_final
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	La funcion recibe los datos, dos fechas para hacer el intervalo y el genero que se quiere estudiar
Salidas	Retorna una lista con dos elementos con la misma estrucura (lista): i. diccionario con la informacion del lobo, ii. Informacion del recorrido, iii. Lista de diccionarios con la informacion de los 3 vertices

Implementado (Sí/No)	Si , Manuel Pinzon

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: Se filtra por fechas	O(N)
Paso 2 : Luego se usa una funcion auxiliar para filtrar los lobos dentro de estas fechas y por genero	O(N)
Paso 3: Se itera con respecto al resultado obtenido en el paso anterior	O(N)
Paso 4: Se usa un funcion auxiliar para sacar todos los vertices por los que vamos a intentar formar recorridos	O(1)
Paso 5: Se usa una funcion auxiliar la cual hace los recorridos entre vertices del mismo lobo, y saca si se le indico el mas pequeño o mayor	O(N*(V+A))
Paso 6: Se repite el mismo paso 3-4-5 para encontrar en este caso el menor	O(N*(V+A)
Paso 7: Se usa una funcion auxiliar para presentar la informacion como se queria, la informacion del lobo y la informacion del recorrido como la de algunos arcos	O(V+A)
TOTAL	***O(N*(V+A))***

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

- “012-11-28 00:00” y “2014-05-17 23:59” con temperaturas entre -17.3 °C y 9.7 °C.

Fueron los datos que se utilizaron

Entrada	**Tiempo (s)**

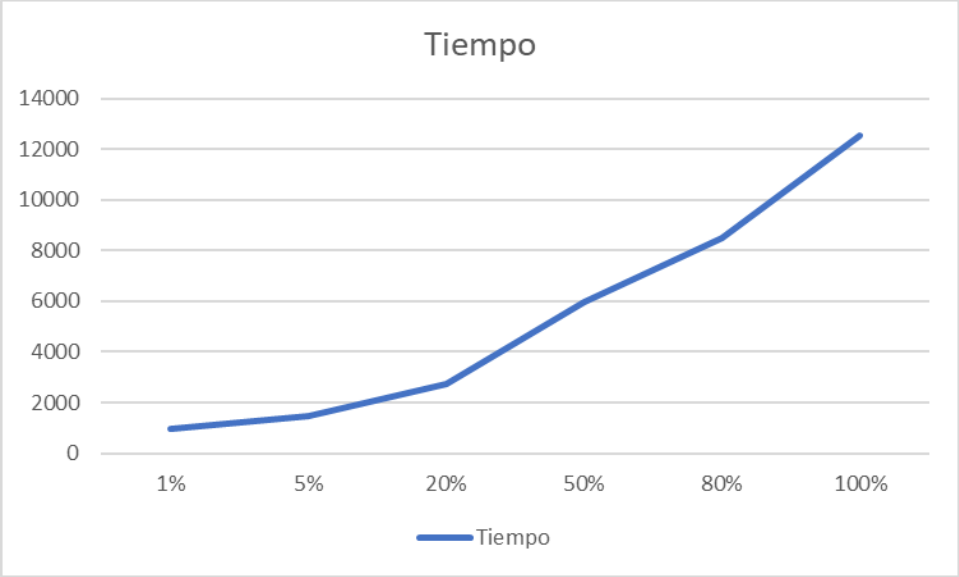
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Tamaño	Tiempo
1%	956.35
5%	1456.81
20%	2748.96
50%	5951.57
80%	8514.76
100%	12548.24

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Este requerimiento fue bastante complejo, ya que los algoritmos que hemos usado solo han servido principalmente para encontrar caminos más pequeños. Entonces el reto fue como podria evitar este inconveniente, para esto decidimos que lo mejor seria escoger todos los vertices que se van a estudiar y a partir de ahí ir haciendo los recorridos. Aunque me imagino que hay mejores algoritmos mas avanzados para esto, teniendo en cuenta lo que teniamos a la mano, fue la opcion mas eficaz

Requerimiento 7

Descripción

```
def req_7(data_structs,time1,time2,temp1,temp2):
    """
    Función que soluciona el requerimiento 7
    """
    crear_grafo_filtrado(data_structs,time1,time2,temp1,temp2)
    grafo=data_structs['grafo']
    kosaraju = scc.KosarajuSCC(grafo)
    "los puntos conectados "
    total = scc.connectedComponents(kosaraju)
    keys = mp.keySet(kosaraju["idscc"])
    mapa = mp.newMap()

    for manada in lt.iterator(keys):
        "invertir las llaves como valores dentro de una lista y el valor se volvio la llave"
        actual = devolver_value(kosaraju["idscc"],manada)
        esta = mp.contains(mapa, actual)
        if esta == False:
            lista = lt.newList()
            lt.addFirst(lista,manada)
            mp.put(mapa,actual,lista)
        else:
            agregar = devolver_value(mapa, actual)
            lt.addLast(agregar, manada)


    llaves_scc = mp.keySet(mapa)
    i = 1
    final = lt.newList()
```

```
    while i <= 3:
        mayor = 0
        sccc = 0
        a = 1

        while a <= lt.size(llaves_scc):
            sccdid = lt.getElement(llaves_scc,a)
            cantidad_list = devolver_value(mapa,sccdid)
            if lt.size(cantidad_list) > mayor:
                mayor = lt.size(cantidad_list)
                sccc = sccdid
                pos = a
            a += 1
        lt.addLast(final,sccc)
        lt.deleteElement(llaves_scc, pos)
        i +=1
    e = 1
    while e <= 3:
        mayor = 99999999999
        sccc = 0
        a = 1

        while a <= lt.size(llaves_scc):
            sccdid = lt.getElement(llaves_scc,a)
            cantidad_list = devolver_value(mapa,sccdid)
            if lt.size(cantidad_list) < mayor:
                mayor = lt.size(cantidad_list)
                sccc = sccdid
                pos = a
            a += 1
        lt.addLast(final,sccc)
        lt.deleteElement(llaves_scc, pos)
        e +=1

    respuesta1 = pedido(data_structs, mapa, final)
```

```
    respuesta2 = []
    vez = 0
    for manada in lt.iterator(final):
        dic = {}
        actual = devolver_value(mapa,manada)

        vertice2 = lt.getElement(actual,lt.size(actual))
        res = mayordfs(grafo,actual,vertice2)
        dic["SCCID"] = manada
        dic["SCC size "] = lt.size(actual)
        dic ["min-lat"] = respuesta1[vez]["min-lat"]
        dic["max-lat"] = respuesta1[vez]["max-lat"]
        dic ["min-lon"] = respuesta1[vez]["min-lon"]
        dic ["max-lon"] = respuesta1[vez]["max-lon"]
        dic["LP node count"] = res[0]
        dic["LP edge count"] = res[0] -1
        dic["LP distance km"] = res[1]

        vez +=1
        respuesta2.append(dic)


    return total, respuesta1, respuesta2
```

El requerimiento pide observar el efecto de los cambios en las condiciones climáticas en la movilidad de las manadas y en el territorio que pueden cubrir a lo largo del tiempo, para lo cuel primero crea un data structs reducido a con elementos solo dentro de los rangos dados.

Para hacer esto primero filtra cada uno de los arrays del mapa lobos con las funciones enuciadas en la nota de la carga de datos, y a partir de este mapa lobos filtrado crea las demás estructuras, incluido un grafo.

Prosiguiendo esto se utiliza el algoritmo de kosajaru para filtrar como se mueven las manadas unas con las otras, despues se le encuentra el top 3 y las 3 que menos se movieron. Finalmente, segun cada manada se hace un dfs para cada una y se evalua comparando a las demas para mostrar cual de estas tiene el mayor camino.

Entrada	Parámetros necesarios para resolver el requerimiento.
Salidas	Respuesta esperada del algoritmo.
Implementado (Sí/No)	Si, Samuel Peña y Tomás Díaz

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
paso 1:Filtrar mapa lobos (iteración)	O(n)
paso 2: Crear nuevo data_structs mediante iteraciones sucesivas	O(n)
Paso 3: se invoca kosaraju al grafo	O(V+E)
Paso 4 : el mapa de los connectados	O(n) los vertices que se conectan entre si

Paso 5: agrupo según el sccid y lo pongo como llave de un diccionario y el valor son todos los nodos en una lista	O(n) los vertices coneectados
Paso 6: según el sccid encuentro el top 3 sccid con mayor cantidad de nodos y los 3 con menos nodos (hay mas de tres con la misma cantidad)	O(n)
Paso 7: Encuentro todo lo que me piden con sccid, encuentro mayores y menos latitudes y longitudes, cuantos lobos por manda y los detalles de los primeros 3 lobos de la manada	O(N) de los nodos según su id
Punto8: para la segunda parte se encuentra la mayor sumatoria de arcos egun el sccid, esto se le saca un dfs y se va sumando arco tras arcos	O(n* (V+E))
Punto 9 :	O(1) Se consigue todo lo necesari
Paso ….	O(...)
TOTAL	***O(***O(n* (V+E)) ***)***

Pruebas Realizadas

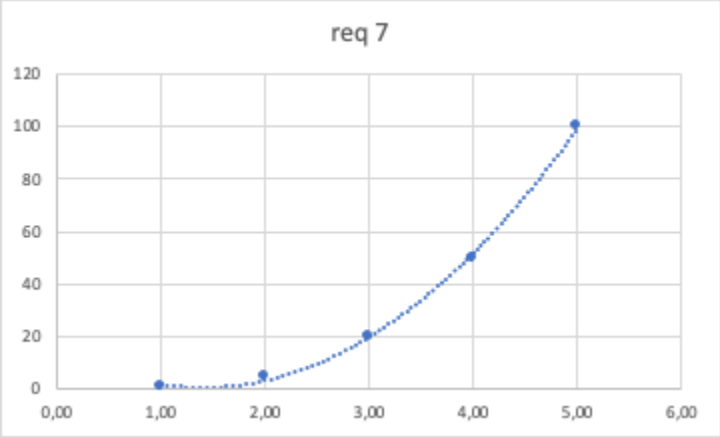
Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

- 012-11-28 00:00” y “2014-05-17 23:59” con temperaturas entre -17.3 °C y 9.7 °C.

porcentaje	tiempo
1	1.624,47
5	52471, 39
20	209885,56
50	524713,9
100	1049427,8

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.

En este es el que se presenta mayor complejidad, porque se toca volver a crear el grafo desde cero siendo filtrado por las caracteristcias que se le piden y ademas de eso a la hora de implementar la funcion de kpsaraju y el dfs para encontrar el mayor recorrido, esto hace que se tomen bastante tiempo su elaboracion.

Anexos

D M A

Diagrama Estructuras de Datos Proto IX EDA

Data Structs

- Lista total: Array de todos los eventos cargados. Los eventos a su vez son diccionarios creados a partir de las columnas del CSV.
- Lista Archivo Lobos: Array que representa la carga del archivo "Wolf". Sus elementos son también diccionarios.
- Mapa lobos: Map con parejas llave-valor de la forma: Llave : animal id
  - Valor: array de eventos asociados a ese lobo, ordenado por fecha y sin 2 eventos consecutivos en el mismo lugar.
- Mapa archivo lobos: Map con parejas: Llave : animal id.
  - Valor: Dic con la info del individuo.
- Mapa Localización: Map con parejas : Llave: coordenada long-lat en formato p=., m=-.
  - Valor: array de eventos asociados.
- Mapa nodos de seguimiento: Map tq : Llave: id del nodo.
  - Valor: coordenada.
- Mapa nodos de encuentro: Map tq - Llave: id del nodo
  - Valor: Array con ids de nodos de seguimiento adyacentes.
- Grafo: Grafo dirigido con los nodos de seguimiento y encuentro con las especificaciones dadas en la guía.
- Grafo NO dirigido: Grafo no dirigido con las mismas características.