ANÁLISIS DEL RETO 4

Sergio Cañar, 202020383, s.canar

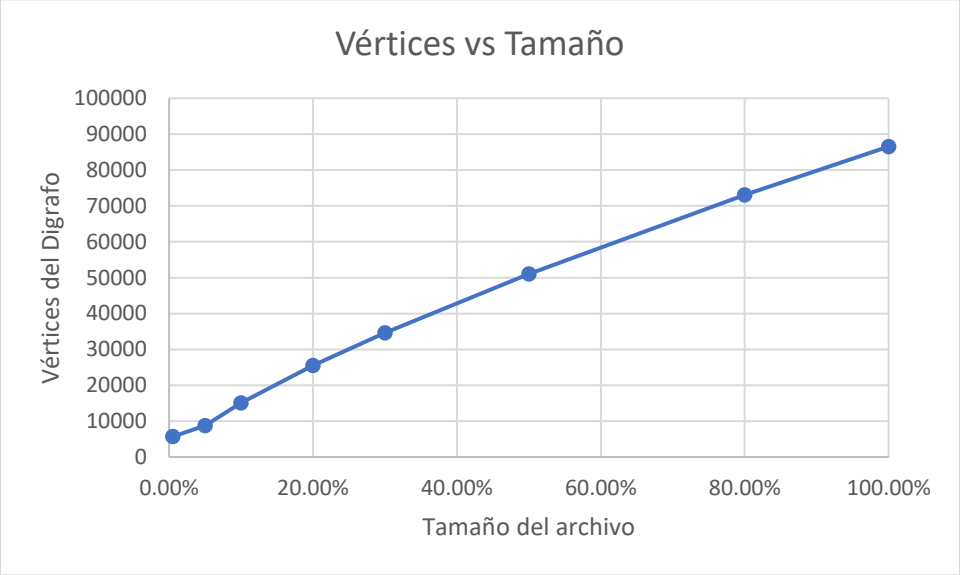
Juan Martín Vásquez, 202113214, j.vasquezc

Juan Bernardo Parra, 202021772, j.parrah

Carga de datos

Se realiza un análisis para la carga de datos, con el fin de observar el comportamiento de los vértices conforme aumenta el tamaño del archivo:

Archivo	Vértices Digrafo
0.50%	5677
5%	8724
10%	15069
20%	25516
30%	34636
50%	51043
80%	73041
100%	86546



Requerimiento 1

Descripción

```python
def req_1(tracker, meeting_point1 , meeting_point2):
    """
    Función que soluciona el requerimiento 1 (Usar DFS)
    """
    # TODO: Realizar el requerimiento 1

    dfs_structure = dfs.DepthFirstSearch(tracker['connections'], meeting_point1)

    flag = dfs.hasPathTo(dfs_structure, meeting_point2)

    if flag:
        ltpath_st = dfs.pathTo(dfs_structure, meeting_point2)

    ltpath = lt.newList("ARRAY_LIST")
    while st.isEmpty(ltpath_st)!=True:
        a = st.pop(ltpath_st)
        lt.addLast(ltpath,a)


    nWolf = 0
    nMeet = 0
    disttot = 0
    r1 = lt.newList('ARRAY_LIST')
    cont = 1
```

```python
for i in lt.iterator(ltpath):
    ltrow = lt.newList('ARRAY_LIST')
    if i.count("_") == 1:
        nMeet += 1
    else:
        nWolf += 1
    long, lat = revertirfromato(i)
    wolfs_str, wolfs = getWolfFromNode(tracker, i)
    indCount = lt.size(wolfs)
    if cont == lt.size(ltpath):
        edgeTo = 'Unknown'
    else:
       edgeTo = lt.getElement(ltpath, cont + 1)

    if edgeTo != 'Unknown':
        dist = round((gr.getEdge(tracker['connections'], i, edgeTo))['weight'],3)
        disttot+= dist
    else:
        dist = 'Unknown'

    #headers = ['lat', 'long', 'id', 'indi ids', 'wolf count','edge-to', 'dist']
    lt.addLast(ltrow,(long))
    lt.addLast(ltrow,(lat))
    lt.addLast(ltrow,(i))
    lt.addLast(ltrow,(wolfs_str))
    lt.addLast(ltrow,(indCount))
    lt.addLast(ltrow,(edgeTo))
    lt.addLast(ltrow,(dist))

    lt.addLast(r1,ltrow)
    cont += 1
r2 = recortarLista(r1)
return r2, flag, nWolf, nMeet, disttot
```

En el requerimiento 1 se quiere encontrar una ruta que conecte a dos puntos de encuentro, ingresados por el usuario. Dadas las características del problema, en el cual no se pide encontrar un camino con características exactas (por ejemplo: camino más corto; camino con menos puntos de encuentro) sino solo se requiere saber si hay un camino, se usa el algoritmo DFS. Se empieza por implementar este

algoritmo sobre el grafo “connections” de nuestro data_structs, empezando por el primer vértice indicado por el usuario. Luego, usando la función hasPathTo se comprueba si existe un camino entre los dos puntos de encuentro deseados. Si la función retorna True, se crea un stack en el cual se vaciará y sus datos se llenarán en una lista normal (esto con el fin de obtener la orientación correcta del camino). Por último, se itera sobre esta lista para obtener los valores de los parámetros deseados (número de lobos, longitud, latitud, etc.). Esta información se añade a la lista r1, la cual se recorta para obtener el formato pedido. Se retorna la lista recortada.

Entrada	data_structs, punto de encuentro 1 y punto de encuentro 2
Salidas	Lista con la información y los nodos del camino, distancia total, total de puntos de encuentro
Implementado (Sí/No)	Si

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1 Hacer DFS	O(V+E)
Paso 2 Recorrer la ruta entregada	O(q) (q corresponde a la constante, de elementos en el path dado por el DFS)
TOTAL	***O(V+E)***

Pruebas Realizadas

Las pruebas se realizaron en un computador con las siguientes características:

Procesador	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
Memoria RAM (GB)	12.0 GB (11.7 GB utilizable)
Sistema Operativo	Windows 11 Home Single Language – 64 bits

Para la medición de tiempos se usaron las funciones getTime y deltaTime de la librería time. Estas se ubicaron en el controller para medir el tiempo de ejecución de los requerimientos de la siguiente manera:

```
start_time = getTime()
ds = control["model"]

if memflag is True:
    tracemalloc.start()
    start_memory = getMemory()
req1,flag, nWolf, nMeet, disttot = model.req_1(ds, origen, destino)
 # toma el tiempo al final del proceso
stop_time = getTime()
# calculando la diferencia en tiempo
delta_time = deltaTime(stop_time, start_time)
# finaliza el proceso para medir memoria
if memflag is True:
    stop_memory = getMemory()
    tracemalloc.stop()
    # calcula la diferencia de memoria
    delta_memory = deltaMemory(stop_memory, start_memory)
```

Por otro lado, se usó la librería tracemalloc para rastrear el consumo de memoria. Se usaron las funciones de tracemalloc.start, getMemory, deltaMemory y tracemalloc.stop para el rastreo de memoria. Hay que destacar que este rastreo solo se ejecuta cuando el usuario indica que se haga, por lo que se implementó una variable booleana (llamada memflag) para confirmar esta operación, como se puede ver en la imagen:

```
start_time = getTime()
ds = control["model"]

if memflag is True:
    tracemalloc.start()
    start_memory = getMemory()
req1,flag, nWolf, nMeet, disttot = model.req_1(ds, origen, destino)
 # toma el tiempo al final del proceso
stop_time = getTime()
# calculando la diferencia en tiempo
delta_time = deltaTime(stop_time, start_time)
# finaliza el proceso para medir memoria
if memflag is True:
    stop_memory = getMemory()
    tracemalloc.stop()
    # calcula la diferencia de memoria
    delta_memory = deltaMemory(stop_memory, start_memory)
```

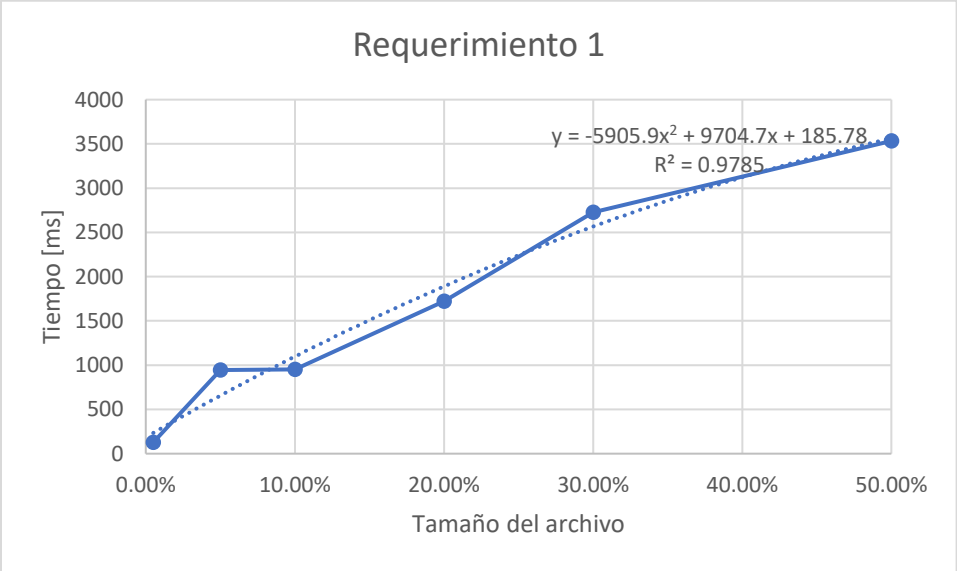
Además, para no interferir en el proceso de la medición de velocidad, ningún dato de tiempo fue recolectado midiendo la memoria de manera simultánea.

Para las mediciones de tiempo de cada requerimiento se ingresaron como parámetro los datos dados en el enunciado para todos los tamaños de archivo. En el requerimiento 1 se usaron los puntos m111p862_57p449 y m111p908_57p427.

Tablas de datos

1		
Archivo	Vértices Digrafo	Tiempo [ms]
0.50%	5677	127.012
5%	8724	944.914
10%	15069	950.254
20%	25516	1721.3355
30%	34636	2728.9279
50%	51043	3532.935
80%	73041	
100%	86546	

Graficas



Análisis

Como se puede observar, la gráfica para el crecimiento temporal del requerimiento 1 sigue un comportamiento parecido al lineal, pese a ciertos datos atípicos que no siguen la tendencia. Esto confirma la hipótesis planteada anteriormente, en la cual se dice que la complejidad temporal del requerimiento 1 es de O(V+E).

Requerimiento 2

Descripción

```python
def req_2(tracker, meeting_point1 , meeting_point2):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    bfs_structure = bfs.BreadhtFisrtSearch(tracker['connections'], meeting_point1)

    flag = bfs.hasPathTo(bfs_structure, meeting_point2)

    if flag:
        ltpath_st = bfs.pathTo(bfs_structure, meeting_point2)

    ltpath = lt.newList("ARRAY_LIST")
    while st.isEmpty(ltpath_st)!=True:
        a = st.pop(ltpath_st)
        lt.addLast(ltpath,a)

    nWolf = 0
    nMeet = 0
    disttot = 0
    r1 = lt.newList('ARRAY_LIST')
    cont = 1
    for i in lt.iterator(ltpath):
        ltrow = lt.newList('ARRAY_LIST')
        if i.count("_") == 1:
            nMeet += 1
        else:
            nWolf += 1
        long, lat = revertirfromato(i)
        wolfs_str, wolfs = getWolfFromNode(tracker, i)
        indCount = lt.size(wolfs)
        if cont == lt.size(ltpath):
            edgeTo = 'Unknown'
        else:
            edgeTo = lt.getElement(ltpath, cont + 1)

        if edgeTo != 'Unknown':
            dist = round((gr.getEdge(tracker['connections'], i, edgeTo))['weight'],3)
            disttot+= dist
        else:
            dist = 'Unknown'

        #headers = ['lat', 'long', 'id', 'indi ids', 'wolf count','edge-to', 'dist']
        lt.addLast(ltrow,(long))
        lt.addLast(ltrow,(lat))
        lt.addLast(ltrow,(i))
        lt.addLast(ltrow,(wolfs_str))
        lt.addLast(ltrow,(indCount))
        lt.addLast(ltrow,(edgeTo))
        lt.addLast(ltrow,(dist))

        lt.addLast(r1,ltrow)
        cont += 1
    r2 = recortarLista(r1)
    return r2, flag, nWolf, nMeet, disttot
```

En este requerimiento se desea hallar el camino mas corto entre 2 nodos, por ello se utiliza el BFS. Se llaman a las funciones de la librería que implementan este requerimiento, luego se saca la ruta del stack en el que llegan. A partir de esta ruta se extrae la información solicitada en el requerimiento.

Entrada	tracker, meeting_point1 , meeting_point2
Salidas	Primeros 3 y últimos 3 nodos de la ruta en el DFs, nodos
Implementado (Sí/No)	Si

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1 Hacer DFS	O(V+E)
Paso 2 Recorrer la ruta entregada	O(q) (q corresponde a la constante, de elementos en el path dado por el DFS)
TOTAL	***O(V+E)***

Pruebas Realizadas

Las pruebas se realizaron en un computador con las siguientes características:

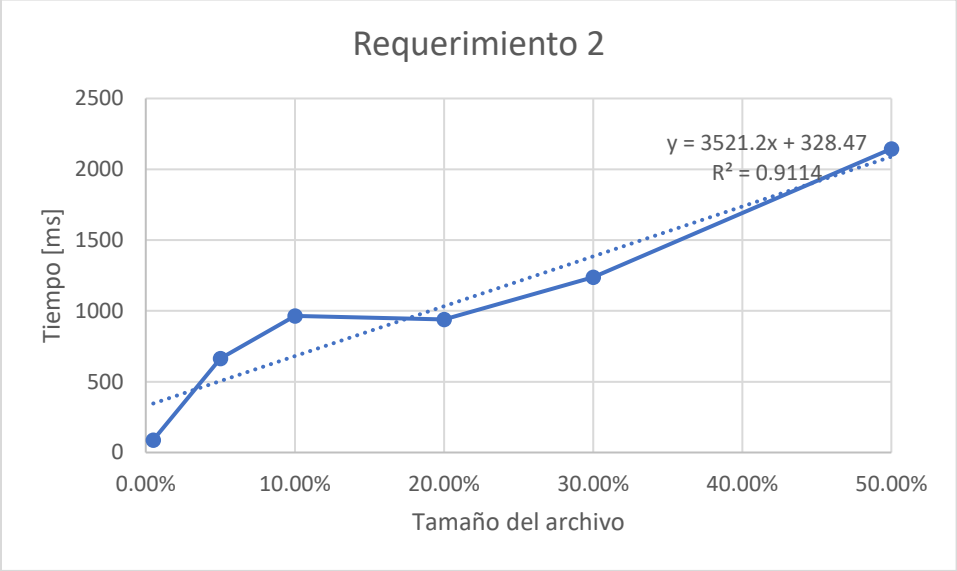
Procesador	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
Memoria RAM (GB)	12.0 GB (11.7 GB utilizable)
Sistema Operativo	Windows 11 Home Single Language – 64 bits

Las pruebas se hicieron con los datos dados en el ejemplo del enunciado. En el caso de este requerimiento, las pruebas para archivos del 10% o menos, se realizaron con meeting points diferentes, puesto que con los del enunciado no se encontraba respuesta.

Tablas de datos

2		
Archivo	Vértices Digrafo	Tiempo [ms]
0.50%	5677	88.2
5%	8724	664.194
10%	15069	963.389
20%	25516	939.222
30%	34636	1238.483
50%	51043	2144.316
80%	73041	
100%	86546	

Graficas



Análisis

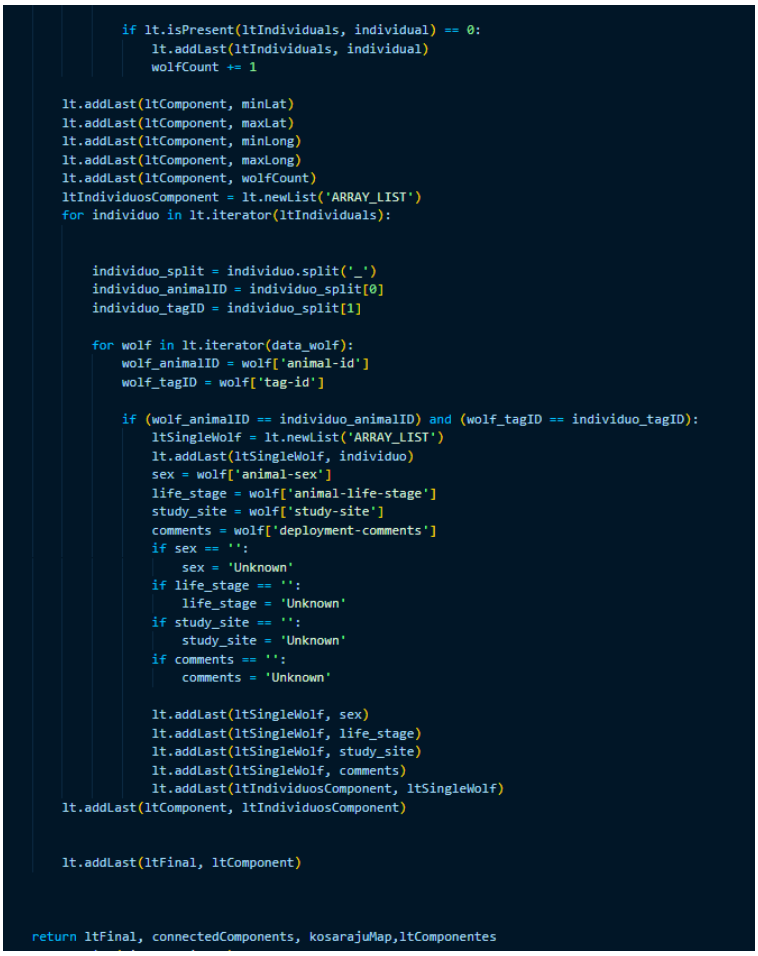
Analizando el código del requerimiento, se llegó a la conclusión de que el crecimiento temporal del req2 debía ser O(E+V). Como se puede ver en la gráfica, el tiempo necesitado para completar el requerimiento es linealmente proporcional al tamaño del archivo, el cual es a su vez proporcional con el número de arcos y aristas. Por esta razón, se puede confirmar los resultados obtenidos en el análisis teórico y concluir que la complejidad temporal del 2do requerimiento es de O(E+V).

Requerimiento 3: Implementado por Sergio Cañar

Descripción

```python
def req_3(tracker):
    """
    Paramétros: El tracker de los lobos

    Como guardabosques del área deseo conocer los territorios de las manadas8 de lobos presentes dentro del
    hábitat del bosque. Cuantas manadas existen, quienes son sus miembros, sus características, los puntos de
    encuentro que frecuentan y las posiciones que dominan.

    Retorna: Estructura de datos que contiene las manadas, cantidad de elementos fuertemente conectados del grafo


    """
    data_wolf = tracker['data_wolfs']

    kosarajuMap = scc.KosarajuSCC(tracker['connections'])

    ltNodes = mp.keySet(kosarajuMap['idscc'])

    connectedComponents = scc.connectedComponents(kosarajuMap)

    mapaComponentes = mp.newMap(1000, maptype='PROBING', cmpfunction=compareWolfID)

    for node in lt.iterator(ltNodes):
        componentID = mp.get(kosarajuMap['idscc'], node)['value']
        entry = mp.get(mapaComponentes, componentID)
        if entry is None:
            ltNode = lt.newList('ARRAY_LIST')
            lt.addLast(ltNode, node)

            mp.put(mapaComponentes, componentID, ltNode)
        else:
            ltNode = entry['value']
            lt.addLast(ltNode, node)
            mp.put(mapaComponentes, componentID, ltNode)


    ltComponentes = listaFromMap(mapaComponentes)

    componentSorted = merg.sort(ltComponentes, compareSize)
```

```python
componentSorted = merg.sort(ltComponentes, compareSize)

ltFinal = lt.newList('ARRAY_LIST')

for component in lt.iterator(componentSorted):
    ltComponent =lt.newList('ARRAY_LIST')
    componentIdentificator, ltStrongNodes = component
    lt.addLast(ltComponent, componentIdentificator) #(ID de la manada)
    lt.addLast(ltComponent, ltStrongNodes) #Lista de nodos fuertemente conectados (manada)
    size = lt.size(ltStrongNodes) #Cantidad de nodos fuertemente conectados (manada)
    lt.addLast(ltComponent, size)

    maxLat = -100000000000
    minLat = 100000000000

    maxLong = -100000000000
    minLong = 100000000000

    wolfCount = 0

    ltIndividuals = lt.newList('ARRAY_LIST')


    for node in lt.iterator(ltStrongNodes):

        coords = getLocationFromNode(node)

        long = float(coords[0])
        lat = float(coords[1])

        if long > maxLong:
            maxLong = long
        if long < minLong:
            minLong = long
        if lat > maxLat:
            maxLat = lat
        if lat < minLat:
            minLat = lat


        flag = meetingOrTracking(node)

        if flag:
            individual = getIndividualFromNode(node)
```



```
            if lt.isPresent(ltIndividuals, individual) == 0:
                lt.addLast(ltIndividuals, individual)
                wolfCount += 1

    lt.addLast(ltComponent, minLat)
    lt.addLast(ltComponent, maxLat)
    lt.addLast(ltComponent, minLong)
    lt.addLast(ltComponent, maxLong)
    lt.addLast(ltComponent, wolfCount)
    ltIndividuosComponent = lt.newList('ARRAY_LIST')
    for individuo in lt.iterator(ltIndividuals):


        individuo_split = individuo.split('_')
        individuo_animalID = individuo_split[0]
        individuo_tagID = individuo_split[1]

        for wolf in lt.iterator(data_wolf):
            wolf_animalID = wolf['animal-id']
            wolf_tagID = wolf['tag-id']

            if (wolf_animalID == individuo_animalID) and (wolf_tagID == individuo_tagID):
                ltSingleWolf = lt.newList('ARRAY_LIST')
                lt.addLast(ltSingleWolf, individuo)
                sex = wolf['animal-sex']
                life_stage = wolf['animal-life-stage']
                study_site = wolf['study-site']
                comments = wolf['deployment-comments']
                if sex == '':
                    sex = 'Unknown'
                if life_stage == '':
                    life_stage = 'Unknown'
                if study_site == '':
                    study_site = 'Unknown'
                if comments == '':
                    comments = 'Unknown'

                lt.addLast(ltSingleWolf, sex)
                lt.addLast(ltSingleWolf, life_stage)
                lt.addLast(ltSingleWolf, study_site)
                lt.addLast(ltSingleWolf, comments)
                lt.addLast(ltIndividuosComponent, ltSingleWolf)
    lt.addLast(ltComponent, ltIndividuosComponent)


    lt.addLast(ltFinal, ltComponent)


return ltFinal, connectedComponents, kosarajuMap,ltComponentes
```

Para solucionar el requerimiento 3 se implementó el algoritmo de Kosaraju para obtener los componentes fuertemente conectados del dígrafo creado en la carga de datos. Posteriormente, para obtener la información solicitada por requerimiento se creó un mapa de los componentes fuertemente conectados cuya llave es el ID de dicho componente y tiene como valor un arreglo de todos los nodos que están en dicho componente. Por consiguiente, se obtuvo una lista de lista de mapa para poder iterar utilizando la librería de DISClib esta lista se ordena por tamaño utilizando Merge sort. A partir de esto se itera sobre cada componente para poder iterar sobre cada lista perteneciente a este para obtener la información solicitada. Por último, se itera sobre los datos de los lobos de tal forma que se termina la lista para retornar la información necesaria.

Entrada	Tracker
Salidas	Lista de la información de las manadas, numero de componentes fuertemente conectados, mapa resultante del algoritmo de kosaraju, lista del mapa mencionado previamente
Implementado (Sí/No)	Si

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Paso 1: Implementación del algoritmo de Kosaraju	O(V+E)
Paso 2: Creación del mapa del algoritmo de Kosaraju	O(n), donde n es el número de vértices fuertemente conectados.
Paso 3: Creación de la lista de listas del mapa	O(n), donde n es el número de componentes fuertemente conectados
Paso 4: Merge sort para ordenar la lista por tamaños	O(nlog(n))
Paso 5: Iterar sobre los componentes fuertemente conectados para obtener la información solicitada.	$O(n^2)$, donde n es el número de componentes fuertemente conectados
TOTAL	***$O(n^2)$***

Pruebas Realizadas

Las pruebas se realizaron en un computador con las siguientes características:

Procesador	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
Memoria RAM (GB)	12.0 GB (11.7 GB utilizable)
Sistema Operativo	Windows 11 Home Single Language – 64 bits

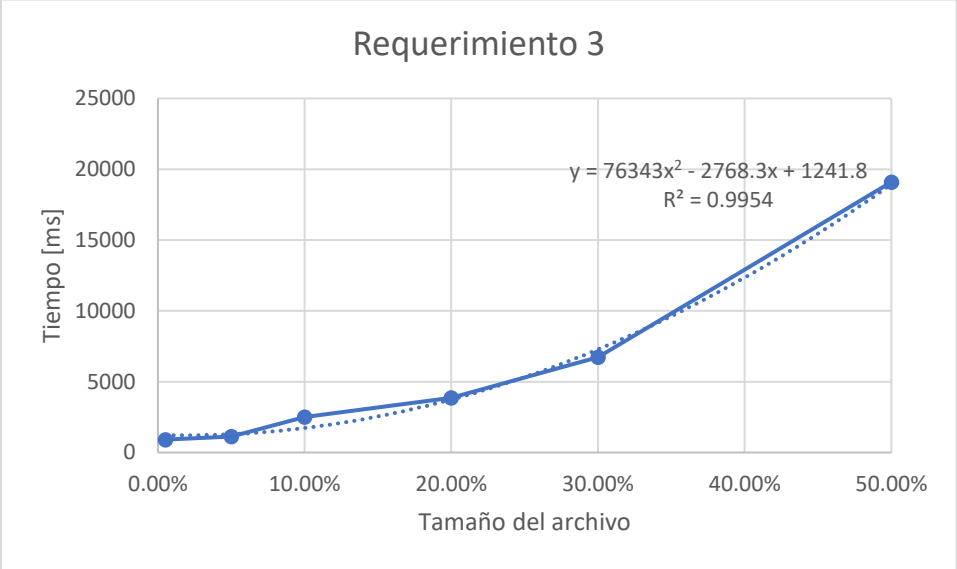
Las pruebas se realizaron con los datos dados en el ejemplo del enunciado.

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

3		
Archivo	Vértices Digrafo	Tiempo [ms]
0.50%	5677	909.9823
5%	8724	1120.6199
10%	15069	2513.4388
20%	25516	3857.144
30%	34636	6722.012
50%	51043	19097.113
80%	73041	
100%	86546	

Graficas



Análisis

A partir de los resultados obtenidos de las pruebas realizadas se logra evidenciar que el requerimiento 3 tiene un comportamiento $O(n^2)$ justo como se evaluó previamente en el análisis de complejidad temporal del requerimiento. Lo anterior, se debe principalmente a los recorridos necesarios para obtener la información solicitada en la impresión puesto que tocaba recorrer la lista de los tracks y lobos para obtener la información de los lobos que estaban en cierta manada.

Requerimiento 4: Implementado por Juan Martín Vásquez

Descripción

```
def req_4(tracker,origen,destino):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4
    NodoMascercano_O, distMin1 = NodoMascercano(tracker,origen)
    NodoMascercano_D, distMin2 = NodoMascercano(tracker,destino)

    bfs_structure = djk.Dijkstra(tracker['connections'], NodoMascercano_O)

    flag = djk.hasPathTo(bfs_structure, NodoMascercano_D)

    if flag:
        ltpath_st = djk.pathTo(bfs_structure, NodoMascercano_D)

    ltpath = lt.newList("ARRAY_LIST")
    while st.isEmpty(ltpath_st)!=True:
        a = st.pop(ltpath_st)
        lt.addLast(ltpath,a)

    tablaOrigen = lt.newList("ARRAY_LIST")
    longO, latO = revertirfromato(NodoMascercano_O)
    lt.addLast(tablaOrigen,NodoMascercano_O)
    lt.addLast(tablaOrigen,longO)
    lt.addLast(tablaOrigen,latO)
    lt.addLast(tablaOrigen,  getWolfFromNode(tracker, NodoMascercano_O)[0])

    tabladestino = lt.newList("ARRAY_LIST")
    longD, latD = revertirfromato(NodoMascercano_D)
    lt.addLast(tabladestino,NodoMascercano_D)
    lt.addLast(tabladestino,longD)
    lt.addLast(tabladestino,latD)
    lt.addLast(tabladestino,  getWolfFromNode(tracker, NodoMascercano_D)[0])

    nWolf = 0
    nMeet = 0
    disttot = 0
    #numWolfInds = 0
    r1 = lt.newList('ARRAY_LIST')
    r3 = lt.newList('ARRAY_LIST')
    listaNodos = lt.newList('ARRAY_LIST')

    for i in lt.iterator(ltpath):
        actual = i['vertexA']
        edgeTo = i['vertexB']

        if lt.isPresent(listaNodos,actual) == False:
            lt.addLast(listaNodos,actual)
        if lt.isPresent(listaNodos,edgeTo) == False:
            lt.addLast(listaNodos,edgeTo)

        ltrow = lt.newList('ARRAY_LIST')
```

```
    if lt.isPresent(listaNodos,actual) == False:
        lt.addLast(listaNodos,actual)
    if lt.isPresent(listaNodos,edgeTo) == False:
        lt.addLast(listaNodos,edgeTo)

    ltrow = lt.newList('ARRAY_LIST')

    long, lat = revertirfromato(actual)
    wolfs_str, wolfs = getWolfFromNode(tracker, actual)
    indCount = lt.size(wolfs)

    edgeTolat,edgeToLog = revertirfromato(edgeTo)

    dist = i['weight']
    disttot+= dist

    #headers = ['lat', 'long', 'id', 'indi ids', 'wolf count','edge-to', 'dist']
    lt.addLast(ltrow,(actual))
    lt.addLast(ltrow,(long))
    lt.addLast(ltrow,(lat))
    lt.addLast(ltrow,(edgeTo))
    lt.addLast(ltrow,(edgeToLog))
    lt.addLast(ltrow,(edgeTolat))
    lt.addLast(ltrow,(dist))
    lt.addLast(r1,ltrow)

for i in lt.iterator(listaNodos):
    ltrow2 = lt.newList('ARRAY_LIST')
    long, lat = revertirfromato(i)
    wolfs_str, wolfs = getWolfFromNode(tracker, i)
    indCount = lt.size(wolfs)

    lt.addLast(ltrow2,(i))
    lt.addLast(ltrow2,(long))
    lt.addLast(ltrow2,(lat))
    lt.addLast(ltrow2,(wolfs_str))
    lt.addLast(ltrow2,(indCount))
    lt.addLast(r3,ltrow2)

r2 = recortarLista2(r1)
r4 = recortarLista2(r3)
return NodoMascercano_O, tablaOrigen, distMin1, NodoMascercano_D, tabladestino, distMin2,r2, nWolf, nMeet, disttot, r4
```

Para hallar el camino más corto entre 2 ubicaciones primero se hallaron los nodos mas cercanos a estas ubicaciones iterando sobre todos los nodos y calculando la distancia a los puntos por parámetro. Luego se implementa en el algoritmo de Dijkstra. Se extrae la información de la ruta del stack y se pone en una lista, se itera sobre esta lista para encontrar el resto de información solicitada.

Entrada	tracker, origen,destino
Salidas	Nodo mas cercano al origen, tabla de la info nodo más cercano al origen, distancia mínima de las coordenadas del usuario a un meeting point, nodo mas cercano al Destino, tabla con la información del destino distancia mínima al destino, tabla con los primeros 3 y últimos tres nodos en la ruta, numero de meeting point, numero de trackieng points, distancia total recorrida, información de los nodos
Implementado (Sí/No)	Si

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1 Encontrar los nodos mas cercanos a las coordendas por parámetros	O(V-k) (k corresponde a los vértices que son meeting points)
Paso 2 Utilizar el algoritmo de Dijkstra	O(V + Elog V)
Paso 3 recorrer el camino dado Dijkstra	O(q) (q corresponde a la cantidad de nodos en el camino)
TOTAL	***O(V + Elog V)***

Pruebas Realizadas

Las pruebas se realizaron en un computador con las siguientes características:

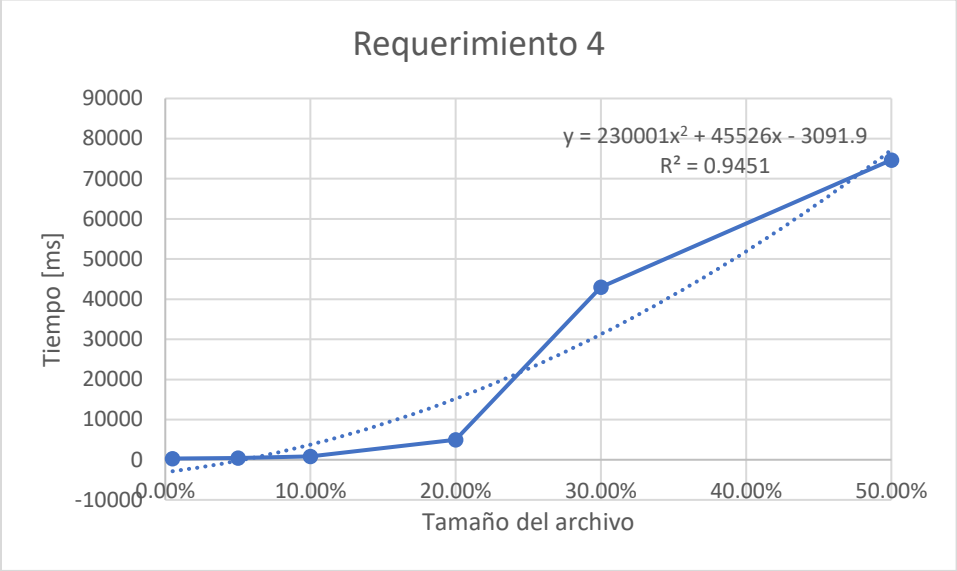
Procesador	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
Memoria RAM (GB)	12.0 GB (11.7 GB utilizable)
Sistema Operativo	Windows 11 Home Single Language – 64 bits

Las pruebas se realizaron con los datos dados en el ejemplo del enunciado.

Tablas de datos

4		
Archivo	Vértices Digrafo	Tiempo [ms]
0.50%	5677	303.3894
5%	8724	465.672
10%	15069	820.915
20%	25516	5014.01
30%	34636	43029.914
50%	51043	74678.372
80%	73041	
100%	86546	

Graficas



Análisis

A partir de las pruebas realizadas se puede observar como el cálculo de complejidad fue acertado, pues se puede confirmar un comportamiento similar al de nlogn. Si bien existen datos atípicos en las mediciones de datos (como el salto en tiempo requerido entre el archivo del 20% y el del 30%), la línea de tendencia demuestra un comportamiento similar al enunciado de O(n log n).

Requerimiento 5: Implementado por Juan Bernardo Parra

Descripción

```python
def req_5(data_structs, origen, maxdist, minMP):
    """
    Función que soluciona el requerimiento 5
    """
    # TODO: Realizar el requerimiento 5
    datos=data_structs["connections"]
    maxdist=maxdist/2
    mst=prim.PrimMST(datos,origen)
    dist=0
    visitados=lt.newList("ARRAY_LIST")
    actual=origen
    listatree=om.valueSet(mst["edgeTo"])

    for i in lt.iterator(listatree):
        format=i["vertexA"].split("_")
        nodof="_".join(format[:2])
        if nodof==actual:
            lt.addLast(visitados,i["vertexB"])
            actual=i["vertexB"]
            dist+=i["weight"]
        if dist>=maxdist or lt.size(visitados)>=minMP:
            break
    vislist=visitados["elements"]
    return vislist, dist
```

En el requerimiento 5 se desea obtener el corredor migratorio con mayor longitud, dados una distancia máxima, un número mínimo de puntos de encuentro y un punto de encuentro inicial. Para resolver este problema, se empieza por obtener un árbol de expansión mínima usando el algoritmo de Prim. Esto nos devuelve un árbol, el cual contiene todos los nodos del grafo y es acíclico. Se extraen los valores de las parejas llave-valor del árbol usando la función om.valueSet. Se itera sobre esta lista, en la cual se construye la ruta conforme pasan los nodos y se actualiza la distancia recorrida. La iteración para si se completan los nodos buscados o si se recorrió la distancia completa. Se devuelve la lista con los nodos recorridos y la distancia recorrida.

Entrada	Data_structs, punto de origen, distancia máxima, mínimos meeting points.
Salidas	Puntos de encuentro visitados, distancia recorrida, secuencia recorrida.
Implementado (Sí/No)	Si - resultado incorrecto

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1 Algoritmo de Prim	O(E log V)
Paso 2 Usar la función om.ValueSet	O(V)
Paso 3 iterar sobre listatree	O(V
TOTAL	***O(E logV)***

Pruebas Realizadas

Las pruebas se realizaron en un computador con las siguientes características:

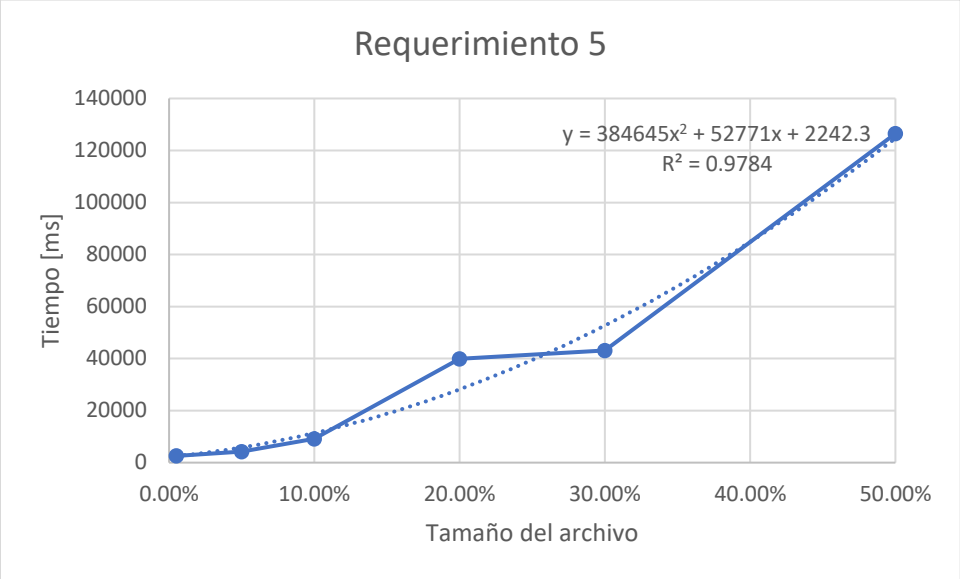
Procesador	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
Memoria RAM (GB)	12.0 GB (11.7 GB utilizable)
Sistema Operativo	Windows 11 Home Single Language – 64 bits

Las pruebas se realizaron con los datos dados en el ejemplo del enunciado.

Tablas de datos

5		
Archivo	Vértices Digrafo	Tiempo [ms]
0.50%	5677	2583.959
5%	8724	4227.526
10%	15069	9127.5135
20%	25516	39874.58
30%	34636	43094.098
50%	51043	126478.856
80%	73041	
100%	86546	

Graficas



Análisis

Al igual que en el requerimiento anterior, si bien existen valores diferente a lo esperado (Por ejemplo, el tiempo requerido con el archivo de 20% es casi idéntico al del archivo de 30%), el conjunto de datos se puede modelar como una función de tipo nlogn. El hecho de que los datos atípicos tanto de este requerimiento como del anterior se encuentren en los mismos tamaños de archivos puede indicar que el error es sistemático y no aleatorio. Por otro lado, se confirma la hipótesis planteada en el análisis del código, donde se estimó la complejidad a O(E log V)

```python
def req_6(tracker,FechaI_raw, FechaF_raw, Sex):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6
    formato = '%Y-%m-%d %H:%M'
    FechaI = datetime.strptime(FechaI_raw, formato)
    FechaF = datetime.strptime(FechaF_raw, formato)

    wolfs_TOT = tracker['data_wolfs']
    wolf_sex = lt.newList('ARRAY_LIST')
    for i in lt.iterator(wolfs_TOT):
        AnimalSex = i['animal-sex']
        if AnimalSex == "f":
            lt.addLast(wolf_sex, i)

    wolf_tracks_TOT = tracker['data_tracks']
    wolf_tracks_Dates = lt.newList('ARRAY_LIST')
    for i in lt.iterator(wolf_tracks_TOT):
        date = i['timestamp']
        #formato = '%d/%m/%Y %H:%M'
        formato = '%Y-%m-%d %H:%M'
        date1 = datetime.strptime(date, formato)
        if FechaI < date1 and date1 < FechaF:
            lt.addLast(wolf_tracks_Dates, i)

    wolfMap_new = mp.newMap(100, maptype='PROBING', cmpfunction=compareWolfID)

    for wolf_lindo in lt.iterator(wolf_sex):
        id_wolf_individual = createIndividualID(wolf_lindo)
        for track_lindo in lt.iterator(wolf_tracks_Dates):
            id_wolf_track = createIndividualID_tracks(track_lindo)
            if id_wolf_track == id_wolf_individual:
                addTrackMapReq6(wolfMap_new, id_wolf_track, track_lindo)

    wolfMap_new = sortWolfTracksbyDateReq6(wolfMap_new)

    wolfByDist = lt.newList('ARRAY_LIST')
    WolfKeys = mp.keySet(wolfMap_new)

    for i in lt.iterator(WolfKeys):
        wolfDist = wolf_dist(i,wolfMap_new)
        thing = (i, wolfDist)
        lt.addLast(wolfByDist, thing)

    wolfByDist = merg.sort(wolfByDist, compareWolfTracksbyDist)

    wolfmxInfo = lt.newList('ARRAY_LIST')
    wolfminInfo = lt.newList('ARRAY_LIST')
    wolfmax = lt.getElement(wolfByDist, 1)
    wolfmix = lt.getElement(wolfByDist, lt.size(wolfByDist))

    wolfMAXRuteRaw = lt.newList('ARRAY_LIST')
    wolfMINRuteRaw = lt.newList('ARRAY_LIST')
```

App > model.py > req_6

```
1298        wolfMINRuteRaw = lt.newList('ARRAY_LIST')
1299
1300        for i in lt.iterator(wolf_sex):
1301            if createIndividualID(i) == wolfmax[0]:
1302                lt.addLast(wolfmxInfo,wolfmax[0])
1303                lt.addLast(wolfmxInfo,i['animal-taxon'])
1304                lt.addLast(wolfmxInfo,i['animal-life-stage'])
1305                lt.addLast(wolfmxInfo,i['study-site'])
1306                lt.addLast(wolfmxInfo,wolfmax[1])
1307                lt.addLast(wolfmxInfo,i['deployment-comments'])
1308
1309
1310            if createIndividualID(i) == wolfmix[0]:
1311                lt.addLast(wolfminInfo,wolfmix[0])
1312                lt.addLast(wolfminInfo,i['animal-taxon'])
1313                lt.addLast(wolfminInfo,i['animal-life-stage'])
1314                lt.addLast(wolfminInfo,i['study-site'])
1315                lt.addLast(wolfminInfo,wolfmix[1])
1316                lt.addLast(wolfminInfo,i['deployment-comments'])
1317
1318        wolfMaxTracks  = mp.get(wolfMap_new,wolfmax[0])['value']
1319        wolfMinTracks  = mp.get(wolfMap_new,wolfmix[0])['value']
1320
1321        for i in lt.iterator(wolfMaxTracks):
1322            fila = lt.newList('ARRAY_LIST')
1323            lt.addLast(fila, createTrackPointID(i))
1324            lt.addLast(fila, i['location-long'])
1325            lt.addLast(fila, i['location-lat'])
1326            lt.addLast(fila, wolfmax[0])
1327            lt.addLast(fila, 1)
1328            lt.addLast(wolfMAXRuteRaw,fila)
1329
1330        for i in lt.iterator(wolfMinTracks):
1331            fila = lt.newList('ARRAY_LIST')
1332            lt.addLast(fila, createTrackPointID(i))
1333            lt.addLast(fila, i['location-long'])
1334            lt.addLast(fila, i['location-lat'])
1335            lt.addLast(fila, wolfmix[0])
1336            lt.addLast(fila, 1)
1337            lt.addLast(wolfMINRuteRaw,fila)
1338
1339        nodesRMAX = lt.size(wolfMAXRuteRaw)
1340        nodesRMIN= lt.size(wolfMINRuteRaw)
1341
1342        if nodesRMAX > 6:
1343            wolfMAXRute = recortarLista2(wolfMAXRuteRaw)
1344        else:
1345            wolfMAXRute = wolfMAXRuteRaw
1346
1347        if nodesRMIN > 6:
1348            wolfMINRute = recortarLista2(wolfMINRuteRaw)
1349        else:
1350            wolfMINRute = wolfMINRuteRaw
1351
1352        return wolfmax, wolfmix, wolfmxInfo, wolfminInfo, nodesRMAX, nodesRMIN, wolfMAXRute ,wolfMINRute
```

En este requerimiento en primer lugar se “limpiaron” los datos de acuerdo a las condiciones planteadas por parámetros, luego se creo un nuevo mapa con un lobo como llave y sus tracks como valor, los cuales se organizaron por fecha. A partir de este mapa se calcula la distancia entre cada uno de os tracks

organizados para cada lobo y estas se organizan. Se toman los valores extremos de la lista organizada y se extrae información sobre estos.

Entrada	tracker,FechaI_raw, FechaF_raw, Sex.
Salidas	El id del lobo que más recorre, Id del lobo que menos recorre, información del lobo que más recorre, información del lobo que menos recorre, numero de nodos de la ruta del lobo que más recorre, numero de nodos que recorre el lobo que menos recorre, rutas
Implementado (Sí/No)	Si

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1 filtrar los datos para las condiciones	O(n) (donde n son el número de tracks en el archivo)
Paso 2 Crear un nuevo mapa con llave el id del lobo y valor una lista con sus tracks	O(n*l) (donde n es el numero de tracks y l cada uno de los lobos)
Paso 3 Organizar la lista de tracks por fecha	O(l*n logn) (donde n es el numero de tracks de un lobo y l es el número de lobos)
Paso 4 Organizar los lobos por distancia recorrida	O(l log l)(donde n es el número de lobos)
Paso 5 iterar sobre los tracks de los lobos extremos	O(l) (donde n es el número de lobos)
TOTAL	***O(l*n logn)***

Pruebas Realizadas

Las pruebas se realizaron en un computador con las siguientes características:

Procesador	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
Memoria RAM (GB)	12.0 GB (11.7 GB utilizable)
Sistema Operativo	Windows 11 Home Single Language – 64 bits

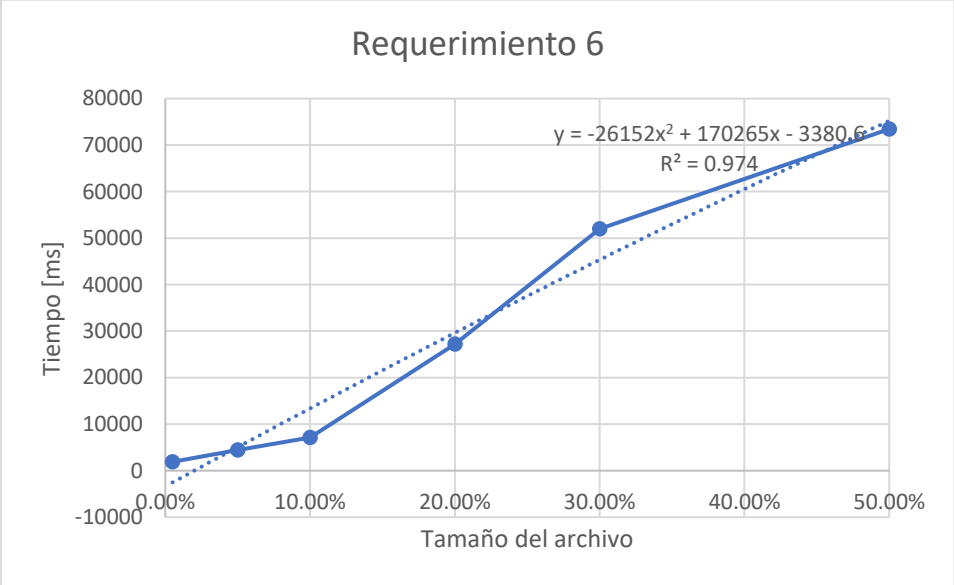
Las pruebas se realizaron con los datos dados en el ejemplo del enunciado.

Tablas de datos

6		
Archivo	Vértices Digrafo	Tiempo [ms]
0.50%	5677	1901.5342
5%	8724	4402.7
10%	15069	7128.991
20%	25516	27229.7692
30%	34636	51978.503

50%	51043	73465.737
80%	73041	
100%	86546	

Graficas



Análisis

A partir de las pruebas realizadas se puede observar como el cálculo de complejidad fue acertado, pues se puede confirmar un comportamiento similar al de nlogn.

Requerimiento 7

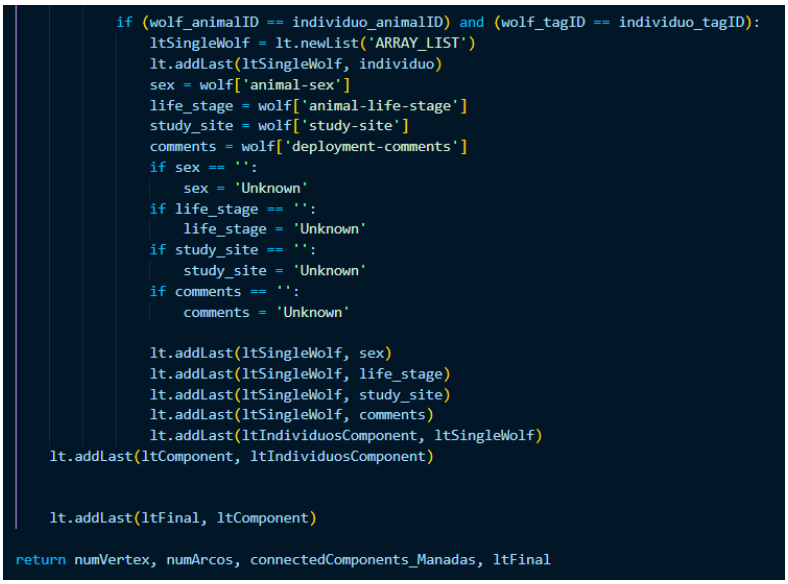
Descripción

```python
def req_7(tracker, dateStart, dateEnd, minTemp, maxTemp):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7

    dateStart = fechaLinda(dateStart)
    dateEnd = fechaLinda(dateEnd)

    data_tracks = tracker['data_tracks']

    data_wolf = tracker['data_wolfs']

    ltTracksFiltered = lt.newList('ARRAY_LIST')

    #Filtro de información para que tenga los parametros de fecha y temperatura

    for track in lt.iterator(data_tracks):

        date = track['timestamp']
        date = fechaLinda(date)

        if dateStart <= date <= dateEnd and minTemp <= track['external-temperature'] <= maxTemp:
            lt.addLast(ltTracksFiltered, track)
        else:
            pass

    #Creación del grafo

    graphManadas = gr.newGraph(datastructure="ADJ_LIST",
                directed=True,
                size=1000,
                cmpfunction=None)

    #Creacion del mapa auxiliar

    mapaManadas = mp.newMap(1000, maptype='PROBING', cmpfunction=compareWolfID)

    #Creación de los nodos

    #Tracking points

    listatrackIDS = getTrackIDsReq7(ltTracksFiltered) #Lista de los ids de los trayectos de los lobos
```

```python
listaMPIDs = getMeetingPointsIDReq7(ltTracksFiltered) #Lista de los ids de los puntos de encuentro de los lobos

graphManadas_pt1 = addTrackNodeReq7(graphManadas, listatrackIDS) #Adiciona los nodos de los trayectos de los lobos al grafo

#Meeting points

listaMeetingPoints = getMeetingPointsIDReq7(ltTracksFiltered) #Lista de los ids de kos trayectos de los lobos

graphManadas_tuple = addMeetingNodeReq7(graphManadas_pt1, listaMeetingPoints) #Adiciona los nodos de los puntos de encuentro al grafo



#Creación de los arcos

addMAPIdTrackReq7(mapaManadas, data_wolf, ltTracksFiltered) #Adiciona los tracks de los lobos al mapa

graphManadas_Connected = connectWolfPointsReq7(graphManadas_tuple, mapaManadas) #Conecta los puntos de seguimiento de los lobos

graphManadas_Final = connectMeetingPointsReq7(graphManadas_Connected, listaMPIDs, data_tracks) #Conecta los puntos de encuentro de los lobos

#Fin de la creación del grafo

#Elementos para la impresión

numVertex = gr.numVertices(graphManadas_Final)
numArcos = gr.numEdges(graphManadas_Final)

#Reconocimiento de los componentes fuertmente conectados de las manadas

mapaKosaraju_Manadas = scc.KosarajuSCC(graphManadas_Final)

ltStrongNodes_Manadas = mp.keySet(mapaKosaraju_Manadas['idscc'])

connectedComponents_Manadas = scc.connectedComponents(mapaKosaraju_Manadas)
```

```
ltStrongNodes_Manadas = mp.keySet(mapaKosaraju_Manadas['idscc'])

connectedComponents_Manadas = scc.connectedComponents(mapaKosaraju_Manadas)

mapaComponentes = mp.newMap(1000, maptype='PROBING', cmpfunction=compareWolfID)

for node in lt.iterator(ltStrongNodes_Manadas):
    componentID = mp.get(mapaKosaraju_Manadas['idscc'], node)['value']
    entry = mp.get(mapaComponentes, componentID)
    if entry is None:
        ltNode = lt.newList('ARRAY_LIST')
        lt.addLast(ltNode, node)
        mp.put(mapaComponentes, componentID, ltNode)
    else:
        ltNode = entry['value']
        lt.addLast(ltNode, node)
        mp.put(mapaComponentes, componentID, ltNode)


ltComponentes = listaFromMap(mapaComponentes)

componentSorted = merg.sort(ltComponentes, compareSize)

ltFinal = lt.newList('ARRAY_LIST')

for component in lt.iterator(componentSorted):
    ltComponent =lt.newList('ARRAY_LIST')
    componentIdentificator, ltStrongNodes = component
    lt.addLast(ltComponent, componentIdentificator) #(ID de la manada)
    lt.addLast(ltComponent, ltStrongNodes) #Lista de nodos fuertemente conectados (manada)
    size = lt.size(ltStrongNodes) #Cantidad de nodos fuertemente conectados (manada)
    lt.addLast(ltComponent, size)

    maxLat = -100000000000
    minLat = 100000000000

    maxLong = -100000000000
    minLong = 100000000000

    wolfCount = 0

    ltIndividuals = lt.newList('ARRAY_LIST')
```

```
    for node in lt.iterator(ltStrongNodes):

        coords = getLocationFromNode(node)

        long = float(coords[0])
        lat = float(coords[1])

        if long > maxLong:
            maxLong = long
        if long < minLong:
            minLong = long
        if lat > maxLat:
            maxLat = lat
        if lat < minLat:
            minLat = lat


        flag = meetingOrTracking(node)

        if flag:
            individual = getIndividualFromNode(node)

            if lt.isPresent(ltIndividuals, individual) == 0:
                lt.addLast(ltIndividuals, individual)
                wolfCount += 1

    lt.addLast(ltComponent, minLat)
    lt.addLast(ltComponent, maxLat)
    lt.addLast(ltComponent, minLong)
    lt.addLast(ltComponent, maxLong)
    lt.addLast(ltComponent, wolfCount)
    ltIndividuosComponent = lt.newList('ARRAY_LIST')
    for individuo in lt.iterator(ltIndividuals):


        individuo_split = individuo.split('_')
        individuo_animalID = individuo_split[0]
        individuo_tagID = individuo_split[1]

        for wolf in lt.iterator(data_wolf):
            wolf_animalID = wolf['animal-id']
            wolf_tagID = wolf['tag-id']
```



```
            if (wolf_animalID == individuo_animalID) and (wolf_tagID == individuo_tagID):
                ltSingleWolf = lt.newList('ARRAY_LIST')
                lt.addLast(ltSingleWolf, individuo)
                sex = wolf['animal-sex']
                life_stage = wolf['animal-life-stage']
                study_site = wolf['study-site']
                comments = wolf['deployment-comments']
                if sex == '':
                    sex = 'Unknown'
                if life_stage == '':
                    life_stage = 'Unknown'
                if study_site == '':
                    study_site = 'Unknown'
                if comments == '':
                    comments = 'Unknown'

                lt.addLast(ltSingleWolf, sex)
                lt.addLast(ltSingleWolf, life_stage)
                lt.addLast(ltSingleWolf, study_site)
                lt.addLast(ltSingleWolf, comments)
                lt.addLast(ltIndividuosComponent, ltSingleWolf)
    lt.addLast(ltComponent, ltIndividuosComponent)


    lt.addLast(ltFinal, ltComponent)

return numVertex, numArcos, connectedComponents_Manadas, ltFinal
```

En primer lugar, se utilizó una lógica similar a la carga de datos puesto que se creó un dígrafo a partir de los tracks filtrados con respecto a los parámetros de entrada. Posteriormente, para solucionar el requerimiento 7 se implementó el algoritmo de Kosaraju para obtener los componentes fuertemente conectados del dígrafo creado en la carga de datos. Posteriormente, para obtener la información solicitada por requerimiento se creó un mapa de los componentes fuertemente conectados cuya llave es el ID de dicho componente y tiene como valor un arreglo de todos los nodos que están en dicho componente. Por consiguiente, se obtuvo una lista de lista de mapa para poder iterar utilizando la librería de DISClib esta lista se ordena por tamaño utilizando Merge sort. A partir de esto se itera sobre cada componente para poder iterar sobre cada lista perteneciente a este para obtener la información solicitada. Por último, se itera sobre los datos de los lobos de tal forma que se termina la lista para retornar la información necesaria.

Entrada	Tracker, fecha de inicio, fecha final, temperatura inicial, temperatura final
Salidas	Numero de vértices, número de arcos, número de componentes fuertemente conectados, lista con la información solicitada.
Implementado (Sí/No)	Si

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: Filtrado de información a partir de los parámetros de entrada	O(n), donde n es el número de tracks del archivo
Paso 2: Creación del grafo	$O(n^2)$, donde n es la cantidad de tracks que cumplen los parámetros de entrada
Paso 1: Implementación del algoritmo de Kosaraju	O(V+E)
Paso 2: Creación del mapa del algoritmo de Kosaraju	O(n), donde n es el número de vértices fuertemente conectados.

Paso 3: Creación de la lista de listas del mapa	O(n), donde n es el número de componentes fuertemente conectados
Paso 4: Merge sort para ordenar la lista por tamaños	O(nlog(n))
Paso 5: Iterar sobre los componentes fuertemente conectados para obtener la información solicitada.	$O(n^2)$, donde n es el número de componentes fuertemente conectados
TOTAL	***$O(n^2)$***

Pruebas Realizadas

Las pruebas se realizaron en un computador con las siguientes características:

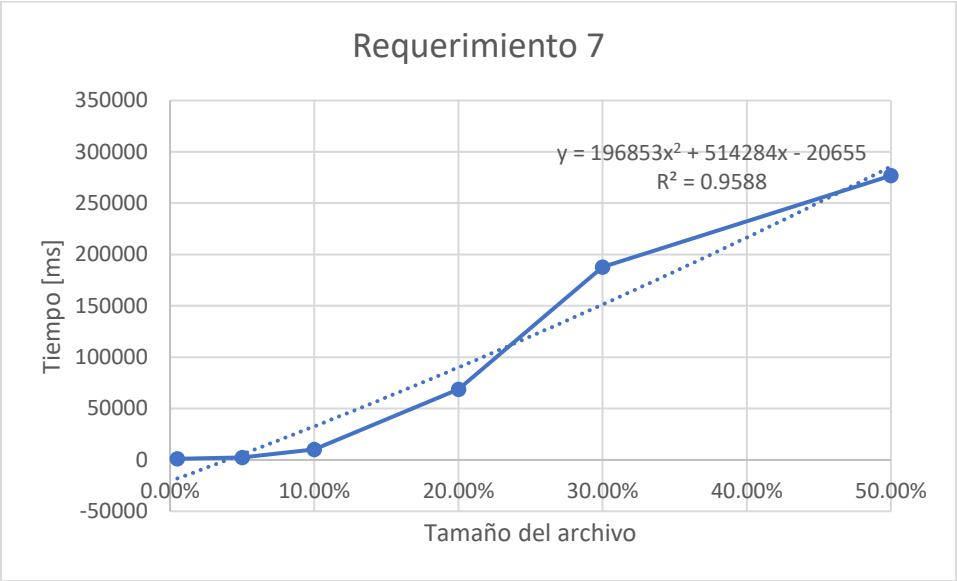
Procesador	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
Memoria RAM (GB)	12.0 GB (11.7 GB utilizable)
Sistema Operativo	Windows 11 Home Single Language – 64 bits

Las pruebas se realizaron con los datos dados en el ejemplo del enunciado.

Tablas de datos

7		
Archivo	Vértices Digrafo	Tiempo [ms]
0.50%	5677	1109.37018
5%	8724	2341.4144
10%	15069	10161.2116
20%	25516	68936.278
30%	34636	187953.889
50%	51043	276838.016
80%	73041	
100%	86546	

Graficas



Análisis

Se puede observar como la gráfica tiene un comportamiento cuadrático, confirmando así la hipótesis enunciada de que la complejidad temporal del requerimiento 7 es de $O(n^2)$

Diagrama de las estructuras de datos usadas